

Abstract versus Concrete Computability: The Case of Countable Algebras

J.V. Tucker

*Department of Computer Science,
University of Wales Swansea, Singleton Park, Swansea SA2 8PP, Wales
J.V.Tucker@swansea.ac.uk*

J.I. Zucker*

*Department of Computing and Software,
McMaster University, Hamilton, Ontario L8S 4K1, Canada
zucker@mcmaster.ca*

Abstract

We consider the distinction between abstract computability, in which computation is independent of data representations, and concrete computability, in which computations are dependent on data representations. The distinction is useful for current research in computability theories for continuous data and uncountable structures, including topological algebras and higher types. The distinction is also interesting in the seemingly simple case of discrete data and countable algebras. We give some theorems about equivalences and inequivalences between abstract models (*e.g.*, computation with ‘while’ programs) and concrete models (*e.g.*, computation via numberings) in the countable case.

* The research of the second author was supported by a grant from the Natural Sciences and Engineering Research Council of Canada

1 Introduction

By a *computability theory* we mean a theory of functions and sets that are definable using a model of computation. By a *model of computation* we mean a model of some general method of calculating the value of a function or of deciding, or enumerating, the elements of a set. We allow the functions and sets to be made from any kind of data.

With this terminology, Classical Computability Theory on the set \mathbb{N} of natural numbers is made up of many computability theories. The computable functions and computably enumerable sets are definable by scores of models of computation, based on scores of ideas about machines, programs, algorithms, specifications, rewriting systems, and calculi. It was an important early discovery, in 1936, that different models of computation can be shown to define the same classes of functions and sets. The fact that diverse computability models lead to the same classes of functions and sets on \mathbb{N} is the main pillar supporting the Church-Turing Thesis, which gives the classical theory on \mathbb{N} its extraordinary unity.

Starting in the 1940s, computability theories have been created for other special sets of data, including:

- higher types over the natural numbers,
- ordinals and set hierarchies,
- real numbers and Banach spaces, and
- higher types over real numbers.

More generally, computability theories have been created for axiomatically defined classes of structures, such as

- groups, rings, fields,
- equational and first order theories of universal algebras,
- topological and metric spaces and algebras,
- domains, and
- categories.

Over the years, however, the classification and the proofs of equivalences of models of computation — and, hence, the search for generalised Church-Turing-type Theses and the theoretical unity they represent — have proved much more difficult to achieve for these data types. Cases exist where different models of computation that were of equal conceptual value have been shown not to be equivalent. For example, in higher types, there is Tait's theorem that the fan functional on total functions is recursively continuous but not computable in Kleene's schemes S1-S9 [Tai62, Nor80]. Another more recent example, in computability on real numbers, is the non-equivalence of the models of computability studied in Computable Analysis [PER89, Wei00] and in the theory developed in [BSS89, BCSS98].

There has been much research activity in the above areas in the past decade. For example, Computable Analysis has been greatly extended by mathematicians and computer scientists using *competing* models. Of course, seen from the point of view of Classical Com-

putability Theory, this exciting and rapid growth of Computable Analysis is also rather messy.

In this paper, we seek a general explanation for this phenomenon of non-equivalent computability theories by focussing on the treatment of the data in models of computation. We classify computability theories into two types by introducing the following two concepts.

In an *abstract computability theory* the computations are *independent* of *all* the representations of the data. Computations are *uniform* over all representations and are isomorphism invariants.

In a *concrete computability theory* the computations are *dependent* on *some* representation of the data. Computations are *not* uniform, and different representations can yield different results. Computations are not isomorphism invariants.

Typical of abstract models of computation are models based on abstract ideas of *program*, *equation*, *scheme*, or *logical formula*. Typical of concrete models of computation are those based on concrete ideas of *coding*, *numbering*, or *data representations* using numbers or functions.

Over the years, many people have considered particular problems where abstract and concrete theories can be compared. Whilst the distinction can hardly be new, it is rarely made even in particular cases. One example of the distinction is that of Dag Normann's *internal* (\approx abstract) and *external* (\approx concrete) in higher type computability [Nor82].

For instance, studying examples and analysing algorithms are important in investigating computation on topological algebras. This is usually done using pseudocode, a high-level informal algorithmic language. Typically, the high-level programming and specification languages are designed using abstract models, and low-level implementations are designed using concrete models. It is to be expected there is a need for both abstract and concrete models, and an understanding of their relationship.

Now the distinction of abstract versus concrete is also directly relevant to the seemingly stable and unified classical world of computability on countable algebras. We will consider the distinction in this special case.

First, in Section 2, we describe the basic concepts in more detail and define the ideas of *soundness*, *adequacy* and *completeness* between abstract and concrete models. We point out some examples of the difference between these kinds of computability theory for topological algebras and higher types. Then, in the rest of the paper, we examine the problem of abstract versus concrete for countable algebras. In Section 3 we introduce our concrete models based on codings or numberings of algebras using the natural numbers. In Section 4 we define our abstract models based on 'while' programs equipped with arrays and non-deterministic choice, which may be interpreted over any algebra. In Section 5 we discuss the soundness of the abstract for the concrete, under weak assumptions on countable algebras. In Section 6 we discuss adequacy and completeness and give some simple completeness theorems under stronger assumptions. Finally, we note connections with other work in the countable case (*e.g.*, the Ash-Nerode theorem), and pose some problems.

We wish to thank Ulrich Berger, Dag Normann and Viggo Stoltenberg-Hansen for helpful discussions on the subject of this paper, and an anonymous referee for useful comments.

2 Abstract and Concrete Computability Theories

In this section we elaborate the general ideas given in the Introduction. First, let us make some relevant reflections on Computability Theory.

2.1 Reflections on Computability Theory and Computer Science

Over the past decade or so, a number of surveys and personal reflections on Computability Theory have appeared. There have been the *Handbook of Computability Theory* [Gri99] and the two-volume *Handbook of Recursive Mathematics* [EGNR98], and chapters on the subject in other Handbooks, for example [AGM92]. The reflection [Soa96] proposed a modernisation or, at least, a “make-over” of the subject. The reflection [Fen02] looks back at the distinction between structure and algorithm in generalised computability theory, debated at the Oslo meetings in the 1970s. Some of the technical development of generalised computability theory of that period was influenced by the reflection [Kre71].

Indeed, encouraged by the symbolism of the year 2000, there have been plenty of surveys and reflections on mathematics. Several propose the idea that *algorithms and computations are returning to prominence throughout mathematics*, enriched by the abstract mathematics of the 20th century [Sma98, Gro98] and encouraged by the new ubiquity of mathematics in the contemporary world [Bou01]. The return to prominence of computation in mathematics can be explained by reflecting on the origins of abstraction and structure in 19th century mathematics. But it is, of course, more accurate to say that what helps make the idea sweet enough for many to swallow is the ever growing, all pervasive influence of computing.

The world is a lot more mathematical than it was, thanks to Computer Science.

This is good news for Computability Theory. Computability Theory is a pure mathematical subject that can be said to have *led* the early development of Computer Science. Since its inception, Computability Theory has provided deep conceptual insights, new algorithmic and programming ideas and techniques, and mathematical theories on which to found software technology. For example, in the standard courses on computability one can find the origins of the following *core* ideas of Computer Science: universal computer; recursion; lambda calculus; rewrite systems, formal specification of computations; higher types; decision problems and their classification; data representations; data type implementations; computational complexity; and so on.

From the standpoint of contemporary Computer Science, the mathematical investigations in Computability Theory that led to these fundamental ideas can be seen as brilliant theoretical speculations whose intellectual and practical value is rising daily. Quite simply, Computability Theory is pure mathematics at its best!

Today, computability retains this important speculative role in our quest to understand the big ideas of algorithm, data, specification, program, machine, and, not surprisingly, many computability theorists — including ourselves — are working as computer scientists.

Now, in Computer Science, it is *obvious* that a computation is fundamentally dependent on its data. By a *data type* we mean (i) data, together with (ii) some primitive operations on data. Often we also have in mind the ways these data are (iii) axiomatically specified

and (iv) represented or implemented. To compute, we think hard about what forms of data the user may need, how we might model the data in designing a system — where some high level but formal understanding is important — and how we might implement the data in some favoured programming language.

Seen from Computer Science, we propose that

A computability theory should be focussed equally on the data and the algorithms.

Now this idea may be difficult to appreciate if one works in one of the classical computability theories of the natural numbers, for data representations rarely seem to be an important topic there. Although the translation between Turing machines and register machines involve data transformations, these can be done on an *ad hoc* basis.

