

Notions of computability at higher types I

John Longley

August 4, 2001

Abstract

We discuss the conceptual problem of identifying the natural notions of computability at higher types (over the natural numbers). We argue for an eclectic approach, in which one considers a wide range of possible approaches to defining higher type computability and then looks for regularities. As a first step in this programme, we give an extended survey of the different strands of research on higher type computability to date, bringing together material from recursion theory, constructive logic and computer science. The paper thus serves as a reasonably complete overview of the literature on higher type computability. Two sequel papers will be devoted to developing a more systematic account of the material reviewed here.

0 Introduction

In elementary recursion theory, one begins with the question: what does it mean for a (total or partial) function on the set \mathbb{N} of natural numbers to be “computable”? As is well known, many different approaches to defining a notion of computable function — via recursion schemas, Turing machines, lambda calculus, Markov algorithms, flowcharts etc. — lead to the same answer, namely the class of (total or partial) recursive functions. Indeed, *Church’s thesis* proposes that we identify the informal notion of *effectively computable* function with the precise mathematical notion of *recursive* function.

The fact that so many *prima facie* independent mathematical characterizations yield the same class of functions strongly suggests that, if nothing else, the recursive functions constitute a very robust and mathematically natural class of functions. Moreover, since many of the individual characterizations are based on intuitively appealing formalizations of the idea of computation, and since no other serious contenders for a class of effectively computable functions are known, most of us are happy to accept Church’s thesis most of the time.

We might now ask: what does it mean for a function of higher type to be “computable”—say, a second order function taking first order functions on \mathbb{N} as arguments, or even a third order function taking such second order functions as arguments? A moment’s reflection shows that many choices confront us if we wish to formulate a definition of higher type computability. For example:¹

¹Many of these points are also made previously in a survey article by Cook [Coo90], whose point of view is close to ours in many respects.

- **Domain of definition.** Do we want to consider partial or total computable functions? Do we want them to act on partial functions of the next type down, or just on total functions? Should they act only on the “computable” functions of this type, or on “all” functions in some wider class?
- **Representation of functions.** If we wish to perform “computations” on functions, how do we regard the functions as given to us? As infinite mathematical objects (e.g. as graphs)? As oracles or “black boxes” for which only the input/output behaviour is visible? As algorithms or “programs” of some kind? If so, what kind of programs?
- **Protocol for computation.** What ways of interacting with functions do we allow in computations? For example, do we insist that calls to functions are performed sequentially, or do we allow parallel function calls? Do we insist that (terminating) computations are in some sense finite objects — as must presumably be the case if we are seeking a genuinely effective notion of computability — or do we allow infinite computations in accordance with the infinite nature of the arguments?
- **Extensionality.** Do we want to restrict our attention to computable *functions* (as assumed in the previous points), or do we want to consider computability for other, possibly non-extensional, operations of higher type? If the latter, what do we mean by an “operation”?

The spirit in which we are asking these questions is not to demand definitive answers to them, but to make the point that many choices are possible. Indeed, as we shall see, the definitions of higher type computability proposed in the literature exemplify many different responses to the above issues. Moreover, the effects of these choices escalate rapidly as we climb up the types: for example, if two definitions yield different classes of computable functions of type σ , then it may be difficult even to compare these definitions at type $\sigma \rightarrow \mathbb{N}$, since the domains of the functions may differ.

It is thus clear that very many approaches to defining higher type computability are possible, but it is not obvious *a priori* whether some approaches are more sensible than others, or which approaches lead to equivalent notions of computability. In short, it is unclear in advance whether at higher types there is really just one natural notion of computability (as in ordinary recursion theory), or several, or no really natural notions at all.

This paper is the first of a planned series of three articles devoted to the conceptual problem of finding good, natural notions of higher type computability.² Whereas previous work has explored various *particular* notions of computability in some detail, we wish to take a step back and look at the overall picture, with a view to identifying the important notions. Our main objectives are as follows:

²In using the word “natural” here we are appealing to the intuitive idea, shared by many practising mathematicians and expressed by them in a variety of ways, that the mathematical landscape is not homogeneous but that some mathematical objects or concepts have greater intrinsic importance than others. It seems to us that one can adopt this idea whether or not one holds a Platonist view of mathematical truth.

- To discover what natural notions of computability exist at higher types, and to collect evidence for their naturalness.
- To develop some basic “recursion theory” for each of these notions, analogous to the elementary parts of ordinary recursion theory.
- To investigate how these notions of computability are related.
- To provide a coherent framework for pulling together and organizing the existing knowledge in the area.

Many ideas and results relevant to our enterprise are already known, though they are rather widely scattered across the literature in recursion theory, constructive logic and computer science, and have never previously been presented together as contributions to a single subject. In the present paper we will give a fairly comprehensive survey of the work to date on different approaches to higher type computability — this will amass some raw material for our project. In two sequel papers, we will present a more systematic view of much of this material, proposing some simple general frameworks for discussing the “space of possible notions of computability”, and showing that within these frameworks a reasonably cohesive picture does indeed emerge from the disparate strands.

To expand on our working philosophy a little further: It appears that *a priori* considerations are by themselves of limited use in determining what are the natural notions of higher type computability — any definition one can write down involves some choices which might be felt to be arbitrary. We are therefore to adopt a more empirical attitude: we can explore a range of possible definitions, and see what natural notions emerge. Various criteria may be used to determine which notions of computability count as “natural”, for instance:

- Whether they admit a wide range of independent characterizations—the more independent the better.
- Whether they arise from some intuitively appealing concept of “computation”.
- Whether they occupy some special position within the space of all possible notions of computability.

There has already been much research over the last fifty years exploring different approaches to higher type computability, and we feel the time is now opportune for bringing this material together and trying to make sense of the big picture. Here we favour an eclectic attitude — since we do not know in advance where to look for good notions of computability, we should cast our net as wide as possible and embrace the diversity of definitions that have been proposed in the literature.

To anticipate the outcome of our project, we will argue that, for computable *functionals* at least, there is in fact a handful of six or seven natural and robust notions of higher type computability, each with a variety of different characterizations and some pleasing intrinsic properties, and with some interesting relationships between them. Although it is possible that there are other equally natural notions of computable functional awaiting discovery, the fact that very

many attempts to defining a notion of computable functional lead to one of the known notions suggests, in the author’s opinion, that the current picture is probably reasonably complete. For non-extensional notions of computability, the situation is at present more open-ended, but we are at least able to unify much of what is currently known in a satisfying way.

I am very grateful to the organizers of the Logic Colloquium for giving me the opportunity to present much of this material in a series of tutorials, for providing me with the stimulus to write this paper, and for waiting so long for me to finish it. Many people have helped me to fill gaps in my knowledge of the area, including Samson Abramsky, Ulrich Berger, Solomon Feferman, Martin Hyland, Dag Normann, Gordon Plotkin and Stan Wainer. This research was supported by the EPSRC Research Grant GR/L89532 “Notions of computability for general datatypes”.

0.1 Motivations

Before proceeding further, we should mention some of the reasons why computability at higher types is potentially interesting. Besides its intrinsic mathematical and conceptual interest, the subject lies at an intriguing juncture between several areas of mathematical logic and computer science, and has (potential or actual) connections with the following areas. For reasons of space, however, we will say relatively little about these applications in the rest of the paper, choosing rather to focus on clarifying the fundamental notions of the subject.

0.1.1 Constructive logic and metamathematics

Historically, the first applications of the ideas of higher type computability were to the metamathematics of constructive systems. Computable objects of finite type can often be used to give interpretations of logics — e.g. *realizability* interpretations — that embody some kind of constructive content. On a technical level, such interpretations can be used to obtain consistency and independence results; on a conceptual level, they can be helpful for clarifying various constructive views of mathematics (often from a classical standpoint).

An good early discussion of possible applications of this kind appears in Kreisel [Kre59, Sections 1,2]. The area was developed above all by Troelstra and his school [Tro73], who focused on realizability and related interpretations. In a somewhat similar spirit is Feferman’s use of computable objects of higher type to give interpretations for theories of finite type [Fef77b]. Although in this case the logical systems are classical, they are typically intended to reflect “semi-constructivist” viewpoints that suffice to support most of mathematical practice.

0.1.2 Descriptive set theory

Logical quantifiers may be regarded as objects of higher type: for instance, existential quantification over the natural numbers can be seen as a (non-computable) object ${}^2\exists : (\mathbb{N} \rightarrow 2) \rightarrow 2$, where $2 = \{0, 1\}$. There are interest-

ing relationships between computability relative to such quantifiers and logical complexity: for instance, a function on \mathbb{N} is computable relative to ${}^2\exists$ (in a certain sense) iff it is hyperarithmetical (see Section 2.2 below). This aspect of higher type recursion theory was one of the motivations behind Kleene’s early work, and has been further developed by Moschovakis, Sacks and others (see e.g. [Sac99]).

0.1.3 Semantics and logic of programming languages

In computer science it is natural to want to understand the notion of “computability” embodied by a given programming language, and it turns out that different languages embody different such notions. Finding mathematical characterizations of these notions is often tantamount to finding well-matched semantic models for the programming language; as argued in [LP97, Lon99a], this can often help us to design a good program logic for the language. The finite (simple) types are a good target for study because many other important datatypes (e.g. lazy lists and functions on them) arise naturally as *retracts* of simple types.

In addition, an understanding of higher type computability is likely to contribute to the semantics of object oriented programming languages, since these typically support higher order styles of programming.

0.1.4 Complexity at higher types

Various ideas from higher type computability have informed the definition of complexity-theoretic notions (see e.g. [Coo90]), but it appears that many of the basic definitions of higher type complexity are still open to discussion. Indeed, in order to formulate good notions of feasible computation at higher types (for instance), it seems that one will have to confront all the choices mentioned above, and others besides. It therefore seems that a clear understanding of the basic notions of higher type computability should provide a good starting-point for an attempt to elucidating the fundamental notions of higher type complexity. The current state of the art is described in detail in [IKR01a, IKR01b].

Similar remarks apply to notions of *subrecursion* at higher types, as considered in [Sch91, Nig93].

0.1.5 Real number computability

The connection between computability over the reals and higher type computability was recognized as far back as [Lac55a, Lac55b, Lac55c], and later manifested itself in the context of constructive interpretations of systems for analysis (e.g. in [Tro73]). Notions of computability over the reals and other metric spaces underpin constructive recursive analysis of the Markov school on the one hand (see [Abe80, Bee85]), and classical computable analysis on the other (see [PER89, Wei00]). Interest in real number computability has also recently been reawakened within computer science (see e.g. [Esc96]), and exact real number computation appears as an attractive application area for higher

type programming. Higher types over the reals and the associated notions of computability have recently been considered by Normann [Nor98, Nor00b].

0.1.6 Computability in physics

Notions of computability in analysis can in turn be applied to questions of computability in various physical theories. Some of this territory is explored in the book by Pour-El and Richards [PER89], who work with one particular definition of computability, but it is not immediately clear whether this definition is the only one of interest. An appreciation of the mathematically possible notions of computability should therefore put us in a better position for discussing issues of computability in physics.

0.2 Overview of the series

This series of articles is intended as a fairly comprehensive account of what is currently known about higher type computability over the natural numbers.

The present Part I is a historical survey of the literature on higher type computability, tracing the various strands of research which have contributed to our present understanding. This is intended to serve various purposes: firstly, to offer a gentle introduction to the main ideas of the subject; secondly, to document the genesis of these ideas; thirdly, to facilitate comparison between different strands of work by placing them side by side; and fourthly, to provide a reasonably complete map of the rather bewildering literature of the subject. We have tried to include just enough technical detail to make the mathematical substance intelligible, without losing the broad sweep of the story. The point of view we wish to advocate is that all the strands that we describe can fruitfully be seen, with hindsight, as contributions to a single coherent subject; in the remaining papers in the series we will attempt a more systematic exposition of this subject.

In Part II [Lon01a], we will try to organize most of what is known about notions of computable *functional* (that is, extensional operations of higher type), and the relationships between them. Here we will work within a general framework given by some simple definitions involving *finite type structures*. Although this framework is very simple and somewhat crude, it suffices for clarifying much of the existing material. After developing the necessary general concepts, we will consider separately the various good notions of *total* and *partial* computable functional. In both the total and partial settings, we present arguments for the impossibility of a “Church’s thesis for higher types”. We then discuss ways in which total and partial functionals can be related or combined.

In Part III [Lon01b] we will consider, more generally, notions of computable *operation* (not necessarily extensional) — for example, the notions of computability embodied by various non-functional programming languages. In order to articulate these notions, we use a more sophisticated general framework based on ideas of *realizability* — this extends and refines the theory of Part II in a mathematically satisfying manner. Once again, we develop the general theory, then survey within this framework some of the notions of computability

that appear to have some claim to naturalness.

Naturally, much of the material covered in Part I will be treated again from a different perspective in Parts II and III. We feel that this kind of overlap is justified in the interests of presenting a rounded view that takes account of both the historical and the purely logical aspects of the subject. Indeed, the historical and logical parts of the series are intended to be complementary, in the sense that each tends to emphasize ideas that receive scant attention in the other.

0.3 Outline of the present paper

As a glance at Figure 1 will confirm, the study of higher type computability has not developed in a coherent, orderly fashion. Rather, it is mostly the result of the parallel activity of several research communities, each with their own set of motivations, and it is only in retrospect that the various strands can be seen as parts of a coherent subject. It is therefore not surprising that the history of the subject appears as somewhat chaotic.

The parallel nature of the subject's development, in particular, makes the history difficult to describe: some compromise between a strictly chronological presentation and a thematic one is necessary, and no linear ordering of the material seems completely satisfactory for an expository point of view. Here we have adopted the following course. In Section 1 we describe, more or less chronologically, the early work on computability at type 2, taking us up to about 1958. Around this time, several notions of computability at all higher types made their debut; at this point the subject effectively split into several streams, and it is only in the last few years that these have started to converge again. For the main body of the paper (Sections 2 and 3) we therefore treat each of the main notions of higher type computability in turn, taking them (roughly) in order of their first appearance in the literature, and giving separate chronological accounts of the developments relating to each of them. In Section 2 we discuss around four different notions of *total* computable functional, and in Section 3 we consider a similar number of notions of *partial* computable functional. (Fortunately for our scheme, all of the total notions made their first appearance before any of the partial ones!) Turning to more recent developments, in Section 4 we describe some ideas from realizability which cross-cut many of these streams. Finally, in Section 5 we briefly discuss a few strands of research relating to *non-functional* notions of “computable operation” at higher types.

We presuppose a knowledge of basic recursion theory, the simply typed lambda calculus, and basic category theory (including cartesian closed categories).

0.4 Basic definitions and notation

The following basic concepts are central to the entire paper and will be used ubiquitously.

We have in mind the types given by the following grammar:

$$\sigma ::= \bar{0} \mid \sigma_1 \rightarrow \sigma_2$$

where $\bar{0}$ represents the type of natural numbers, and $\sigma_1 \rightarrow \sigma_2$ represents the type of functions from σ_1 to σ_2 . For convenience, we define the *pure types* \bar{n} inductively by $\overline{n+1} = \bar{n} \rightarrow \bar{0}$; we also define the *level* of a type inductively by

$$\text{level } \bar{0} = 0, \quad \text{level } \sigma \rightarrow \tau = \max(1 + \text{level } \sigma, \text{level } \tau).$$

We will also make incessant use of the notion of *type structure*. For the present purposes, a type structure A will consist of a family of sets A_σ (one for each type) together with “application” operations $\cdot_{\sigma\tau} : A_{\sigma \rightarrow \tau} \times A_\sigma \rightarrow A_\tau$. (A more comprehensive armoury of definitions relating to type structures will be given in Part II.) By a *type n object* of A we will mean an element of some A_σ where $\text{level } \sigma = n$. For simplicity we will often write A_n in place of $A_{\bar{n}}$. We say A is a type structure *over* X if $A_{\bar{0}} = X$. A type structure A is *extensional* if for all types σ, τ and all $f, g \in A_{\sigma \rightarrow \tau}$ we have

$$(\forall x \in A_\sigma. f \cdot x = g \cdot x) \implies f = g.$$

Thus, the extensional type structures are those in which the objects are (essentially) functions; for such type structures, we will often refer to objects of type ≥ 2 as *functionals*.

A more comprehensive armoury of definitions relating to type structures will be presented in Part II, along with a discussion of the relationships between alternative definitions.

We write \mathbb{N} for the set of natural numbers (including 0); in general we use N for the natural number object in a category. For any set X , we write X_\perp for the set $X \sqcup \{\perp\}$. We sometimes tacitly identify partial functions $X \rightarrow Y$ with total functions $X \rightarrow Y_\perp$. We also write:

- $(\mathbb{N} \rightarrow \mathbb{N})$ or $\mathbb{N}^{\mathbb{N}}$ for the set of all (set-theoretic) total functions from \mathbb{N} to \mathbb{N} ,
- $(\mathbb{N} \rightarrow \mathbb{N})$ or $\mathbb{N}_\perp^{\mathbb{N}}$ for the set of all partial functions from \mathbb{N} to \mathbb{N} ,
- $(\mathbb{N} \rightarrow \mathbb{N})_{\text{rec}}$ or $\mathbb{N}_{\text{rec}}^{\mathbb{N}}$ for the set of total recursive functions from \mathbb{N} to \mathbb{N} ,
- $(\mathbb{N} \rightarrow \mathbb{N})_{\text{rec}}$ or $\mathbb{N}_{\perp \text{rec}}^{\mathbb{N}}$ for the set of partial recursive functions from \mathbb{N} to \mathbb{N} .

We write $\text{Seq}(X)$ for the set of finite sequences over a set X ; we will use the notation $[x_1, \dots, x_n]$ to display such sequences. We will suppose $\langle - \rangle : \text{Seq}(\mathbb{N}) \rightarrow \mathbb{N}$ is some fixed effective coding for finite sequences, and write $\langle x_1, \dots, x_n \rangle$ in place of $\langle [x_1, \dots, x_n] \rangle$. Given $f : \mathbb{N} \rightarrow \mathbb{N}$, we define $\bar{f} : \mathbb{N} \rightarrow \mathbb{N}$ by

$$\bar{f}(n) = \langle f(0), \dots, f(n-1) \rangle.$$

We also suppose we have some effective indexing scheme for the partial recursive functions, given for example by an effective enumeration of Turing

machines. For $e \in \mathbb{N}$, we will write ϕ_e or for the partial recursive function $\mathbb{N} \rightarrow \mathbb{N}$ with recursive index e . We will sometimes write $e \bullet x$ in place of $\phi_e(x)$ (*Kleene application*).

We will use the following notational conventions in connection with potentially non-denoting expressions:

- $e \downarrow$ means “the value of e is defined”.
- $e \uparrow$ means “the value of e is not defined”.
- $e = e'$ means “the values of e and e' are both defined and they are equal”.
- $e \simeq e'$ means “if either e or e' is defined then so is the other, and their values are equal” (Kleene equality).
- $e \succeq e'$ means “if e' is defined then so is e and they are equal”.

We will generally use boldface letters as abbreviations for vectors, or lists of variables, when we do not care exactly how many elements there are. More precisely, a symbol such as \mathbf{x} , wherever it occurs, will textually abbreviate $x_1 \dots x_{l_x}$ or x_1, \dots, x_{l_x} (as demanded by the context), where $l_x \geq 0$ is regarded as a fresh variable associated with \mathbf{x} .

1 Early work: Computability at types 1 and 2

1.1 Type 1 computability

The main ideas concerning computability for type 1 functions of course date back to the development of basic recursion theory in the 1930s by Gödel [Göd31], Church [Chu36], Turing [Tur36, Tur37], Kleene [Kle36a, Kle36b] and Post [Pos36], who accumulated several characterizations of the class of (partial) recursive functions as mentioned in the introduction. The first explicit formulation of Church’s thesis appears in [Chu36]. In [Tur39, Section 4] Turing introduced the notion of a computing machine equipped with an oracle for deciding non-computable properties, but considered this only as a means of defining type computability relative to a *fixed* oracle, so cannot truly be said to have introduced the concept of a computable type 2 function.

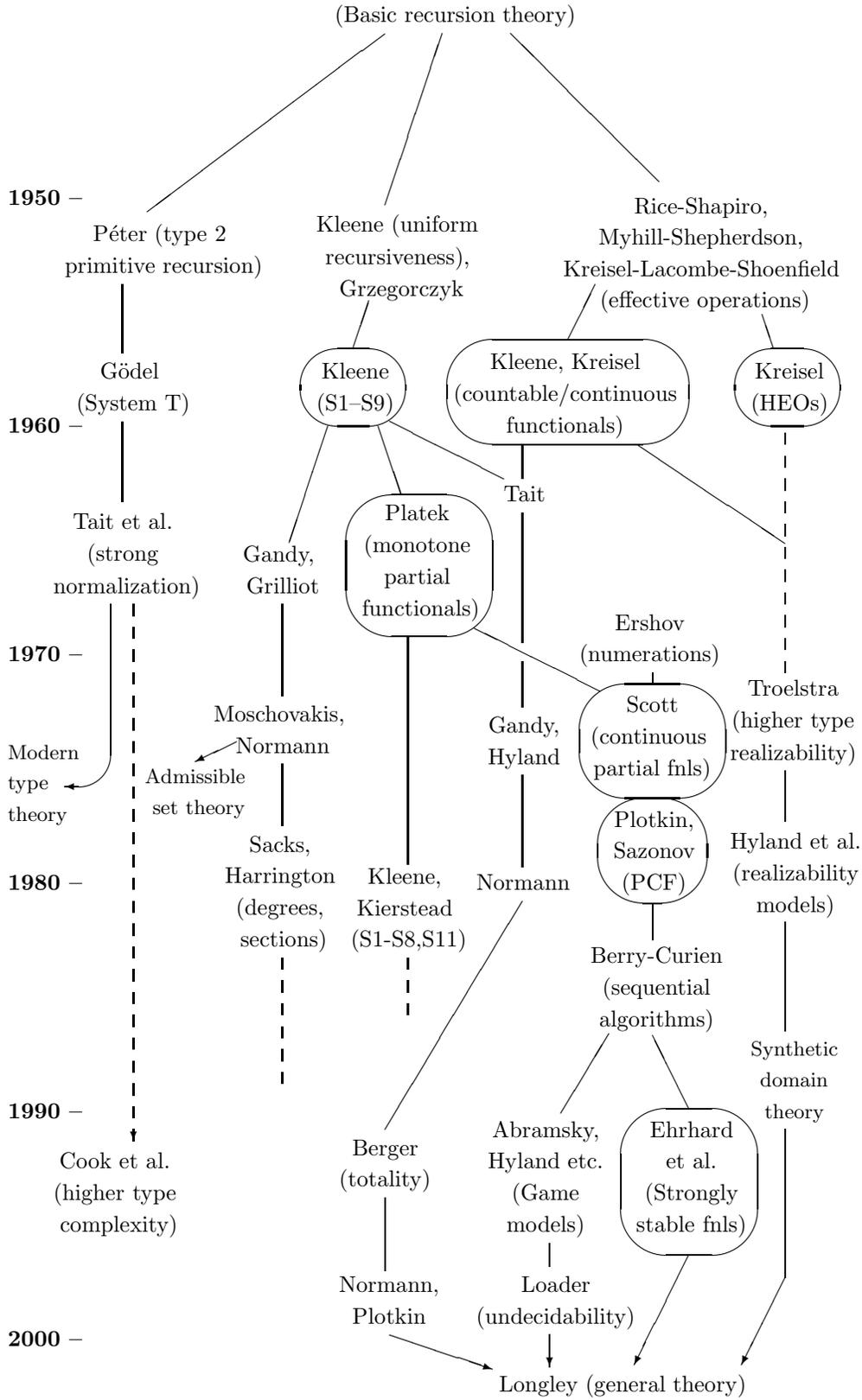
1.2 Banach-Mazur functionals

A very early definition of a class of type 2 functionals involving a notion of computability is due to Banach and Mazur [BM37] (see also [Maz63]):

Definition 1.1 *A total function $F : (\mathbb{N} \rightarrow \mathbb{N})_{\text{rec}} \rightarrow \mathbb{N}$ is Banach-Mazur if, for every total recursive function $h : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, the function $\lambda x.F(\lambda y.h(x, y)) : \mathbb{N} \rightarrow \mathbb{N}$ is total recursive.*

Notice that this condition says that, in some sense, F carries computable functions to computable functions, but it does not tell us how given g one might compute $F(g)$ in any sense. For this reason, the notion is rather tangential to the story we tell here — we do not regard it as a genuine candidate for a notion

Figure 1: History of higher type computability: a selective outline.



of computable functional, but rather as a property which computable functionals may possess. Early results showed that every computable functional (in senses discussed below) is Banach-Mazur but not *vice versa* (see [Fri58a]); various relationships to other conditions on functionals are also studied in [PE60]. Most of this material is helpfully summarized in [Rog67, §15.3].

The Banach-Mazur functionals later reappear in the work of Lawvere and Mulry [Mul82], whose *recursive topos* provides a natural generalization of the notion to higher types.

1.3 Computations on pure functions

The first explicit definition of a genuine notion of type 2 computability, as far as we are aware, was given by Péter in [Pét51a] and [Pét51b, Chapter 13]. Here Péter considered a schema for “primitive recursion of the second degree” as a means of defining total functions with arguments of type $(\mathbb{N} \rightarrow \mathbb{N})$ as well as \mathbb{N} . Sacrificing some generality for the sake of clarity, the basic idea is as follows: from functions G, H one may construct a function F such that

$$\begin{aligned} F(0, x, g) &= G(x, g), \\ F(n + 1, x, g) &= H(n, x, \lambda y. F(n, y, g), F(n, x, g)). \end{aligned}$$

Péter showed, moreover, that a certain class of “transfinite recursions” at type 1 could be systematically replaced by primitive recursions at type 2; thus, the Ackermann function, though not primitive recursive in the usual sense, could be defined by a type 2 primitive recursion.

In his famous book [Kle52], Kleene gave schemata for defining primitive, total and partial recursive functions *uniformly in* a finite list of functions of type $(\mathbb{N} \rightarrow \mathbb{N})$ (§47, §58, §63), and made explicit the possibility of regarding these as type 2 functionals. (However, Kleene’s notion of primitive recursion was weaker than Péter’s, because whereas Péter’s scheme allows type 1 functions to be varied during the course of a recursive computation, in Kleene’s definition the type 1 functions enter the computation simply as additional “basic functions” which remain fixed throughout. Thus, Kleene’s definition yields just the usual class of primitive recursive functions at type 1.) Kleene also showed that his partial recursive functionals can be characterized equivalently as those computable by Turing machines with oracles (*op. cit.*, Chapter XIII).³

In two papers from 1954, Grzegorzcyk [Grz55b, Grz55a] considered a class of computable total functionals with arguments in \mathbb{N} and $(\mathbb{N} \rightarrow \mathbb{N})$, defined as the smallest class containing some basic functionals and closed under substitution and minimization operations. It is fairly easy to see directly (and is immediate from Kleene’s later results — see Section 2.2 below) that Grzegorzcyk’s notion coincides with Kleene’s notion of uniform total recursiveness, though this does

³Kleene introduced, somewhat peripherally, a notion of partial recursiveness uniformly in a list of *partial* functions $\mathbb{N} \rightarrow \mathbb{N}$ [Kle52, §63], but did not study it in detail. As pointed out by Platek [Pla66, pp. 128–130], Kleene’s particular definition has some rather undesirable features; nevertheless, it turns out to give rise to the class of parallel-computable type 2 functions, see Section 3.

not seem to have been noted in the literature of the time. In [Grz55b] Grzegorzczuk showed that his definition was equivalent to an Herbrand-Gödel style definition in terms of an equational calculus. In [Grz55a] he investigated some properties of these functionals, notably a *continuity* property and the existence of certain computable “modulus of continuity” functionals.

The relationship between computability and continuity was to become a recurring theme in the subject. In the case of Grzegorzczuk’s result (and others in the same vein) the basic intuition is simple: if we apply a computable type 2 function F to a type 1 function g , the “computation” of $F(g)$ can only interrogate g at finitely many arguments, so for any other function g' agreeing with g on this finite portion we would have $F(g) = F(g')$. Thus F is continuous with respect to the familiar Baire topology.

1.4 Computations on Kleene indices

All the approaches to type 2 computability mentioned so far share the feature that the type 1 arguments are presented simply as oracles or “black boxes”. In parallel with this was another strand of research which considered notions of type 2 computability in which the type 1 arguments were presented as recursive indices (say, as Gödel numbers for Turing machines or for recursive definitions in some formal language). Given a recursive index we can of course do everything that we can do with an oracle, since we can always apply the index to a type 0 argument. However, it would seem *a priori* that one might be able to do more with an index than with an oracle, since we are given extra *intensional* information — intuitively, we can “look inside” the black box.