However, representations are *always* important. Indeed, representations are a subject in themselves. This is true even in the simple cases of discrete data forming countable sets, from Gödel numberings of syntax to general numberings of countable sets and structures. From the beginning there has been a great interest in the differences and non-equivalences between numberings, for example in Mal'cev's theory of computability on sets and structures [Mal71b]. The study of different notions of reduction and equivalence of numberings, and the space or spectrum of numberings, has had a profound influence on the theory and has led to quite remarkable results, such as Goncharov's Theorem and its descendants [EGNR98, SHT99a]. These notions also include the idea of invariance under computable isomorphisms, prominent in computable algebra, starting in [FS56]. In the general theory of computing with numbered structures that are countable, there was always the possibility of computing relative to a reasonable numbering that was standard in some sense. For example, earlier work on the word problem for groups, such as [MKS76], and on computable rings and fields, such as [Rab60], did not need to worry about numberings in order to develop fine theorems. Indeed, some authors dispensed with explicit numberings altogether and worked with algebras of natural numbers, thus removing what we consider to be the basic notion of representation.

We see the importance of representations even more clearly when computing with continuous data forming uncountable sets. For example, in computing with real numbers, it has long been known that care must be taken over the representations: if one represents the reals by decimal expansions then one cannot even compute multiplication. But if one chooses the Cauchy sequence representations then a great deal is computable [BH02].

Now there is a chasm between computing with discrete data in countable sets and with continuous data in uncountable sets. Today, it is a prominent problem to understand computability on continuous data because of the possible “new” role of computation in mathematics (*e.g.*, algebra and analysis), and because in computer science there are problems in understanding exact computation with real numbers, and high level programming languages for continuous data. Indeed, in physics there are applications to computing by experiment (*e.g.*, quantum computation) and the debate on the existence of non-computable physical systems; see the survey in [BT04].

We need a unification of the many specialised computability theories that can handle arbitrary data and enable us to integrate discrete and continuous data in specifications,

computation and proofs. This could have applications in understanding digital and analogue data in hybrid computing and communication systems; first order and higher order data in specifications, programs and verifications; and discrete and continuous data in simulation, and experiments with physical systems.

2.2 Abstract and Concrete Models for Computing

We present a number of *informal* ideas about computing functions and sets on *any* kind of data.

Definition (Model of computation). A *model of computation* is a mathematical model of some general method of calculating the values of functions, or deciding, or enumerating, the elements of sets. A *computability theory* is a theory of functions and sets that are definable using a model of computation.

In an *abstract model of computation* the idea is to compute strictly within a set D of data using programs based on some primitive operations on D . The choice of operations on the data creates a structure which defines how data is created and tested in programs. Here is an attempt to define this.

Definition (Abstract model of computation).

- (a) An *abstract model of computation* for computing with a set D of data consists of
 - (i) a structure A containing the data D ;
 - (ii) a set $\mathbf{Prog}(\Sigma)$ of programs based on the signature Σ of the structure, naming the primitive functions and relations of A ; and
 - (iii) a semantics

$$\llbracket _ \rrbracket^A: \mathbf{Prog}(\Sigma) \rightarrow \mathbf{Func}(D)$$

that defines partial functions on D .

- (b) A partial function $f: D \rightarrow D$ is *computable* (according to this model) if there exists a program $P \in \mathbf{Prog}(\Sigma)$ that defines it, *i.e.*,

$$f = \llbracket P \rrbracket^A.$$

- (c) A subset S of D is *semicomputable* if it is the domain of a computable partial function.

The intention is that since the programs can use only the functions and predicates of the structure, the programs are at exactly the same level of abstraction as the primitive operations of the structure, as far as the data is concerned. In fact, we expect the programs to be independent of the representations of the data and to generate computations uniformly over all representations: a program $P \in \mathbf{Prog}(\Sigma)$ can be run on different structures of common signature Σ . In an abstract model of computation, we may expect that the programs and computations are invariant under isomorphisms. In symbols:

Invariance Principle Let $\phi: A \rightarrow B$ be a Σ -isomorphism of structures. For all $a \in A$,

$$\phi(\llbracket P \rrbracket^A(a)) = \llbracket P \rrbracket^B(\phi(a)).$$

Thus, an abstract model of computation is “abstract” in exactly the same way that an abstract algebra is “abstract”: it ignores the “nature” of the elements and concentrates on the properties of operations on the elements. This explains the choice of the name *abstract*.

The structures we have in mind include:

- particular single-sorted structures (*e.g.*, strings, natural numbers, integers, reals);
- classes of single sorted structures (*e.g.*, groups, rings and fields);
- particular, and classes of, finitely sorted structures (*e.g.*, hereditarily finite sets and vector spaces, modules, normed spaces, metric spaces);
- particular, and classes, of infinitely sorted structures (*e.g.*, functions of all finite types).

In a *concrete model of computation*, the idea is to implement computations in a set D of data using computations on another “simpler” set R of data, using a “simpler” computability theory on R . Here is an attempt to define this.

Definition (Concrete model of computation). A *concrete model of computation* for computing with a set of data D consists of

- (i) a representation of the data in D by data from another set R via a mapping $\alpha: R \rightarrow D$,
- (ii) a computability theory $\mathbf{Comp}(R)$ on R .

A partial function $f: D^m \rightarrow D$ is *computable with respect to* α if there is a computable function $\varphi \in \mathbf{Comp}(R)$ that tracks f in R in the sense that, for appropriate $r \in R$,

$$f(\alpha(r)) = \alpha(\varphi(r));$$

equivalently, if the following diagram commutes:

$$\begin{array}{ccc} D^m & \xrightarrow{f} & D \\ \alpha^m \uparrow & & \uparrow \alpha \\ R^m & \xrightarrow{\varphi} & R \end{array}$$

In a concrete model the computations are dependent on some choice of data representation α . Computations need not be isomorphism invariant. In fact they need not be uniform, and different representations can yield different results. An important task is to define when one representation reduces, or is equivalent to, another.

A concrete model of computation is “concrete” in exactly the same way as a specific concrete algebra is “concrete”. The term “concrete” is chosen as an opposite of “abstract”.

In most examples, the representation R is constructed from, or can be reduced to, the natural numbers \mathbb{N} and functions and functionals on \mathbb{N} , and the computability theory $\mathbf{Comp}(R)$ is some form of classical computability theory on \mathbb{N} or $\mathbb{N} \rightarrow \mathbb{N}$.

2.3 Soundness, Adequacy and Completeness

In general, what is the relationship between abstract and concrete computability models with a common set of data D ?

Let $\mathbf{AbstComp}_A(D)$ be the set of functions on the data set D that are computable in an abstract model of computation associated with a structure A containing D . Let $\mathbf{ConcComp}_\alpha(D)$ be the set of functions on D that are computable in a concrete model of computation with representation α .

Definition (Soundness). An abstract model of computation $\mathbf{AbstComp}_A(D)$ is *sound* for a concrete model of computation $\mathbf{ConcComp}_\alpha(D)$ if

$$\mathbf{AbstComp}_A(D) \subseteq \mathbf{ConcComp}_\alpha(D).$$

Definition (Adequacy). An abstract model of computation $\mathbf{AbstComp}_A(D)$ is *adequate* for a concrete model of computation $\mathbf{ConcComp}_\alpha(D)$ if

$$\mathbf{ConcComp}_\alpha(D) \subseteq \mathbf{AbstComp}_A(D).$$

Definition (Completeness). An abstract model of computation $\mathbf{AbstComp}_A(D)$ is *complete* for a concrete model of computation $\mathbf{ConcComp}_\alpha(D)$ if it is both sound and adequate, *i.e.*,

$$\mathbf{AbstComp}_A(D) = \mathbf{ConcComp}_\alpha(D).$$

In general, when comparing abstract and concrete models of computation for a common set D of data, we must compare a structure A and a representation α .

In the abstract model, the choice of Σ -structure A and the set of programs $\mathbf{Prog}(\Sigma)$ will determine what is, and is not, computable on D . Change the operations or programming constructs, and the model changes. In the concrete model, the choice of representation $\alpha: R \rightarrow D$, and the computability model $\mathbf{Comp}(R)$ on R , will determine what is, and is not, computable on D .

In comparing them, first we ask if the primitive operations on the structure A of the abstract model are computable using the concrete model. If not then the abstract model cannot be sound for the concrete model. If the answer is yes, then it is sensible to ask about soundness and adequacy. We then meet the following problem. Let $\mathbf{ConcRep}(D)$ be some class of concrete representations of the form $\alpha: R \rightarrow D$.