The following definition will allow us to state the main results succinctly:

Definition 1.2 (Effective operation) *Let R be either of the sets $(\mathbb{N} \rightarrow \mathbb{N})_{\text{rec}}$, $(\mathbb{N} \rightarrow \mathbb{N})_{\text{rec}}$. A partial function $F : R \rightarrow \mathbb{N}$ is a partial effective operation on R if there exists a partial recursive function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that for any $e \in \mathbb{N}$ with $\phi_e \in R$ we have $F(\phi_e) = f(e)$. If additionally F is total, we say it is a total effective operation on R .*

In fact, all four possible combinations of total and partial function spaces were investigated in the early literature. First, the theorem of Rice [Ric53], originally phrased in terms of decidable properties of r.e. sets, essentially says the following:

Theorem 1.3 (Total acting on partial) *Every total effective operation on $(\mathbb{N} \rightarrow \mathbb{N})_{\text{rec}}$ is constant.*

This led, via the theorem of Rice-Shapiro [Ric56], to the following theorem due to Myhill and Shepherdson [MS55] and independently to Uspenskii [Usp55]. A similar result was obtained by Nerode [Ner57]. Here we suppose $\theta \mapsto \hat{\theta}$ is an effective coding of the graphs of *finite* partial functions $\theta : \mathbb{N} \rightarrow \mathbb{N}$ as natural numbers.

Theorem 1.4 (Partial acting on partial) *A function $F : (\mathbb{N} \rightarrow \mathbb{N})_{\text{rec}} \rightarrow \mathbb{N}$ is a partial effective operation iff*

- F is monotone and continuous (i.e. F preserves existing lubs of chains in $(\mathbb{N} \rightarrow \mathbb{N})_{\text{rec}}$), and
- F acts effectively on finite elements (i.e. there is a partial recursive function h such that for every finite $\theta : \mathbb{N} \rightarrow \mathbb{N}$ we have $F(\theta) = h(\tilde{\theta})$).

Moreover, every partial effective operation F extends uniquely to a monotone and continuous function $\bar{F} : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$.

The other main positive result was obtained slightly later by Kreisel, Lacombe and Shoenfield [KLS57, KLS59], and independently by Čaitin [Č59]. Our formulation here is somewhat less general than the original version.

Theorem 1.5 (Total acting on total) *A function $F : (\mathbb{N} \rightarrow \mathbb{N})_{\text{rec}} \rightarrow \mathbb{N}$ is a total effective operation iff it is the restriction of a uniformly partial recursive function $\bar{F} : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$ (in Kleene’s sense) whose domain contains $(\mathbb{N} \rightarrow \mathbb{N})_{\text{rec}}$.*

It follows easily that every total effective operation on $(\mathbb{N} \rightarrow \mathbb{N})_{\text{rec}}$ is continuous with respect to the usual Baire topology. Proofs of Theorem 1.5 may be found in many texts, e.g. [Rog67, Bee85, Abe80, Odi89]. A very short proof using Kleene’s second recursion theorem was given by Gandy [Gan62], but it is too serendipitous for most ordinary mortals.

Theorems 1.4 and 1.5 both provide very striking examples of the connection between computability and continuity. These results obviously say something deeper than the result of Grzegorzczuk mentioned earlier, since here we place no restriction on how recursive indices may be manipulated except for the requirement of extensionality: we must get the same answer from all possible indices for a type 1 function. However, the final theorem of our quartet, due to Friedberg [Fri58b], shows that the connection breaks down if totality and partiality are mixed:

Theorem 1.6 (Partial acting on total) *There exists a partial effective operation on $(\mathbb{N} \rightarrow \mathbb{N})_{\text{rec}}$ which is not continuous (hence not the restriction of a Kleene partial recursive functional on $(\mathbb{N} \rightarrow \mathbb{N})$).*

Most of the above theorems (and their proofs) are covered in [Rog67, §15.3].

Most of the remaining natural questions about type 2 computability turn out to have negative answers. For example, not every total effective operation $(\mathbb{N} \rightarrow \mathbb{N})_{\text{rec}} \rightarrow \mathbb{N}$ is the restriction of a uniformly total recursive function $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$. This is shown by the celebrated *Kleene tree*, one of the most important counterexamples in the subject.

Theorem 1.7 (Kleene tree) *There exists a binary tree K (i.e. a prefix-closed set of finite sequences over $\{0, 1\}$) such that*

- K is primitive recursive (that is, there is a primitive recursive function f such that $f(\alpha) = 0$ iff $\alpha \in K$).
- K contains finite paths of arbitrary length (hence, by König’s Lemma, K contains infinite paths).

- K contains no recursive infinite paths.

The Kleene tree is so named after its appearance as Theorem LII of [Kle59b], although examples of the same phenomenon also appear in [Lac55d] and [Zas62, ZC62]. We may now obtain a total effective operation κ that does not extend to a total recursive function on $\mathbb{N}^{\mathbb{N}}$, by defining

$$\kappa(f) = \mu n. [f(0), \dots, f(n-1)] \notin K.$$

A few other negative results will help to complete the picture at type 2. A simple example due to Pour-El [PE60] (see [Rog67, §15.3, Theorem XXXV]) shows the failure of a plausible analogue of Theorem 1.4 for total functionals acting on total functions: not every continuous functional which acts effectively on a suitable basis of “finite” functions (namely, the eventually constant functions $\mathbb{N} \rightarrow \mathbb{N}$) is an effective operation. Finally, the analogue of Theorem 1.5 for partial functionals acting on partial functions fails because the uniformly partial recursive functions $(\mathbb{N} \multimap \mathbb{N}) \multimap \mathbb{N}$ are all *sequentially* computable whereas the partial effective operations include *parallel* functions (see Section 3.2) — though this distinction seems not to have been made explicitly until [Pla66].

These early results show that the situation at type 2 is already quite complicated, and they serve to illustrate many of the general characteristics of the whole subject. Firstly, they exhibit a considerable diversity of approaches to defining a notion of “computable functional”. Secondly, they show how quite different approaches sometimes lead to the same class of functionals, providing evidence for the intrinsic importance of this class (Theorem 1.5 is a good example of this). On the other hand, we have seen that not all definitions of computability conveniently collapse to a single notion, and that an important role is played by negative results and counterexamples. Finally, we have already seen several instances of the connection between computability and continuity.

2 Total computable functionals

Once various notions of type 2 computability had been considered, the possibility of trying to extend them to higher types was obvious. (Péter [Pét51b] speculated informally on this possibility, but did not develop it.) The years 1957–59 saw the appearance of no fewer than four important notions of higher type computability, represented by Gödel’s System T, Kleene’s schemata S1–S9, the Kleene-Kreisel countable or continuous functionals (and their effective substructure), and Kreisel’s hereditarily effective operations. It is worth noting that all these notions were concerned with hereditarily *total* functionals — good notions of computability for partial functionals of all higher types were a later development. (see Section 3).

Roughly speaking, each of the above notions gave rise to its own strand of research, so that the study of higher type computability appears as somewhat fragmented from 1960 onwards. We now consider these four notions in turn and the lines of research they gave rise to.

2.1 System T and related type systems

Gödel's System T, introduced in [Göd58], provides a higher type analogue of the notion of primitive recursive computation; it is one of a number of syntactic formalisms that define “restricted” classes of computable functionals, in the sense that the computational complexity of definable functionals is somehow limited. These formalisms fall somewhat outside our primary area of interest, since they make no claim to defining a “complete” class of computable functionals in any interesting sense, but they are close enough to our concerns to merit some attention.

2.1.1 Gödel's T

System T is essentially a simply-typed λ -calculus over the ground type $\bar{0}$, with constants for zero, successor and primitive recursors of higher type:

$$0 : \bar{0}, \quad S : \bar{0} \rightarrow \bar{0}, \quad R_\sigma : \sigma \rightarrow (\sigma \rightarrow \bar{0} \rightarrow \sigma) \rightarrow \bar{0} \rightarrow \sigma$$

and equipped with the following reduction rules in addition to the usual β -reduction:

$$R_\sigma MN 0 \rightarrow M, \quad R_\sigma MN(Sx) \rightarrow N(R_\sigma MNx)x$$

(where x is a variable of type $\bar{0}$). Thus, System T naturally extends Péter's notion of second-order primitive recursion to higher types.

One can consider this system either as a standalone syntactic formalism, or as a language for denoting functionals in some other type structure. For instance, let \mathbf{S} be the full set-theoretic type structure over \mathbb{N} ; then any closed term $M : \sigma$ of System T denotes an element $\llbracket M \rrbracket \in \mathbf{S}_\sigma$ in an obvious way.

Gödel's purpose in introducing this system was to give an interpretation of first order Heyting arithmetic (the so-called *Dialectica interpretation*) in which quantifier complexity was replaced by the type complexity of objects of finite type. Specifically, Gödel gave a translation from formulae $\phi(\mathbf{x})$ of HA to formulae $\phi'(\mathbf{x}) \equiv \exists \mathbf{y}.\forall \mathbf{z}.\phi^*(\mathbf{x}, \mathbf{y}, \mathbf{z})$ in the language of System T, in which ϕ^* is quantifier-free and the variables \mathbf{y}, \mathbf{z} may be of arbitrary finite type. For the sake of completeness we give the definition here, though the details are not too important for our purposes:

Definition 2.1 (Dialectica translation) *For atomic formulae α , define $\alpha' \equiv \alpha$. Given $\phi'(\mathbf{x}) \equiv \exists \mathbf{y}.\forall \mathbf{z}.\phi^*(\mathbf{x}, \mathbf{y}, \mathbf{z})$ and $\psi'(\mathbf{u}) \equiv \exists \mathbf{v}.\forall \mathbf{w}.\psi^*(\mathbf{u}, \mathbf{v}, \mathbf{w})$, define*

$$\begin{aligned} (\phi \wedge \psi)' &\equiv \exists \mathbf{y}\mathbf{v}.\forall \mathbf{z}\mathbf{w}.\phi^*(\mathbf{x}, \mathbf{y}, \mathbf{z}) \wedge \psi^*(\mathbf{u}, \mathbf{v}, \mathbf{w}) \\ (\phi \vee \psi)' &\equiv \exists \mathbf{y}\mathbf{v}t.\forall \mathbf{z}\mathbf{w}.(t = 0 \wedge \phi^*(\mathbf{x}, \mathbf{y}, \mathbf{z})) \vee (t = 1 \wedge \psi^*(\mathbf{u}, \mathbf{v}, \mathbf{w})) \\ (\phi \Rightarrow \psi)' &\equiv \exists \mathbf{V}\mathbf{Z}.\forall \mathbf{y}\mathbf{w}.\phi^*(\mathbf{x}, \mathbf{y}, \mathbf{Z}(\mathbf{y}, \mathbf{w})) \Rightarrow \psi^*(\mathbf{u}, \mathbf{V}(\mathbf{y}), \mathbf{w})^4 \\ (\forall s.\phi)' &\equiv \exists \mathbf{Y}.\forall s\mathbf{z}.\phi^*(\mathbf{x}, \mathbf{Y}(s), \mathbf{z}) \\ (\exists s.\phi)' &\equiv \exists s\mathbf{y}.\forall \mathbf{z}.\phi^*(\mathbf{x}, \mathbf{y}, \mathbf{z}) \end{aligned}$$

Moreover, it can be shown that if ϕ is provable in HA then there are terms \mathbf{Y} of System T such that $\phi^*(\mathbf{x}, \mathbf{Y}(\mathbf{x}), \mathbf{z})$ is provable in System T with quantifier-free intuitionistic logic; thus, the terms \mathbf{Y} are thought of as embodying the constructive content of ϕ . Gödel regarded it as immediately apparent that there was a notion of “computable function of finite type” satisfying the axioms of System T in which the proof of $\phi^*(\mathbf{x}, \mathbf{Y}(\mathbf{x}), \mathbf{z})$ can be interpreted.⁵ This gives us a consistency proof for Heyting arithmetic, and hence (in view of the double-negation translation) for Peano arithmetic.

However, the interest of System T seems to go beyond this original application. Gödel himself gave a proof-theoretic characterization of the expressive power of System T: the type 1 functions definable in T are precisely the functions provably total in first order (Heyting or Peano) arithmetic. Grzegorzczuk [Grz64] gave some alternative presentations of System T definable functionals in the style of combinatory logic, though from a modern perspective these do not seem very different from the original presentation. Tait and several others [Tai67, Dra68, Hin67, Hin66a, San67, Sho67] proved independently that System T is *strongly normalizing* — that is, for any System T term, all reduction paths terminate yielding the same normal form. (A remarkable alternative proof was later given by Gandy [Gan80].) This result, whose proofs depend explicitly or implicitly on transfinite induction up to ϵ_0 , shows that the extensional term model for System T (consisting of closed System T terms modulo extensional equality) is itself a type structure with all the properties needed for the Dialectica interpretation. That is, a functional interpretation of arithmetic can be given in terms of System T itself, without need for an interpretation of System T in another type structure. (This idea was already implicit in a remark of Kreisel [Kre59, §3.4].)

The Dialectica interpretation also inspired work by Kreisel [Kre59] and later Spector [Spe62] on interpretations for analysis (essentially second order arithmetic) in a similar spirit, but these go beyond System T itself and make use of some of the other type structures we shall discuss in this paper. Other mathematical results pertaining specifically to System T include the theorems of Howard ([How73]; see also [Gir87, annex 7.A]) that all T-definable functionals are *hereditarily majorizable*, and (hence) that there is no T-definable *modulus of extensionality* functional. A further mathematical analysis of System T in terms of functors over the category of ordinals is promised in [Gir, Chapter 12].

Other questions concern *theories* for System T: for example, what equivalence relations on System T terms are induced by their interpretation in various type structures? The largest interesting theory (which we shall call T_0) is clearly

⁴Here, for instance, if \mathbf{Z} abbreviates $Z_1 \dots Z_r$, we write $\mathbf{Z}(\mathbf{y}, \mathbf{w})$ to abbreviate $Z_1(\mathbf{y}, \mathbf{w}), \dots, Z_r(\mathbf{y}, \mathbf{w})$. Gödel motivated the clause for implication as follows. We identify a proposition $(\exists \mathbf{y} \dots) \Rightarrow (\exists \mathbf{v} \dots)$ with the existence of computable functions \mathbf{V} that to each sequence \mathbf{y} making the antecedent true assign a sequence \mathbf{v} making the consequent true. Moreover, we identify a proposition $(\text{forall } \mathbf{z} \dots) \Rightarrow (\forall \mathbf{w} \dots)$ with the existence of computable functions \mathbf{Z} that to each sequence \mathbf{w} making the consequent false assign a sequence \mathbf{z} making the antecedent false.

⁵All that is required for Gödel’s argument is that *some* such notion of computable functional should exist. A footnote in a revised version of the paper [Göd72], however, suggests he had in mind Kreisel’s hereditarily recursive operations (see Section 2.4).

that given by the extensional closed term model of System T itself. An argument due to Kreisel (appearing as one of the final set of exercises in [Bar84]) shows that the interpretation of System T in \mathbf{S} induces a strictly smaller equivalence relation — in computer science terms, this interpretation is not *fully abstract*. In [Loa97], Loader considers a system stronger than, but conservative over, System T, and shows that the interpretation of this system in the usual category of PERs is fully abstract; it follows that the interpretation of System T itself in the type structure \mathbf{HEO} (see below) induces exactly the theory T_0 .

Further discussions of System T can be found in [Kre59], [Tro73, §3.5], [Bar84, Appendix A.2], and [Gir87, annex 7.A ff.]. The Dialectica interpretation is described in [Tro73, Gir87]. A good survey of later research related to System T can be found in Section 5 of Troelstra’s introductory note to Gödel’s original paper in [Göd90].

2.1.2 Kleene’s S1–S8

We have already observed that System T can be interpreted in other type structures, such as the full set-theoretic type structure. Indeed, one reason why System T is interesting from our point of view is that it admits an interpretation of this kind in practically all the type structures we shall have occasion to consider. It therefore provides a kind of common “skeleton” for all these type structures, and in general yields an effectively enumerable class of functions that can play a role analogous to that of the primitive recursive functions in ordinary recursion theory.

However, in this respect there is nothing particularly distinguished about System T — many other systems would serve the same purpose. For example, a weaker analogue of primitive recursion is given by the typed λ -calculus with constants 0 , S and

$$\hat{R}_\sigma : \sigma \rightarrow (\sigma \rightarrow \bar{0} \rightarrow \sigma) \rightarrow \bar{0} \rightarrow \sigma$$

together with reduction rules

$$\hat{R}_\sigma MN0\mathbf{P} \rightarrow M\mathbf{P}, \quad \hat{R}_\sigma MN(Sx)\mathbf{P} \rightarrow N(\hat{R}_\sigma MNx)x\mathbf{P}$$

where $\sigma = \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \bar{0}$ and \mathbf{P} textually abbreviates $P_1 \dots P_n$. This system corresponds exactly in expressive power to Kleene’s schemata S1–S8 (see Section 2.2 below); it is more restrictive than System T just as Kleene’s notion of type 2 primitive recursiveness is more restrictive than Péter’s. Kleene referred to the elements of \mathbf{S} definable using S1–S8 as the *primitive recursive functionals*; the type 1 functions of this kind are exactly the primitive recursive functions in the usual sense (see [Kle59b, §1]). (Our notation is borrowed from [Fef77b], where the constants R_σ are called *N-recursion operators* and the \hat{R}_σ are called *elementary recursion operators*. Kleene’s primitive recursive functionals are also mentioned in [Tro73, §2.8.2].)

In some sense S1–S8 provides a better common core of “simple” functionals than System T, inasmuch as it can be interpreted in a wider class of type structures. It would seem, however, that Kleene’s S1–S8 is of less mathematical interest than Gödel’s T.

System T also provided a point of departure for two other strands of later work, namely modern type theory (which typically considers stronger systems than System T), and higher type complexity theory (which typically considers weaker systems). Both of these are significant areas of research in theoretical computer science. For detailed information on modern type theory, a good reference is [Bar92]; for higher type complexity, we refer the reader to the recent survey articles of Irwin, Kapron and Royer [IKR01a, IKR01b].

2.2 Kleene computability: S1–S9

2.2.1 Kleene’s work

The first serious attempt at a full-blown generalization of the notion of recursive function to all type levels was Kleene’s definition of higher type computability via the schemata S1–S9 [Kle59b, Kle63]. In this approach to higher type computability, we suppose we are given a type structure A of total functionals over \mathbb{N} , and we define a class of computable *partial* functions over A by means of the certain computation schemata.

The spirit behind the above definition is easily grasped: we are allowed to perform effective computations involving elements of A , in which the elements of function type are treated simply as simply as *oracles* or *black boxes*. In particular, we may feed an element of function type with numbers or functions that are themselves computable (in the same sense), and observe the numerical results, but we are not granted access to information about such elements in any other way. Thus, Kleene’s definition embodies the ideal of computing with functions as pure extensions, without reference to how they are represented or implemented. Indeed, A will in general include non-computable objects, so that we have to imagine the oracles as working “by magic”.

Kleene in fact concentrated his attention to computations over objects of *pure* type, though this was not an especially significant decision. Because of its historical importance, we reproduce Kleene’s original definition here in all its glory, with slightly modified notation.⁶ Given a type structure A , we write $X(A)$ for the disjoint union of all sets $A_{\sigma_1} \times \cdots \times A_{\sigma_r}$ ($r \geq 1$) where the σ_i are pure types. We write $[-]$ for the coding of pure types as natural numbers given by $[\bar{t}] = t$, and take $\langle \cdot \cdot \cdot \rangle$ to be an effective coding of finite sequences of natural numbers as natural numbers. We abbreviate x_1, \dots, x_r by \mathbf{x} and $x_1 : \sigma_1, \dots, x_r : \sigma_r$ by $\mathbf{x} : \boldsymbol{\sigma}$. We also write $[\boldsymbol{\sigma}]$ for $\langle [\sigma_1], \dots, [\sigma_r] \rangle$.

The definition proceeds by introducing a system of *indexing* whereby natural numbers z encode definitions of computable partial functions $\{z\}$; this is reminiscent of the indexing of partial recursive functions in ordinary recursion theory.

Definition 2.2 (Kleene computability) *Let A be an extensional type structure over \mathbb{N} . We inductively define a partial operation $\{-\}(-) : \mathbb{N} \times X(A) \rightarrow \mathbb{N}$,*

⁶Kleene’s definition was restricted to the case $A = \mathbf{S}$. Moreover, Kleene adopted the peculiar convention that only the order of arguments within each type was material, so his version of S6 was slightly different from the one given here.

called index application, by means of the following clauses (which we interpret as applying whenever they are well-typed).

S1 Successor function: $\{\langle 1, [\sigma] \rangle\}(\mathbf{x} : \sigma) = x_1 + 1$.

S2 Constant functions: For any $q \in \mathbb{N}$, $\{\langle 2, [\sigma], q \rangle\}(\mathbf{x} : \sigma) = q$.

S3 Projections: $\{\langle 3, [\sigma] \rangle\}(\mathbf{x} : \sigma) = x_1$.

S4 Composition: For any $g, h \in \mathbb{N}$, $\{\langle 4, [\sigma], g, h \rangle\}(\mathbf{x} : \sigma) \simeq \{g\}(\{h\}(\mathbf{x}), \mathbf{x})$.

S5 Primitive recursion: For any $g, h \in \mathbb{N}$, if $z = \langle 5, [\sigma], g, h \rangle$ then

$$\{z\}(0, \mathbf{x}) \simeq \{g\}(\mathbf{x}), \quad \{z\}(n+1, \mathbf{x}) \simeq \{h\}(n, \{z\}(n, \mathbf{x}), \mathbf{x}).$$

S6 Permutation of arguments: For any $g \in \mathbb{N}$ and $1 \leq k < r$,

$$\{\langle 6, [\sigma], k, g \rangle\}(\mathbf{x} : \sigma) \simeq \{g\}(x_{k+1}, x_1, \dots, x_k, x_{k+2}, \dots, x_r).$$

S7 Type 1 application: If $\sigma_1 = \bar{1}$ and $\sigma_2 = \bar{0}$, then $\{\langle 7, [\sigma] \rangle\}(\mathbf{x} : \sigma) = x_1(x_2)$.

S8 Higher type application: For any $h \in \mathbb{N}$ and $t \geq 2$, then

$$\{\langle 8, [\sigma], t, h \rangle\}(y : \bar{t}, \mathbf{x} : \sigma) \simeq y(\lambda z : \overline{t-2}. \{h\}(y, z, \mathbf{x}))$$

whenever $\{h\}(y, z, \mathbf{x})$ is defined for all $z \in A_{t-2}$.

S9 Index invocation: $\{\langle 9, [\sigma], [\tau] \rangle\}(x : \bar{0}, \mathbf{y} : \sigma, \mathbf{z} : \tau) \simeq \{x\}(\mathbf{y})$.

We say $F : A_{\sigma_1} \times \dots \times A_{\sigma_r} \rightarrow \mathbb{N}$ is (Kleene) computable over A if there is a number e (called an index for F) such that $F(\mathbf{x}) \simeq \{e\}(\mathbf{x})$ for all \mathbf{x} .

It is curious that Kleene, one of the pioneers of the λ -calculus, did not make greater use of the typed λ -calculus in formulating this notion of computability. A definition of Kleene computability in a more modern spirit will be given in Part II.

As already noted, S1–S8 by themselves define a perfectly good class of *total* functionals — the schema S9 is the only one that introduces partiality into the definition. Notice that even if the indexing system were redesigned so that every natural number indexed one function at each type, partial functions would still arise for reasons of circularity: if e were an index for the function $F = \lambda xy. \{x\}(y)$, then $F(e, e)$ would be undefined according to the inductive definition above.

Kleene generally conceived computations in this setting as infinitely branching trees of transfinite depth — when invoking S8 one computes the entire graph of $\lambda z. H(y, z, \mathbf{x})$ by means of auxiliary computations before presenting it to the oracle y . This is perhaps not the only possible conception: for example, one might imagine that the magic of the oracles included the ability to recognize functions from finitary descriptions of them, in which case these auxiliary computations would not be needed. But whatever conception one adopts, Kleene's schemata exhibit a curious tension between effective, finitary computational

processes and calls to numinous or infinitistic oracles, so that the computational significance of Kleene’s definition is sometimes difficult to assess.

Kleene’s motivation seems to have sprung from two distinct strands of his earlier work. On the one hand, he was certainly seeking an appropriate higher type generalization of the basic notions of ordinary recursion theory — including, if possible, some analogue of Church’s thesis at higher types. On the other hand, he was seeking to explain his theory of logical complexity for classical predicate logic (embodied in the arithmetic and analytic hierarchies [Kle55a, Kle55b]) in terms of computability relative to certain higher type objects (hence the emphasis on *quantifiers* in his papers on higher type computability). This latter motivation goes some way towards explaining two features of Kleene’s definition that sometimes appear puzzling to modern readers: the restriction to computations on *total* functionals, and the specialization to the type structure \mathbf{S} rather than any more constructively given class of functionals.

Kleene developed some basic results for this notion of computability analogous to those of ordinary recursion theory: for example, higher type versions of the normal form theorem [Kle59b, §5], and a restricted version of the first recursion theorem [Kle63, §10]. Other results made precise the connection with logical complexity: for instance:

Theorem 2.3 (Kleene [Kle59b, §10]) *A total function $F : \mathbb{N}^r \rightarrow \mathbb{N}$ is hyperarithmetical (that is, its graph is Δ_1^1) iff it is Kleene computable relative to the type 2 object ${}^2\exists$, where*

$${}^2\exists(f) = \begin{cases} 0 & \text{if } \exists n. f(n) = 0, \\ 1 & \text{otherwise.} \end{cases}$$

Kleene also followed up his definition via S1–S9 with a clutch of papers [Kle62b, Kle62c, Kle62a, Kle61] showing that several alternative definitions of higher type computability — via Turing machines, lambda calculus,⁷ and Herbrand–Gödel-style recursive definitions — give rise to the same class of computable functionals. These results of course closely parallel the corresponding results of ordinary recursion theory, but they are not especially deep extensions of the familiar results, since all these definitions are based on essentially the same idea of computation with oracles. Nevertheless, these results go some way towards establishing the robustness of Kleene’s class of computable functionals.

Despite these reassuring results, however, Kleene’s notion of higher type computability presents us with some strange anomalies. We mention two of these here, both arising somehow from the curious infinitary side-condition in S8.

The first of these is the notorious fact that the computable partial functionals are not closed under some basic substitution operations. For example,

⁷In fact, in [Kle62a] Kleene used a system based on what is now known as Church’s λI calculus

consider the functions h, Φ, G defined by

$$\begin{aligned} h(e : \bar{0}, x : \bar{0}) &\simeq \begin{cases} 0 & \text{if } \neg T(e, e, x), \\ \text{undefined} & \text{if } T(e, e, x) \end{cases} \\ \Phi(F : \bar{2}, e : \bar{0}) &\simeq F(\lambda x. h(e, x)) \\ G(f : \bar{1}) &= 0. \end{aligned}$$

Then the partial functions Φ and G are both Kleene computable, but $\lambda e. \Phi(G, e)$ is not (if it were, we could solve the halting problem). This is obviously rather worrying — indeed, it even seems questionable in what sense we have defined a computable functional Φ if we are not allowed to plug in the total computable functional G as its first argument. Essentially the same problem lies behind the failure of the natural generalization of the first recursion theorem. There is also other evidence that the notion of partial computability is pathological: for instance, a set that is semi-recursive and co-semi-recursive need not be recursive (see [Pla66, p. 131]), and the union of two semi-recursive sets need not be semi-recursive (see [Gri69b, p. 233]).

One way of responding to these problem is to accept that the pure notion of *partial* Kleene computable functional is not a very good one, and to restrict one’s attention to the *total* computable functionals, which behave well with respect to substitution and indeed give rise to a cartesian closed category. In most interesting cases, of course, the total Kleene computable functions over A will themselves all be elements of A (in this case we say A is *closed under Kleene computation*), so we may regard Kleene’s definition as picking out an important substructure of A . We find this a compelling point of view and will pursue it further in Part II.

A second curious feature of Kleene’s definition, less often noted, is its *non-absoluteness*. The set of indices that define total functionals, for instance, may vary from one type structure A to another — intuitively, the fewer elements of type σ there are in A , the easier it is for an index to define a total functional of type $\sigma \rightarrow \bar{0}$. Even for the case of computability over \mathbb{S} , it is unclear *a priori* whether the totality or otherwise of indices is an absolute property, unless one is willing to accept the notion of “the” full set-theoretic type structure as absolute. In other words, might there be Kleene-style computations whose termination is dependent on controversial principles of set theory? This point was essentially noted by Kreisel in [Kre61]; again, we will return to it in Part II.

Other presentations of the Kleene computable functionals were later given by Gandy [Gan67a] and Platek [Pla66]. Both of these treatments sought to avoid some of the arbitrary features of Kleene’s definitions and to emphasize the naturalness of this notion of computability. Gandy gave a more perspicuous definition involving register machines, and argued that one was led ineluctably to Kleene’s notion of higher type computability if (a) one restricted attention to (hereditarily) total arguments, and (b) one treated functions as “pure extensions”. Platek gave a characterization of Kleene’s functionals within a general framework for recursion theory that stressed *definability* of functions by recursion rather than computability. In particular, he showed how a very natural theory of recursive definability could be developed for type structures hereditarily partial functionals — this gave rise to Kleene’s functionals via some simple

relationships between the partial and total type structures (see Section 3.1 below). Platek’s approach via inductive definability was later streamlined by Moschovakis [Mos76], who clarified the relationship with Kleene’s original definition (this treatment is adopted in [KM77]; see also [Fef77a]).

Beyond the results already mentioned, however, the pure notion of Kleene computability over \mathbf{S} appears to hold rather little mathematical interest. For this reason, the later study of recursion theory on \mathbf{S} concentrated almost entirely on questions of relative computability and/or certain kinds of non-computable object. (We say an object $x \in A$ is *Kleene computable relative to* $y \in A$, and write $x \preceq y$, if there is a Kleene computable partial function f such that $f(y) = x$. If $x \preceq y \preceq x$ we say x, y are of the same *Kleene degree*.) Thus, the subject largely lost contact with anything that might be considered a genuinely effective notion of computability. Although they are somewhat peripheral to the story told in this paper, a brief account of these developments is in order.

2.2.2 Recursion in normal objects

Most of the later work has concentrated on the theory of normal functionals and normal classes. Let us write ${}^k\exists (k \geq 2)$ for the object of \mathbf{S}_k embodying existential quantification over \mathbf{S}_{k-2} . A functional M of type level k is said to be *normal* if ${}^k\exists \preceq M$. We will be interested in notions of M -computability (that is, Kleene computability relative to M) where M is normal, and in particular in the notions of ${}^k\exists$ -computability.

The study of normal objects and the associated notions of computability turns out to yield a very rich and beautiful theory. Roughly speaking, if ${}^k\exists$ is deemed computable, then the theory of computability is good up to and including type level k . One way to see intuitively why this should be so is to observe that the infinitary side-condition in S8 becomes decidable in the presence of the ${}^k\exists$, so that the anomalies mentioned earlier for pure Kleene computability are washed out in this setting. The study of computability in the presence of all the ${}^k\exists$ is sometimes E-recursion theory [Sac99]. (Note, however, that even the set of ${}^k\exists$ -computable type 1 functions increases strictly with k .)

The significance of normal objects had already emerged from Kleene’s early work, but their good properties were first seriously exploited by Gandy [Gan67b], who introduced the key concepts of stage comparison and number selection. Informally, a *stage comparison function* for two functionals F and G is a partial function $\chi_{F,G}(\mathbf{x}, \mathbf{y})$ that tells us which of the two Kleene computation trees for $F(\mathbf{x}), G(\mathbf{y})$ has the smaller ordinal depth, assuming that at least one of these trees is well-founded. In particular, it can tell us which of the two computations will terminate, given that at least one of them will. The following theorem was stated for type 2 in [Gan67b], proved for type 3 in [Mos67], and extended to higher types in [Mos76]:

Theorem 2.4 (Stage comparison) *Let M be a normal functional of level k . If F, G are M -computable and of level $\leq k$, then $\chi_{F,G}$ is also M -computable.*

One can often use stage comparison where in ordinary recursion theory one might use interleaving arguments — for instance, in showing that (if M is

normal of level k) the union of two M -semidecidable subsets of \mathbf{S}_k is M -semidecidable, and that a subset of \mathbf{S}_k is M -decidable iff both it and its complement are M -semidecidable.

Given a predicate $P(\mathbf{x}, n : \bar{0})$, a *selection function* for P is a partial functional $F(\mathbf{x})$ that “selects” a value of n satisfying $P(\mathbf{x}, n)$ whenever there is one. Again, the basic result is proved for type 2 in [Gan67b] and generalized in [Mos67, Mos76]:

Theorem 2.5 (Number selection) *Let M be normal of level $k \geq 2$. For any M -semidecidable predicate $P(\mathbf{x}, n : \bar{0})$ on \mathbf{S} , there is an M -computable function $F(\mathbf{x})$ such that*

$$\forall \mathbf{x}. (\exists n. P(\mathbf{x}, n)) \implies F(\mathbf{x}) \downarrow \wedge P(\mathbf{x}, F(\mathbf{x})).$$

It follows, for example, that the class of M -semidecidable subsets of \mathbf{S}_k is closed under (uniform) \mathbf{N} -indexed unions, and that a partial function of level k is M -computable iff its graph is M -semidecidable. The basics of stage comparison and number selection up to this point are covered in the survey article of Kechris and Moschovakis [KM77].

A selection theorem of another kind is the following. It first appeared in [Gri69b] but with an incorrect proof. A correct proof was given in [HM76].

Theorem 2.6 (Grilliot selection) *Let M be normal of level $k + 2$ ($k \geq 1$). For any inhabited M -semidecidable subset P of \mathbf{S}_{k-1} , there is an inhabited M -decidable subset Q such that $Q \subseteq P$.*

Much attention has also been devoted to the study of *sections* as defined by Kleene in [Kle63]. Given an object $x \in \mathbf{S}$, the *section* of x is the set $\{y \in \mathbf{S} \mid y \prec x\}$; likewise the k -*section* of x is the set $\{y \in \mathbf{S}_k \mid y \prec x\}$.⁸ One of the high points of the theory is the following result due to Sacks [Sac74, Sac77]:

Theorem 2.7 (Plus-one theorem) *Suppose $0 < k < n$. Then for any normal object $F \in \mathbf{S}_n$ there is a normal object $G \in \mathbf{S}_{k+1}$ such that F and G have the same k -section.*

Later Sacks initiated a more fine-grained study of sections in the spirit of classical degree theory and considered the higher type analogues of problems from degree theory such as Post’s problem [Mac72, Sac80, Sac85]. and the study of the \mathbf{E} -recursively enumerable degrees [Sac86, Gri80]. Slaman obtained \mathbf{E} -recursive versions of the splitting and density theorems [Sla81, Sla85].

Sections are to total functionals what *envelopes* are to partial functionals. If $\sigma = \sigma_1 \rightarrow \dots \rightarrow \sigma_r \rightarrow \bar{0}$, let us write \mathcal{P}_σ for the set of all partial functions $\mathbf{S}_{\sigma_1} \times \dots \times \mathbf{S}_{\sigma_n} \rightarrow \mathbf{N}$. The k -*envelope* of $x \in \mathbf{S}$ is the set $\{f \in \mathcal{P}_\sigma \mid \text{level}(\sigma) \leq k, f \text{ semicomputable relative to } x\}$. Moschovakis [Mos74c] showed that the analogue of the Plus-one theorem for envelopes fails: indeed, for *any* normal type 3 object, its 1-envelope is not the 1-envelope of any normal type 2 object. However, we have the following result due to Harrington [Har73]:

⁸The k -section is sometimes taken to include all objects of types $\leq k$ computable in x , but it makes little difference.

Theorem 2.8 (Plus-two theorem) *Suppose $0 < k < n - 1$. Then for any normal object $F \in \mathcal{S}_n$ there is a normal object $G \in \mathcal{S}_{k+2}$ such that F and G have the same k -envelope.*

Regarding elements of the k -envelope of a normal M as representing M -semidecidable predicates, it is natural to ask what closure properties these enjoy. It is clear from earlier remarks that the 1-envelope of a normal type 2 object is closed under existential quantification over type 0 (in an obvious sense). The situation at higher types is more interesting:

Theorem 2.9 *Let M be normal of level $m \geq 3$. Then the $(m - 1)$ -envelope of M is closed under existential quantification over \mathcal{S}_j for $j < m - 2$ [HM76], but not under existential quantification over \mathcal{S}_{m-2} [Mos67, Gri67].*

Both Sacks and Harrington worked with definitions of recursion in normal functionals that differed somewhat from Kleene's, but the equivalences are verified in [Low76]. The proofs of many of the above results have been aided and streamlined by adopting the perspective of abstract recursion theory, which isolates the essential features of the domains (in the this case, the \mathcal{S}_σ) over which we are computing. This point of view is worked out in the books [Mol77, Fen80, Sac90].

There has been little activity specifically in these areas over the last decade, though the ideas of recursion in normal objects have left their mark on descriptive and admissible set theory. For readers interested in these applications, some references to follow up are [Mos74a, Mos74b, Nor78b], as well as the survey article [Sac99] and other papers cited therein.

2.2.3 Hierarchies

Another area of investigation has been the search for *hierarchies* for the functionals computable from a given object. The starting point for this endeavour is the fact that the 1-section of ${}^2\exists$ consists of exactly the hyperarithmetic or Δ_1^1 functions, which (as shown by Kleene [Kle55b]) can be classified according to their logical complexity by means of an ordinal hierarchy of height ω_1^{CK} , known as the hyperarithmetic hierarchy. Kleene in [Kle63] introduced the general idea of seeking similar ordinal stratifications for various sections of other higher type objects.

Tugué [Tug60] considered in particular the normal type 2 object E_1 embodying the *Suslin quantifier*:

$$E_1(f : \bar{1}) = \begin{cases} 0 & \text{if } \exists g : \bar{1}. \forall n. f(\bar{g}(n)), \\ 1 & \text{otherwise.} \end{cases}$$

Tugué and later Richter [Ric67] gave ordinal hierarchies for the 1-section of E_1 . Kleene had conjectured that this class might coincide with the Δ_2^1 functions, but this was refuted by Shoenfield in [Sho62].

A pleasing generalization of Kleene's result to the 1-section of an arbitrary normal type 2 object was given independently by Shoenfield [Sho68] and Hinman [Hin66b, Hin69]. The two versions are essentially the same; the key point

in both cases is that the set of ordinal notations, rather than being fixed in advance, is generated simultaneously with the hierarchy itself. Later, Wainer [Wai74] showed how a somewhat more delicate construction yields a hierarchy for the 1-section even of an arbitrary non-normal type 2 object.

A natural challenge was to obtain similar hierarchy results for 2-sections of ${}^3\exists$ and other type 3 objects. In [Kle63] Kleene outlined the construction of a *hyperanalytic hierarchy* for type 2 objects; this hierarchy was studied in detail by his student Clarke [Cla64], who conjectured that it did not exhaust the type 2 objects computable in ${}^3\exists$. This conjecture was confirmed by Moschovakis [Mos67], who however constructed an alternative hierarchy that does exhaust them, using a much more powerful system for generating ordinal notations.

Other objects of interest are the *superjump* operators nS ($n \geq 2$), defined by

$${}^nS(e : \bar{0}, x : \overline{n-1}) = \begin{cases} 0 & \text{if } \{e\}(x) \downarrow, \\ 1 & \text{if } \{e\}(x) \uparrow. \end{cases}$$

The operator 2S is the ordinary jump operator of classical recursion theory, which is equivalent in strength to ${}^2\exists$. The superjump operator 3S , however, is strictly weaker than ${}^3\exists$ (and hence non-normal). Recursion in the superjump has been studied in [Gan67b, Pla71, AH74]. Superjump operators of even higher types have been considered by Harrington [Har73, Har74].

Several of the above hierarchy results are covered in detail in Hinman's book [Hin78].

2.3 The total continuous functionals

2.3.1 Early work: Kleene and Kreisel

Whereas computations over the full set-theoretic type structure \mathbf{S} seem to have connections with the metamathematics of classical logic, the study of the *total continuous functionals* has been motivated by interest in more constructive interpretations. The type structure \mathbf{C} of total continuous functionals was obtained around the same time by Kleene [Kle59a] (who called them the *countable functionals*) and Kreisel [Kre59] (who called them the *continuous functionals*). Both definitions were rather complicated and neither were obviously natural, but they both embodied the basic idea that any finite piece of information about the output from a functional F was determined by a finite amount of information about the input.

Kleene's definition centres around the observation that any continuous type 2 function $F : \mathbb{N}^{\mathbb{N}} \rightarrow \mathbb{N}$ is completely determined by the function $\alpha_F : \mathbb{N} \rightarrow \mathbb{N}$ given by

$$\alpha_F \langle n_0, \dots, n_{r-1} \rangle = \begin{cases} m + 1 & \text{if } F(\beta) = m \text{ whenever } \beta(i) = n_i \text{ for all } i < r, \\ 1 & \text{if no } m \text{ exists with this property.} \end{cases}$$

Reversing this idea, we may define a partial "application" operation $- | - : \mathbb{N}^{\mathbb{N}} \times \mathbb{N}^{\mathbb{N}} \rightarrow \mathbb{N}$ by

$$\alpha | \beta \simeq \alpha(\bar{\beta}(r)) - 1, \\ \text{where } r \text{ is minimal such that } \alpha(\bar{\beta}(r)) > 0.$$

Thus, type 1 objects can be used to encode type 2 objects. Once this is in place, it easy to use type 1 objects can be used to encode objects of arbitrary pure types:

Definition 2.10 (Kleene’s countable functionals) *For pure types \bar{n} , define sets C_n and notions of associate for elements of C_n as follows:*

- Take $C_0 = \mathbb{N}$, $C_1 = \mathbb{N}^{\mathbb{N}}$, and declare each $\alpha \in C_1$ to be an associate for itself.
- For $n \geq 1$, say $\alpha \in C_1$ is an associate for the function $F : C_n \rightarrow \mathbb{N}$ iff whenever β is an associate for $G \in C_n$ we have $\alpha \mid \beta = F(G)$; and take C_{n+1} to be the set of functions $C_n \rightarrow \mathbb{N}$ that have an associate.

Kleene extended this definition from pure types to arbitrary types by means of the usual type-changing techniques.

In Kreisel’s definition, continuity was built in by defining functionals first on *formal neighbourhoods* of elements, and only then on the elements themselves. The details are a little complicated and we will omit them here (a slightly simplified version of Kreisel’s definition will appear in Part II). The equivalence between these definitions, though not mathematically trivial, was recognized at the time of their discovery and is mentioned in both the original papers (see also [HT69]). Tait [Tai62] later gave a somewhat cleaner, more axiomatic treatment of the continuous functionals in the spirit of Kreisel’s definition.

Both Kleene and Kreisel identified a natural effective substructure RC of C, consisting of the *recursively countable* or *recursively continuous* functionals. In Kleene’s terms, these are the functionals with at least one recursive associate; in Kreisel’s terms, they are defined via recursive functions on bases of neighbourhoods. An important result, the *density theorem* (see [Kre59, Appendix]), ensures that each $F \in C_{\sigma \rightarrow \tau}$ is completely determined by its effect on the recursive (or even the “finite”) elements of C_σ ; it follows that RC is an extensional type structure in its own right.

Kreisel, in particular, was interested in using C to give constructive interpretations for systems such as intuitionistic second order arithmetic. The idea is as follows: given a formula ϕ , first apply the Gödel Dialectica translation to obtain a formula $\phi' \equiv \exists \mathbf{y}.\forall \mathbf{z}.\phi^*$ in which ϕ^* is quantifier-free and the \mathbf{y}, \mathbf{z} may be of any finite type. We may now consider the interpretations of ϕ' arising by allowing the \mathbf{y} and \mathbf{z} to range over various kinds of object of finite type. Kreisel suggested that we get an interpretation that fits well with the informal notion of constructive truth if we let \mathbf{z} range over functionals in C and \mathbf{y} over functionals in RC. The intuition is that the constructive content of ϕ should be embodied by an effectively given operation (the \mathbf{y}) which nevertheless works for arbitrary continuous data such as free choice sequences (the \mathbf{z}).

Kreisel also considered rival interpretations of this kind which are of interest for independence proofs, e.g., let \mathbf{z} ranging over arbitrary functionals in S and \mathbf{y} over Kleene-computable ones; or let \mathbf{y} and \mathbf{z} both range over the hereditarily effective operations. He also showed that the above interpretation could be combined with the double-negation translation to give a very satisfactory “no-counterexample interpretation” for *classical* second-order arithmetic. The

present author deems Kreisel’s paper [Kre59] to be a classic in the field, which remains well worth reading today as a discussion of the metamathematical applications of higher type functionals.

One can also consider the notion of Kleene computable functional over \mathbf{C} as given by Definition 2.2. It was clear to Kleene and Kreisel that every total Kleene computable functional over \mathbf{C} was a recursively continuous functional — intuitively, anything that is computable in Kleene’s “extensional” sense is computable at the more intensional level of associates. Moreover, at type 2 the recursively continuous functionals clearly coincide with the Kleene computable functionals (over \mathbf{C} or \mathbf{S}), since e.g. any $F \in \mathbf{C}_2$ is Kleene computable relative to an associate for F .

It was raised as an open problem in [Kre59, §4.4] whether all elements of \mathbf{RC} at types 3 and above were Kleene computable. This was answered negatively by Tait [Tai62], who gave as a counterexample the type 3 *modulus of uniform continuity* functional Φ (also known as the *fan functional* because it embodies the constructive content of Brouwer’s Fan Theorem). The existence of Φ depends on the fact that (classically) any continuous function $F : 2^{\mathbb{N}} \rightarrow 2$ is uniformly continuous. First define $\text{binary}(f : \bar{1}) \equiv \forall x. f(x) = 0 \vee f(x) = 1$; then take

$$\begin{aligned} \Phi(F : \bar{2}) &= \mu n. \forall f, g. \text{binary}(f) \wedge \text{binary}(g) \wedge (\forall m < n. f(m) = g(m)) \\ &\implies F(f) = F(g). \end{aligned}$$

Tait showed that Φ is recursively continuous but not Kleene computable over \mathbf{C} — nor, indeed, is it the restriction of \mathbf{C}_2 of a function Kleene computable over \mathbf{S} . (Published proofs of Tait’s result may be found in [GH77, §4] or [Nor99, §5].) We thus have two distinct notions of computability for the total continuous functionals.

2.3.2 Further characterizations

Apart from Tait’s work, interest in the total continuous functionals appears to have waned during the 1960s. The 70s, however, saw a wealth of new developments which were mostly of two kinds: results providing further characterizations of \mathbf{C} and \mathbf{RC} , illuminating their basic character and confirming their natural status; and results relating to various notions of relative definability and degree structures on \mathbf{C} .

Many of the later characterizations of \mathbf{C} were much simpler and more appealing than the original definitions. Ershov [Ers74a, Ers77a] showed that the continuous functionals could be obtained from the type structure \mathbf{P} of partial continuous functionals via an extensional collapse construction (see Theorem 3.10 below). A very similar characterization of \mathbf{C} was obtained independently by Scott, using a type structure arising from the category of algebraic lattices (see [Hyl75]).

Another pleasing characterization of \mathbf{C} is that based on the idea of *sequential continuity*. This may be presented very simply as follows. We use the notation $[x_i]$ for the infinite sequence $[x_0, x_1, \dots]$.

Definition 2.11 (Sequential continuity) For each type σ , we define a set C_σ together with a relation $[x_i] \downarrow x$ (read as “[x_i] converges to x ”) between infinite sequences and elements of C_σ :

- $C_0 = \mathbb{N}$, and $[x_i] \downarrow x$ if $x_i = x$ for all sufficiently large i ;
- $C_{\sigma \rightarrow \tau}$ is the set of all $f : C_\sigma \rightarrow C_\tau$ such that $[f(x_i)] \downarrow f(x)$ whenever $[x_i] \downarrow x$. Moreover $[f_i] \downarrow f$ in $C_{\sigma \rightarrow \tau}$ iff $[f_i(x_i)] \downarrow f(x)$ whenever $[x_i] \downarrow x$.

This is tantamount to the definition of type structure over \mathbb{N} in the cartesian closed category of L -spaces (see [Kur52]). Scarpellini [Sca71] considered the type structure defined in this way as a model for bar recursion at higher types, apparently without realizing that it coincided with the Kleene-Kreisel continuous functionals. The equivalence was shown by Hyland [Hyl75, Hyl79], who collected together the known characterizations and clarified the relationships between them. He also discovered some new characterizations of C , e.g., as the type structure over \mathbb{N} in the cartesian closed category of *filter spaces*, or in the CCC of compactly generated Hausdorff spaces. (A published proof of the latter fact appeared first in [Nor80].) Normann later gave yet another characterization via a hyperfinite type structure in the sense of non-standard analysis [Nor83].

Bergstra [Ber76, Ber78a] gave an interesting characterization of C as a maximal type structure (subject to some basic closure requirements) in which all type 2 functions were continuous. Intuitively, one can construct C by taking, at each type level ≥ 3 , all functions that can be added without inducing any discontinuous type 2 functions. Grilliot [Gri71] had already shown that a type 2 functional F is continuous iff ${}^2\exists$ is *not* Kleene computable relative to F and a type 1 function; we therefore have a characterization of C as a maximal type structure closed under Kleene computation and not containing ${}^2\exists$.

By this stage it was very clear that C was the canonical choice of a full continuous type structure of total functionals over \mathbb{N} . Moreover, Hyland’s work had also shown that all the constructions of C that admitted a natural effectivization gave rise to the same effective substructure, namely RC .

2.3.3 Degrees and relative computability

Most of the work on degrees and relative computability for C has focussed on the notions arising from relative Kleene computability. Hinman [Hin73] gave an example of an *irreducible* element of C_2 — that is, one whose Kleene degree is not the Kleene degree of any type 1 function. Hyland in [GH77, §5] gave a simpler example of this phenomenon, making use of the Kleene tree. This paper also contained an example, due to Gandy, of a type 3 object $\Gamma \in RC_3$ which is not Kleene computable relative to the fan functional.

In view of this last result, it was natural to ask whether any good “basis” of functionals could be given relative to which all functionals in RC were Kleene computable. (A closely related question, discussed by Feferman [Fef77a], was whether one could give a definition of RC as a substructure of C via monotone inductive schemata in the spirit of generalized recursion theory.) It is fairly easy to specify an infinite basis for RC consisting of *partial* recursively continuous functions, one for each type level (see [Ber78a, §1]). Normann [Nor79b, Nor81a]

gave a similar infinite basis consisting of total recursively continuous functionals (i.e. elements of RC), but showed that no finite subset of this sufficed; hence no *finite* basis consisting of this kind for RC is possible.

Another group of results from this period concerned properties of 1-sections for continuous type 2 objects. Early results in this vein were obtained by Grilliot [Gri71], who pointed out that such a 1-section is never closed under the ordinary jump operator. Bergstra [Ber76] showed that there are objects $F \in C_2$ whose 1-section is not the 1-section of any $f \in C_1$ (this improves on the result of Hinman mentioned above). Results relating to ordinal hierarchies for 1-sections of continuous objects are obtained in [BW77, Nor78a, NW80].

A simple but beautiful result of Normann [Nor81b] (who attributes the ideas to Kreisel [Kre59]) reveals a connection between the continuous functionals and some classical logical complexity classes for subsets of N^N . Normann showed that many facts about 1-sections and 2-envelopes flow from this theorem. A proof of the theorem also appears in [Nor99].

Theorem 2.12 (Projective hierarchy) *Let $k > 0$, $A \subseteq N^N$. Then*

(i) *A is Π_k^1 iff there is a primitive recursive predicate R (e.g. definable by Kleene's S1–S8 over C) such that*

$$f \in A \Leftrightarrow \forall G \in C_k. \exists n \in N. R(f, G, n).$$

(ii) *A is Σ_k^1 iff there is a primitive recursive R such that*

$$f \in A \Leftrightarrow \forall G \in C_{k+1}. \exists n \in N. R(f, G, n).$$

and moreover there is a uniform algorithm (e.g. a Kleene computable partial functional over C) which given any $f \notin A$ returns a G such that $\forall n. \neg R(f, G, n)$.

Finally, we mention that an alternative degree structure on C can be obtained by considering a more generous notion of relative computability: take $F \preceq_c G$ iff there is a recursive type 2 functional which transforms any associate for G into an associate for F . Most of what is known about degrees, sections and envelopes with respect to \preceq_c is summarized in [Hyl78].

Normann's book [Nor80] contains most of what was known about the total continuous functionals by 1980.

2.3.4 Recent work

Work on the continuous functionals lapsed again during the 1980s, but was renewed somewhat in the 90s, owing largely to interest from the computer science community. Modern treatments have tended to favour versions of the Ershov/Scott construction of C via domains or information systems [Ber93, SHLG94, Sch96, Nor99]. A particular focus of recent work has been the search for abstract formulations of the concept of *totality* for elements of such domains. Berger [Ber93] has shown the significance of the dual notions of *density* and *codensity* (= totality) for subsets of domains, and given generalized versions of the density theorem and Kreisel-Lacombe-Shoenfield theorem in a

domain-theoretic framework. A related approach to totality has been pursued by Normann [Nor89, Nor97]. More recent work has been concerned with extending the construction of \mathbf{C} and the associated results on domains, density and totality to transfinite types Martin-Löf-style dependent types with universes [Ber97, KN97]. Bauer and Birkedal [BBar] have shown how much of this material fits smoothly into the framework of Scott's *equilogical spaces* (see Section 4).

Other recent results of Normann [Nor00a] and Plotkin [Plo97] involving \mathbf{C} will be mentioned in Section 3.3 below.

2.4 The hereditarily effective operations

As we have seen, one notion of total computable functional of higher type is given by the effective submodel \mathbf{RC} of the total continuous functionals; here we have computable functionals acting on (possibly arbitrary) continuous data. One might also ask if there are interesting type structures based on the idea of computable functions acting only on computable data. Several ways of constructing such a type structure, e.g.

- We might consider higher type generalizations of the definition of type 2 effective operations based on recursive indices (see Definition 1.2).
- We might consider effective *analogues* of various definitions of \mathbf{C} : that is, when building the type structure we only take the effective functionals at each type level.

Both kinds of construction were considered by Kreisel in [Kre59, §4.2].

As an instance of the first kind of construction, he defined an extensional type structure \mathbf{HEO} , together with a closely related non-extensional type structure \mathbf{HRO} .

Definition 2.13 (Hereditarily effective operations) *For each type σ , define a partial equivalence relation \equiv_σ on \mathbf{N} as follows:*

- $x \equiv_0 x'$ iff $x = x'$;
- $x \equiv_{\sigma \rightarrow \tau} x'$ iff whenever $y \equiv_\sigma y'$ we have $\phi_x(y) \downarrow$, $\phi_{x'}(y') \downarrow$ and $\phi_x(y) = \phi_{x'}(y')$.

Now let \mathbf{HEO}_σ be the set of \equiv_σ -equivalence classes. Since there are well-defined total application operations $\mathbf{HEO}_{\sigma \rightarrow \tau} \times \mathbf{HEO}_\sigma \rightarrow \mathbf{HEO}_\tau$ induced by recursive index application, we have a total extensional type structure \mathbf{HEO} over \mathbf{N} .

Definition 2.14 (Hereditarily recursive operations) *For each type σ , define a subset $\mathbf{HRO}_\sigma \subseteq \mathbf{N}$ as follows:*

- $\mathbf{HRO}_0 = \mathbf{N}$;
- $x \in \mathbf{HRO}_{\sigma \rightarrow \tau}$ iff whenever $y \in \mathbf{HRO}_\sigma$, $\phi_x(y) \downarrow$ and $\phi_x(y) \in \mathbf{HRO}_\tau$.

This defines a total non-extensional type structure \mathbf{HRO} over \mathbf{N} , with application given by recursive index application.

The above names, and the designations HEO, HRO, were introduced later by Troelstra, who independently rediscovered these structures around 1970 (see [Tro73, §2.4.17]). Note that the structure HEO is independent (up to isomorphism) of the choice of enumeration for the partial recursive functions, whereas this is not true for HRO — hence it is misleading, strictly speaking, to talk about “the” hereditarily recursive operations.

As an instance of the second kind of construction, Kreisel proposed a recursive analogue of his construction of C via formal neighbourhoods. We give here an equivalent definition based on a recursive analogue of Kleene’s construction of C via associates (Definition 2.10).

Definition 2.15 (Hereditarily recursively countable functionals) *For pure types \bar{n} , define sets HRC_n and notions of associate for elements of HRC_n as follows:*

- Take $\text{HRC}_0 = \mathbb{N}$, $\text{HRC}_1 = \mathbb{N}_{\text{rec}}^{\mathbb{N}}$, and declare each $\alpha \in \text{HRC}_1$ to be an associate for itself.
- For $n \geq 1$, say $\alpha \in \text{HRC}_1$ is an associate for $F : \text{HRC}_n \rightarrow \mathbb{N}$ iff whenever β is an associate for $G \in \text{HRC}_n$ we have $\alpha \mid \beta = F(G)$; and take HRC_{n+1} to be the set of functions $\text{HRC}_n \rightarrow \mathbb{N}$ that have an associate.

The definition of HRC may be extended to arbitrary types by the usual methods. Again, the above name is due to Troelstra.

Kreisel noted the remarkable fact that these two approaches to constructing a class of computable functionals coincide — that is:

Theorem 2.16 $\text{HEO} \cong \text{HRC}$.

At type 2 this is essentially the Kreisel-Lacombe-Shoenfield theorem (Theorem 1.5). The generalization to higher types was noted in [Kre59, §4.2] without detailed proof. Some further details appeared in [Tai62], but a complete published proof first appeared in [Tro73, §2.6].

Some further remarks may be helpful in clarifying the relationship between HRC (the recursive *analogue* of C) and RC (the recursive *submodel* of C). To build RC, one first builds the whole of C and then extracts the effective elements. To build HRC, one considers at each type level only the effective total functionals on the set of effective objects of the next type down. Thus, for instance, the Kleene tree κ lives in HRC_2 , but it is not (the restriction of) an element of RC_2 since it has no computable extension to the whole of $\mathbb{N}^{\mathbb{N}}$. We therefore have a proper inclusion $\text{RC}_2 \hookrightarrow \text{HRC}_2$. Conversely, there is no element in HRC_3 corresponding to the fan functional $\Phi \in \text{RC}_3$, essentially because any computable functional $\Phi' \in \text{HRC}_3$ that extended Φ (with respect to the above inclusion) would be undefined on κ . There is therefore no canonical mapping from RC_3 to HRC_3 or *vice versa*.