General Completeness Problem.

$$\text{Is } \mathbf{AbstComp}_A(D) = \bigcap_{\alpha \in \mathbf{ConcRep}(D)} \mathbf{ConcComp}_\alpha(D)?$$

Now, the abstract model relies *exclusively* on these primitive operations of A . However, the representation α allows *all* the properties of the *particular* structure R to be used in defining computations on D . Thus, we may expect that in practice,

if the primitive operations are concretely computable, then computing with a concrete representation R of the data of D enables more functions on these data to be computed than computing with abstract programs based solely on the operations of a structure A containing these data.

For example, when A is some finite or countably infinite abstract structure *without* an ordering and R is \mathbb{N} , then a concrete model of computation is able to use the total ordering of the data representations in \mathbb{N} as the basis of a global search operator, that is then computable in the concrete model but may not be in the abstract model.

One reason for designing an abstract model is to abstract away from the details of data representations and to define computations that are *uniform* over a class of representations, i.e., to design an abstract model that is sound for the concrete model. What about adequacy and completeness? These properties we might expect to be possible in special cases, where we can limit the class of representations, or allow access to low-level operations in the high-level programs. But high-level languages that capture all the algorithms that may be needed are clearly desirable. Designing high-level languages (= abstract models) that can do this is a familiar problem.

2.4 Many-sorted signatures and algebras

We give a short introduction on many-sorted algebras. More details may be found in any of [TZ99, TZ00, TZ04a, TZ04b].

Given a signature Σ with finitely many *sorts* s, \dots and *function symbols*

$$F: s_1 \times \dots \times s_m \rightarrow s, \quad (2.1)$$

a Σ -algebra A consists of a carrier A_s for each Σ -sort s , and a function

$$F^A: A_1 \times \dots \times A_m \rightarrow A_s$$

for each Σ -function symbol as in (2.1). In general, functions F^A are assumed to be partial. Special further possible properties of Σ and the Σ -algebra A are:

- (1) **Standard algebras.** Σ and A are *standard* if they contain the sort `bool` of *booleans* and the corresponding carrier $A_{\text{bool}} = \mathbb{B} = \{\text{t}, \text{f}\}$, together with the standard boolean and boolean-valued operations, including the conditional at all sorts, and equality at certain sorts (“equality sorts”);
- (2) **N-standard algebras.** Σ and A are *N-standard* if they contain the sort `nat` of *natural numbers* and the corresponding carrier $A_{\text{nat}} = \mathbb{N} = \{0, 1, 2, \dots\}$, together with the standard arithmetical operations of zero, successor, equality and order on \mathbb{N} .

In this paper we will always assume (1), *i.e.*,

Assumption 2.4.1 (Standardness assumption). *All signatures Σ and Σ -algebras A are standard.*

We will also usually assume N-standardness, but will state that assumption explicitly.

In any case, any standard signature Σ can be *N-standardised* to a signature Σ^N by adjoining the sort `nat` and the standard arithmetical operations listed in (2) above. Correspondingly, any standard Σ -algebra A can be *N-standardised* to an algebra A^N by adjoining the carrier \mathbb{N} together with the corresponding standard functions.

- (3) **Array algebras.** We also consider *array signatures* Σ^* and *array algebras* A^* , which are formed from *N-standard signatures* Σ and algebras A by adding, for each sort s , an *array sort* s^* , with corresponding carrier A_s^* consisting of all arrays or finite sequences over A_s , together with certain standard array operations, as specified in [TZ00, §2.7] or (in an equivalent but simpler version) in [TZ99, §2.4].

2.5 Computability theories on many-sorted algebras

There are many abstract computability theories on algebras. Some have their beginnings in

- (a) generalisations of computability theory (*e.g.*, prime and search computability of Moschovakis [Mos69a, Mos69b], definability of Montague [Mon68], and the finite algorithmic procedures of Friedman [Fri71] and Shepherdson [She73];
- (b) the theory of programs (*e.g.*, flow charts, program schemes, ‘while’ programs).

A huge number of models of computation and computability theories have been classified, and there many equivalence theorems between disparate models. Most models are based on ideas about programming, but some are based on ideas about definability and specification. The literature has been surveyed in [She85, TZ88, TZ00].

Abstract models of computability theories on algebras have decent mathematical theories, inspired by, but distinct from, the theory of computable functions on the natural numbers. In particular, the abstract case is well understood, and there are stable proposals for generalised Church-Turing Theses [TZ00].

The primary method of modelling concrete representations for computing on algebras is by means of numberings of the form

$$\alpha: \Omega_\alpha \rightarrow A \quad (\Omega_\alpha \subseteq \mathbb{N}).$$

for all, or part, of an algebra A , and in which the operations, or their restrictions, are recursive on codes. If A is countable, α can be taken to be surjective.

The problem of “abstract versus concrete” for countable algebras is examined in the following sections. We will choose the ‘while’ language and its extensions as our abstract model of computation, and find conditions under which

$$\mathbf{WhileCC}^*(A) = \bigcap_{\alpha \in \mathbf{ConcRep}(A)} \mathbf{Comp}_\alpha(A).$$

2.6 Example: Computability theories on topological many-sorted algebras

Consider many-sorted algebras equipped with a topology and continuous operations. Analysis makes heavy use of such algebraic structures, through topological groups and vector spaces, Banach spaces, Hilbert spaces, C^* algebras, and many more. These many-sorted topological algebras specify

- (i) some basic operations;
- (ii) normal forms for the algebraic representation of elements (*e.g.*, using bases);
- (iii) structure-preserving operators (*i.e.*, homomorphisms such as linear operators);
- (iv) approximations, through inner products, norms, metrics and topologies.

We can create abstract computability theories by simply applying the abstract models of §2.4 to these algebras. However, thanks to the method of approximation (iv), we obtain two classes of functions: the *computable* functions and the *computably approximable* functions, defined as follows:

Definition. Let A be an algebra with metric $d: A^2 \rightarrow \mathbb{R}$. A function $f: A \rightarrow A$ is *computably approximable* in an abstract model if there is an abstract program $P \in \mathbf{Prog}(\Sigma)$ such that

$$d(f(a), \llbracket P \rrbracket^A(n, a)) < 2^{-n}$$

for all $a \in A$.

Despite the simplicity of this abstract approach, most computable analysis is done using *concrete models* of computation. In the case of concrete computability, there have been a number of general approaches to the analysis and classification of metric and topological structures:

- *Effective metric spaces* [Cei59, Mos64];
- *Computable sequence structures for Banach spaces* [PER89];
- *Type 2 enumerations* [Wei00];
- *Algebraic domain representations* [SHT88, SHT95];
- *Continuous domain representations* [Eda97];
- *Numbered spaces* [Spr98].

In fact, for certain basic topological algebras, most of these concrete computability theories have been shown to be essentially equivalent [SHT99b]

The comparison of abstract and concrete models of computation for topological algebras is a promising research area.

First, we must look at the primitive operations of the algebras. In the concrete models computable functions are continuous. Thus tests, like equality and order, which are total functions with discontinuities, cannot be primitive operations. If the discontinuities are removed and the functions are made partial then the functions become continuous and may be allowed as operations. For example, the continuous partial form of equality $=_p: A^2 \rightarrow \mathbb{B}$

is defined by

$$(x =_p y) \simeq \begin{cases} \uparrow & \text{if } x = y \\ \text{ff} & \text{otherwise.} \end{cases}$$

Thus, partial algebras play an essential role.

Next, the *computably approximable functions* have been found to be necessary to bridge the gap between abstract and concrete in the case of total functions on the real numbers [TZ99]. Furthermore, it has become clear that multivaluedness is necessary to bridge the abstract and concrete for general classes of topological algebras. In [TZ04a] there is a detailed comparison using our ‘while’-array language with nondeterministic countable choice construct

$$\mathbf{x} := \text{choose } \mathbf{z} : b(\mathbf{z}, \mathbf{x}, \mathbf{y})$$

(see Section 4). Soundness is proved in considerable generality, but completeness demands a number of special hypotheses, though there is no shortage of examples. These equivalence results have been extended to include local coverings of metric algebras in [TZ04b]. We show that for effectively locally uniformly continuous functions, and a wide class of metric algebras, approximation by ‘while’-array programs with countable choice is equivalent to a simple general concrete computational model based on effective metric spaces.

Briefly, let us consider the case of the real numbers. To compute on the set \mathbb{R} of real numbers with an abstract model of computation, we have to choose a structure A in which \mathbb{R} is a carrier set. There are infinitely many choices of operations with which to make an algebra, and so there are infinitely many choices of classes of computable functions. Thanks to the theory of computable functions on many-sorted algebras (§2.4), all these classes of abstractly computable functions on \mathbb{R} will have decent mathematical theories. For example, computability on rings and fields of reals, with and without orderings, forms the basis of abstract computability theories in [BCSS98], which have been extended to complexity theory. A problem with these particular abstract theories is that, when interpreted on \mathbb{R} , equality and order are assumed total, violating continuity (see remarks above). Earlier such theories have been given by Herman and Isard [HI70] and Engeler [Eng68].