The idea here is that κ and Φ cannot live together in the same universe of computable total functionals. Indeed, in Part II we will formulate and prove an “anti-Church’s thesis for total functionals”, to the effect that there is *no* type structure of computable functionals over \mathbb{N} that subsumes both HRC and RC.

This shows that, for hereditarily total functions at least, the dichotomy between “computable acting on computable” and “computable acting on continuous” is a fundamental one.

We have seen that the recursive analogues of both Kleene’s and Kreisel’s definition of \mathbf{C} yield the same type structure \mathbf{HRC} . Later results confirmed the impression that whenever any natural definition of \mathbf{C} admits a recursive analogue, this analogue turns out to be \mathbf{HRC} . Several such equivalences are verified in [Hyl75]. In addition, Ershov [Ers76b, Ers77a] showed that just as the hereditarily total elements of \mathbf{P} give rise to \mathbf{C} (see Theorem 3.10), so the hereditarily total elements of \mathbf{P}^{eff} give rise to \mathbf{HEO} . Ershov also made explicit a higher type generalization of the Kreisel-Lacombe-Shoenfield theorem implicit in the fact that $\mathbf{HEO} \cong \mathbf{HRC}$: namely, that all operations in \mathbf{HEO} are “continuous” in the sense of the neighbourhood structure given by Kreisel’s or Ershov’s definition of \mathbf{HRC} .

This contrasts with the negative result, discovered by Gandy around 1965, that not all the \mathbf{HEO} s are *sequentially* continuous in the sense of Definition 2.11. His ingenious construction of a counterexample at type 3 appears in [GH77, §8].

Later, Bezem [Bez85a] gave another characterization of \mathbf{HEO} as the extensional collapse of \mathbf{HRO} :

Theorem 2.17 *Define partial equivalence relations \sim_σ on each \mathbf{HRO}_σ as follows: $x \sim_{\bar{0}} y$ iff $x = y$; and $f \sim_{\sigma \rightarrow \tau} g$ iff $fx \sim_\tau gy$ whenever $x \sim_\sigma y$. Then the type structure \mathbf{HRO}/\sim constituted by the partial equivalence classes is isomorphic to \mathbf{HEO} .*

What is perhaps surprising is that this result is decidedly non-trivial. The proof essentially goes via the equivalence with \mathbf{HRC} as defined via Kleene associates (see Definition 2.15).

Another descendant of Kreisel’s definition of \mathbf{HEO} was Girard’s category \mathbf{PER} of *partial equivalence relations* on the natural numbers [Gir72]. We will give the definition of this category in Section 4; meanwhile, we simply remark that \mathbf{HEO} can be naturally seen as the finite type structure over N in the cartesian closed category \mathbf{PER} .

3 Partial computable functionals

The picture outlined above exhibits various oddities which may suggest that a theory of hereditarily partial objects of higher type might be much smoother than one for total objects. Intuitively, any computational paradigm powerful enough to generate all total computable functions will naturally generate partial ones as well. Any restriction to total functionals thus seems somewhat artificial, and this artificiality is exacerbated as one passes to higher types. The most striking instance of this is the brood of difficulties associated with Kleene’s $\mathbf{S8}$ (see Section 2.2.1). Another tension arising from the insistence on totality can perhaps be discerned in the incompatibility of the Kleene tree and the fan functional.

The idea that “partial is easier than total” was first aired in [Kle63, §9.3], and was frequently discussed in the later literature, see e.g. [Pla66, GH77,

Fef77a, Ers77a]. We will see in this section that these suspicions are amply justified — that notions of partial computable functional do indeed lead to much simpler theories, and enjoy more pleasant properties, than the total notions. (This accords with our experience in ordinary recursion theory: for example, the partial recursive functions are recursively enumerable but total recursive functions are not.) This is not to say the total notions are necessarily of less interest than the partial ones — we have already seen that certain total type structures have good mathematical credentials — but as we shall see, a study of the partial notions greatly enriches our understanding of the total ones.

Nevertheless, there are some important conceptual questions to be addressed in formulating notions of hereditarily partial object of higher type. For instance, should our type 1 objects be partial functions $N \rightarrow N$ (i.e., functions $N \rightarrow N_\perp$), or should our treatment be so thoroughly partial that we interpret type 0 as N_\perp , in which case our type 1 objects might be (for instance) functions $N_\perp \rightarrow N_\perp$? Questions of this kind will proliferate as we pass to higher types, and at first sight it is unclear how we should respond. One can speculate that a lurking unease about these issues may partly account for the relatively late development of good notions of partial higher type object.

With the benefit of hindsight, we can see that these choices are not as significant as they might seem, since all reasonable choices turn out to be “equivalent” in some sense. We will discuss these issues in detail in Part II, where a precise statement to this effect will be given. For the purposes of our historical survey, however, it will be useful to introduce here two rival definitions of “partial type structure” that both figured in the development of the subject:

Definition 3.1 (Call-by-name, call-by-value) (i) A call-by-name (or CBN) structure is simply an extensional type structure over N_\perp (in the sense of Section 0.4).

(ii) A call-by-value (or CBV) structure is a family of sets A_σ (one for each type) such that $A_{\bar{0}} = N$, together with partial application operations $\cdot_{\sigma\tau} : A_{\sigma \rightarrow \tau} \times A_\sigma \rightarrow A_\tau$ (or $A_{\sigma \rightarrow \tau} \times A_\sigma \rightarrow A_{\tau_\perp}$) satisfying the following extensionality condition:

$$(\forall x \in A_\sigma. f \cdot x \simeq g \cdot x) \implies f = g.$$

This terminology is taken from computer science, and does not appear in the earlier recursion theory literature. The significance of the terms can be appreciated by considering the nature of type 1 objects under the two definitions. In a CBN structure, these are functions $N_\perp \rightarrow N_\perp$, and the argument fed to such a function is not itself a natural number but a *name* for one — a computational process which may or may not evaluate to yield a natural number. In a CBV structure, type 1 objects are functions $N \rightarrow N_\perp$, which demands a natural number as its argument — that is, the computation of the *value* of the argument must take place before the function is called.

As we shall see in Part II, in all cases of interest CBN and CBV structures are equivalent, in that from any CBN structure we may recover a corresponding CBV structure and *vice versa*. We can therefore regard corresponding CBN

and CBV structures simply as different concrete presentations of the same underlying notion of computability. However, this perspective was probably not available to the pioneers of the subject.

Throughout this section, if X is a poset, we write X_{\perp} for the poset obtained by adding a new least element \perp .

3.1 Partial monotone functionals

As far as we are aware, the earliest formulation of a class of hereditarily partial computable functionals at all finite types is due to Davis [Dav59]. Davis (together with Putnam) realized that some restriction on set-theoretic partial functionals was needed to obtain a good theory, and proposed (essentially) a call-by-name structure A of *consistent* partial functionals ($f : A_n \rightarrow \mathbb{N}$ is consistent if whenever $v, w \in \text{dom}(f)$ and for some $t \in A_n$ we have $v \subseteq t$, $w \subseteq t$ then $f(v) = f(w)$), and a notion of computable element of A . Certain features of Davis's definition closely foreshadow Scott's later work on domain theory (see Section 3.2). However, the consistency condition appears to be rather too weak to be really fruitful, and Davis's type structure seems lacking in good properties at higher types.

3.1.1 Platek's thesis

A very satisfactory notion of partial functional was introduced and studied by Platek in his monumental Ph.D. thesis [Pla66]. His first fundamental insight was that a good theory could be obtained by restricting attention to *monotone* partial functionals. Essentially, he considered recursion theory over the following call-by-value structure:

Definition 3.2 (Hereditarily monotone functionals) *For each type σ define a poset M_{σ} as follows: take $M_0 = \mathbb{N}$ with the discrete order; and for $\sigma = \sigma_1 \rightarrow \dots \rightarrow \sigma_r \rightarrow \bar{0}$, let M_{σ} be the set of all monotone functions $M_{\sigma_1} \times \dots \times M_{\sigma_r} \rightarrow \mathbb{N}_{\perp}$ equipped with the pointwise order. Application in M is defined by*

$$f \cdot g = \lambda x_2 \dots x_r. f(g, x_2, \dots, x_r).^9$$

The full total type structure S can be naturally embedded in M via injections $\Psi_{\sigma} : S_{\sigma} \rightarrow M_{\sigma}$: take $\Psi_{\bar{0}}(x) = x$, and for $\sigma = \sigma_1 \rightarrow \dots \rightarrow \sigma_r \rightarrow \bar{0}$ we let $\Psi_{\sigma}(F)(\mathbf{g}) \simeq y \in \mathbb{N}$ iff there exist $G_i \in S_{\sigma_i}$ with $\Psi_{\sigma_i}(G_i) \sqsubseteq g_i$ and $F(\mathbf{G}) = y$. We say $f \in M_{\sigma}$ represents $F : S_{\sigma_1} \times \dots \times S_{\sigma_r} \rightarrow \mathbb{N}$ if $f(\Psi(\mathbf{G})) \simeq F(\mathbf{G})$ for all \mathbf{G} . If $F \in S_{\sigma}$, clearly $\Psi(F)$ represents (the uncurried form of) F .

⁹Platek used the term *consistent* rather than *monotone*. The structure M can be construed as a call-by-value structure in our sense, though it does not quite coincide with the "natural" full monotone call-by-value structure over \mathbb{N} , for which one would take $M_{\sigma \rightarrow \tau}$ to be the set of monotone functions $M_{\sigma} \rightarrow M_{\tau}$. However, for pure types the two definitions coincide exactly, and the minor difference can be glossed over.

Platek actually defined such a structure relative to an arbitrary set of objects in place of \mathbb{N} , since one of his aims was to give a uniform treatment of recursion theory which applied also to other domains, such as ordinals and transitive sets.

One can give a computational motivation for the restriction to monotone functionals: if f is any computable partial functional, we would expect an increase in the available information about the input to f to allow (if anything) an increase in the output information produced by f . Platek's main motivation, however, was to provide a setting in which definitions by recursion make sense. Given any recursion equation

$$f(x) = F(f, x)$$

where $x : \sigma, f : \sigma \rightarrow \tau$ are variables and $F \in \mathbf{M}_{(\sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau}$, Tarski's fixed point theorem ensures that we have a unique least solution $f \in \mathbf{M}_{\sigma \rightarrow \tau}$ (satisfying the equation for all $x \in \mathbf{M}_\sigma$), which may be obtained via a transfinite iteration of F :

$$f_0(x) = \perp, \quad f_{\alpha^+}(x) = F(x, f_\alpha), \quad f_\lambda = \bigcup_{\alpha < \lambda} f_\alpha, \quad f = \bigcup f_\alpha.$$

As mentioned earlier, Platek was more interested in questions of *definability* than of effective computability, so the transfinite nature of the “computation” here was not seen as a problem.

Platek's second fundamental insight was that definitions by recursion can be conveniently expressed by means of *fixed-point operators*. Specifically, for each type $\rho = \sigma \rightarrow \tau$ there is an element $Y_\rho \in \mathbf{M}_{(\rho \rightarrow \rho) \rightarrow \rho}$ such that for all $F \in \mathbf{M}_{\rho \rightarrow \rho}$, YF is the unique least solution f to $f(x) = F(f, x)$.

Other (simpler) ways of explicitly defining new elements of \mathbf{M} from old ones are encapsulated by the elements $I_\sigma, K_{\sigma\tau}, S_{\sigma\tau\nu}, D \in \mathbf{M}$ defined by

$$\begin{aligned} I_\sigma x^\sigma &= x, \\ K_{\sigma\tau} x^\sigma y^\tau &= x, \\ S_{\sigma\tau\nu} f^{\sigma \rightarrow \tau \rightarrow \nu} g^{\sigma \rightarrow \tau} x^\sigma &= (fx)(gx), \\ Dx^{\bar{0}} y^{\bar{0}} f^{\bar{1}} g^{\bar{1}} &= \begin{cases} f & \text{if } x = y, \\ g & \text{otherwise.} \end{cases} \end{aligned}$$

(Here D stands for “definition by cases”.) These lead us to the following definition of recursive definability over \mathbf{M} :

Definition 3.3 *Given any set $B \subseteq \mathbf{M}$, define the set $\mathcal{R}(B)$ of elements recursively definable from B to be the smallest subset of \mathbf{M} closed under application and containing B and all the elements Y, I, K, S, D .*

In the case of the monotone type structure over \mathbf{N} , we will usually be interested in the case where B contains “enough” basic computable elements (zero, successor and predecessor suffice¹⁰). Let us call an element of \mathbf{M} *recursively definable* if it is recursively definable from these basic elements alone.

Platek showed that the above definition can also be cast in terms of schemata in the spirit of Kleene's S1–S9, but with an important difference: *any* definition of a computation written using these schemata has a meaning in \mathbf{M} . Thus

¹⁰Gandy pointed out that in fact zero and successor suffice in Platek's setting.

the theory does not suffer from the difficulties arising from the infinitary side-condition in S8.

Moreover, one can recover Kleene’s notion of computability over \mathbf{S} as follows: any partial function $F : \mathbf{S}_{\sigma_1} \times \cdots \times \mathbf{S}_{\sigma_r} \rightarrow \mathbf{N}$ is Kleene computable iff it is represented by some recursively definable $f \in \mathbf{M}$. Using this as an alternative definition of Kleene computability, Platek was able to obtain clearer proofs of many of Kleene’s results plus some new ones, such as that the first recursion theorem holds subject to a type level restriction (see p. 123). However, the equivalence with Kleene’s definition is non-trivial and depends on the difficult substitution theorems of [Kle59b]; this seems to reflect the unmanageability of the schemata S1–S9 (see the discussions on pp. 114–5, 168–9).

Platek was also the first to draw attention to the famous *parallel or* function and the issues it raises (pp. 127–131). Let us suppose $0 \in \mathbf{N}$ stands for “true” and any other natural number stands for “false”, and consider the element $por \in \mathbf{M}_{\bar{0} \rightarrow \bar{0} \rightarrow \bar{0}}$ (called *strong or* by Platek) defined by

$$por\ x\ y = \begin{cases} 0 & \text{if } x \text{ is true,} \\ 0 & \text{if } y \text{ is true,} \\ 1 & \text{if } x \text{ and } y \text{ are both false,} \\ \perp & \text{otherwise.} \end{cases}$$

Platek pointed out that *por* is not recursively definable, although it is definable in an Herbrand-Gödel-style equation calculus, and is arguably computable if we allow a kind of non-determinism in computations with regard to the order of evaluation of arguments. As we shall see below, this dichotomy between different notions of computability was to become a major theme in later work.

In addition to the above definition of recursive definability, Platek gave a characterization in terms of a formal language based on the lambda calculus, closely foreshadowing PCF (see below). Rather curiously, the language he introduced is an *untyped* lambda calculus — this allowed him to make use of the counterparts of Y and D as well as I, K, S in pure lambda calculus. Platek’s calculus also featured some sophisticated machinery for allowing a mixture of syntax and semantics in computations — e.g. a constant $[f]$ for every element $f \in \mathbf{M}$ was admitted, and the definition of the evaluation relation $M \rightarrow n$ included rules of the following kind (similar to Kleene’s S8):

$$\begin{array}{l} \text{if } f \in \mathbf{M}_{(\sigma \rightarrow \bar{0}) \rightarrow \bar{0}}, g \in \mathbf{M}_{\sigma \rightarrow \bar{0}} \\ \text{and } M[h] \rightarrow g(h) \text{ for all } h \in \mathbf{M}_{\sigma} \text{ such that } g(h) \in \mathbf{N}, \\ \text{then } [f]M \rightarrow f(g). \end{array}$$

The infinitary character of such rules means that computations can be of transfinite length, although for terms containing no constants of type level ≥ 2 , computations are always finite.

One of the major results of [Pla66] is the following (proved under mild conditions on B):

Theorem 3.4 *An element $f \in \mathbf{M}_{\sigma_1 \rightarrow \dots \rightarrow \sigma_r \rightarrow \bar{0}}$ is recursively definable from $B \subseteq \mathbf{M}$ iff f is λ -definable from B (in the sense that there is a λ -term M , containing*

no constants except those corresponding to elements of B , such that for all $g_i \in M_{\sigma_i}$ we have $M[g_1, \dots, g_r] \rightarrow n$ iff $f\mathbf{g} = n$.

The forwards implication here is easily shown by supplying lambda terms corresponding to I, K, S, Y, D . To show the converse, Platek proved the existence of recursively definable enumerators $E_\sigma \in M_{\bar{0} \rightarrow \sigma}$ such that if M λ -defines $f \in M_\sigma$ then $E_\sigma(\lceil M \rceil) = f$ (where $\lceil M \rceil$ is a Gödel-number for M). (Essentially the same result and proof were rediscovered in [LP97]).

In summary, Platek’s thesis was a major achievement which both shed considerable light on Kleene’s earlier work on S1–S9 computability, and introduced many of the fundamental ideas explored in later work on PCF. Indeed, his recursively definable elements of M are exactly the PCF-definable ones in the sense of Scott and Plotkin; and his work shows the existence of a programming language with a *finitary* notion of computation for expressing these elements. With a little charity, one can read into the results of his Chapter 4 a proof of adequacy for M as a model of combinatory PCF.

Platek’s thesis is unfortunately not generally available, but a detailed published account of much of the material appears in [Mol77].

3.1.2 Kleene’s later work

In his later years, Kleene returned to the study of higher type computability and published a series of papers under the title “Recursive functionals and quantifiers of finite types revisited” [Kle78, Kle80, Kle82, Kle85, Kle91]. Of these papers, [Kle85] provides the best introduction to the whole series.

Kleene’s motivation was spelt out in [Kle78, §1.2]:

I aim to generate a class of functions $\phi(\mathcal{M})$ which shall coincide with all the partial functions which are “computable” or “effectively decidable”, so that Church’s 1936 thesis (IM §62) will apply with the higher types included (as well as to partial functions, IM p. 332).

(This has been called “Kleene’s problem” by Hyland and Ong [HO00].) Like Platek, Kleene sought to avoid the anomalies of the original S1–S9 theory by developing a theory of hereditarily partial objects. However, whereas Platek was interested primarily in questions of definability, Kleene was more interested in the concrete nature of computations at higher types.

Kleene concentrated mostly on the type levels $j = 0, 1, 2, 3$. In [Kle78] he introduced a collection of schemata for defining a class of computable functions: a group of schemata S0–S7 (similar in spirit to the earlier S1–S8 though different in a few details), together with a new schema S11 to take over the role of S9. (No one seems to know what happened to S10.) We adapt Kleene’s formulation slightly for consistency with the notation of Definition 2.2:

S11 *General recursion:* $\{\langle 11, [\sigma], g \rangle\}(\mathbf{x} : \sigma) \simeq \{g\}(\lambda \mathbf{x}. \{\langle 11, [\sigma], g \rangle\}(\mathbf{x}), \mathbf{x})$
 (more briefly, $\{h\}(\mathbf{x}) \simeq \{g\}(\{h\}, \mathbf{x})$ where $h = \langle 11, [\sigma], g \rangle$).

Whereas in the earlier theory the schema S9 had given rise to restricted versions of the first recursion theorem, the new schema S11 has the effect of building a

natural formulation of the first recursion directly into the definition, and closure under S9 can be derived as a consequence.

In order to give an interpretation of these schemata which makes sense of the apparent circularity in S11, Kleene gave in [Kle78] a syntactic description of computations as computation trees labelled by formal expressions. (Kleene uses the term *j-expression* for a formal expression of type \bar{j} .) The essential difference between these and the computation trees of [Kle59b, Kle63] is that in order to evaluate an expression such as $\alpha(\lambda x.\beta^1(x), y^0)$ (where α, β are themselves defined via the schemata), we are no longer required to launch a subcomputation compute the whole graph of $\lambda x.\beta^1(x)$, but instead may plough on with the main computation, carrying around the $\lambda x.\beta(x)$ as an unevaluated formal expression. Only if we later require a particular value for β , such as $\beta(5)$ do we engage in a computation for β . Computation therefore takes on a much more syntactic character: we are really computing with formal expressions rather than with the semantic objects they denote, and so it need not matter that not every formal expression denotes a semantic object.

In general, the formal expressions in Kleene’s computations are allowed to contain free variables of type levels 0, 1, 2, 3: computations then proceed relative to an interpretation of these free variables as semantic objects. For this reason, a successful computation tree for a 0-expression (that is, one that results in a numerical value) may be infinitely branching and hence transfinitely deep: for example, if we wish to compute $\alpha^2(B)$ where α^2 is a free variable and B is a 1-expression, we really do need to compute $B(n)$ for each numeral n . However, a successful computation tree for a *closed* 0-expression will always be a finite object. Indeed, Kleene’s definition of computations closely foreshadows the operational definition of PCF — though as in his earlier work, his seeming reluctance to make fuller use of the typed λ -calculus is puzzling.

In [Kle80] and the following papers, Kleene proceeded to develop a semantics for his formal expressions which allow us to give an interpretation for the objects considered at all stages during a computation. In effect, Kleene constructed (the first few levels of) a call-by-name type structure U consisting of what he termed the *unimonotone* functionals (*monotone* with a *unique* and *intrinsically* determined basis). Here one takes U_0 to be N_\perp and U_{j+1} to be a certain set of monotone functions from U_j to U_0 , ordered pointwise. As in Platek’s work, the monotonicity reflects the computational intuition that if we are able to obtain some output value without seeing a certain piece of input information, this value should not be affected if we later see it. Furthermore, Kleene restricted attention to those monotone functionals that could be computed by an oracle who followed a strategy or protocol of a certain kind. Kleene gave detailed explicit descriptions of the possible behaviours of such oracles at types 2 and 3, which foreshadow later work on game semantics for PCF. However, Kleene’s definitions, couched in terms of the colourful imagery oracles presented with envelopes containing other oracles, were somewhat lacking in clarity and mathematical crispness, and moreover the generalization to higher types was left unclear. It seems that the expression of Kleene’s ideas would have been facilitated by some of the basic ideas and terminology of domain theory and even category theory which were then available.

Nevertheless, several interesting ideas emerged from Kleene’s work. For instance, Kleene isolated an important feature of “sequential” strategies for oracles: a type 2 oracle O , unless she computes a constant type 2 function, must be able of her own accord to produce a type 0 object x (i.e. an element of \mathbb{N}_\perp) to serve as the first question she poses to her type 1 argument f (so that if $f(x) = \perp$ then the whole computation hangs up). Kleene originally conjectured that a similar property would hold at type 3: that a non-constant type 3 oracle would always be able to come up with a particular type 1 object to serve as the first question posed to its type 2 argument. However, this was refuted by Kleene’s student Kierstead, who gave as a simple counterexample the type 3 functional defined by

$$\Psi(F^2) = F(\lambda x^0. F(\lambda y^0. x)).$$

Intuitively, the first question asked to the argument F here is the type 1 function $\lambda x. F(\lambda y. x)$, but this function is itself dependent on information about F which only emerges later in the dialogue. This example shows that the possible interactions between oracles and their arguments are somewhat more subtle than Kleene had at first envisaged.

In the subsequent papers, Kleene was able to draw his description of type 3 oracles to a satisfactory completion. An important feature of Kleene’s semantics is its intensional character: his oracles work by acting on other oracles rather than on pure function extensions. Indeed, the requirement that oracles behave extensionally (that is, give the same result when presented with two different oracles for the same function) has to be explicitly incorporated into the definition at certain points. This intensional character is also an important aspect of much of the work in computer science on the semantics of higher type computation: for instance, Berry and Curien’s sequential algorithms model [BC82], or the game models of PCF [AJM00, HO00, Nic94].

Kleene’s unimonotone semantics can be considered as one in which computable objects act only on computable objects, if “computable” here is understood to mean realized by some oracle. However, the behaviour of oracles is not required to be effectively given, and so unimonotone functionals need not be computable in a genuinely effective sense. Even more strikingly, they need not be continuous: indeed, Kleene seems to have made a quite deliberate decision to allow infinite computation trees, and moreover to allow the behaviour of oracles to depend on infinitely much information about their arguments. In this regard, Kleene’s work contrasts sharply with most work in the computer science tradition; indeed, from a modern perspective, Kleene’s decision to impose such tight and computationally motivated constraints in the direction of unimonotonicity but not in the direction of continuity stands as something of a curiosity.

In the meantime, Kierstead [Kie80, Kie83] developed an alternative semantics for Kleene’s S0–S7 and S11, closer in spirit to Platek’s work in that it makes use of the full monotone type structure over \mathbb{N}_\perp . Many of Kierstead’s results are closely parallel to Platek’s: for instance, he embeds the total set-theoretic type structure \mathbb{S} in the monotone one in a way which respects Kleene

computability, and from this is able to deduce certain substitution properties for the total Kleene computable functionals. One difference is that Kierstead considers the call-by-name monotone type structure whereas Platek considers the call-by-value one — this appears to be correlated to some difference in the conceptual status of the models in the two approaches, but from a purely technical point of view it does not seem that this is a major difference. (Some relevant results are obtained in the Ph.D. thesis of van Draanen [vD95].)

The present author feels that Kleene’s work in this area contributed some important ideas, although most of these have been more clearly expressed and more fully explored in the subsequent computer science literature. A useful paper by Bucciarelli [Buc93a] (see also [Buc93b]) makes explicit the relationships between Kleene’s work and subsequent developments in computer science, and in particular explores the connections between Kleene’s oracles and Berry-Curien sequential algorithms (see Section 3.3 below). The influence of Kleene’s ideas on the computer science tradition is further discussed in [HO00, Section 1.4].

3.2 Partial continuous functionals

Undoubtedly one of the most compelling notions of higher type computability was discovered independently by Scott [Sco93, Sco69] and Ershov [Ers72, Ers73]. Coming from somewhat different directions, these authors both arrived at the same type structure \mathbf{P} of *partial continuous functionals* and an effective substructure \mathbf{P}_{eff} of partial computable functionals.

3.2.1 Scott’s approach

Scott’s work can be seen as bringing together Platek’s idea of using monotone partial functionals in recursion theory, and the Kleene-Kreisel idea of using continuity to capture the finitary nature of computations. This leads to a theory of computable functionals which is arguably simpler and better behaved than either Platek’s or Kleene and Kreisel’s. Scott was also motivated by the idea of giving a mathematical theory of the meanings of computer programs — an abstract view of the mathematical functions they compute as distinct from a purely symbolic machine-oriented account of how they behave — and his work lies at the root of the modern computer science tradition in denotational semantics.

The simplest definition of the partial continuous functionals was formulated in [Sco93]:

Definition 3.5 (Partial continuous functionals) (i) A complete partial order (CPO) is a poset (X, \sqsubseteq) with a least element, in which every chain $x_0 \sqsubseteq x_1 \sqsubseteq \dots$ has a least upper bound $\bigsqcup x_i$.

(ii) A function $f : X \rightarrow Y$ between CPOs is continuous if for every chain $x_0 \sqsubseteq x_1 \sqsubseteq \dots$ in X we have $f(\bigsqcup x_i) = \bigsqcup f(x_i)$.

(iii) Define a call-by-name type structure \mathbf{P} as follows: let \mathbf{P}_0 be \mathbf{N}_\perp (considered as a CPO with the usual partial ordering), and $\mathbf{P}_{\sigma \rightarrow \tau}$ is the CPO of monotone and continuous functions $f : \mathbf{P}_\sigma \rightarrow \mathbf{P}_\tau$.

Thus, \mathbf{P} is simply the natural type structure over \mathbf{N}_\perp in the cartesian closed category of CPOs. Scott showed that \mathbf{P} served as a model for a formal language \mathcal{L} directly inspired by Platek’s work, consisting of the well-typed expressions built up via application from the combinators $\mathbf{I}, \mathbf{K}, \mathbf{S}, \mathbf{Y}$ and \mathbf{if} , together with constants $\mathbf{zero}, \mathbf{succ}, \mathbf{pred}$ denoting basic arithmetical operations.¹¹ Here the combinator \mathbf{Y}_σ is interpreted by the function Y_σ which assigns to any $f \in \mathbf{P}_{\sigma \rightarrow \sigma}$ the *least* fixed point of f in \mathbf{P}_σ . The combinator $\mathbf{if} : \bar{0} \rightarrow \bar{0} \rightarrow \bar{0}$ takes over the role of Platek’s \mathbf{D} ; the corresponding element of \mathbf{P} is defined by

$$\mathbf{if}(0)(x)(y) = x, \quad \mathbf{if}(n+1)(x)(y) = y, \quad \mathbf{if}(\perp)(x)(y) = \perp.$$

Scott also introduced a logic for proving simple assertions about the functions definable in \mathcal{L} , conceived as a way of proving semantic properties of programs. The language \mathcal{L} later became known as PCF (Programming language for Computable Functions),¹² and the associated logic as LCF (Logic of Computable Functions).

The class of \mathcal{L} -definable elements gives rise to one possible notion of computable function in \mathbf{P} , including all partial recursive functions at type 1 and in some ways analogous to the notion of Kleene computable element of \mathbf{C} . However, a more intrinsic notion of computable element of \mathbf{P} can be given, by exploiting the fact that the CPOs \mathbf{P}_σ are all *domains*. Intuitively, a domain is a CPO in which there is a good notion of “finite piece of information” about an element, and in which every element is determined by the set of finite pieces of information about it. The following definitions were introduced by Scott in [Sco69]; for more details see any standard text on domain theory (e.g. [SHLG94]).

Definition 3.6 (Scott domains) (i) A subset D of a poset (X, \sqsubseteq) is directed if it is non-empty and for all $x, y \in D$ there exists $z \in D$ with $x \sqsubseteq z$ and $y \sqsubseteq z$. A DCPO is a poset (X, \sqsubseteq) with a least element \perp in which every directed subset D has a least upper bound $\bigsqcup D$. A function $f : X \rightarrow Y$ between DCPOs is a continuous map if it is monotone and preserves lubs of directed sets.

(ii) In a DCPO X , we say x, y are consistent (and write $x \uparrow y$) if they have an upper bound in X . A DCPO X is consistently complete if whenever $x, y \in X$ have an upper bound, they have a least upper bound $x \sqcup y$.

(iii) An element e of a DCPO X is finite if whenever D is directed and $e \sqsubseteq \bigsqcup D$, we have $e \sqsubseteq d$ for some $d \in D$. We write F_x for the set of finite elements $e \sqsubseteq x$.

(iv) A DCPO is algebraic if for every element x , F_x is directed and has lub x . A DCPO is ω -algebraic if it is algebraic and its set of finite elements is countable.

(v) A domain is a consistently complete, ω -algebraic DCPO.

¹¹Actually Scott’s system had a ground type of truth-values as well as one of integers. Here, as earlier, we will content ourselves with representing “true” by zero and “false” by any other natural number.

¹²We will use the name PCF to refer to either the combinatory language described here, or its λ -calculus counterpart as presented e.g. in [Plo77]. The possibility of these variants was already clear to Scott in [Sco93], and the differences between them are of minor importance for our purposes.

Scott showed that the CPOs P_σ are all domains, and moreover that in each P_σ one can give an *effective enumeration* e_0, e_1, \dots of the finite elements, in such a way that the relations $e_i \uparrow e_j$ and $e_i \sqcup e_j = e_k$ are semirecursive in i, j, k . One can then call an element $x \in P_\sigma$ *effective* if the finite pieces of information about x are recursively enumerable, that is, if $\{i \mid e_i \sqsubseteq x\}$ is r.e. The effective elements of P are closed under application and so constitute a substructure P^{eff} .

Whereas it is easy to see that every \mathcal{L} -definable element is effective in this sense, the converse is not true, as the “parallel-or” function is effective but not \mathcal{L} -definable. As Scott pointed out in [Sco93], there is a genuine choice as to whether we wish to regard this function as computable — evidently the kind of algorithmic procedure needed to compute it is of a flavour significantly different from those which suffice to compute the \mathcal{L} -definable elements.

Neither [Sco93] nor [Sco69] was published at the time, though they circulated widely in manuscript and had a considerable influence on later work. ([Sco93] was eventually published in the Böhm Festschrift in 1993.) In his subsequent papers [Sco70, Sco72, Sco76], Scott shifted his attention from domains to *complete lattices*, which give rise a very similar theory but have a rather simpler and more familiar definition. Despite the elegance of the lattice-theoretic approach, however, it can be argued that it takes us a step further away from computationally meaningful type structures: for instance, even the type of natural numbers now needs to be represented by a poset with a top element, construed as “inconsistent information” regarding the natural number in question. In the 1980s Scott returned to the original domain-theoretic ideas and gave a more concrete presentation of them in terms of *information systems* [Sco82], in which the finite elements (finite pieces of information) were taken as primary.

3.2.2 Ershov’s approach

In the meantime, Ershov had given a construction of P^{eff} of a quite different character, arising from his theory of *enumerated sets*. This theory offers a framework for studying a wide range of mathematical structures from an algorithmic or recursion-theoretic point of view (for a recent survey and further references, see [Ers99]). The basic definitions are as follows:

Definition 3.7 (Enumerated sets) (i) An enumerated set is simply a set X equipped with a total function $\nu : \mathbb{N} \rightarrow X$ (called an enumeration). If $\nu(n) = x$, we may say that n is a code or recursive index for x .

(ii) A morphism from (X, ν) to (X', ν') is a function $\mu : X \rightarrow X'$ such that there exists a recursive function f satisfying $\mu \circ \nu = \nu' \circ f$.

In [Ers71b, Ers71a] Ershov introduced the category **EN** of enumerated sets, and considered the question of when the set of morphisms $(X, \nu) \rightarrow (X', \nu')$ can be endowed with an enumeration making it into the category-theoretic exponential $(X', \nu')^{(X, \nu)}$. In particular, he defined certain classes $\mathcal{C}_2, \mathcal{C}_{20}$ of enumerated sets, and proved:

Theorem 3.8 If $E \in \mathcal{C}_2$ and $E' \in \mathcal{C}_{20}$, then the exponential E'^E exists in **EN** and belongs to \mathcal{C}_{20} .

(We omit here the somewhat technical definitions of \mathcal{C}_2 and \mathcal{C}_{20} , which are reproduced e.g. in Section 3.3 of [Ers99].) Since \mathbf{N} (with the identity enumeration) is in \mathcal{C}_2 and \mathbf{N}_\perp (with an obvious enumeration) is in \mathcal{C}_{20} , it follows that we can construct both call-by-name and call-by-value type structures by repeated exponentiation in \mathbf{EN} .

Significantly, these type structures are constructed purely out of computable objects acting on computable objects — no notion of continuity is involved in the definition.

In [Ers72] Ershov gave a topological presentation of these structures using the concept of f_0 -spaces:

Definition 3.9 (i) For any T_0 topological space X , let \sqsubseteq_X be the partial order defined by

$$x \sqsubseteq_X y \Leftrightarrow \text{for every open set } V, \text{ if } x \in V \text{ then } y \in V.$$

Call an open non-empty set V an f -set if it contains a least element with respect to \sqsubseteq_X .

(ii) An f_0 -space is a T_0 space X in which the family of f -sets, together with the empty set, is closed under binary intersections and forms a basis for the topology on X , and moreover the whole of X is an f -set.

(iii) Let us write X_0 for the set of elements $x \in X$ for which $\{y \mid x \sqsubseteq y\}$ is open. We say $I \subseteq X_0$ is an ideal if it is downward closed under \sqsubseteq and every pair of elements in I has a least upper bound in I . An f_0 -space X is complete if (X, \sqsubseteq) is canonically isomorphic to the set of ideals in X_0 ordered by inclusion.

Ershov showed, in effect, that the category of complete f_0 -spaces and continuous maps is cartesian closed, and moreover that the (call-by-name) type structure over \mathbf{N}_\perp in \mathbf{EN} coincided with the substructure of “effective elements” in the type structure over the complete f_0 -space \mathbf{N}_\perp . (It is natural to think of this as a higher type generalization of the Myhill-Shepherdson theorem.) In [Ers72, Section 8] he noted in passing that this class of computable functionals had many pleasing recursion-theoretic properties, remarking that

It is fully justified to consider [this class] as the most natural generalization (more precisely, extension) of the class of partially recursive functions.

Thus far Ershov’s work had proceeded independently of Scott’s, but the connections were made explicit in [Ers73]. Indeed, it is not hard to see that domains and complete f_0 -spaces are essentially equivalent, and that Ershov’s type structures coincide exactly with \mathbf{P} and \mathbf{P}^{eff} . One may say that Ershov’s definitions made greater use of topological concepts where Scott’s had emphasized the order-theoretic ideas, though in fact both authors made good use of the interplay between the two perspectives.

In [Ers74a, Ers77a] Ershov obtained the following relationship between \mathbf{P} and the Kleene-Kreisel type structure \mathbf{C} :

Theorem 3.10 (Hereditarily total elements) *Define a substructure $T \subseteq \mathbf{P}$ as follows: $T_0 = \mathbf{N} \subset \mathbf{N}_\perp$; and $T_{\sigma \rightarrow \tau} = \{f \in \mathbf{P}_{\sigma \rightarrow \tau} \mid \forall x \in T_\sigma. fx \in T_\tau\}$. Define total equivalence relations \sim_σ on T_σ as follows: $x \sim_{\bar{0}} y$ iff $x = y$; and $f \sim_{\sigma \rightarrow \tau} g$ iff $\forall x \in T_\sigma. fx \sim_\tau gx$. Then the sets T_σ / \sim_σ constitute a type structure that is canonically isomorphic to \mathbf{C} .*

Moreover, if $T_\sigma^{\text{eff}} = T_\sigma \cap \mathbf{P}_\sigma^{\text{eff}}$, the sets $T_\sigma^{\text{eff}} / \sim_\sigma$ constitute a type structure isomorphic to \mathbf{RC} .

An analogous result for effective type structures — that the extensional collapse of \mathbf{P}^{eff} gives rise to \mathbf{HEO} — was proved in [Ers76b].

Ershov’s main results on \mathbf{P} and \mathbf{P}^{eff} are conveniently summarized in [Ers77a]. Some further discussion of the relationship between the partial and total type structures also appears in Sections 9 and 10 of [GH77]. A streamlined presentation of the main results on f_0 -spaces appears in [GL84], together with some applications to the semantics of *untyped* λ -calculus.

3.2.3 Parallel PCF and after

The next major result about \mathbf{P}^{eff} was obtained independently by Plotkin [Plo77] and Sazonov [Saz76a], who showed that the gap between the notions of PCF computability and effectivity could be bridged by augmenting PCF with two operations of a “parallel” flavour:

Theorem 3.11 *Let \mathbf{PCF}^{++} be the language PCF extended with the two constants*

$$\text{parallel-or} : \bar{0} \rightarrow \bar{0} \rightarrow \bar{0}, \quad \text{exists} : (\bar{0} \rightarrow \bar{0}) \rightarrow \bar{0}$$

and extend the interpretation of PCF in \mathbf{P} by interpreting parallel-or by the function por defined as in Section 3.1.1, and exists by the functional exists defined by

$$\text{exists}(f) = \begin{cases} 0 & \text{if } f(x) = 0 \text{ for some } x \in \mathbf{N}, \\ 1 & \text{if } f(\perp) = n + 1 \text{ for some } n \in \mathbf{N}, \\ \perp & \text{otherwise.} \end{cases}$$

Then the elements of \mathbf{P}^{eff} are exactly the \mathbf{PCF}^{++} -definable elements of \mathbf{P} .¹³

As pointed out in [Fef77a], this was a significant result in that it showed that this notion of computability on \mathbf{P} could be captured by a finite collection of inductive schemata, somewhat in the spirit of Kleene’s S1–S9. Interestingly, it seems that Kleene himself never considered parallel operations as a way of getting more expressive power, although he was explicitly interested in the problem of identifying a class of “all” computable functions at higher types.

In addition, Plotkin in [Plo77] gave an operational semantics (that is, a set of symbolic evaluation rules) for a mild variant of \mathbf{PCF}^{++} , showing that as a programming language this system could stand alone as mathematical object in its own right. Indeed, the type structure \mathbf{P}^{eff} can be characterized

¹³In fact Plotkin considered a parallel conditional operator rather than parallel or, but it is easy to show that these are interdefinable in PCF (see [Sto91a]).

up to isomorphism as the closed term model for PCF^{++} modulo observational equivalence. (A set of evaluation rules for PCF in this spirit will be given below.) It is also worth noting here a later result of Stoughton [Sto91b] that \mathbf{P} is up to isomorphism the *unique* extensional continuous model for PCF with parallel-or.

Another important structural property was obtained by Plotkin in [Plo78]:

Theorem 3.12 *The Scott domain \mathbb{T}^ω of functions $\mathbf{N} \rightarrow 2_\perp$ ordered pointwise is a universal domain: that is, for any Scott domain X there are continuous maps $f_X : X \rightarrow \mathbb{T}^\omega$ and $g_X : \mathbb{T}^\omega \rightarrow X$ such that $g_X \circ f_X = \text{id}_X$ (we say X is a retract of \mathbb{T}^ω). Moreover, if X is an effective domain then these maps are computable.*

It follows that any domain that is rich enough to contain \mathbb{T}^ω as a computable retract (such as \mathbf{P}_1) contains all the domains \mathbf{P}_σ as computable retracts. Indeed, the whole of \mathbf{P} can be reconstructed from \mathbf{P}_1 together with its set of continuous endofunctions; likewise, \mathbf{P}^{eff} can be reconstructed from the monoid of computable endofunctions on \mathbf{P}_1^{eff} .

In some sense all the most important facts about \mathbf{P} and \mathbf{P}^{eff} were understood by this stage, though a few later results and characterizations were forthcoming. For instance, several people noted that Ershov's category \mathbf{EN} has poor closure properties (e.g. it is not cartesian closed), and proposed larger categories to remedy this. Mulry [Mul82] showed that \mathbf{EN} could be embedded into his *recursive topos* preserving existing exponentials, so that the type structure over a suitable object N_\perp in this topos coincides with \mathbf{P}^{eff} .¹⁴ This is very close in spirit to another characterization of \mathbf{P}^{eff} given by Longo and Moggi [LM84b], which gives the remarkable appearance of magically dispensing with any explicit computability requirement at higher types:

Definition 3.13 *Let $fst, snd : \mathbf{N} \rightarrow \mathbf{N}$ be the projections associated with some recursive pairing function $\mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$. For each type σ we define a set \mathbf{P}_σ^{eff} together with a set $\mathbf{P}_\sigma^{\omega,eff}$ of functions $\mathbf{N} \rightarrow \mathbf{P}_\sigma^{eff}$ as follows:*

$$\begin{aligned} \mathbf{P}_0^{eff} &= \mathbf{N}_\perp \\ \mathbf{P}_0^{\omega,eff} &= \{\text{monotone computable functions } \mathbf{N}_\perp \rightarrow \mathbf{N}_\perp\} \\ \mathbf{P}_{\sigma \rightarrow \tau}^{eff} &= \{f : \mathbf{P}_\sigma^{eff} \rightarrow \mathbf{P}_\tau^{eff} \mid \forall g \in \mathbf{P}_\sigma^{\omega,eff}. f \circ g \in \mathbf{P}_\tau^{\omega,eff}\} \\ \mathbf{P}_{\sigma \rightarrow \tau}^{\omega,eff} &= \{f : \mathbf{N} \rightarrow \mathbf{P}_{\sigma \rightarrow \tau}^{eff} \mid \forall g \in \mathbf{P}_\sigma^{\omega,eff}. \lambda n. f(fst\ n)(g(snd\ n)) \in \mathbf{P}_\tau^{\omega,eff}\} \end{aligned}$$

(This is a mild variation on the definition given in [LM84b]— the latter relies on a theorem ensuring the existence of suitable pairing operations at higher types.) Inspired by this characterization, Longo and Moggi [LM84a] introduced another cartesian closed category extending \mathbf{EN} : the category of *generalized numbered sets*. More recently, Moggi [Mog96] has clarified the relationship between this category and the recursive topos.

¹⁴It is worth noting, in passing, that the type structure over N in this topos does not coincide with any of our type structures of total computable functionals, but rather yields the *generalized Banach-Mazur functionals*: see [Mul82] and cf. Section 1.2.

Later work has tended to concentrate on the category of PERs on \mathbb{N} , into which \mathbf{EN} can be trivially embedded. An brief account of this category, together with a few other results on \mathbf{P} and \mathbf{P}^{eff} arising from the study of realizability models, will be given in Section 4 below.

3.3 PCF and sequential computability

Although, as we have seen, not all the effective elements of \mathbf{P} are PCF-definable, the sense persisted that the notion of computability embodied by PCF is a natural one and of interest in its own right. For reference, we give here a suitable set of deterministic reduction rules for PCF as defined above, much in the spirit of [Mil77]:

Definition 3.14 (Operational semantics for PCF) *We inductively define a binary relation $M \rightarrow N$ on closed PCF terms of the same type via the following clauses (we write \hat{k} as sugar for $(\text{succ}^k \text{zero})$):*

- $\lambda M \rightarrow M, KMN \rightarrow M, SMNP \rightarrow (MP)(NP), YM \rightarrow M(YM).$
- $\text{pred zero} \rightarrow \text{zero}, \text{pred } k \hat{+} 1 \rightarrow \hat{k}.$
- if $\text{zero } MN \rightarrow M, \text{if } k \hat{+} 1 MN \rightarrow N$
- If $M \rightarrow M'$ then $MN \rightarrow M'N.$
- If $M \rightarrow M' : \bar{0}$ then $\text{succ } M \rightarrow \text{succ } M', \text{pred } M \rightarrow \text{pred } M', \text{if } MN P \rightarrow \text{if } M' N P.$

We write \rightarrow^* for the reflexive transitive closure of \rightarrow . A term $M : \bar{0}$ evaluates to some (necessarily unique) natural number k if $M \rightarrow^* \hat{k}$. A closed term $M : \bar{0}$ converges if it evaluates to some k ; otherwise M diverges.

In addition, any closed term $M : \sigma$ can be seen as denoting an element $\llbracket M \rrbracket \in \mathbf{P}_\sigma$. The fundamental result connecting the operational and denotational semantics is the following (essentially Theorem 3.1 of [Plo77]):

Theorem 3.15 (Adequacy) *For any closed term $M : \bar{0}$, M evaluates to k iff $\llbracket M \rrbracket = k$.*

Intuitively, computations in PCF proceed in a “sequential” manner in the sense that there is a single thread of computation — we never find two disjoint subterms of a term being evaluated in parallel. From a computer science perspective, since almost all (deterministic, uniprocessor) programming languages have this sequential character, PCF therefore held some interest as a “prototypical” programming language for theoretical study.

3.3.1 PCF versus S1–S9

Before surveying the main body of research relating to PCF, we digress briefly to comment on its relationship to Kleene’s S1–S9. It has sometimes been informally remarked that PCF is essentially equivalent to S1–S9, but in our view this idea needs to be treated with great caution, and there is an issue here which has never been clearly explained in the published literature.

Even if we restrict attention to the type structure \mathbf{P} , taking a literal reading of S1–S9 as given in [Kle59b] (or our Definition 2.2) it is not true that the Kleene computable elements of \mathbf{P} are exactly the PCF-definable ones. This is because there is a difference between saying that $\{e\}(\mathbf{x})$ is not defined (by the inductive definition of the ternary relation $\{e\}(\mathbf{x}) = y$) and saying that $\{e\}(\mathbf{x}) = \perp$. Indeed, the fact that in \mathbf{P} the notion of undefinedness has been objectified by the element \perp means that if we wish $\{e\}(\mathbf{x})$ to be \perp then this triple must be explicitly generated by the inductive procedure. Now suppose we attempt to construct an index e for, say, the fixed point operator $Y_{\bar{1}} \in \mathbf{P}_{(\bar{1} \rightarrow \bar{1}) \rightarrow \bar{1}}$. By invoking S9 and appealing to Kleene’s second recursion theorem, we can obtain an index e with the property that $\{e\}(F) \simeq F(\{e\}(F))$ for all $F \in \mathbf{P}_{\bar{1} \rightarrow \bar{1}}$. But now, even if we specialize f to the constant function $\lambda g.\lambda x.0$, we cannot conclude that $\{e\}(F) = 0$, since strictly speaking we cannot assign a denotation to $F(\{e\}(F))$ before $\{e\}(F)$ has been given a denotation! Indeed, it can be shown that although all Kleene computable elements of \mathbf{P} are PCF-definable, the element $Y_{\bar{1}}$ is not Kleene computable under this strict interpretation.

One can overcome this problem by reinterpreting S1–S9 not as the inductive definition of a set of triples (e, \mathbf{x}, y) , but as the recursive definition of a total function $\{-\}(-) : \mathbf{N} \times X(\mathbf{P}) \rightarrow \mathbf{N}_{\perp}$ obtained as a massive simultaneous least fixed point. (Alternatively, one could perhaps recast it as the definition of a relation $\{e\}(\mathbf{x}) \sqsupseteq y$). This, we take it, is generally what people have in mind when referring to Kleene computability over \mathbf{P} , and it *is* true that this more generous notion coincides with PCF-definability.

Even so, it seems questionable whether this latter interpretation is true to the *spirit* of S1–S9 as manifest in Kleene’s papers. As remarked earlier, the schemata S1–S9 were introduced by Kleene to capture the ideal of computing with functions as pure extensions, whose characteristic behaviour is that when presented with a specified object of lower type they simply return an answer (or perhaps diverge). It was a conscious shift in perspective on Kleene’s part to regard computations as acting on more intensional representations of functions, such as formal expressions in [Kle59b] or the oracles described in [Kle63]; and for this purpose Kleene introduced S11. Now the strategy required to compute a functional such as $Y_{\bar{1}}$ makes essential use of the idea that an object F of type $\bar{1} \rightarrow \bar{1}$ can not only provide answers when presented with specified arguments, but also ask questions when presented with unspecified (or incompletely specified) arguments; and if an F can do this, it is behaving as something more than a pure extension. It therefore seems to us more accurate, both in letter and in spirit, to say that PCF essentially corresponds to Kleene’s S1–S8 plus S11 (this is stated explicitly by Nickau in [Nic94]). It is worth remarking that Kleene himself never used S9 in connection with hereditarily partial functionals.

It is not hard to define weaker languages than PCF that do correspond in expressive power to the strict interpretation of S1–S9: for instance, one can take Gödel’s System T extended with the minimization operator μ of ordinary recursion theory. This and related systems have indeed been considered recently by Berger in [Ber00], where some preliminary expressivity results are obtained. We feel that the notion of computability given by this language is philosophically a very natural one and deserves further study — certainly it captures an

intuitively appealing idea of “purely extensional computation”. We will say a little more about this notion of computability in Part II.

3.3.2 The full abstraction problem

Plotkin’s work on PCF marked something of a break between the older recursion theory tradition and what was to become the modern computer science tradition. Roughly speaking, whereas the former usually treated formal languages or inductive schemata principally as ways of picking out a class of computable elements in some predefined mathematical structure such as \mathbf{S} or \mathbf{P} , the tendency in computer science has been to take the programming languages as primary and then look for mathematical structures in which they can be interpreted. (The dichotomy is not hard-and-fast and one can think of many exceptions; nevertheless, some cultural difference of this kind can be clearly discerned in the literature and to some extent persists to this day.) The techniques of operational semantics, which gave tractable ways of giving standalone syntactic definitions of programming languages, are what made the latter approach feasible.

The operational semantics of PCF gives rise to a notion of *observational equivalence* of PCF programs. We say the PCF terms $M, N : \sigma$ are observationally equivalent (and write $M \approx N$) if, for all term contexts $C[-]$ such that $C[M], C[N]$ are closed terms of type $\bar{0}$, $C[M]$ evaluates to k iff $C[N]$ evaluates to k . This is a natural notion from a programming point of view, since $M \approx N$ means that it is always safe to replace the block of code M by the code N without affecting the result of the computation. It is therefore natural to ask whether one can give a mathematical characterization of observational equivalence (in PCF or in other languages).

An important result of Milner [Mil77] shows that two PCF terms are observationally equivalent iff, intuitively, they yield the same functions on terms of lower type:

Theorem 3.16 (Context Lemma) *For closed PCF terms $M, N : \sigma_1 \rightarrow \dots \rightarrow \sigma_r \rightarrow \bar{0}$, we have that $M \approx N$ iff, for all closed $P_1 : \sigma_1, \dots, P_r : \sigma_r$, we have that $MP_1 \dots P_r$ evaluates to k iff $NP_1 \dots P_r$ does.*

The idea that the behaviour of PCF terms can be adequately modelled by mathematical functions of some kind is expressed by saying that PCF is a *purely functional* programming language.

It follows easily from the adequacy theorem that if $\llbracket M \rrbracket = \llbracket N \rrbracket$ in \mathbf{P} then $M \approx N$, but the converse is false. This is because one can find terms $M, N : \sigma \rightarrow \tau$ such that $\llbracket M \rrbracket(x) = \llbracket N \rrbracket(x)$ for all PCF-definable elements $x \in \mathbf{P}_\sigma$, but $\llbracket M \rrbracket(x) \neq \llbracket N \rrbracket(x)$ for some non-definable elements such as parallel or (see citeLCF-considered). This suggested that if one could give a semantic characterization of the “sequential” functionals at all higher types, one would be able to provide a denotational model of PCF that precisely captured observational equivalence (such a model is called *fully abstract*).

The early investigators had in mind a denotational model consisting of CPOs of some kind, though chain-completeness is not an essential requirement.

Plotkin and Milner [Mil77] showed for a large class of potential CPO models of PCF that full abstraction holds iff all the finite elements are PCF-definable; it follows that there is up to isomorphism only one (order-extensional) fully abstract CPO model. The full abstraction problem was thus intimately related to the problem of characterizing the sequential functionals.

Semantic characterizations of sequentiality were obtained easily enough for first order functions $N_{\perp}^r \rightarrow N_{\perp}$. The following definition (by induction on r) was given by Milner:

Definition 3.17 *A monotone function $f : N_{\perp}^r \rightarrow N_{\perp}$ is sequential if either it is constant, or there is some i ($1 \leq i \leq r$) such that for all $x_i \in N_{\perp}$ the function $\lambda x_1 \dots x_{i-1} x_{i+1} \dots x_r. f x_1 \dots x_i \dots x_r : N_{\perp}^{r-1} \rightarrow N_{\perp}$ is sequential.*

Intuitively, f is sequential if at each stage in the computation of $f x_1 \dots x_r$, either f can return an answer or there is a “next argument” x_i which needs to be evaluated in order for the computation to proceed. (Essentially the same ideas were present in Kleene’s definition of unimonotonicity for type 2 oracles — see Section 3.1.2. A different but equivalent characterization was also obtained independently by Vuillemin [Vui73].) It is easily shown that a *finite* element of $P_{\bar{0}^r \rightarrow \bar{0}}$ is sequential iff it is PCF-definable, so that a model for PCF that included only sequential functions at first order types would be fully abstract for types of level 2. However, a counterexample due to Trakhtenbrot ([Tra75]; see also [Saz76a]) shows that not every *effective* sequential element of $P_{\bar{0}^r \rightarrow \bar{0}}$ is PCF-definable — it can happen that at some stage some appropriate choice of index i exists but there is no effective way to compute it.

The problem of characterizing “higher order” sequentiality turned out to be much more difficult, and was the focus of a great deal of research effort in theoretical computer science. Milner [Mil77] and later Mulmuley [Mul87] gave constructions of the fully abstract CPO model, but both of these referred in some way to the operational semantics of PCF and so were felt not to qualify as an independent mathematical characterization of sequentiality. Several other models for PCF were constructed: Berry’s *stable* and *bistable* models [Ber78b]; the Berry-Curien *sequential algorithms* model [BC82, Cur93] based on Kahn and Plotkin’s notion of *concrete data structure* [KP93]; and the Bucciarelli-Ehrhard *strongly stable* model [BE91b, Ehr93]. (For a detailed survey of this material and further references, we recommend [Ong95]). This line of research gave much insight into the difficulty of the full abstraction problem, and generated many counterexamples illustrating the subtlety of the notion of observational equivalence. We will not describe the above models in detail here, since as far as PCF is concerned the main point about them is that they are *not* fully abstract. However, the sequential algorithms model and the strongly stable model turned out to be important in connection with other notions of computability, and we will return to them below.

Given that the only known constructions of a fully abstract model were felt to be unsatisfactory, it was natural to ask what exactly were the criteria for a “good” solution to the full abstraction problem. In [JS93] Jung and Stoughton proposed one possible criterion: that when restricted to *finitary* PCF (that is, the fragment of PCF generated by the ground type of booleans), the model

construction should be *effective*. In other words, from a simple type σ over the booleans it should be possible to effectively compute a complete description of the (finite) semantic object representing σ . This criterion rules out syntactic constructions such as Milner’s; it admits the other models mentioned above although these are not fully abstract.

If an extensional fully abstract model satisfying the Jung-Stoughton criterion existed, it would follow that observational equivalence for finitary PCF was decidable. However, the latter statement was refuted by Loader [Loa96] using a delicate encoding of semi-Thue problems. Loader’s result represents one of the most important advances in our understanding of PCF: it closes the door on a large class of attempts at the full abstraction problem, and indeed shows that it was not possible to give any good finitary analysis of PCF sequentiality considered purely as a property of functions.

3.3.3 Intensional semantics for PCF

Another possible approach is to seek a good mathematical description of the *algorithms* embodied by PCF terms rather than of the functions they compute. This approach does in fact lead to good models of sequential computation, albeit of an “intensional” nature. One can regard such models as occupying a kind of middle ground between operational and denotational semantics as traditionally conceived: elements of the model are typically computation strategies with a dynamic operational behaviour, but many of the inessential details typically present in an operational definition of a language are abstracted out.