In contrast, to compute on the set \mathbb{R} of real numbers with a concrete model of computation, we choose an appropriate concrete representation of the set \mathbb{R} , such as computable Cauchy sequences. The study of the computability of the reals began with [Tur36], but was only later taken up in a systematic way, in [Ric54, Lac55, Grz55, Grz57] for example.

For a special partial metric algebra \mathcal{R}_p^N over \mathbb{R} , with continuous partial operations of equality and order, and a special total metric algebra \mathcal{R}_t^N over \mathbb{R} , with a continuous total division operation of reals by naturals, (both algebras containing the sort of naturals as well as reals), the following can be shown [TZ04b, §4.4]:

Theorem. *Let $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ be a total function that is effectively locally uniformly continuous on a standard open exhaustion of \mathbb{R}^n . Then the following are equivalent:*

- (i) *f is Grzegorzczuk-Lacombe computable on \mathbb{R} ;*
- (ii) *f is effectively trackable on the computable reals \mathbb{R}_C ;*

- (iii) f is effectively locally \mathbb{Q} -polynomial approximable on \mathbb{R} ;
- (iv) f is locally ‘while’ approximable on \mathcal{R}_t^N ;
- (v) f is locally ‘while’-array approximable on \mathcal{R}_t^N ;
- (vi) f is locally ‘while’-array with countable choice approximable on \mathcal{R}_p^N .

There is much to explore on the borderline between abstract and concrete computability: notions of approximation, limit processes, nondeterminism and multivaluedness. There are the important results of Brattka [Bra96, Bra97] that show that by strengthening a fundamental abstract computability model (relations defined by primitive recursion, minimalisation and a limit operation) it is possible to characterise a fundamental concrete computability model (relations defined by type 2 enumerability). The implications of Brattka’s results for other abstract models need further investigation.

2.7 Computability theories on higher types

The computation of functionals on higher types over the natural numbers \mathbb{N} has been analysed in many different abstract and concrete computability theories the classification of which has been, and continues to be, a hugely important undertaking. Higher type computations are technically complicated. One must consider total and partial functionals on total and partial functions as arguments, which may or may not be continuous. See the concise survey [Nor99] for an introduction and the invaluable historical survey [Lon04] for a full description.

Abstract models for computable partial higher type functionals begin to be studied in earnest with the Kleene schemes S1–S9, which were first proposed in [Kle59] for computing on all the total functionals, and later reinterpreted on the total continuous functionals. These schemes have an enumeration scheme (S9) involving algorithms — but not data — and were refined by other equivalent abstract models based on the λ -calculus, in [Pla66] and [Mol77], for the hereditarily monotone functionals. A basic abstract model of computation on higher types is a simply typed λ -calculus with fixed points. An example in computer science is the language PCF and its extensions, which are abstract models of functional programming.

Different concrete models were also introduced by Kleene and Kreisel [Kle59, Kre59] for total functionals with total arguments. Kleene’s *recursively continuous functionals* are the total functionals on total functions that are continuous and have recursive associates. Other concrete models include the *hereditarily recursive continuous functions* and the *hereditarily effective operators*. Ershov proved the latter are equivalent, thus generalising the Kreisel-Lacombe-Schoenfield Theorem to all finite types. However, they are incomparable with the recursively continuous functionals.

The comparison of the early models of higher type computation encountered problems of comparing abstract and concrete models. Up to type 2, Kleene’s S1–S9 computability is complete for his notion of recursively continuous. Indeed, at all finite types, Kleene’s S1–S9 computability is sound for the recursively continuous functionals. However at type 3, we have Tait’s theorem that the fan functional at type 3 is recursively continuous but not computable by Kleene’s S1–S9. Thus, completeness fails at type 3 [Tai62, Nor80].

However the first bridging theorems between abstract and concrete were proved for partial (rather than total) continuous functionals defined on domains. Computable functions on domains are defined to have recursively enumerable sets of approximations [SHLG94]; they generalise the recursively continuous functionals.

In the remarkable study [Plo77] of the language PCF, it was shown that whilst PCF was sound for all the computable partial functionals over the domain of naturals, it was not complete. However, PCF augmented with a parallel conditional and existential operator was complete for the computable partial functionals.

Berger conjectured that if one restricts to total functions on total arguments then the parallel constructs are not needed and PCF is complete w.r.t. the computable total continuous functionals [Ber93]. This striking fact was proved by Normann [Nor00].

Although PCF and S1-S9 are equivalent over partial continuous functionals, this result does not contradict Tait's Theorem, because in that case S1-S9 are interpreted over arguments that are total, or total continuous, functionals (*cf.* scheme S8), whereas (as stated above) the domain semantics of PCF is defined over partial arguments.

Recently, computability in higher types has also been studied over the real numbers. Escardo [Esc96] shows that a language Real PCF, containing parallel extensions, is complete with the computable partial functionals over the interval domain representation of the reals. However, the corresponding version of Berger's Conjecture is open.

Other examples of abstract versus concrete models at higher type can be extracted from [Lon04]. An overview on totality, continuity, computability can be found in [Ber02].

3 Concrete computation: Numbering of algebras

In this section we assume A is a countable \mathbb{N} -standard partial Σ -algebra.

3.1 Numberings

Definition 3.1.1 (Numbering). A *numbering* of A is a family

$$\alpha = \langle \alpha_s : \Omega_{\alpha,s} \rightarrow A_s \mid s \in \mathbf{Sort}(\Sigma) \rangle$$

of surjective maps $\alpha_s : \Omega_{\alpha,s} \rightarrow A_s$, for some family

$$\Omega_{\alpha} = \langle \Omega_{\alpha,s} \mid s \in \mathbf{Sort}(\Sigma) \rangle$$

of sets $\Omega_{\alpha,s} \subseteq \mathbb{N}$, the *code sets* of α ; we say that that A is *numbered* by α . If $u = s_1 \times \cdots \times s_m$, we write $\Omega_{\alpha}^u =_{df} \Omega_{\alpha,s_1} \times \cdots \times \Omega_{\alpha,s_m}$. We assume that $\alpha_{\mathbf{nat}} = \mathbf{id}_{\mathbb{N}}$ and that $\alpha_{\mathbf{bool}} : 0 \mapsto \mathbf{f}$, $n(\neq 0) \mapsto \mathbf{t}$.

Let $\alpha : \Omega \rightarrow A$ be a fixed numbering of A .

Definition 3.1.2 (Kernel). The *kernel* of α is the family of relations

$$\equiv_{\alpha} = \{ \equiv_{\alpha,s} \subseteq \mathbb{N}^2 \mid s \in \mathbf{Sort}(\Sigma) \}$$

where for all Σ -sorts s

$$\equiv_{\alpha,s} = \{(k, l) \in \Omega_{\alpha,s}^2 \mid \alpha_s(k) = \alpha_s(l)\}.$$

Definition 3.1.3 (Tracking function). Let $f: A^u \rightarrow A_s$ and $\varphi: \Omega_\alpha^u \rightarrow \Omega_{\alpha,s}$. Then φ is called an α -tracking function for f if the following diagram commutes:

$$\begin{array}{ccc} A^u & \xrightarrow{f} & A_s \\ \alpha^u \uparrow \cdot & & \cdot \uparrow \alpha_s \\ \mathbb{N}^m & \xrightarrow{\varphi} & \mathbb{N} \end{array}$$

in the sense that for all $k \in \Omega_\alpha^u$

$$f(\alpha^u(k)) \downarrow \implies \varphi(k) \downarrow \in \Omega_{\alpha,s} \wedge f(\alpha^u(k)) = \alpha_s(\varphi(k))$$

and

$$f(\alpha^u(k)) \uparrow \implies \varphi(k) \uparrow.$$

Note that a possible alternative definition of “ α -tracking function” would replace the second clause above by

$$f(\alpha^u(k)) \uparrow \implies \varphi(k) \uparrow \vee \varphi(k) \downarrow \notin \Omega_{\alpha,s}.$$

However the completeness proof for **WhileCC*** computability given in [TZ04a, Thm B] requires this (apparently) stronger definition of α -tracking function.

Note also that nothing is said about the behaviour of φ off Ω_α^u .

3.2 Computable functions and semicomputable sets

Definitions 3.2.1 (α -computability and α -semicomputability).