An approach of this kind was first successfully carried through by Sazonov [Saz75, Saz76b, Saz76c], whose work was explicitly formulated in terms of Scott’s LCF but was independent of the work of Milner and Plotkin (and indeed remained little known in the West until a brief description of it appeared in [HO00]). In effect, Sazonov gave a model of PCF-style sequential computation in terms of Turing machines with oracles, much in the same spirit as Kleene’s characterization of S1–S9 computations in [Kle62b, Kle62c]. A Turing machine for a function of type \bar{n} communicates with its argument (an oracle of type $\overline{n-1}$) by feeding it with a description of a Turing machine for type $\overline{n-2}$. As in Kleene’s work, computations have the character that functionals of type $\overline{n-k}$ are represented by (codes for) Turing machines when k is even, and by pure oracles when k is odd. At the heart of Sazonov’s work is his notion of the strategy followed by a Turing machine: sequentiality is enforced by a requirement that when a machine invokes an oracle, it must receive an answer before it can continue. Although Sazonov’s formalization is somewhat complicated and rather dependent on the use of explicit codings for Turing machines, this approach succeeds in capturing the notion of PCF computability at all finite types and very closely anticipates many features of later work. (In fact, it could be argued that Sazonov had already accomplished what Kleene was trying to achieve in the papers from [Kle78] onwards!)

A somewhat similar approach was pursued for many years by Gandy and his student Pani, who however concentrated more on the problem of characterizing the PCF-definable elements of P . This approach was apparently influenced

by [Kle78], and emphasized the idea of computations as dialogues between two participants. Gandy’s insights had a significant influence on the computer science community, though unfortunately Gandy never completed a written account of his ideas and their exact form remains unclear.

Within computer science itself, an abstract formulation of the essence of PCF sequentiality in terms of dialogue games was attained around 1993, when three closely related models of PCF were obtained (simultaneously and more or less independently) by Abramsky, Jagadeesan and Malacaria [AJM00], Hyland and Ong [HO00], and Nickau [Nic94]. The formal details can appear rather complicated, but the essential ideas can be easily grasped via an example. Using λ -calculus notation and some other syntactic sugar here for convenience, suppose we are given the PCF terms

$$\begin{aligned} F &\equiv \lambda gh. (g\ 3) + (g(Yh)) && : \bar{1} \rightarrow \bar{1} \rightarrow \bar{0} \\ g &\equiv \lambda x. x + 2 && : \bar{1} \\ h &\equiv \lambda x. 4 && : \bar{1} \end{aligned}$$

and we wish to evaluate Fgh . The computation can be modelled as a dialogue between a Player P , who follows a strategy determined by F , and an Opponent O , whose strategy is determined by g and h . Moves in the game are either questions (?), or answers (natural numbers) which are matched to previous questions:

$$\begin{array}{rcl} & (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow & (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \\ O : & & ? \\ P : & ? & \\ O : & ? & \\ P : & 3 & \\ O : & 5 & \\ P : & ? & \\ O : & ? & \\ P : & & ? \\ O : & & 4 \\ P : & 4 & \\ O : & 6 & \\ P : & & 11 \end{array}$$

(In fact, the order of moves here precisely matches the order of interactions between Turing machines and oracles in Sazonov’s approach.) In the model of [HO00], for instance, two main constraints on strategies are imposed: *well-bracketing* (every answer must match the most recent pending question) and *innocence* (intuitively, the participants must decide on their moves purely on the basis of answers received to previous questions, not on how they were obtained). Another feature of this model is that every move apart from the first one is explicitly *justified* by some previous move: answers are justified by the questions they answer, and questions are justified by earlier questions which open up the appropriate part of the game.

All of the game models mentioned yield definability results for PCF: every recursive strategy is the denotation of some PCF term. In our view, the main

insight offered by the perspective of game semantics is the idea that a *parity* (O or P) may be consistently assigned to the implicit interactions between the subterms of a PCF term. This parity is genuinely extra structure present in the game models — it is not at all present in the raw operational definition of PCF. The game-theoretic analysis seems to us to be deep enough to resolve some interesting and purely syntactic questions about PCF, but results of this nature still need to be worked out in detail.

3.3.4 Other work

We now mention a few miscellaneous topics to conclude our survey of what is known about PCF computability.

A remarkable result of Sieber [Sie92] gives a complete characterization of the PCF-definable finite elements at types of level 2 as those that are invariant under a class of logical relations known as sequentiality relations. The idea of using invariance under logical relations to construct a model was carried much further by O’Hearn and Riecke [OR94] who obtained a fully abstract model in this way; however, their result depends only on general facts about λ -definability and so reveals nothing about sequentiality as such.

Some attention has also been given to other variants of PCF. The version considered above is the original call-by-name one, but one can equally well define a version of PCF with a call-by-value evaluation strategy, which can be naturally interpreted in CBV type structures (the idea essentially appears in [Plo83, Chapter 3]). There is also a third possibility: the *lazy PCF* of [BR89]. The relationships between these languages were investigated by Sieber [Sie90], Riecke [Rie93] and Longley [Lon95, Chapter 6]. The picture that has emerged is that these languages are sufficiently interencodable that, from a denotational perspective at least, they can all be regarded as embodying the same abstract “notion of computability”. We will explain this point of view in more detail in Part II.

Another area of recent interest has been the relationship between PCF computability and the notions of *total* functional considered in Section 2. The general idea here is to ask what total functionals can be computed in PCF, though this question can be made precise in several different ways. For example, we have already seen that \mathbf{C} arises as an extensional collapse of \mathbf{P} ; one can therefore ask which elements of \mathbf{C} are represented by PCF-definable elements of \mathbf{P} . It is not hard to see, for instance, that all Kleene computable elements of \mathbf{C} are PCF-definable in this way. More surprising is the fact that the fan functional Φ mentioned in Section 2.3 is PCF-definable, by means of a clever higher type recursion. This fact appeared in Berger’s thesis [Ber90]; it was also known (independently) by Gandy. The following was conjectured by Cook [Coo90] and Berger [Ber93], and proved by Normann [Nor00a].

Theorem 3.18 *The PCF-definable elements of \mathbf{C} are exactly the elements of \mathbf{RC} .*

Equivalently (in the light of Theorem 3.10) the PCF-definable elements of \mathbf{C} coincide exactly with the PCF^{++} -definable ones. Normann’s proof is highly

non-trivial and is in our view one of the highlights of the subject. Around the same time, Plotkin [Plo97] considered other possible notions of totality in PCF (such as that given by an extensional collapse of the term model for PCF itself) and investigated the differences between the various notions.

Finally, a few papers have been devoted to the degree theory induced by the notion of relative PCF-definability. An element $x \in \mathcal{P}^{eff}$ (for example) is PCF-definable relative to $y \in \mathcal{P}^{eff}$ if there is a PCF-definable element f such that $f(y) = x$; the corresponding equivalence classes are known as *degrees of parallelism*. The lattice of degrees of parallelism was introduced by Sazonov [Saz76a]. Like other lattices of degrees, its structure would appear to be extremely complicated. Sazonov mentioned several examples of distinct degrees and some relationships between them: for instance, the parallel-or and *exists* operations represent incomparable degrees. A few results in a similar spirit appear in [Tra75]. Bucciarelli [Buc95] undertook a somewhat more systematic study of degrees of parallelism for first order functions; this line of investigation was pursued further by Lichtenthaler [Lic96]. Degrees of parallelism can be seen as representing notions of computability intermediate between PCF and \mathcal{P}^{++} ; however, to date none of these intermediate notions have established themselves as being of independent mathematical interest.

3.4 Another notion of sequentiality

Until the mid 1990s, it seemed plausible that the only two respectable notions of partial computable functional were those embodied by PCF and \mathcal{P}^{++} — these were typically referred to as “sequential” and “parallel” computability. However, it has recently emerged that there is another good class of computable functionals which can reasonably be seen as embodying an alternative notion of “sequential” computability, more generous than the PCF one. This class is in some sense the effective counterpart of the *strongly stable* functionals introduced by Bucciarelli and Ehrhard [BE91b], and later studied by Longley [Lon98] as the *sequentially realizable (SR)* functionals.

The basic idea can be given by a simple example. Let M be the set of monotone computable functions $\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$. and consider the function $F : M \rightarrow \mathbb{N}_\perp$ defined by

$$F(g) = \begin{cases} 0 & \text{if } g\perp \in \mathbb{N} \\ 1 & \text{if } g\perp = \perp \text{ but } g(0) \in \mathbb{N} \\ \perp & \text{otherwise.} \end{cases}$$

Intuitively, the function F can be computed via the following strategy: given a function g , feed it the object $0 \in \mathbb{N}_\perp$ (that is, a computation which when run would terminate yielding the value 0), and then watch g closely to see whether it ever “looks at” its argument (that is, whether this computation is in fact run). If g returns a result without looking at its argument, we return 0; if g returns a result having looked at the argument, we return 1.

This is a perfectly effective and intuitively “sequential” way of computing F , as long as g is presented to us in some form about which it is sensible to ask whether it ever looks at its argument. The computation of F thus has to operate

on some kind of algorithm or intensional representation for g rather than on its pure extension, although the *result* $F(g)$ is completely determined by the extension g . As argued in Section 3.3.1, the same is true for PCF computations — the only difference is that here the computation of F manipulates the algorithm for g in a way that is not admitted in PCF. Indeed, it is easy to see that the function F cannot be definable in PCF: if $g_1 = \lambda x.if(x)(0)(0)$ and $g_2 = \lambda x.0$ then $g_1 \sqsubseteq g_2$ in the pointwise order but $F(g_1) \not\sqsubseteq F(g_2)$; hence F does not exist even in \mathbf{P} since it is not (pointwise) monotone.

It may appear puzzling that a non-monotone function can be considered computable in some sense. The explanation hinges on the fact that computations operate on intensional objects (as is the case with many of the other definitions we have seen), and at this level computable operations are indeed monotone. Thus, although $g_1 \sqsubseteq g_2$, the *algorithm* that computes g_2 is not obtained by extending the algorithm that computes g_1 . It is also worth noting that even at the extensional level, functions like F are monotone with respect to a different ordering known as the *stable* order. (Curiously, Kleene came across the possibility of algorithms such as the one described above for F — see [Kle85, §13.3] — but decided to rule them out by imposing an extrinsic monotonicity requirement.)

There is a mathematically natural type structure containing F and “all things like it”, which we shall denote by \mathbf{R} . This first appeared in the literature as the type structure arising from the *strongly stable* model of [BE91b], a domain-theoretic model of PCF intended to capture certain aspects of sequential functionals, and conceived partly as a line of attack on the PCF full abstraction problem. The objects in this model are *dI-domains with coherence*, which are certain CPOs equipped with a class of finite subsets designated as *coherent sets*. The morphisms (equivalently the elements of function spaces) are the *strongly stable functions*, the continuous functions between such CPO which preserve coherent sets and least upper bounds of coherent sets. Though rather complicated to formulate, the construction of this model is as finitary and effective as could be desired, so that (for instance) equality of finite elements is decidable. Ehrhard later gave a simplified presentation of this type structure in the framework of *hypercoherences* [Ehr93]; a slightly more abstract analysis of the relevant structure was given in [BE93].

Another interesting characterization, given by Colson and Ehrhard in [CE94], showed that \mathbf{R} in some sense arises naturally from the class of (infinitary) first order sequential functions $\mathbf{N}_\perp^{\mathbf{N}} \rightarrow \mathbf{N}_\perp$. (For functions of this type, there is evidently only one reasonable notion of sequentiality.) Note the analogy with Definition 3.13.

Definition 3.19 *For each type σ we define a set SR_σ , and a set R_σ^ω of functions from $\mathbf{N}_\perp^{\mathbf{N}}$ to R_σ , as follows:*

- $R_{\bar{0}} = \mathbf{N}_\perp$.
- $R_{\bar{0}}^\omega$ is the set of continuous functions $\mathbf{N}_\perp^{\mathbf{N}} \rightarrow \mathbf{N}_\perp$ that are sequential in e.g. the Milner-Vuillemin sense.

- $R_{\sigma \rightarrow \tau}$ is the set of all functions $f : R_\sigma \rightarrow R_\tau$ such that for all $g \in R_\sigma^\omega$ we have $f \circ g \in R_\tau^\omega$.
- $R_{\sigma \rightarrow \tau}^\omega$ is the set of all functions $f : N_\perp^N \rightarrow R_{\sigma \rightarrow \tau}$ such that for all $g \in R_\sigma^\omega$ the function $\lambda r : N_\perp^N. f(\text{fst } r)(g(\text{snd } r))$ is in R_τ^ω .

From the above results it is still not obvious whether any of the non-PCF-definable elements of R at higher types are sequentially computable in any reasonable sense. However, Ehrhard showed in [Ehr96] that the notion of strong stability admits a computational interpretation: every element of R can be in some sense computed by a Berry-Curien sequential algorithm. This line of investigation was continued in [Ehr97], where Ehrhard showed that R is in fact the extensional collapse of the sequential algorithms model.

A somewhat simpler characterization in the same vein was discovered independently by van Oosten [vO99] and Longley [Lon98], who obtained R as the type structure arising from a certain combinatory algebra \mathcal{B} . The construction of \mathcal{B} hinges on the fact that a sequential algorithm for computing a function $N_\perp^N \rightarrow N_\perp$ can be represented by an infinitely branching decision tree, which can itself be coded by an element of N_\perp^N . Likewise, a sequential algorithm of type $N_\perp^N \rightarrow N_\perp^N$ can be represented by an infinite forest of such trees, which can again be coded by an element of N_\perp^N .

Definition 3.20 (i) Let $\text{play} : N_\perp^N \times N_\perp^N \times \text{Seq}(N) \rightarrow N_\perp$ be the smallest partial function such that, for all $f, g \in N_\perp^N$, $\alpha \in \text{Seq}(N)$ and $n, m \in N$,

- if $f\langle\alpha\rangle = !n$ then $\text{play}(f, g, \alpha) = n$,
- if $f\langle\alpha\rangle = ?n$ and $g(n) = m$ then $\text{play}(f, g, \alpha) = \text{play}(f, g, (\alpha; m))$.

(ii) For $f \in N_\perp^N$ and $n \in N$, write f_n for the least function such that $f_n\langle\alpha\rangle = f\langle n; \alpha\rangle$ for all α . Let $|, \bullet$ be the operations defined by

$$f | g = \text{play}(f, g, \epsilon), \quad f \bullet g = \lambda n. f_n | g$$

and let \mathcal{B} be the applicative structure (N_\perp^N, \bullet) .

The construction of R from \mathcal{B} now follows a standard pattern (see also Section 4 below). Define partial equivalence relations \sim_σ on \mathcal{B} by

$$\begin{aligned} f \sim_{\bar{0}} g &\text{ iff } f(0) \simeq g(0) \text{ (for example),} \\ f \sim_{\sigma \rightarrow \tau} g &\text{ iff } \forall x, y \in \mathcal{B}. x \sim_\sigma y \Rightarrow f \bullet x \sim_\tau g \bullet y \end{aligned}$$

We may then define R by taking $R_\sigma = \mathcal{B} / \sim_\sigma$, with application operations induced by \bullet .

This construction makes it especially clear that all the elements of R can be computed or “realized” by sequential algorithms in some sense. In fact, the relationship between \mathcal{B} and the Berry-Curien sequential algorithms model is very close: the relevant objects in the Berry-Curien category can all be obtained as retracts of \mathcal{B} . This and other results connecting up the known characterizations of R appear in [Lon98].

Longley also explicitly considered the effective analogue R^{eff} (which may be constructed either as a standalone type structure or as a substructure of R), and argued that this embodied a natural and compelling notion of sequential computability at higher types. A major result of [Lon98] was that in both R and R^{eff} the type $\bar{2}$ is *universal* in the same sense in which \mathbb{T}^ω is universal in P and P^{eff} (see Theorem 3.12). A closely related fact is the existence of a type 3 functional $H \in R^{eff}$ such that every element of R^{eff} is PCF-definable relative to H ; indeed, one can define a programming language PCF+H with an effective operational semantics whose term model provides an alternative characterization of R^{eff} . (The operation H and other effective SR functionals can in fact be implemented perfectly well in existing higher order programming languages such as Standard ML — see [Lon99c].)

As we have seen, the functional $F \in R^{eff}$ mentioned above is not present in P^{eff} . On the other hand, the parallel-or function in P^{eff} is not present in R^{eff} . We thus have two incomparable notions of partial computable functional — this is somewhat analogous to the situation for the total type structures RC and HRC as described in Section 2.4. Indeed, one can also make precise an “anti-Church’s thesis for partial computable functionals, to the effect that there is no type structure of computable functionals over N_\perp that subsumes both P^{eff} and R^{eff} . One version of this result was given in [Lon98, Section 11]; a slightly stronger version will be presented in Part II.

The type structures P^{eff} and R^{eff} both “contain” Q^{eff} in some sense (to be made precise in Part II). As a curiosity, however, it is worth noting that there are functionals present in both P^{eff} and R^{eff} that are not present in Q^{eff} (see [Lon98, Section 11.2], or else [BE94, BE91a] where a kind of “intersection” of P and R is constructed).

In conclusion we remark on a few points of comparison between the PCF-sequential and SR functionals. As argued in [Lon98, Section 12], the evidence that R and R^{eff} are canonical mathematical objects is very compelling, and at present we possess a much wider range of *prima facie* independent characterizations for R^{eff} than for Q^{eff} . Closely related to this is the fact that R^{eff} appears to enjoy better structural properties than Q^{eff} : for instance, the fact that the finitary version of R is decidable (as witnessed by the strongly stable model), and the existence of a universal type in R, R^{eff} . However, one difficulty with R^{eff} from a practical point of view is that the universal functional H has a high inherent computational complexity (see [Roy00]); this makes it unlikely that R^{eff} will become the staple notion of sequentially computable functional employed by higher order programming languages.

4 Type structures in realizability models

We end our discussion of computable functionals by surveying some ideas from the study of *realizability models* that cross-cut several of the topics we have mentioned. We start with the following standard notion:

Definition 4.1 *A partial combinatory algebra (PCA) is a set A equipped with a partial “application” operation $\cdot : A \times A \rightarrow A$, in which there exist elements*

$k, s \in A$ such that for all $x, y, z \in A$ we have

$$k \cdot x \cdot y = x, \quad s \cdot x \cdot y \downarrow, \quad s \cdot x \cdot y \cdot z = (x \cdot z) \cdot (y \cdot z).$$

The definition of PCA leads to a rich theory of computational structure: for instance, one can show that in any PCA A one can encode the natural numbers and that all recursive functions are representable by elements of A . We may therefore think of a PCA as an untyped universe of computation in some abstract sense; this is a fruitful point of view since many (though not all) naturally arising PCAs are indeed “effective” in nature. Perhaps the leading example of a PCA is *Kleene’s first model* K_1 , consisting of the natural numbers with Kleene application (see Section 0.4); other interesting examples will be mentioned below.

The crucial definition we shall consider here is the following:

Definition 4.2 (i) Given a PCA (A, \cdot) , a partial equivalence relation (PER) on A is a symmetric, transitive binary relation R on A , i.e. an equivalence relation on a subset of A . We write A/R for the set of equivalence classes for R , and $[a]_R$ for the equivalence class containing a when this exists.

(ii) If R, S are PERs on A , the PER S^R is defined by

$$S^R(a, a') \text{ iff } \forall b, b' \in A. R(b, b') \Rightarrow S(a \cdot b, a' \cdot b').$$

A morphism $f : R \rightarrow S$ is an element of $A/(S^R)$; we say a realizes f if $a \in f$. We write $\mathbf{PER}(A)$ for the category of PERs on A and morphisms between them.

It is easy to see that $\mathbf{PER}(A)$ is a cartesian closed category, with exponentials as suggested by the above definition. In fact, the categories $\mathbf{PER}(A)$ turn out to have an extremely rich structure and to provide a common semantic setting for type theories, constructive logics and programming languages.

Girard originally introduced the category $\mathbf{PER}(K_1)$ in order to give a semantics for his System F of second order polymorphic lambda calculus [Gir72]. Later, $\mathbf{PER}(K_1)$ was identified as an important subcategory of Hyland’s *effective topos* [Hy182], a categorical model for higher order logic based on Kleene’s realizability interpretation of arithmetic [Kle45]. (More generally, $\mathbf{PER}(A)$ arises as a subcategory of the *realizability topos* $\mathbf{RT}(A)$; we will refer loosely to categories such as $\mathbf{PER}(A)$, $\mathbf{RT}(A)$ and their close relatives as *realizability models*.) Here, however, we shall concentrate on the connections with programming languages and computability.

One way of thinking about the above definitions (advocated by Mitchell) is to regard a PCA A as a kind of abstract model for “machine level” computation, and to regard $\mathbf{PER}(A)$ as a category of “datatypes” as in a high level programming language implemented on top of this machine. In the case of K_1 , for instance, this accords closely with what happens inside a computer: a high-level datatype consists of values which must ultimately be somehow represented on the machine by bit sequences (or let us say by natural numbers). Since two machine representations might be indistinguishable from the point of

view of the high-level language, we can think of the datatype as corresponding to a partial equivalence relation on \mathbb{N} . Abstracting from this situation, we can imagine any PCA A as a kind of primitive model of computation, and on this view all morphisms of $\mathbf{PER}(A)$ are “computable” in that they are realized by an element of A .

In any category $\mathbf{PER}(A)$, there is (up to isomorphism) a canonical “datatype of natural numbers” N , arising for instance from Curry’s encoding of the natural numbers in the language of combinatory logic. In addition, for most of the particular PCAs of interest, there is an object in $\mathbf{PER}(A)$ that stands out as being the obvious choice for N_{\perp} . We can therefore obtain type structures over \mathbb{N} and \mathbb{N}_{\perp} by repeated exponentiation in $\mathbf{PER}(A)$. From our point of view, these can be seen as type structures of “computable functionals” naturally arising from the abstract notion of computability embodied by A . In the cases where A is a genuinely “effective” PCA, this will yield type structures of effectively computable functionals in some sense. We therefore have a rich supply of interesting constructions of type structures to consider.

In fact, many of the characterizations of type structures that we have already considered are of precisely this kind. As regards total type structures over \mathbb{N} , for instance, Kreisel’s definition of HEO (see Definition 2.13) is nothing other than the definition of the type structure over N in $\mathbf{PER}(K_1)$. Kleene’s definition of \mathbf{C} via associates (Definition 2.10) is trivially equivalent to the type structure over N in the category of PERs over *Kleene’s second model* K_2 (see [KV65]); similarly, the definition of HRC (Definition 2.15) corresponds to the type structure over N in $\mathbf{PER}(K_{2\text{rec}})$. Scott’s characterization of \mathbf{C} via algebraic lattices is very close to the definition of the type structure over N in the *Scott graph model* $\mathcal{P}\omega$.

Certain other constructions can naturally be viewed as type structures in other kinds of realizability models. For instance, Bezem’s construction of the extensional collapse of HRO (see Theorem 2.17) corresponds to the type structure over N in the category $\mathbf{MPER}(A)$ of *modified PERs* on A , a natural subcategory of the *modified realizability topos* on A . In addition, several possible definitions of RC correspond to type structures in *relative realizability* models of the kind considered by Birkedal *et al* [ABS00]. (All the results alluded to above will be covered in more detail in Part II.)

The general programme of trying to identify the total type structures arising from various PCAs was explicitly articulated by Beeson in [Bee85, Chapter VI], where the cases of $\mathcal{P}\omega$ and $\mathcal{P}\omega_{re}$ (giving rise to \mathbf{C} and HRC respectively) were considered. Similar results for other graph models (such as the Plotkin-Scott-Engeler models) were obtained by Bethke [Bet88]. As one might expect, it would seem that all natural PCAs based solely on a notion of continuous function application give rise to the type structure \mathbf{C} , and their recursive analogues give rise to HRC.

Whereas several of the above constructions of type structures over \mathbb{N} predated the explicit definition of $\mathbf{PER}(A)$, the study of type structures over \mathbb{N}_{\perp} in such categories did not really develop until these categories themselves came to be seen as objects worthy of study. This happened largely through the rise of *synthetic domain theory*, initiated by Scott in the early 1980s. The idea of this

enterprise was to look for models of constructive set theory containing objects which by themselves could serve as domains for denotational semantics; domains would then simply be sets in some constructive universe, and the hope was that this might lead to a simpler version of domain theory than the classical one. An early example was provided by the work of McCarty [McC84], who showed that the category of effectively given information systems (essentially equivalent to effective Scott domains) embeds fully in a model of intuitionistic ZF set theory based on Kleene realizability (a model closely related to $\mathbf{PER}(K_1)$).

Later work by Rosolini [Ros86], Hyland [Hyl90] and many other researchers sought to give an axiomatic account of synthetic domain theory: a set of conditions on a model of constructive set theory (usually a topos) which suffice to ensure that it contains a good category of domains. Here realizability models provided the leading source of motivating examples. (See the relevant section in [FJM⁺96] for a brief survey and further references.) One important component of this research was the theory of *dominances* as a way of talking about “computable partial functions” between objects — the key ideas here are due to Mulry and Rosolini. A dominance is a small piece of extra structure on a category which determines an (abstract) notion of “semidecidable predicate”. Under certain conditions on the category, a dominance gives rise to a *lift* operation $X \mapsto X_\perp$ on objects, analogous to lifting in classical domain theory. We may then identify computable partial functions $X \rightarrow Y$ with morphisms $X \rightarrow Y_\perp$. For example, the natural choice of dominance on $\mathbf{PER}(K_1)$ gives rise to an object N_\perp which may be defined (as a PER) by

$$N_\perp(m, n) \Leftrightarrow m \bullet 0 \simeq n \bullet 0.$$

The morphisms $N \rightarrow N_\perp$ then correspond precisely to the partial recursive functions $\mathbb{N} \rightarrow \mathbb{N}$, as we might have hoped.

In [Lon95], Longley developed more explicitly the idea that different PCAs embody different notions of computability. Longley and Simpson [LS97] developed a version of synthetic domain theory that applied uniformly to a wide range of realizability models, showing that very many PCAs gave rise to models of PCF at least. Longley also considered the problem of identifying the type structures over N_\perp in particular models. The following result appeared in in [Lon95, Chapter 7]:

Theorem 4.3 *The type structure over N_\perp (defined as above) in $\mathbf{PER}(K_1)$ is isomorphic to \mathbf{P}^{eff} .*

This was basically a reformulation of McCarty’s result in the setting of PER models. It is also essentially the same in content as Ershov’s result that the type structure over N_\perp in \mathbf{EN} coincides with \mathbf{P}^{eff} ; indeed, there is an obvious embedding of \mathbf{EN} in $\mathbf{PER}(K_1)$ which preserves existing exponentials. However, the characterization in terms of $\mathbf{PER}(K_1)$ seems simpler insofar as here it is immediate that the required exponentials exist.

Longley drew particular attention to the coincidence between the type structure in $\mathbf{PER}(K_1)$ and the term model for \mathbf{PCF}^{++} . Both of these constructions can be seen as attempts at defining a class of hereditarily computable partial

functionals in a very “pure” way (e.g. the definitions do not involve any auxiliary concepts such as continuity), but they are very different in spirit: In $\mathbf{PER}(K_1)$ we think of computations as taking place at the level of recursive indices, and an index realizes a functional whenever its action on indices for functionals on lower type happens to be extensional. In \mathbf{PCF}^{++} , by contrast, computations take place at a higher “symbolic” level, and extensionality is enforced by the syntax of the language itself. That these constructions lead to the same type structure still appears to the author to be a wonderful and surprising fact.

It was also shown in [Lon95] that the type structure over (an obvious object) N_{\perp} in $\mathbf{PER}(\mathcal{P}\omega)$ [resp. in $\mathbf{PER}(\mathcal{P}\omega_{re})$] coincided with \mathbf{P} [resp. \mathbf{P}^{eff}]. This is not surprising in view of the close connections between Scott domains and algebraic lattices. The categories $\mathbf{PER}(\mathcal{P}\omega)$, $\mathbf{PER}(\mathcal{P}\omega_{re})$ also play a central role in recent work of Scott and his colleagues [BBS98], who adopt the perspective that PERs on $\mathcal{P}\omega$ are essentially equivalent to countably based T_0 spaces equipped with an equivalence relation.

Another interesting family of PCAs are the term models for untyped λ -calculi. For instance, let Λ^0/T be the set of closed untyped λ -terms modulo some reasonable theory T , regarded as a combinatory algebra. The following was conjectured implicitly (for a particular T) by Phoa [Pho91], and more explicitly (for a large class of theories T) by Longley [Lon95, Section 7.4], who obtained some partial results and outlined a possible proof strategy:

Conjecture 4.4 *The type structure over (an obvious choice of object) N_{\perp} in $\mathbf{PER}(\Lambda^0/T)$ is isomorphic to \mathbf{Q}^{eff} ; in other words, all functionals in this type structure are PCF-definable.*

The evidence for this conjecture seems to us fairly compelling, but it has resisted extensive attempts at proof (by the author and others). Moreover, even if proved, it is doubtful whether this result would give any useful information about the type structure \mathbf{Q}^{eff} , since the relevant objects in $\mathbf{PER}(\Lambda^0/T)$ appear to be intractable. The main interest in the conjecture is perhaps conceptual: it would provide an example of a highly non-trivial equivalence between two characterizations of \mathbf{Q}^{eff} , which in turn would provide evidence for the mathematical naturalness of this type structure. A interesting variant of the Longley-Phoa conjecture for a lambda calculus extended with certain constants has been established by Streicher *et al* [MRS99], though this is a much more straightforward result.