- (a) A function on A is α -computable if it has a computable (*i.e.*, partial recursive) α -tracking function.
- (b) A relation on A is α -computable if its characteristic function is α -computable.
- (c) A relation on A is α -semicomputable if it is the domain of an α -computable function.

We write $\mathbf{Comp}_\alpha(A)$ for the set of α -computable functions on A , and $\mathbf{SComp}_\alpha(A)$ for the set of α -semicomputable relations on A .

We say that Ω_α is *recursive* (or *r.e.*, or *co-r.e.*) if $\Omega_{\alpha,s}$ is recursive (*etc.*) for each Σ -sort s . There is a similar meaning for \equiv_α being recursive (*etc.*).

Lemma 3.2.2. Let $U \subseteq A^u$. Then U is α -semicomputable \iff

$$(\alpha^u)^{-1}[U] = S \cap \Omega_\alpha^u \quad \text{for some r.e. } S \subseteq \mathbb{N}. \quad (3.1)$$

Proof: (\Rightarrow) Suppose U is α -semicomputable. Then $U = \mathbf{dom}(f)$, where $f: A^u \rightarrow A_s$ is α -computable. Let $\varphi: \Omega_\alpha^u \rightarrow \Omega_{\alpha,s}$ be a computable tracking function for f . Let $S = \mathbf{dom}(\varphi)$. Then S is r.e., and it is easy to check that

$$(\alpha^u)^{-1}[U] = S \cap \Omega_\alpha^u.$$

(\Leftarrow) Suppose (3.1). We must show:

$$U = \mathbf{dom}(f) \quad \text{for some } \alpha\text{-computable } f. \quad (3.2)$$

Define $f: A^u \rightarrow \mathbb{N}$ and $\varphi: \mathbb{N}^m \rightarrow \mathbb{N}$ by

$$f(x) \simeq \begin{cases} 0 & \text{if } x \in U \\ \uparrow & \text{otherwise} \end{cases}$$

and
$$\varphi(k) \simeq \begin{cases} 0 & \text{if } k \in S \\ \uparrow & \text{otherwise.} \end{cases}$$

Then, since S is r.e., φ is computable. Also it is easy to check that φ is an α -tracking function for f , thus proving (3.2). \square

Corollary 3.2.3. *Suppose Ω_α is r.e. Then for $U \subseteq A^u$,*

$$U \text{ is } \alpha\text{-semicomputable} \iff (\alpha^u)^{-1}[U] \text{ is r.e.}$$

3.3 Computability of numberings, code sets and kernels

We are interested in various computability aspects of α , namely α -computability of the Σ -operations, and computability or semicomputability of the *code set* and *kernel*.

Definition 3.3.1 (Σ -effective algebra under α). The partial algebra A is Σ -effective under α , or α - Σ -effective, if for every $F \in \mathbf{Func}(\Sigma)$, F^A is α -computable.

We also describe this by saying that α has computable Σ -operations.

Note that in [TZ04a, TZ04b] the concepts defined in Definitions 3.1.3, 3.2.1 and 3.3.1 were called *strict α -tracking*, *strict α -computability* and *strict Σ -effectivity* of α respectively.

Definition 3.3.2 (Effective algebra under α). The partial algebra A is effective under α , or α -effective, if A is Σ -effective under α , and also Ω_α is r.e.

Definition 3.3.3 (Computable algebra under α). (a) The partial algebra A is *computable* (or *semicomputable*, or *cosemicomputable*) under α , or α -*computable* (or α -*semicomputable*, or α -*cosemicomputable*), if A is α -effective, and also, for all Σ -sorts s there is a recursive (or r.e., or co-r.e.) relation $R \subseteq \mathbb{N}^2$ such that

$$R \cap \Omega_{\alpha,s}^2 = \equiv_{\alpha,s}.$$

(b) A is said to be *computable* (etc.) if it is computable (etc.) under some numbering.

We also say that α is a Σ -*effective* (or *effective*, or *computable*, etc.) numbering of A if A is Σ -effective (etc.) under α .

Let $\Sigma\text{-EffNum}(A)$ and $\text{CompNum}(A)$ be the set of all Σ -effective and computable (respectively) numberings of A .

Note that the following are equivalent:

- (i) α is α -computable by $\mathbf{id}_{\Omega_\alpha}$, which is computable on \mathbb{N} ,
- (ii) Ω_α is r.e.

This is clear from the diagram:

$$\begin{array}{ccc}
 \mathbb{N} & \xrightarrow{\alpha_s} & A_s \\
 \uparrow (\alpha_{\text{nat}} =) \mathbf{id}_{\mathbb{N}} & & \uparrow \alpha_s \\
 \mathbb{N} & \xrightarrow{\mathbf{id}_{\Omega_{\alpha,s}}} & \Omega_{\alpha,s}
 \end{array}$$

Lemma 3.3.4. Suppose Ω_α is r.e., i.e., $\Omega_{\alpha,s}$ is r.e. for all s . For all s , let ν_s be a total recursive function with range $\Omega_{\alpha,s}$. Let $\beta: \mathbb{N} \rightarrow A$ be the numbering of A defined by

$$\beta =_{df} \alpha \circ \nu =_{df} \{ \alpha_s \circ \nu_s \mid s \in \mathbf{Sort}(\Sigma) \}.$$

Then

(a) If a function $f: A^u \rightarrow A_s$ is α -tracked by φ , then it is β -tracked by the function ψ defined as follows:

$$\begin{array}{ccc}
 A^u & \xrightarrow{f} & A_s \\
 \uparrow \alpha^u & & \uparrow \alpha_s \\
 \Omega_\alpha^u & \xrightarrow{\varphi} & \Omega_{\alpha,s} \\
 \uparrow \nu_u & & \uparrow \nu_s \\
 \mathbb{N}^m & \xrightarrow{\psi} & \mathbb{N}
 \end{array}$$

(b) For any Σ -sort s , if

$$\equiv_{\alpha,s} = R \cap \Omega_{\alpha,s}$$

for some computable (or semicomputable, or cosemicomputable) relation R on \mathbb{N} , then $\equiv_{\beta,s}$ is computable (or semicomputable, or cosemicomputable, respectively).

Proof: For (a), the algorithm for ψ is: with input $k \in \mathbb{N}^m$, compute $\varphi(\nu_u(k))$. (If and when this converges, find an l such that $\nu_s(l) = \varphi(\nu_u(k))$ (which exists, by surjectivity of ν on Ω). Output such an l .

Part (b) is proved by noting that $\Omega_{\beta,s} = \mathbb{N}$.

Corollary 3.3.5. *Suppose Ω_α is r.e. Then, with β defined as in Lemma 3.3.4:*

(a) *If A is α -effective, then A is β -effective.*

(b) *If A is computable (or semicomputable, or cosemicomputable) under α , then A is also computable (or semicomputable, or cosemicomputable) under β .*

Remark 3.3.6. Hence if A is α -effective, or α -computable, or α -(co)semicomputable, then we can assume without loss of generality that $\Omega_{\alpha,s} = \mathbb{N}$ for all s .

Further, under this assumption on Ω_α , the definition (3.3.3) of computability (etc.) of an algebra can be simplified to:

A is α -computable (or α -semicomputable, or α -cosemicomputable) iff A is α -effective, and also \equiv_α is recursive (or r.e., or co-r.e.).

Lemma 3.3.7 (Canonical extension of numbering to N-standard and array algebra). *A numbering α of A can be canonically extended to numberings α^N of A^N , and α^* of A^* , such that if A is Σ -effective, effective, computable, or (co-)semicomputable under α , then so are A^N under α^N , and α^* under A^* .*

4 Abstract computation: the *While* language and its extensions

Again, we assume that A is a countable N-standard Σ -algebra.

4.1 The *WhileCC** programming language

The programming language ***WhileCC***(Σ) (‘CC’ for “countable choice”) is an extension of ***While***(Σ) [TZ00, §3] with an extra ‘choose’ rule of term formation. The idea of the semantics for ‘choose’ is to select *all* possible implementations satisfying a given property. The complete description of its syntax and semantics, as well as motivation for it, are given in [TZ04a, TZ04b]. Here we give a brief review.

The language ***WhileCC*** has a ‘choose’ construct in the context of an assignment statement, which has one of three forms:

- (i) $\mathbf{x} := t$ (simultaneous assignment),
- (ii) $\mathbf{x} := \text{choose } \mathbf{z} : b(\mathbf{z}, \dots)$,
- (iii) $\mathbf{x} := \text{choose } \mathbf{z} : P(\mathbf{z}, \dots)$,

where \mathbf{z} is a variable of sort nat, and in (ii) $b(\mathbf{z}, \dots)$ is a *boolean term*, and in (iii) $P(\mathbf{z}, \dots)$ is a *semicomputable predicate* of \mathbf{z} (and other variables), *i.e.*, the halting set of a boolean-valued ***WhileCC*** procedure P with \mathbf{z} among its input variables.