A more semantic example of a PCA which does give rise to \mathbf{Q}^{eff} was constructed by Abramsky, based on the ideas of games and well-bracketed strategies. The proof that it does so is non-trivial and to some extent furnishes the same kind of evidence for the status of \mathbf{Q} as would be provided by the Longley-Phoa conjecture; see [AL00].

Meanwhile, van Oosten and Longley constructed the PCA \mathcal{B} described in Section 3.4; the type structure described there is exactly the type structure over the obvious object N_{\perp} in $\mathbf{PER}(\mathcal{B})$. It was this characterization, arising from the theory of realizability models, that led to the systematic study of the SR functionals in [Lon98].

Various attempts have been made to generalize the construction of realizability models from untyped to typed structures. One seemingly natural way to do this was proposed by the present author in [Lon99b]; we will make considerable use of this perspective in Parts II and III as a means of unifying much of the existing material. This generalization leads to many new ways of constructing type structures, though in most cases these turn out to be isomorphic to one of the type structures mentioned above.

In conclusion, it appears that in almost all known “natural” examples of realizability models, the type structure over N is isomorphic to one of \mathbf{C} , \mathbf{RC} and \mathbf{HRC} ; while the type structure over (a natural choice of) N_{\perp} is isomorphic to one of \mathbf{P} , \mathbf{Q} , \mathbf{R} or their recursive analogues. This is consistent with the overall impression gained from the material in Sections 2 and 3 — namely, that a wide range of approaches to defining plausible notions of higher type computable functional actually leads to a relatively small and manageable handful of type structures. It would be interesting to know whether the Kleene computable functionals over either \mathbf{S} or \mathbf{P} (in the strict sense described in Section 3.3.1) can in any sense be obtained via a realizability construction.

5 Non-functional notions of computability.

Thus far we have concentrated almost entirely on extensional notions of computability — that is, on notions of computable *functional*. One can also ask whether there are reasonable non-extensional notions of “computable operation” at higher types. Such notions have received relatively little attention by comparison with the extensional notions — perhaps because the very idea of an “intensional operation” seems rather hazy and it is unclear *a priori* whether it is amenable to a precise mathematical formulation. We here briefly survey what is known in relation to this problem.

We have seen how notions of computable functional may be naturally embodied by extensional type structures (or substructures thereof). As a first attempt, therefore, we might propose that more general notions of computable operation could be identified simply with type structures without the extensionality requirement. A typical example would be the structure \mathbf{HRO} of Definition 2.14. Many other examples arise from (non-well-pointed) cartesian closed categories: given any object X corresponding to N or N_{\perp} , interpret the simple types by repeated exponentiation and then apply the global elements functor $\mathbf{Hom}(1, -)$. This view seems somewhat unsatisfactory in that it is too concrete: for instance, different Gödel-numbering schemes can give rise to non-isomorphic variants of \mathbf{HRO} , whereas we would presumably wish to consider all these variants as embodying essentially the same “notion of computability”. This particular problem may be addressed by adopting the more refined point of view outlined in Section 5.3 below; in the meantime, however, we may at least collect examples of non-extensional type structures that might embody plausible notions of computable operation.

5.1 Structures over \mathbf{N}

Non-extensional type structures over \mathbf{N} were first systematically studied by Troelstra [Tro73], who exploited them for metamathematical purposes: any such type structure containing suitable basic operations (essentially those of System T) can serve as a model for higher order arithmetic without the extensionality axiom. Two of the type structures considered by Troelstra are of particular interest from our point of view: the structure **HRO** of hereditarily recursive operations, and a type structure **ICF** of *intensional continuous functionals*.

Many of the relevant facts about **HRO** have already been described in Section 2.4. A trivial example of a non-extensional “computable operation” present in **HRO** is the element $G \in \mathbf{HRO}_{\bar{2}}$ which given an operation $f \in \mathbf{HRO}_{\bar{1}}$ returns a Kleene index for f . As a less trivial example, there is a *local modulus of continuity* operation $M \in \mathbf{HRO}_{\bar{2} \rightarrow \bar{1} \rightarrow \bar{0}}$ with the following property: if $F \in \mathbf{HRO}_{\bar{2}}$ represents an extensional operation (that is, $Ff = Ff'$ whenever $fn = f'n$ for all n) and $gn = g'n$ for all $n < MFg$, then $Fg = Fg'$. The existence of such a recursive operation is implicit in the original proof of the Kreisel-Lacombe-Shoenfield theorem [KLS59]. By way of contrast, it is easily shown that no *extensional* local modulus of continuity operation is computable: that is, there is no element $M \in \mathbf{HEO}$ with the above property. Observations such as these can be used to obtain a variety of consistency and independence results for theories of higher order arithmetic.

The type structure **ICF** is the intensional counterpart of **C** (defined as in Definition 2.10) in the way that **HRO** is the intensional counterpart of **C**. That is, it is (essentially) obtained from the PCA K_2 in the way that **HRO** is obtained from K_1 . For pure types, **ICF** may be defined as follows:

- $\mathbf{ICF}_0 = \mathbf{N}$, $\mathbf{ICF}_1 = \mathbf{N}^{\mathbf{N}}$;
- $\mathbf{ICF}_{n+1} = \{\alpha \in \mathbf{N}^{\mathbf{N}} \mid \forall \beta \in \mathbf{ICF}_n. \alpha \mid \beta \downarrow\}$

As with the extensional type structures, one may also consider the recursive analogue of **ICF**, or its recursive substructure. It can be shown that **ICF** contains a local modulus of continuity operation as above, while **RC** does not, and similarly for the recursive variants (see [Tro73, §2.6]).

Troelstra also considered other non-extensional type structures, such as certain term models for System T, though these seem less appealing as candidates for notions of computability.

5.2 Structures over \mathbf{N}_{\perp}

Other non-extensional notions of computability have more recently been considered in computer science, where they arise naturally in connection with typed programming languages containing non-functional features such as exceptions or state. The term models for such programming languages frequently give rise to non-well-pointed CCCs, and hence to non-extensional type structures. One can regard such term models as defining notions of computability by themselves; though as in the extensional case, most of the interest lies in trying to

provide other, more semantic characterizations of these type structures. Since the number of programming languages that have been considered in the computer science literature is very large, and for most of them little of interest is known from the point of view of computability, we will confine our attention here to languages for which some alternative characterization of the implicit notion of computability has been obtained.

One intensional notion of computability that has emerged as having good mathematical credentials is that embodied by the language $\text{PCF}+\text{catch}$ studied by Cartwright, Curien and Felleisen [CF92, CCF94], as well as by PCF with (first order) callcc [KCF93] and by μPCF [OS97]. All these languages are equivalent in the sense that their fully abstract term models (closed terms modulo observational equivalence) are isomorphic at the finite types. Here we will consider $\text{PCF} + \text{catch}$ as a representative of these languages:

Definition 5.1 *Let $\text{PCF}+\text{catch}$ be obtained by adding to the definition of PCF a constant $\text{catch}_r : (\overline{0}^r \rightarrow \overline{0}) \rightarrow \overline{0}$ for each $r > 0$, together with the reduction rules:*

- *If $Mx_0 \dots x_{r-1} \rightarrow^* E[x_i]$ where the $x_i : \overline{0}$ are fresh variables and $E[-]$ is an evaluation context (intuitively, if we cannot proceed further with the reduction without knowing x_i), then $\text{catch}_r M \rightarrow \widehat{i}$.*
- *If $Mx_0 \dots x_{r-1} \rightarrow^* \widehat{k}$, where the $x_i : \overline{0}$ are fresh variables (intuitively, if we can complete the computation without knowing any of the x_i), then $\text{catch}_r M \rightarrow r + k$.*

Informally, $\text{catch}_r M$ evaluates $M f$ and watches to see if f ever has to look at one of its arguments; if so, the computation is aborted and the index for the argument looked at is returned. It is easy to see how the functional F of Section 3.4 may be implemented in $\text{PCF}+\text{catch}$. Unlike F , however, the catch operators give rise to non-functional behaviour: for instance, we have

$$\text{catch}_2(\lambda xy. x + y) \rightarrow^* 0, \quad \text{catch}_2(\lambda xy. y + x) \rightarrow^* 1.$$

The fact that several proposed languages turn out to have the same expressivity as $\text{PCF}+\text{catch}$ is already encouraging, but more significant is the following semantic characterization due to Cartwright, Curien and Felleisen [CCF94]:

Theorem 5.2 *The fully abstract term model for $\text{PCF}+\text{catch}$ is isomorphic to the type structure over \mathbb{N}_\perp arising from the category of effective sequential algorithms (that is, the effective analogue of the Berry-Curien model [BC82]).*

The original definition of the sequential algorithms model is rather heavy and we will not give it here. However, Longley showed that the relevant objects in this model arise naturally as retracts of the van Oosten algebra \mathcal{B} . In particular, one can form the Karoubi envelope of the λ -algebra \mathcal{B}_{eff} ; the type structure over \mathbb{N}_\perp in this category then coincides exactly with the type structure of Theorem 5.2.

Since then, the ideas of game semantics as developed by Abramsky *et al* have been successful in providing a semantic account of the expressivity of a number

of programming languages. In [AM99], for instance, it is shown that by imposing or not imposing the well-bracketing and innocence conditions on strategies (see Section 3.3.3) one obtains a square of four categories of games corresponding to different computational paradigms, of which only one corresponds to a functional notion of computation (namely the model with both well-bracketing and innocence constraints, which corresponds to PCF). The model with innocence but not well-bracketing (studied in detail by Laird [Lai98, Lai97]) turns out to correspond to PCF+catch: more precisely, the fully abstract term model for PCF+catch is an algebraic quotient of the type structure arising from this model. The models without innocence correspond to notions of computation involving memory or state; a full abstraction result for Idealized Algol (with respect to the non-innocent, well-bracketed game model) is obtained in [AHM98]. Very recently, the present author has found some alternative constructions of the non-innocent, non-well-bracketed model, including a programming language characterization of this notion of computability.

In general, it would appear that categories of games and their correlations with programming languages provide a fruitful source of candidates for mathematically natural notions of non-extensional computability, and we expect more developments in this vein in the future.

5.3 A realizability perspective

We have concentrated here on non-extensional type structures as a way to capture notions of computable operation. A slightly more subtle perspective, making use of ideas from realizability, was outlined in [Lon99a, Lon99b]. The idea here is that given a realizability interpretation for a logic (say predicate logic for the simple types over \mathbb{N} or \mathbb{N}_\perp) in which the realizers are drawn from some computational universe A (such as a PCA or a type structure of some kind), the set of realizable sentences gives information about what kinds of operation are computable in A . For example, whether the sentence

$$\forall F : \bar{2}. \forall g : \bar{1}. \exists n : \bar{0}. \forall g' : \bar{1}. (\forall m < n. g(m) = g'(m)) \Rightarrow F(g) = F(g')$$

is realizable in A corresponds to whether local moduli of continuity are computable in A . We might propose, therefore, to *identify* notions of higher type computability with notions of realizability for such a logic. In some sense, this allows to us say what intensional operations are computable without having to commit ourselves to any concrete definition of “computable operation”. This perspective will be further advocated and developed in detail in Part III.

This point of view is very general and leads to a large class of potential notions of computability: for instance, any untyped PCA implicitly embodies a notion of computable operation at higher types. Fortunately, however, the theory also gives us a natural way of saying when two structures A are equivalent from the point of view of computability, and this significantly reduces the number of distinct notions. Even so, there are very many notions of computability competing for our attention, and much territory remains to be explored. It seems unclear as yet whether it is reasonable to hope ultimately for a small

collection of genuinely natural notions, such as we have in the extensional setting, or whether there is in practice an unlimited range of equally reasonable notions.

6 Conclusion and prospectus

In this paper we have tried to trace the various lines of research to date that are relevant to the study of computability at higher types. Although these strands of research have been rather widely scattered across different areas of mathematical logic and computer science, it is our contention that when viewed together, the outline of a coherent subject area may be discerned.

The central conceptual question we are concerned with is “What are the good notions of computability at higher types, if there are any?” Our approach in this paper has been rather empirical: to collect a variety of different attempts at defining such a notion, and see what natural notions emerge. Our main concern has been to chart the history of the ideas that bear on the problem. The development of these ideas has (naturally enough) been rather haphazard, and we have made little attempt here to organize the material beyond what has been necessary to tell a coherent story.

Regarding notions of computable functional, there are some grounds for believing that the emerging picture is probably now reasonably complete: the territory has been fairly thoroughly explored, and as observed at the end of Section 4, almost any reasonable attempt to define a class of computable functionals seems to lead to one of the already known notions. Regarding more general notions of computable operation, a wealth of individual examples are known and many of these can be described within a uniform framework, though the overall picture seems far from complete and it is unclear how many good notions to expect.

Having collected and reviewed all this material, we are in a position to attempt a more systematic treatment. In Part II we will survey the material on computable functionals within a uniform framework, presenting the important notions of computability, their various characterizations, their intrinsic properties, the relationships between different notions, and some philosophical discussion of the conceptual issues. In particular, we will include some results that can be interpreted as showing the impossibility of a “Church’s thesis” for higher type functionals.

In Part III we will develop in more detail the realizability framework mentioned in Section 5.3 and sketched in [Lon99b]. Within this framework we will collect and organize what is known about general notions of computable operation. We will see that many results from recursion theory and computer science can be formulated very simply within this setting. The framework used in Part III in one sense subsumes that of Part II, but the flavour is rather different and we believe both perspectives to be valuable.

Remarks on bibliography

In the following list of references, we have attempted to provide reasonably complete bibliography for the field of higher type computability as delineated by this article. One of our aims in this survey has been to provide an accessible guide to the literature of the subject, and since most of the works listed below are cited somewhere in the text, it is possible to view the entire article as an extended commentary on the bibliography.

We have tried to include all relevant published works, along with any Ph.D. theses and other unpublished works that made important contributions to the subject. Unpublished technical reports whose contributions appeared soon after in the published literature are usually not listed.

References

- [Abe80] O. Aberth. *Computable Analysis*. McGraw-Hill, New York, 1980.
- [ABS00] S. Awodey, L. Birkedal, and D.S. Scott. Local realizability toposes and a modal logic for computability. To appear in *Mathematical Structures in Computer Science.*, 2000.
- [AH74] P. Aczel and P.G. Hinman. Recursion in the superjump. In J.E. Fenstad and P.G. Hinman, editors, *Generalized Recursion Theory*, pages 3–41. North-Holland, 1974.
- [AHM98] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *Proc. 13th Annual Symposium on Logic in Computer Science*. IEEE, 1998.
- [AJM00] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163:409–470, 2000.
- [AL00] S. Abramsky and J.R. Longley. Some combinatory algebras for sequential computation. In preparation, 2000.
- [AM99] S. Abramsky and G. McCusker. Game semantics. In H. Schwichtenberg and U. Berger, editors, *Computational Logic: Proceedings of the 1997 Marktoberdorf Summer School*, pages 1–56. Springer, 1999.
- [Bar84] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.
- [Bar92] H.P. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, volume 2*, pages 117–309. Oxford University Press, 1992.
- [BBar] A. Bauer and L. Birkedal. Continuous functionals of dependent types and equilogical spaces. In *Proc. Computer Science Logic 2000*, To appear.
- [BBS98] A. Bauer, L. Birkedal, and D.S. Scott. Equilogical spaces. To appear in *Theoretical Computer Science.*, 1998.
- [BC82] G. Berry and P.-L. Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20(3), 1982.
- [BE91a] A. Bucciarelli and T. Ehrhard. Extensional embedding of a strongly stable model of PCF. In *Proc. Eighteenth Int. Conf. on Automata, Languages and Programming, Madrid*, volume 510 of *Lecture Notes in Computer Science*, pages 35–46. Springer, 1991.

- [BE91b] A. Bucciarelli and T. Ehrhard. Sequentiality and strong stability. In *Proc. 6th Annual Symposium on Logic in Computer Science*, pages 138–145. IEEE, 1991.
- [BE93] A. Bucciarelli and T. Ehrhard. A theory of sequentiality. *Theoretical Computer Science*, 113:273–291, 1993.
- [BE94] A. Bucciarelli and T. Ehrhard. Sequentiality in an extensional framework. *Information and Computation*, 110:265–296, 1994.
- [Bee85] M. Beeson. *Foundations of Constructive Mathematics*. Springer, 1985.
- [Ber76] J.A. Bergstra. *Computability and Continuity in Finite Types*. PhD thesis, University of Utrecht, 1976.
- [Ber78a] J.A. Bergstra. The continuous functionals and 2E . In J.E. Fenstad, R.O. Gandy, and G.E. Sacks, editors, *Generalized Recursion Theory II*, pages 39–53. North-Holland, 1978.
- [Ber78b] G. Berry. Stable models of typed lambda-calculi. In *Proc. 5th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 62*, pages 72–89. Springer, 1978.
- [Ber90] U. Berger. *Totale Objekte und Mengen in der Bereichstheorie*. PhD thesis, Munich, 1990.
- [Ber93] U. Berger. Total sets and objects in domain theory. *Annals of Pure and Applied Logic*, 60:91–117, 1993.
- [Ber97] U. Berger. Continuous functionals of dependent and transitive types. Technical report, Habilitationsschrift, Ludwig-Maximilians-Universität München, 1997.
- [Ber00] U. Berger. Minimisation vs. recursion on the partial continuous functionals. Draft paper., 2000.
- [Bet88] I. Bethke. *Notes on Partial Combinatory Algebras*. PhD thesis, Universiteit van Amsterdam, 1988.
- [Bez85a] M. Bezem. Isomorphisms between HEO and HRO^E, ECF and ICF^E. *Journal of Symbolic Logic*, 50:359–371, 1985.
- [Bez85b] M. Bezem. Strongly majorizable functionals of finite type: a model for bar recursion containing discontinuous functionals. *Journal of Symbolic Logic*, 50:652–660, 1985.
- [Bez88] M. Bezem. Equivalence of bar recursors in the theory of functionals of finite type. *Archive of Mathematical Logic*, 27:149–160, 1988.
- [Bez89] M. Bezem. Compact and majorizable functionals of finite type. *Journal of Symbolic Logic*, 54:271–280, 1989.
- [BM37] S. Banach and S. Mazur. Sur les fonctions calculables. *Ann. Soc. Pol. de Math.*, 16:223, 1937.
- [BR89] B. Bloom and J.G. Riecke. LCF should be lifted. In *Proc. Conf. Algebraic Methodology and Software Technology*. Dept. Comp. Sci., University of Iowa, 1989.
- [BS91] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ -calculus. In *Proc. 6th Annual Symposium on Logic in Computer Science*, pages 203–211. IEEE, 1991.

- [Buc93a] A. Bucciarelli. Another approach to sequentiality: Kleene’s unimonotone functions. In *Proc. 9th Symp. Mathematical Foundations of Programming Semantics, New Orleans*, volume 802 of *Lecture Notes in Computer Science*, pages 333–358. Springer, 1993.
- [Buc93b] A. Bucciarelli. *Sequential models of PCF: some contributions to the domain-theoretic approach to full abstraction*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1993.
- [Buc95] A. Bucciarelli. Degrees of parallelism in the continuous type hierarchy. In *Proc. 9th Int. Conf. Mathematical Foundations of Programming Semantics*, 1995.
- [BW77] J. Bergstra and S.S. Wainer. The “real” ordinal of the 1-section of a continuous functional (abstract). *Journal of Symbolic Logic*, 42:440, 1977.
- [CCF94] R. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract semantics for observably sequential languages. *Information and Computation*, 111(2):297–401, 1994.
- [CE94] L. Colson and T. Ehrhard. On strong stability and higher-order sequentiality. In *Proc. 9th Annual Symposium on Logic in Computer Science*, pages 103–108. IEEE, 1994.
- [CF92] R. Cartwright and M. Felleisen. Observable sequentiality and full abstraction. In *Proc. 19th POPL*, pages 328–342. ACM Press, 1992.
- [Chu36] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.
- [Cla64] D.A. Clarke. Hierarchies of predicates of finite types. *Mem. Amer. Math. Soc.*, 51, 1964.
- [Coo90] S. Cook. Computability and complexity of higher type functions. In Y. Moschovakis, editor, *Proc. MSRI Workshop on Logic from Computer Science*, pages 51–72. Springer, 1990.
- [Cur93] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Birkhäuser, second edition, 1993.
- [Dav58] M. Davis. *Computability and unsolvability*. McGraw Hill, 1958.
- [Dav59] M. Davis. Computable functionals of arbitrary finite type. In A. Heyting, editor, *Constructivity in Mathematics: proceedings of the colloquium held at Amsterdam, 1957*, pages 281–284. North-Holland, 1959.
- [Dra68] A.G. Dragalin. The computation of primitive recursive terms of finite type, and primitive recursive realization. *Zap. Nauch. Sem. Leningrad, Otdel Mat. Inst. Steklov*, 8:32–45, 1968.
- [Ehr93] T. Ehrhard. Hypercoherences: a strongly stable model of linear logic. *Mathematical Structures in Computer Science*, 3:365–385, 1993.
- [Ehr96] T. Ehrhard. Projecting sequential algorithms on strongly stable functions. *Annals of Pure and Applied Logic*, 77:201–244, 1996.
- [Ehr97] T. Ehrhard. A relative PCF-definability result for strongly stable functions and some corollaries. To appear in *Information and Computation*, 1997.
- [Ers71a] Yu.L. Ershov. Computable numerations of morphisms. *Algebra i Logika*, 10(3):247–308, 1971.

- [Ers71b] Yu.L. Ershov. La théorie des énumérations. In *Actes du Congrès International des Mathématiciens, Nice 1970, Tome 1*, pages 223–227. Gauthier-Villars, Paris, 1971.
- [Ers72] Yu.L. Ershov. Computable functionals of finite type. *Algebra i Logika*, 11(4):203–277, 1972. English translation in *Algebra and Logic*, AMS.
- [Ers73] Yu.L. Ershov. The theory of A-spaces. *Algebra i Logika*, 12:369–416, 1973. English translation in *Algebra and Logic*, AMS.
- [Ers74a] Yu.L. Ershov. Maximal and everywhere defined functionals. *Algebra i Logika*, 13(4):210–255, 1974. English translation in *Algebra and Logic*, AMS.
- [Ers74b] Yu.L. Ershov. On the model G of the theory BR. *Soviet Math. Doklady*, 15(4):1158–1160, 1974.
- [Ers76a] Yu.L. Ershov. Constructions ‘by finite’. In *Proceedings of the Fifth International Congress on Logic, Methodology and Philosophy of Science*, pages 3–9. London, Ontario, 1976.
- [Ers76b] Yu.L. Ershov. Hereditarily effective operations. *Algebra i Logika*, 15(6):642–654, 1976. English translation in *Algebra and Logic*, AMS.
- [Ers77a] Yu.L. Ershov. Model C of the partial continuous functionals. In *Logic Colloquium ’76*, pages 455–467. North-Holland, 1977.
- [Ers77b] Yu.L. Ershov. *The Theory of Enumerations*. Monographs in Mathematical Logic and Foundations of Mathematics. Nauka, Moscow, 1977.
- [Ers99] Yu.L. Ershov. Theory of numberings. In E.R. Griffor, editor, *Handbook of Computability Theory*, pages 473–503. North-Holland, 1999.
- [Esc96] M.H. Escardó. PCF extended with real numbers. *Theoretical Computer Science*, 162:79–115, 1996.
- [Fef77a] S. Feferman. Inductive schemata and recursively continuous functionals. In *Logic Colloquium ’76*, pages 373–392. North-Holland, 1977.
- [Fef77b] S. Feferman. Theories of finite type related to mathematical practice. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 913–971. North-Holland, 1977.
- [Fen78] J.E. Fenstad. On the foundation of general recursion theory: Computations versus inductive definability. In J.E. Fenstad, R.O. Gandy, and G.E. Sacks, editors, *Generalized Recursion Theory II*, pages 99–110. North-Holland, 1978.
- [Fen80] J.E. Fenstad. *General recursion theory*. Perspectives in Mathematical Logic. Springer, 1980.
- [Fit81] M.C. Fitting. *Fundamentals of Generalized Recursion Theory*, volume 105 of *Studies in Logic*. North-Holland, 1981.
- [FJM⁺96] M.P. Fiore, A. Jung, E. Moggi, P. O’Hearn, J. Riecke, G. Rosolini, and I. Stark. Domains and denotational semantics: History, accomplishments and open problems. *Bulletin of the European Association for Theoretical Computer Science*, 59:227–256, 1996.
- [Fri58a] R.M. Friedberg. Four quantifier completeness: a Banach-Mazur functional not uniformly partial recursive. *Bull. Acad. Polon. Sci. Série des sciences mathématiques, astronomiques et physiques*, 6:1–5, 1958.

- [Fri58b] R.M. Friedberg. Un contre-exemple relatif aux fonctionnelles récursives. *Comptes rendus hebdomadaires des séances de l'Académie des Sciences (Paris)*, 247:852–854, 1958.
- [Gan62] R.O. Gandy. Effective operations and recursive functionals (abstract). *Journal of Symbolic Logic*, 27:378–9, 1962.
- [Gan67a] R.O. Gandy. Computable functionals of finite type I. In J.N. Crossley, editor, *Sets, Models and Recursion Theory*, pages 202–242. North-Holland, 1967. Part II never appeared.
- [Gan67b] R.O. Gandy. General recursive functionals of finite type and hierarchies of functionals. *Ann. Fac. Sci. Univ. Clermont-Ferrand*, 35:5–24, 1967.
- [Gan80] R.O. Gandy. Proofs of strong normalization. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.
- [GH77] R.O. Gandy and J.M.E. Hyland. Computable and recursively countable functions of higher type. In *Logic Colloquium '76*, pages 407–438. North-Holland, 1977.
- [Gir] J.-Y. Girard. Proof theory and logical complexity, volume II. To appear.
- [Gir72] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Paris, 1972.
- [Gir87] J.-Y. Girard. *Proof theory and logical complexity, Volume I*. Bibliopolis, 1987.
- [GL84] P. Giannini and G. Longo. Effectively given domains and lambda-calculus models. *Information and Control*, 62:36–63, 1984.
- [Göd31] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.
- [Göd58] K. Gödel. Über eine bisher noch nicht Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958. English translation in [Göd90].
- [Göd72] K. Gödel. On an extension of finitary mathematics which has not yet been used. First published in [Göd90], 1972.
- [Göd90] Kurt Gödel. *Collected Works Volume II*. Oxford University Press, 1990. Edited by S. Feferman *et al.*
- [Gri67] T.J. Grilliot. *Recursive functions of finite higher types*. PhD thesis, Duke University, 1967.
- [Gri69a] T.J. Grilliot. Hierarchies based on objects of finite type. *Journal of Symbolic Logic*, 34:177–182, 1969.
- [Gri69b] T.J. Grilliot. Selection functions for recursive functionals. *Notre Dame Journal of Formal Logic*, X:225–234, 1969.
- [Gri71] T.J. Grilliot. On effectively discontinuous type-2 objects. *Journal of Symbolic Logic*, 36:245–248, 1971.
- [Gri80] E.R. Griffor. *E-recursively Enumerable Degrees*. PhD thesis, MIT, Cambridge, Mass., 1980.
- [Grz55a] A. Grzegorzcyk. Computable functionals. *Fund. Math.*, 42:168–202, 1955.

- [Grz55b] A. Grzegorzcyk. On the definition of computable functionals. *Fund. Math.*, 42:232–239, 1955.
- [Grz64] A. Grzegorzcyk. Recursive objects in all finite types. *Fundamenta Mathematicae*, 54:73–93, 1964.
- [Har73] L.A. Harrington. *Contributions to Recursion Theory in Higher Types*. PhD thesis, MIT, Cambridge, Mass., 1973.
- [Har74] L.A. Harrington. The superjump and the first recursively Mahlo ordinal. In J.E. Fenstad and P.G. Hinman, editors, *Generalized Recursion Theory*, pages 43–52. North-Holland, 1974.
- [Hin66a] Y. Hinatani. Calculabilité des fonctionnels recursives primitives de type fini sur les nombres naturels. *Ann. Japan Assoc. Philos. Sci.*, 3:19–30, 1966.
- [Hin66b] P.G. Hinman. *Ad astra per aspera: hierarchy schemata in recursive function theory*. PhD thesis, University of California, Berkeley, 1966.
- [Hin67] S. Hinata. Calculability of primitive recursive functionals of finite type. *Science Reports of the Tokyo Kyoiku Daigaku, A*, 9:218–235, 1967.
- [Hin69] P.G. Hinman. Hierarchies of effective descriptive set theory. *Trans. Amer. Math. Soc.*, 142:111–140, 1969.
- [Hin73] P.G. Hinman. Degrees of continuous functionals. *Journal of Symbolic Logic*, 38:393–395, 1973.
- [Hin78] P.G. Hinman. *Recursion-Theoretic Hierarchies*. Perspectives in Mathematical Logic. Springer, 1978.
- [HK75] L.A. Harrington and A.S. Kechris. On characterizing Spector classes. *Journal of Symbolic Logic*, 40:19–24, 1975.
- [HM76] L.A. Harrington and D. MacQueen. Selection in abstract recursion theory. *Journal of Symbolic Logic*, 41:153–158, 1976.
- [HO00] J.M.E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III. *Information and Computation*, 163:285–408, 2000.
- [How73] W.A. Howard. Hereditarily majorizable functionals of finite type. In A.S. Troelstra, editor, *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*, pages 454–461. Springer, 1973.
- [HT69] S. Hinata and S. Tugué. A note on continuous functionals. *Annals of the Japan Association for Philosophy of Science*, 3:138–145, 1969.
- [Hyl75] J.M.E. Hyland. *Recursion theory on the countable functionals*. PhD thesis, University of Oxford, 1975.
- [Hyl78] J.M.E. Hyland. The intrinsic recursion theory on the countable or continuous functionals. In J.E. Fenstad, R.O. Gandy, and G.E. Sacks, editors, *Generalized Recursion Theory II*, pages 135–145. North-Holland, 1978.
- [Hyl79] J.M.E. Hyland. Filter spaces and continuous functionals. *Ann. Math. Logic*, 16:101–143, 1979.
- [Hyl82] J.M.E. Hyland. The effective topos. In *The L.E.J. Brouwer Centenary Symposium*. North-Holland, 1982.

- [Hy190] J.M.E. Hyland. First steps in synthetic domain theory. In A. Carboni, M.C. Pedicchio, and G. Rosolini, editors, *Category Theory, Proceedings, Como*, volume 1488 of *Lecture Notes in Mathematics*, pages 131–156. Springer, 1990.
- [IKR01a] R. Irwin, B. Kapron, and J. Royer. On characterizations of the basic feasible functionals, Part I. *Journal of Functional Programming*, 11:117–153, 2001.
- [IKR01b] R. Irwin, B. Kapron, and J. Royer. On characterizations of the basic feasible functionals, Part II. Draft available from <ftp://ftp.cis.syr.edu/users/royer>, 2001.
- [JS93] A. Jung and A. Stoughton. Studying the fully abstract model of PCF within its continuous function model. In *Proc. Int. Conf. Typed Lambda Calculi and Applications, Utrecht*, number 664 in *Lecture Notes in Computer Science*, pages 230–245. Springer, 1993.
- [KCF93] R. Kanneganti, R. Cartwright, and M. Felleisen. SPCF: its model, calculus, and computational power. In *Proc. REX Workshop on Semantics and Concurrency, Lecture Notes in Computer Science 666*, pages 318–347. Springer, 1993.
- [Kec73] A.S. Kechris. The structure of envelopes: A survey of recursion theory in higher types. MIT Logic Seminar notes, 1973.
- [Kie80] D.P. Kierstead. A semantics for Kleene’s j -expressions. In J. Barwise, H.J. Keisler, and K. Kunen, editors, *The Kleene Symposium*, pages 353–366. North-Holland, 1980.
- [Kie83] D.P. Kierstead. Syntax and semantics in higher-type recursion theory. *Trans. Amer. Math. Soc.*, 276:67–105, 1983.
- [Kle36a] S.C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112:727–742, 1936.
- [Kle36b] S.C. Kleene. λ -definability and recursiveness. *Duke Mathematical Journal*, 2:340–353, 1936.
- [Kle45] S.C. Kleene. On the interpretation of intuitionistic number theory. *J. Symb. Logic*, 10, 1945.
- [Kle52] S.C. Kleene. *Introduction to Metamathematics*. Wolter-Noordhoff and North-Holland, 1952.
- [Kle55a] S.C. Kleene. Arithmetical predicates and function quantifiers. *Trans. Amer. Math. Soc.*, 79:312–340, 1955.
- [Kle55b] S.C. Kleene. Hierarchies of number-theoretic predicates. *Bull. Amer. Math. Soc.*, 61:193–213, 1955.
- [Kle59a] S.C. Kleene. Countable functionals. In A. Heyting, editor, *Constructivity in Mathematics: proceedings of the colloquium held at Amsterdam, 1957*, pages 81–100. North-Holland, 1959.
- [Kle59b] S.C. Kleene. Recursive functionals and quantifiers of finite types I. *Trans. Amer. Math. Soc.*, 91:1–52, 1959.
- [Kle61] S.C. Kleene. Herbrand-Gödel-style recursive functionals of finite types. *Proc. Symp. Pure Math.*, 5:49–75, 1961.
- [Kle62a] S.C. Kleene. Lambda-definable functionals of finite types. *Fund. Math.*, 50:281–303, 1962.

- [Kle62b] S.C. Kleene. Turing-machine computable functionals of finite types I. In *Logic, methodology and philosophy of science, Stanford*, pages 38–45, 1962.
- [Kle62c] S.C. Kleene. Turing-machine computable functionals of finite types II. *Proc. London Math. Soc.*, 12:245–258, 1962.
- [Kle63] S.C. Kleene. Recursive functionals and quantifiers of finite types II. *Trans. Amer. Math. Soc.*, 108:106–142, 1963.
- [Kle69] S.C. Kleene. Formalized recursive functionals and formalized realizability. *Mem. Amer. Math. Soc.*, 89, 1969.
- [Kle78] S.C. Kleene. Recursive functionals and quantifiers of finite types revisited I. In J.E. Fenstad, R.O. Gandy, and G.E. Sacks, editors, *Generalized Recursion Theory II*, pages 185–222. North-Holland, 1978.
- [Kle80] S.C. Kleene. Recursive functionals and quantifiers of finite types revisited II. In J. Barwise, H.J. Keisler, and K. Kunen, editors, *The Kleene Symposium*, pages 1–29. North-Holland, 1980.
- [Kle82] S.C. Kleene. Recursive functionals and quantifiers of finite types revisited III. In G. Metakides, editor, *Patras Logic Symposium*, pages 1–40. North-Holland, 1982.
- [Kle85] S.C. Kleene. Unimonotone functions of finite types (recursive functionals and quantifiers of finite types revisited IV). In A. Nerode and R.A. Shore, editors, *Recursion Theory*, volume 42 of *Proc. Symposia in Pure Mathematics*, pages 119–138, 1985.
- [Kle91] S.C. Kleene. Recursive functionals and quantifiers of finite types revisited V. *Trans. Amer. Math. Soc.*, 325:593–630, 1991.
- [KLS57] G. Kreisel, D. Lacombe, and J.R. Shoenfield. Fonctionnelles récursivement définissable et fonctionnelles récursives. *Compt. Rend. Acad. Sci.*, 245:399–402, 1957.
- [KLS59] G. Kreisel, D. Lacombe, and J.R. Shoenfield. Partial recursive functionals and effective operations. In A. Heyting, editor, *Constructivity in Mathematics: proceedings of the colloquium held at Amsterdam, 1957*, pages 101–128. North-Holland, 1959.
- [KM77] A.S. Kechris and Y.N. Moschovakis. Recursion in higher types. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 681–737. North-Holland, 1977.
- [KN97] L. Kristiansen and D. Normann. Total objects in inductively defined types. *Archive of Mathematical Logic*, 36:405–436, 1997.
- [KP93] G. Kahn and G.D. Plotkin. Concrete domains. *Theoretical Computer Science*, 121:187–277, 1993. First appeared in French as INRIA-LABORIA technical report, 1978.
- [Kre59] G. Kreisel. Interpretation of analysis by means of functionals of finite type. In A. Heyting, editor, *Constructivity in Mathematics: proceedings of the colloquium held at Amsterdam, 1957*, pages 101–128. North-Holland, 1959.
- [Kre61] G. Kreisel. Set-theoretic problems suggested by the notion of potential totality. In *Infinitistic Methods, Proc. Symp. Foundations of Mathematics, Warsaw*, page 103. Pergamon, 1961.
- [Kur52] C. Kuratowski. *Topologie Vol. I*. Warsaw, 1952.

- [KV65] S.C. Kleene and R.E. Vesley. *The Foundations of Intuitionistic Mathematics*. North-Holland, 1965.
- [Lac55a] D. Lacombe. Extension de la notion de fonction récursive aux fonctions d'une ou plusieurs variables réelles I. *Comptes Rendus Acad. Sc. Paris*, 240:2478–2480, 1955.
- [Lac55b] D. Lacombe. Extension de la notion de fonction récursive aux fonctions d'une ou plusieurs variables réelles II. *Comptes Rendus Acad. Sc. Paris*, 241:13–14, 1955.
- [Lac55c] D. Lacombe. Extension de la notion de fonction récursive aux fonctions d'une ou plusieurs variables réelles III. *Comptes Rendus Acad. Sc. Paris*, 241:151–153, 1955.
- [Lac55d] D. Lacombe. Remarques sur les opérateurs récursifs et sur les fonctions récursives d'une variable réelle. *Comptes Rendus Acad. Sc. Paris*, 241:1250–1252, 1955.
- [Lai97] J. Laird. Full abstraction for functional languages with control. In *Proc. 12th Annual Symposium on Logic in Computer Science*, pages 58–67. IEEE, 1997.
- [Lai98] J. Laird. *A Semantic Analysis of Control*. PhD thesis, University of Edinburgh, 1998. Examined March 1999.
- [Lic96] B. Lichtenthäler. Degrees of parallelism. Technical Report 96-01, Fachgruppe Informatik, Siegen, 1996.
- [LM84a] G. Longo and E. Moggi. Cartesian closed categories of enumerations for effective type structures, Parts I and II. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 235–255. Springer, 1984.
- [LM84b] G. Longo and E. Moggi. The hereditary partial functionals and recursion theory in higher types. *Journal of Symbolic Logic*, 49:1319–1332, 1984.
- [Loa96] R. Loader. Finitary PCF is not decidable. To appear, 1996.
- [Loa97] R. Loader. Equational theories for inductive types. *Annals of Pure and Applied Logic*, 84:175–217, 1997.
- [Lon95] J.R. Longley. *Realizability Toposes and Language Semantics*. PhD thesis, University of Edinburgh, 1995. Available as ECS-LFCS-95-332.
- [Lon98] J.R. Longley. The sequentially realizable functionals. Technical Report ECS-LFCS-98-402, Department of Computer Science, University of Edinburgh, 1998. To appear in *Annals of Pure and Applied Logic*.
- [Lon99a] J.R. Longley. Matching typed and untyped realizability. In L. Birkedal, J. van Oosten, G. Rosolini, and D.S. Scott, editors, *Proc. Workshop on Realizability, Trento*, 1999. Published as Electronic Notes in Theoretical Computer Science 23 No. 1, Elsevier. Available via <http://www.elsevier.nl/locate/entcs/volume23.html>.
- [Lon99b] J.R. Longley. Unifying typed and untyped realizability. Electronic note, available at <http://www.dcs.ed.ac.uk/home/jr1/unifying.txt>, 1999.
- [Lon99c] J.R. Longley. When is a functional program not a functional program? In *Proc. 4th International Conference on Functional Programming, Paris*, pages 1–7. ACM Press, 1999.

- [Lon01a] J.R. Longley. Notions of computability at higher types II. 50 pages approx, in preparation. Draft available from <http://www.dcs.ed.ac.uk/home/jr1>, 2001.
- [Lon01b] J.R. Longley. Notions of computability at higher types III. 30 pages approx, in preparation. Draft available from <http://www.dcs.ed.ac.uk/home/jr1>, 2001.
- [Low76] F. Lowenthal. Equivalence of some definitions of recursion in a higher type object. *Journal of Symbolic Logic*, 41:427–435, 1976.
- [LP97] J.R. Longley and G.D. Plotkin. Logical full abstraction and PCF. In J. Ginzburg et al., editor, *Tbilisi Symposium on Language, Logic and Computation*, pages 333–352. SILLI/CSLI, 1997.
- [LS97] J.R. Longley and A.K. Simpson. A uniform approach to domain theory in realizability models. *Mathematical Structures in Computer Science*, 7:469–505, 1997.
- [Mac72] D. MacQueen. *Post's problem for Recursion in Higher Types*. PhD thesis, MIT, Cambridge, Mass., 1972.
- [Maz63] S. Mazur. Computable analysis. *Rozprawy Matematyczne*, 33, 1963.
- [McC84] D.C. McCarty. Information systems, continuity and realizability. In E. Clarke and D. Cozen, editors, *Logics of Programs*, number 164 in Lecture Notes in Computer Science, pages 341–359. Springer, 1984.
- [Mil77] R. Milner. Fully abstract models of typed λ -calculi. *Theoretical Computer Science*, 4, 1977.
- [Mog96] E. Moggi. Partial morphisms in categories of effective objects. Details needed, 1996.
- [Mol77] J. Moldestad. *Computations in higher types*, volume 574 of *Lecture Notes in Mathematics*. Springer, 1977.
- [Mos67] Y.N. Moschovakis. Hyperanalytic predicates. *Trans. Amer. Math. Soc.*, 129:249–282, 1967.
- [Mos69] Y.N. Moschovakis. Abstract first order computability I, II. *Trans. Amer. Math. Soc.*, 138:427–464, 465–504, 1969.
- [Mos74a] Y.N. Moschovakis. *Elementary Induction on Abstract Structures*. North-Holland, 1974.
- [Mos74b] Y.N. Moschovakis. On non-monotone inductive definability. *Fund. Math.*, 82:39–83, 1974.
- [Mos74c] Y.N. Moschovakis. Structural characterizations of classes of relations. In J.E. Fenstad and P.G. Hinman, editors, *Generalized Recursion Theory*, pages 53–79. North-Holland, 1974.
- [Mos76] Y.N. Moschovakis. On the basic notions in the theory of induction. In *Proceedings of the Fifth International Congress in Logic, Methodology and Philosophy of Science*. London, Ontario, 1976.
- [Mos80] Y.N. Moschovakis. *Descriptive Set Theory*. North-Holland, 1980.
- [Mos81] Y.N. Moschovakis. On the Grilliot-Harrington-MacQueen theorem. In *Logic Year 79–80*, volume 859 of *Lecture Notes in Mathematics*, pages 246–267. Springer, 1981.

- [MRS99] M. Marz, A. Rohr, and T. Streicher. Full abstraction and universality via realisability. In *Proc. 14th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 1999.
- [MS55] J. Myhill and J.C. Shepherdson. Effective operations on partial recursive functions. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 1:310–317, 1955.
- [Mul82] P.S. Mulry. Generalized Banach-Mazur functionals in the topos of recursive sets. *J. Pure Appl. Algebra*, 26:71–83, 1982.
- [Mul87] K. Mulmuley. *Full abstraction and semantic equivalence*. MIT Press, 1987.
- [Ner57] A. Nerode. General topology and partial recursive functionals. In *Cornell Summ. Inst. Symb. Logic*, pages 247–251. Cornell, 1957.
- [Ner59] A. Nerode. Some Stone spaces and recursion theory. *Duke Mathematical Journal*, 26:397–406, 1959.
- [Nic94] H. Nickau. Hereditarily sequential functionals. In *Proc. 3rd Symposium on Logical Foundations of Computer Science, Lecture Notes in Computer Science 813*, pages 253–264. Springer, 1994.
- [Nig93] K.-H. Niggel. Subrecursive hierarchies on scott domains. *Archive of Mathematical Logic*, 32:239–257, 1993.
- [Nor78a] D. Normann. A continuous functional with noncollapsing hierarchy. *Journal of Symbolic Logic*, 43:487–491, 1978.
- [Nor78b] D. Normann. Set recursion. In J.E. Fenstad, R.O. Gandy, and G.E. Sacks, editors, *Generalized Recursion Theory II*, pages 303–320. North-Holland, 1978.
- [Nor79a] D. Normann. A classification of higher type functionals. In F.V. Jensen, B.H. Mayoh, and K.K.Møller, editors, *Proc. 5th Scandinavian Logic Symposium*, pages 301–308. Aalborg University Press, 1979.
- [Nor79b] D. Normann. Nonobtainable continuous functionals. In *Proc. 6th International Congress on Logic, Methodology and Philosophy of Science*, pages 241–249. Hanover, 1979.
- [Nor80] D. Normann. *Recursion on the countable functionals*, volume 811 of *Lecture Notes in Mathematics*. Springer, 1980.
- [Nor81a] D. Normann. The continuous functionals: computations, recursions and degrees. *Annals of Mathematical Logic*, 21:1–26, 1981.
- [Nor81b] D. Normann. Countable functionals and the projective hierarchy. *Journal of Symbolic Logic*, 46:209–215, 1981.
- [Nor83] D. Normann. Characterising the continuous functionals. *Journal of Symbolic Logic*, 48:965–969, 1983.
- [Nor89] D. Normann. Kleene-spaces. In Ferro et al., editor, *Logic Colloquium '88*, pages 91–109. Elsevier, 1989.
- [Nor97] D. Normann. Closing the gap between the continuous functionals and recursion in ³E. *Archive of Mathematical Logic*, 36:269–287, 1997.
- [Nor98] D. Normann. The continuous functionals of finite types over the reals. To appear., 1998.

- [Nor99] D. Normann. The continuous functionals. In E.R. Griffor, editor, *Handbook of Computability Theory*, pages 251–275. North-Holland, 1999.
- [Nor00a] D. Normann. Computability over the partial continuous functionals. *Journal of Symbolic Logic*, 65:1133–1142, 2000.
- [Nor00b] D. Normann. Exact real number computations relative to hereditarily total functionals. To appear in *Theoretical Computer Science.*, 2000.
- [NW80] D. Normann and S.S. Wainer. The 1-section of a countable functional. *Journal of Symbolic Logic*, 45:549–562, 1980.
- [Odi89] P.G. Odifreddi. *Classical Recursion Theory*, volume 125 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1989. Second edition 1999.
- [Ong95] C.-H. L. Ong. Correspondence between operational and denotational semantics. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 4*, pages 269–356. Oxford University Press, 1995.
- [OR94] P.W. O’Hearn and J.G. Riecke. Kripke logical relations and PCF. Preprint, 1994.
- [OS97] C.-H.L. Ong and C.A. Stewart. A Curry-Howard foundation for functional computation with control. In *Proc. Symposium on Principles of Programming Languages*, pages 215–227. ACM Press, 1997.
- [PE60] M.B. Pour-El. A comparison of five “computable” operators. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6:325–340, 1960.
- [PER89] M.B. Pour-El and J.I. Richards. *Computability in Analysis and Physics*. Springer, 1989.
- [Pét51a] R. Péter. Probleme der Hilbertschen Theorie der höheren Stufen von rekursiven Funktionen. *Acta Math. Acad. Sci. Hungar.*, 2:247–274, 1951.
- [Pét51b] R. Péter. *Rekursive Funktionen*. Akademischer Verlag, Budapest, 1951. English translation published as *Recursive Functions*, Academic Press, 1967.
- [Pho91] W.K.-S. Phoa. From term models to domains. In *Proc. of Theoretical Aspects of Computer Software, Sendai*. Springer LNCS 526, 1991.
- [Pla66] R. Platek. *Foundations of recursion theory*. PhD thesis, Stanford University, 1966.
- [Pla71] R. Platek. A countable hierarchy for the superjump. In R.O. Gandy and C.E.M. Yates, editors, *Logic Colloquium ’69*, pages 257–271. North-Holland, 1971.
- [Plo77] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5, 1977.
- [Plo78] G.D. Plotkin. T^ω as a universal domain. *Journal of Computer and System Sciences*, 17:209–236, 1978.
- [Plo83] G.D. Plotkin. Domains. Technical report, Dept. Comp. Sci., University of Edinburgh, 1983.
- [Plo97] G.D. Plotkin. Full abstraction, totality and PCF. *Mathematical Structures in Computer Science*, 9(1):1–20, 1997.

- [Pos36] E.L. Post. Finite combinatory processes—formulation 1. *Journal of Symbolic Logic*, 1:103–105, 1936.
- [Ric53] H.G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.
- [Ric56] H.G. Rice. On completely recursively enumerable classes and their key arrays. *Journal of Symbolic Logic*, 21:304–308, 1956.
- [Ric67] W. Richter. Constructive transfinite number classes. *Bull. Amer. Math. Soc.*, 73:261–265, 1967.
- [Rie93] J.G. Riecke. Fully abstract translations between functional languages. *Math. Struct. in Comp. Science*, 3:387–415, 1993.
- [Rog67] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.
- [Ros86] G. Rosolini. *Continuity and Effectiveness in Topoi*. PhD thesis, Oxford; Carnegie-Mellon, 1986.
- [Roy00] J.S. Royer. On the computational complexity of Longley’s H functional. Presented at Second International Workshop on Implicit Computational Complexity, UC/Santa Barbara, 2000.
- [Sac71] G.E. Sacks. Recursion in objects of finite type. In *Proc. International Congress of Mathematicians*, pages 251–254. Gauthiers-Villars, Paris, 1971.
- [Sac74] G.E. Sacks. The 1-section of a type n object. In J.E. Fenstad and P.G. Hinman, editors, *Generalized Recursion Theory*, pages 81–96. North-Holland, 1974.
- [Sac76] G.E. Sacks. RE sets higher up. In *Proceedings of the Fifth International Congress on Logic, Methodology and Philosophy of Science*. Ontario, 1976.
- [Sac77] G.E. Sacks. The k -section of a type n object. *Amer. J. Math.*, 99:901–917, 1977.
- [Sac80] G.E. Sacks. Post’s problem, absoluteness and recursion in finite types. In J. Barwise, H.J. Keisler, and K. Kunen, editors, *The Kleene Symposium*, pages 201–222. North-Holland, 1980.
- [Sac85] G.E. Sacks. Post’s problem in E-recursion. *Proc. Symp. Pure Math. AMS*, 42:177–193, 1985.
- [Sac86] G.E. Sacks. On the limits of E-recursive enumerability. *Annals of Pure and Applied Logic*, 31:87–120, 1986.
- [Sac90] G.E. Sacks. *Higher Recursion Theory*. Springer, 1990.
- [Sac99] G.E. Sacks. E-recursion. In E.R. Griffor, editor, *Handbook of Computability Theory*, pages 301–314. Elsevier, 1999.
- [San67] L.E. Sanchis. Functionals defined by recursion. *Notre Dame Journal of Formal Logic*, 8:161–174, 1967.
- [San92] L.E. Sanchis. *Recursive Functionals*, volume 131 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1992.
- [Saz75] V.Yu. Sazonov. Sequentially and parallelly computable functionals. In *Proc. Symp. Mathematical Foundations of Computer Science*, volume 37 of *Lecture Notes in Computer Science*, pages 312–318. Springer, 1975.

- [Saz76a] V.Yu. Sazonov. Degrees of parallelism in computations. In *Mathematical Foundations of Computer Science 1976*, volume 45 of *Lecture Notes in Computer Science*, pages 517–523. Springer, 1976.
- [Saz76b] V.Yu. Sazonov. Expressibility of functions in Scott’s LCF language. *Algebra i Logika*, 15:308–330, 1976.
- [Saz76c] V.Yu. Sazonov. Functionals computable in series and in parallel. *Matematicheskii Zhurnal*, 17:648–672, 1976.
- [Sca71] B. Scarpellini. A model for barrecursion of higher types. *Compositio Mathematica*, 23:123–153, 1971.
- [Sch91] H. Schwichtenberg. Primitive recursion on the partial continuous functionals. In M. Broy, editor, *Informatik und Mathematik*, pages 251–269. Springer, Berlin, 1991.
- [Sch96] H. Schwichtenberg. Density and choice for total continuous functionals. In P. Odifreddi, editor, *Kreiseliana. About and Around Georg Kreisel*, pages 335–362. A.K. Peters, Wellesley, Massachusetts, 1996.
- [Sco69] D.S. Scott. A theory of computable functions of higher type. Unpublished seminar notes, University of Oxford. 7 pages., November 1969.
- [Sco70] D.S. Scott. Outline of a mathematical theory of computation. In *Proc. 4th Annual Princeton Conference on Information Science and Systems*, pages 165–176, 1970.
- [Sco72] D.S. Scott. Continuous lattices. In F.W. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*. Springer, 1972.
- [Sco76] D.S. Scott. Data types as lattices. *SIAM Journal of Computing*, 5(3):522–587, 1976.
- [Sco82] D.S. Scott. Domains for denotational semantics. In M. Nielsen and E.M. Schmidt, editors, *Proc. Ninth International Colloquium on Automata, Languages and Programming, Aarhus, Denmark*, number 140 in *Lecture Notes in Computer Science*, pages 577–610. Springer, 1982.
- [Sco93] D.S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993. First written in 1969 and widely circulated in unpublished form since then.
- [SHLG94] V. Stoltenberg-Hansen, I. Lindström, and E.R. Griffor. *Mathematical Theory of Domains*. Number 22 in *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1994.
- [Sho62] J.R. Shoenfield. The form of the negation of a predicate. In J.C.E. Dekker, editor, *Recursive Function Theory: Proceedings of the Fifth Symposium in Pure Mathematics*, pages 131–134. AMS, 1962.
- [Sho67] J.R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, MA, 1967.
- [Sho68] J.R. Shoenfield. A hierarchy based on a type two object. *Trans. Amer. Math. Soc.*, 134:103–108, 1968.
- [Sie90] K. Sieber. Relating full abstraction results for different programming languages. In *Proc. 10th Conference on Foundations of Software Technology and Theoretical Computer Science, Bangalore*. Springer LNCS 472, 1990.

- [Sie92] K. Sieber. Reasoning about sequential functions. In M.P. Fourman, P.T. Johnstone, and A.M. Pitts, editors, *Applications of Categories in Computer Science, Durham 1991*, volume 177 of *London Mathematical Society Lecture Note Series*, pages 258–269. Cambridge University Press, 1992.
- [Sla81] T.A. Slaman. *Aspects of E-recursion*. PhD thesis, Harvard University, 1981.
- [Sla85] T.A. Slaman. The E-recursively enumerable degrees are dense. *Proc. Symp. Pure Math. AMS*, 42:195–213, 1985.
- [Spe62] C. Spector. Provably recursive functionals of analysis: a consistency proof of analysis by means of principles formulated in current intuitionistic mathematics. In *Recursive Function Theory: Proc. Symp. Pure Math. V*, pages 1–27. AMS, 1962.
- [Sto91a] A. Stoughton. Interdefinability of parallel operations in PCF. *Theoretical Computer Science*, 79, 1991.
- [Sto91b] A. Stoughton. Parallel PCF has a unique extensional model. In *Proc. 6th Annual Symposium on Logic in Computer Science*, pages 146–151. IEEE, 1991.
- [Tai62] W.W. Tait. A second order theory of functionals of higher type. In Stanford report on the foundations of analysis, Stanford University (mimeographed notes), 1962.
- [Tai67] W.W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32:198–212, 1967.
- [Tra75] M.B. Trakhtenbrot. On representation of sequential and parallel functions. In *Proc. 4th Symposium on Mathematical Foundations of Computer Science*, volume 32 of *Lecture Notes in Computer Science*. Springer, 1975.
- [Tro73] A.S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer, 1973.
- [Tug60] T. Tugué. Predicates recursive in a type-2 object and Kleene hierarchies. *Comment. Math. Univ. St. Paul*, 8:97–117, 1960.
- [Tur36] A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc. series 2*, 42:230–265, 1936. A correction, *ibid.*, vol. 42, pp. 455–546, 1937.
- [Tur37] A.M. Turing. Computability and λ -definability. *Journal of Symbolic Logic*, 2:153–163, 1937.
- [Tur39] A.M. Turing. Systems of logic based on ordinals. *Proc. London Math. Soc. series 2*, 45:161–228, 1939.
- [Č59] G.S. Čaitin. Algorithmic operators in constructive complete separable metric spaces. *Dokl. Acad. Nauk*, 128:49–52, 1959.
- [Usp55] V.A. Uspenskii. On enumeration operators. *Dokl. Acad. Nauk*, 103:773–776, 1955.
- [vD95] J.-P. van Draanen. *Models for simply typed lambda-calculi with fixed point combinators and enumerators*. PhD thesis, Catholic University of Nijmegen, 1995.
- [vO99] J. van Oosten. A combinatory algebra for sequential functionals of finite type. In S.B. Cooper and J.K. Truss, editors, *Models and Computability*, pages 389–406. Cambridge University Press, 1999.

- [Vui73] J. Vuillemin. Correct and optimal implementations of recursion in a simple programming language. In *Proc. Fifth ACM Symposium on Theory of Computing*, pages 224–239, 1973.
- [Wai74] S.S. Wainer. A hierarchy for the 1-section of any type two object. *Journal of Symbolic Logic*, 39:88–94, 1974.
- [Wai75] S.S. Wainer. Some hierarchies based on higher type quantification. In H. Rose and T. Shepherdson, editors, *Logic Colloquium '73*, pages 305–316. North-Holland, 1975.
- [Wai78] S.S. Wainer. The 1-section of a non-normal type-2 object. In J.E. Fenstad, R.O. Gandy, and G.E. Sacks, editors, *Generalized Recursion Theory II*, pages 407–417. North-Holland, 1978.
- [Wei00] K. Weihrauch. *Computable Analysis: an introduction*. Texts in Theoretical Computer Science. Springer, 2000.
- [Zas62] I.D. Zaslavskii. Some properties of constructive real numbers and constructive functions. *Trudy Mat. Inst. Steklov*, 67:385–457, 1962. Translated in AMS Translations (2) 57 (1966), 1–84.
- [ZC62] I.D. Zaslavskii and G.S. Ceitin. On singular coverings and properties of constructive functions connected with them. *Trudy Mat. Inst. Steklov*, 67:458–502, 1962. Translated in AMS Translations (2) 98 (1971), 41–89.