Thus the semantics of ***WhileCC*** is given by a many-valued function.

In [TZ04a] an algebraic operational semantics is given for ***WhileCC***, whereby a ***WhileCC*** procedure $P : u \rightarrow v$ has a meaning in an N-standard Σ -algebra A :

$$P^A : A^u \rightarrow \mathcal{P}_\omega^+(A^v \cup \{\uparrow\})$$

where $\mathcal{P}_\omega^+(X)$ is the set of all countable *non-empty* subsets of X , and ‘ \uparrow ’ represents a divergent computation. This is also written in “multivalued function” notation:

$$P^A: A^u \rightrightarrows^+ A^{v\uparrow}$$

where X^\uparrow denotes $X \cup \{\uparrow\}$.

4.2 *WhileCC** computable functions and semicomputable sets

Definitions 4.2.1. (a) A partial function on A is ***WhileCC****(Σ) computable on A if it is ***WhileCC***(Σ) computable on A^* .

(b) ***WhileCC***(A) is the class of all ***WhileCC***(Σ) computable partial functions on A .

(c) ***WhileCC****(A) is the class of all ***WhileCC****(Σ) computable partial functions on A .

Lemma 4.2.2. *If A is total then*

(a) ***WhileCC***(A) = ***While***(A).

(b) ***WhileCC****(A) = ***While****(A).

Proof: We can implement $\text{choose } z : b(z)$ as $\mu z : b(z)$, since by totality, for each value $n = 0, 1, 2, \dots$ of z , computation of $b(n)$ converges to a value \mathfrak{t} or \mathfrak{f} .

Definitions 4.2.3 (*WhileCC*-semicomputability).

(a) The *halting set* of a ***WhileCC*** procedure $P: u \rightarrow v$ on A is the set

$$\{x \in A^u \mid P^A(x) \setminus \{\uparrow\} \neq \emptyset\}.$$

(b) A set is ***WhileCC*-semicomputable** on A if it is the halting set on A of a ***WhileCC*** procedure.

(c) ***WhileCC** semicomputability** is defined similarly.

(d) ***SWhileCC***(A) is the class of ***WhileCC*-semicomputable** relations on A .

(e) ***SWhileCC****(A) is defined similarly.

If R is the halting set of the procedure P , then a *code* or *Gödel number* of R is given by a code of P .

4.3 Closure under projections and countable unions

Definition 4.3.1 (Minimal carrier). A carrier A_s of an algebra A is said to be Σ -*minimal* if every element of it is the value of a closed Σ -term of sort s .

Definition 4.3.2 (TEP). (a) The *term evaluation representing function* on A relative to a tuple of variables $\mathbf{x} : u$ and Σ -sort s is the function

$$te_{\mathbf{x},s}^A: \ulcorner \mathbf{Term}_{\mathbf{x},s} \urcorner \times A^u \rightarrow A_s$$

where $\mathbf{Term}_{\mathbf{x},s}$ is the class of Σ -terms of sort s with variables among \mathbf{x} only, and for any term $t \in \mathbf{Term}_{\mathbf{x},s}$ and $a \in A^u$, $\mathbf{te}_{\mathbf{x},s}^A(\ulcorner t \urcorner, a)$ is the value of t when \mathbf{x} is assigned the value a .

(b) The partial algebra A has the *term evaluation property* (TEP) if for all \mathbf{x} and s , the function $\mathbf{te}_{\mathbf{x},s}^A$ is $\mathbf{While}(A)$ computable.

Lemma 4.3.3. *The function $\mathbf{te}_{\mathbf{x},s}^A$ is always $\mathbf{While}^*(A)$ computable.*

Lemma 4.3.4 (Closure under projection).

(a) $\mathbf{SWhileCC}(A)$ and $\mathbf{SWhileCC}^*(A)$ are closed under projection off nat, i.e., existential quantification over \mathbb{N} .

(b) If A_s is minimal, then $\mathbf{SWhileCC}^*(A)$ is closed under projection off sort s , i.e., existential quantification over A_s .

(c) If A_s is minimal and A has TEP, then $\mathbf{SWhileCC}(A)$ is closed under projection off sort s , i.e., existential quantification over A_s .

Proof: (a) Note that $\exists n R(x, n) \iff R(x, \text{choose } n: R(x, n))$.

(b) Note that $\exists y R(x, y) \iff \exists n R(x, \mathbf{te}_s^A(n, \langle \rangle))$.

(c) Like (b). Use Lemma 4.3.3. \square

Lemma 4.3.5 (Closure under effective countable unions). *If A has TEP, then $\mathbf{SWhileCC}^*(A)$ is closed under effective countable unions.*

Proof: From TEP there follows a Universal Function Theorem for $\mathbf{WhileCC}^*$: namely there is a $\mathbf{WhileCC}^*$ function

$$\mathbf{Univ}_{u \rightarrow \text{bool}}^A : \mathbb{N} \times A^u \rightarrow \mathbb{B}$$

which is universal for $\mathbf{Proc}_{u \rightarrow \text{bool}}$ on A , in the sense that for all $P \in \mathbf{Proc}_{u \rightarrow \text{bool}}$ and $x \in A^u$,

$$\mathbf{Univ}_{u \rightarrow \text{bool}}^A(\ulcorner P \urcorner, x) \simeq P^A(x).$$

This can be proved by an adaptation of the methods of [TZ00] (cf. [Jia03]). Now let $f: \mathbb{N} \rightarrow \mathbb{N}$ be a total computable numbering of a sequence of codes of $\mathbf{WhileCC}^*$ semicomputable relations $R_n \subseteq A^u$ ($n = 0, 1, 2, \dots$). We may assume w.l.o.g. that the procedures enumerated by f all have range type bool. Then

$$\begin{aligned} \bigcup_n R_n &= \{x \in A^u \mid \exists n P_{f(n)}(x) \downarrow\} \\ &= \{x \in A^u \mid \exists n \mathbf{Univ}_{u \rightarrow \text{bool}}^A(f(n), x) \downarrow\} \end{aligned}$$

which is in $\mathbf{SWhileCC}^*$ by Lemma 4.3.4(a). \square

4.4 Locality of Computation

The locality of computation theorem, proved for the deterministic **While** language in [TZ00, §3.8], also applies to **WhileCC***. The proof, in broad lines, follows the proof in [TZ00]. (For $X \subseteq A$, we write $\langle X \rangle_v^A$ for the retract to A^v of the Σ -subalgebra of A generated by X .)

Lemma 4.4.1 (Locality for terms). *If $t : s$ and $\text{var}(t) \subseteq \mathbf{x}$ then*

$$\llbracket t \rrbracket^A \sigma \subseteq \langle \sigma[\mathbf{x}] \rangle_v^A.$$

Proof: By structural induction on t , as in [TZ00, §3.8]. The interesting new case is $t \equiv \text{choose } z : b$. \square

Lemma 4.4.2 (Locality for computation trees). *If $\text{var}(S) \subseteq \mathbf{x} : u$, then for all n*

$$\mathbf{CompTreeStage}^A(S, \sigma, n)[\mathbf{x}] \subseteq \langle \sigma[\mathbf{x}] \rangle_u^A.$$

Here the l.h.s. means the set of u -tuples $\sigma'[\mathbf{x}]$ for all states σ' in $\mathbf{CompTreeStage}^A(S, \sigma, n)$, i.e., the first n stages of the computation tree of statement S at state σ [TZ04a, §4.2(e)].

Proof: By induction on n . \square

From this follows:

Lemma 4.4.3 (Locality for procedures).

*If $P^A : u \rightarrow v$ is a **WhileCC***(Σ) procedure, then for all $a \in A^u$,*

$$P^A(a) \setminus \{\uparrow\} \subseteq \langle a \rangle_v^A.$$

Theorem 4.4.4 (Locality for functions).

*If $f : A^u \rightarrow A_s$ is **WhileCC*** computable on A , then for any $a \in A^u$, if $f(a) \downarrow$ then*

$$f(a) \in \langle a \rangle_s^A.$$

5 A general soundness theorem

Again, A is a countable partial Σ -algebra. However we need not assume N-standardness of A in this section.

5.1 Soundness theorem

Theorem 5.1.1 (Soundness of **WhileCC* computation).** *Suppose A is Σ -effective under α . Then*

$$\mathbf{WhileCC}^*(A) \subseteq \mathbf{Comp}_\alpha(A).$$

This is Theorem A_0 in [TZ04a, §7]. A complete proof is given there. Here we present, as an indication of the proof, the last part of the main lemma (Lemma Scheme 7.3.1(g)) from which the theorem easily follows.

Lemma 5.1.2 (Tracking of procedure evaluation). *For a specific triple of lists of variables $\mathbf{a} : u, \mathbf{b} : v, \mathbf{c} : w$, let $\mathbf{Proc}_{\mathbf{a},\mathbf{b},\mathbf{c}}$ be the class of all **WhileCC** procedures of type $u \rightarrow v$, with declaration ‘in \mathbf{a} out \mathbf{b} aux \mathbf{c} ’. The (many-valued) procedure evaluation function localised to this declaration:*

$$\mathbf{PE}_{\mathbf{a},\mathbf{b},\mathbf{c}}^A : \mathbf{Proc}_{\mathbf{a},\mathbf{b},\mathbf{c}} \times A^u \rightrightarrows^+ A^{v\uparrow}$$

defined by

$$\mathbf{PE}_{\mathbf{a},\mathbf{b},\mathbf{c}}^A(P, a) = P^A(a),$$

is α -tracked by a computable function

$$\mathbf{pe}_{\mathbf{a},\mathbf{b},\mathbf{c}}^A : \ulcorner \mathbf{Proc}_{\mathbf{a},\mathbf{b},\mathbf{c}} \urcorner \times \Omega_\alpha^u \dot{\rightarrow} \Omega_\alpha^v,$$

i.e., the following diagram commutes:

$$\begin{array}{ccc} \mathbf{Proc}_{\mathbf{a},\mathbf{b},\mathbf{c}} \times A^u & \xrightarrow{\mathbf{PE}_{\mathbf{a},\mathbf{b},\mathbf{c}}^A} & A^{v\uparrow} \\ \langle \mathbf{enum}, \alpha^u \rangle \uparrow & & \uparrow \alpha^v \\ \ulcorner \mathbf{Proc}_{\mathbf{a},\mathbf{b},\mathbf{c}} \urcorner \times \Omega_\alpha^u & \xrightarrow{\mathbf{pe}_{\mathbf{a},\mathbf{b},\mathbf{c}}^A} & \Omega_\alpha^v \end{array}$$

in the sense that

$$\begin{aligned} \mathbf{pe}_{\mathbf{a},\mathbf{b},\mathbf{c}}^A(\ulcorner P \urcorner, k) \downarrow l &\implies \alpha(l) \in \mathbf{PE}_{\mathbf{a},\mathbf{b},\mathbf{c}}^A(P, \alpha(k)), \\ \mathbf{pe}_{\mathbf{a},\mathbf{b},\mathbf{c}}^A(\ulcorner P \urcorner, k) \uparrow &\implies \uparrow \in \mathbf{PE}_{\mathbf{a},\mathbf{b},\mathbf{c}}^A(P, \alpha(k)). \end{aligned}$$

Here $\mathbf{pe}_{\mathbf{a},\mathbf{b},\mathbf{c}}^A$ is a combination “tracking function” and “selection function”. We can think of $\mathbf{pe}_{\mathbf{a},\mathbf{b},\mathbf{c}}^A$ as giving one possible implementation of $\mathbf{PE}_{\mathbf{a},\mathbf{b},\mathbf{c}}^A$.

Proof of Theorem 5.1.1: Suppose $f : A^u \dot{\rightarrow} A_s$ is **WhileCC*** computable on A . Then there is a deterministic **WhileCC*** procedure

$$P : u \rightarrow s$$

such that for all $a \in A^u$,

$$\begin{aligned} f(x) \downarrow y &\implies P^A(x) = \{y\}, \\ f(x) \uparrow &\implies P^A(x) = \{\uparrow\}. \end{aligned}$$

Hence by Lemma 5.1.2 (substituting a suitable constant for $\lceil P \rceil$) there is a computable (partial) function

$$\varphi: \Omega_\alpha^u \dot{\rightarrow} \Omega_{\alpha,s}$$

which α -tracks f , as required.

Note that in applying Lemma 5.1.2 to prove Theorem 5.1.1, we are implicitly using the canonical extension α^* of α given by Lemma 2.3.8. \square

5.2 Applications

Corollary 5.2.1. *Suppose A is Σ -effective under α . Then for any relation on A ,*

$$\mathbf{WhileCC}^*\text{-semicomputability} \implies \alpha\text{-semicomputability}.$$

Proof: By the Soundness Theorem 5.1.1. \square

Let $\langle =^A \rangle$ be the family of equality relations $\langle =_s^A \mid s \in \mathbf{Sort}(\Sigma) \rangle$.

Corollary 5.2.2.

- (a) *If $\langle =^A \rangle$ is $\mathbf{WhileCC}^*$ -computable, then α has a computable kernel.*
- (b) *If $\langle =^A \rangle$ is $\mathbf{WhileCC}^*$ -semicomputable, then α has a semicomputable kernel.*

Definition 5.2.3. (a) A **DE** (*disjunctive-existential*) Σ -formula is one of the form

$$\bigvee_{i=0}^{\infty} \exists y_i \mathbf{b}_i(\mathbf{x}, y_i)$$

i.e., an infinite disjunction of a *computable sequence* of existentially quantified Σ -booleans, $\mathbf{x} \equiv (\mathbf{x}_1, \dots, \mathbf{x}_k)$ and $y_i \equiv (y_{i1}, \dots, y_{il_i})$.

(b) **DE**(A) is the class of **DE**(Σ) definable relations on A .

Theorem 5.2.4 (Invariance of α -semicomputability). *Let A be a countable Σ -algebra. Then*

$$\mathbf{DE}(A) \subseteq \bigcap_{\alpha \in \Sigma\text{-EffNum}(A)} \mathbf{SComp}_\alpha(A).$$

Proof: Straightforward. To deal with evaluation in A of the sequences y_i of bound variables (of finite but unbounded length), we need term evaluation on A^* , which is \mathbf{While}^* -computable on A^N [TZ00, §4.7], and hence (by Lemma 3.3.7 and the Soundness Theorem 5.1.1) α^* -computable. \square

It follows from Engeler's Lemma [TZ00, §§5.11/12] and the above theorem that

$$\begin{aligned} \text{proj. } \mathbf{While} \text{ semicomputability} &\implies \mathbf{DE}\text{-definability} \\ &\implies \alpha\text{-semicomputability for all } \alpha \in \Sigma\text{-EffNum}(A). \end{aligned}$$

The converse directions, *i.e.*, finding “reasonable” side conditions under which the above arrows can be reversed, remain to be investigated.

6 Examples of Completeness Theorems

We begin with an example of incompleteness of abstract w.r.t. concrete computation, and then find conditions on the algebra which guarantee completeness.

6.1 An example of incompleteness

Consider the single-sorted algebra

$$A = (\mathbb{N}, 0, \text{pd})$$

where pd is the predecessor function on \mathbb{N} : $\text{pd}(0) = 0$, $\text{pd}(n+1) = n$.

This algebra is clearly computable (in the sense of Definition 3.3.3), since it is computable under *e.g.* the identity numbering $\mathbf{id}: \mathbb{N} \rightarrow \mathbb{N}$. For a more general kind of concrete representation of A we have the following incompleteness result. Let $\text{suc}: \mathbb{N} \rightarrow \mathbb{N}$ be the successor function, $\text{suc}(n) = n + 1$.

Theorem 6.1.1. *For any computable numbering α of A , we have*

$$\text{suc} \in \mathbf{Comp}_\alpha(A) \quad \text{but} \quad \text{suc} \notin \mathbf{WhileCC}^*(A)$$

and hence

$$\mathbf{WhileCC}^*(A) \subsetneq \bigcap_{\alpha \in \mathbf{CompNum}(A)} \mathbf{Comp}_\alpha(A).$$

Proof: First, suppose $\text{suc} \in \mathbf{WhileCC}^*(A)$. Then by the locality of computation theorem (4.4.2), we would have $n+1 \in \langle n \rangle^A$. But

$$\langle n \rangle^A = \{0, 1, \dots, n\},$$

and so $\text{suc} \notin \mathbf{WhileCC}^*(A)$.

Next, let $\alpha: \Omega_\alpha \rightarrow \mathbb{N}$ be a computable numbering of A with 0 α -coded by $c \in \Omega_\alpha$ and pd recursively α -tracked by $\varphi: \Omega_\alpha \rightarrow \Omega_\alpha$. Define $\psi: \mathbb{N} \dot{\rightarrow} \mathbb{N}$ by

$$\psi(k) \simeq \text{some } l [l \in \Omega_\alpha \wedge l \not\equiv_\alpha c \wedge \varphi(l) \equiv_\alpha k].$$

Since Ω_α is r.e. and \equiv_α is recursive, ψ is recursive. Since $\varphi: \Omega_\alpha \rightarrow \Omega_\alpha$, it follows that

$$\psi: \Omega_\alpha \rightarrow \Omega_\alpha$$

and for all $k, l \in \Omega_\alpha$:

$$\begin{aligned} \psi(k) \downarrow l &\Rightarrow l \not\equiv_\alpha c \wedge \varphi(l) \equiv_\alpha k \\ &\Leftrightarrow \alpha(l) \neq 0 \wedge \text{pd}(\alpha(l)) = \alpha(k) \\ &\Leftrightarrow \alpha(k) + 1 = \alpha(l). \end{aligned}$$

So φ recursively α -tracks succ .

Call the above sort of naturals (with 0 and predecessor) nat^- . Note that essentially the same counterexample to completeness can be constructed on nat^- even if we N-standardise A , *i.e.*, adjoin the standard sort nat of naturals with 0 and successor, in addition to nat^- .

6.2 Sections, equality and completeness

We turn to conditions which guarantee completeness, *i.e.*, a sort of converse of the Soundness Theorem 6.1.1.

We assume again that A is an N-standard partial Σ -algebra.

Definition 6.2.1 (Sections). A *section* of α is a right inverse of α , *i.e.*, a family

$$\hat{\alpha} = \langle \hat{\alpha}_s \mid s \in \mathbf{Sort}(\Sigma) \rangle$$

of mappings

$$\hat{\alpha}_s: A_s \longrightarrow \mathbb{N}$$

such that

$$\alpha \circ \hat{\alpha} = \mathbf{id}_A, \tag{6.1}$$

i.e., for all $s \in \mathbf{Sort}(\Sigma)$,

$$\alpha_s \circ \hat{\alpha}_s = \mathbf{id}_{A_s}.$$

Note from (6.1) that (by the property of left and right inverses) α is onto (which we already knew) and $\hat{\alpha}$ is 1-1. So for all $a \in A$, $\hat{\alpha}(a)$ selects an element of $\alpha^{-1}(\{a\})$ (which is not empty, since α is onto).

Thus a section $\hat{\alpha}$ of α always exists, by the Axiom of Choice. The interesting question is: when does α have a *computable* section?

Lemma 6.2.2. *Suppose α is **While*** computable. Then the following are equivalent:*

- (1) α has a **WhileCC*** computable section $\hat{\alpha}$,
- (2) $=^A$ is **WhileCC*** computable,
- (3) $=^A$ is **WhileCC*** semicomputable.

Proof: (1) \Rightarrow (2): Assume (1). Then for all $a, b \in A_s$:

$$a = b \iff \hat{\alpha}_s(a) = \hat{\alpha}_s(b) \text{ in } \mathbb{N},$$

which gives a **WhileCC*** decision procedure for $=^A$.

(2) \Rightarrow (3): trivial.

(3) \Rightarrow (1): Suppose $=^A_s$ is **WhileCC*** semicomputable. Define $\hat{\alpha}_s: A_s \rightarrow \mathbb{N}$ as follows. With *input* $a \in A_s$, the *output* is given by

$$\text{choose } \mathbf{k}: \alpha_s(\mathbf{k}) = a.$$

where we are choosing an item satisfying a **WhileCC***-semicomputable predicate. \square

Note that in the proof of (1) \Rightarrow (2), **WhileCC*** computability of α was not used.

Theorem 6.2.3 (Adequacy of WhileCC* computation). *Suppose α is While* computable. Then any of the conditions (1), (2) or (3) of Lemma 6.2.2 implies:*

$$\mathbf{Comp}_\alpha(A) \subseteq \mathbf{WhileCC}^*(A).$$

Proof: Assume any of (1), (2) or (3) of Lemma 6.2.2. (We will actually use (1).) Let f be α -computable on A by a recursive φ :

$$\begin{array}{ccc} A^u & \xrightarrow{f} & A_s \\ \alpha^u \updownarrow \hat{\alpha}_u & & \uparrow \alpha_s \\ \Omega_\alpha^u & \xrightarrow{\varphi} & \Omega_{\alpha,s} \end{array}$$

Then

$$f = \alpha_s \circ \varphi \circ \hat{\alpha}_u$$

which is **WhileCC*** computable. \square

Note that Σ -effectivity of A under α is not assumed in the proof of adequacy (Theorem 6.2.3) or Lemma 6.2.2. It is, however, assumed for the reverse inclusion, soundness (Theorem 5.1.1).

Combining Theorems 5.1.1 (soundness) and 6.2.3 (adequacy) we have

Theorem 6.2.4 (Completeness of WhileCC* computation, Version 1). *Suppose A is Σ -effective under α , and α is While* computable. Then any of the conditions (1), (2) or (3) of Lemma 6.2.2 implies:*

$$\mathbf{Comp}_\alpha(A) = \mathbf{WhileCC}^*(A).$$

Combining this with Lemma 6.2.2 provides another formulation of completeness:

Theorem 6.2.5 (Completeness of WhileCC* computation, Version 2). *Suppose A is Σ -effective under α , and α is WhileCC* computable. Then the following are equivalent:*

- (1) α has a **WhileCC*** computable section,
- (2) $=^A$ is **WhileCC*** computable,
- (3) $=^A$ is **WhileCC*** semicomputable,
- (4) $\mathbf{Comp}_\alpha(A) = \mathbf{WhileCC}^*(A)$ and α has a semicomputable kernel,
- (5) $\mathbf{Comp}_\alpha(A) = \mathbf{WhileCC}^*(A)$ and α has a computable kernel.

Clearly, if A satisfies the assumptions of this theorem, and also has equality at all sorts in its signature, then

$$\mathbf{Comp}_\alpha(A) = \mathbf{WhileCC}^*(A).$$

For our next corollary, we suppose that A is a *finitely generated* Σ -algebra, say

$$A = \langle c_1^A, \dots, c_p^A \rangle_A.$$

Then A has a *canonical numbering* α_c , defined as the composition

$$\mathbb{N} \xrightarrow{\gamma} \mathbf{T}(\Sigma, \mathbf{x}) \xrightarrow{te_c^A} A$$

where $\mathbf{x} \equiv (x_1, \dots, x_p)$ is a tuple of variables of the same type as $\mathbf{c} \equiv (c_1, \dots, c_p)$, $\mathbf{T}(\Sigma, \mathbf{x})$ is the set of Σ -terms generated from \mathbf{x} , γ is a *standard effective numbering* of $\mathbf{T}(\Sigma, \mathbf{x})$, and te_c^A is the evaluation of terms in $\mathbf{T}(\Sigma, \mathbf{x})$ determined by the assignment $\langle \mathbf{x} \mapsto \mathbf{c}^A \rangle$. By [TZ00, Corollary 4.7], α_c is **For*** (and hence **While***) computable. This yields the following Corollary of Theorem 6.2.5.

Corollary 6.2.6 (Completeness for algebras with canonical numberings). *Suppose A is a finitely generated Σ -algebra, and is Σ -effective under the canonical numbering α_c . Then the following are equivalent:*

- (1) α_c has a **WhileCC*** computable section,
- (2) $=^A$ is **WhileCC*** computable,
- (3) $=^A$ is **WhileCC*** semicomputable,
- (4) $\mathbf{Comp}_{\alpha_c}(A) = \mathbf{WhileCC}^*(A)$ and α_c has a semicomputable kernel,
- (5) $\mathbf{Comp}_{\alpha_c}(A) = \mathbf{WhileCC}^*(A)$ and α_c has a computable kernel.

A similar result holds when A is generated not by a finite set of Σ -constants, as above, but by an infinite set c_0, c_1, c_2, \dots , where the function $n \mapsto c_n$ is in Σ .

6.3 Invariance, definability and the Ash-Nerode Theorem

In concrete computability theories based on numbered structures, invariance questions of the following kind arise:

Invariance Problem. Let A be a computable Σ -algebra. What are

$$\bigcap_{\alpha \in \mathbf{CompNum}(A)} \mathbf{Comp}_\alpha(A) \quad \text{and} \quad \bigcap_{\alpha \in \mathbf{CompNum}(A)} \mathbf{SComp}_\alpha(A)?$$

Viewed from the theory of numbered algebras, our work on soundness and completeness can be seen as trying to answer these sorts of questions using abstract computability theories.

However, recalling the role of the arithmetic hierarchy for computability theories on \mathbb{N} , it is also natural to ask if definability in logical languages based on Σ can characterise the computability of functions and sets on a numbered algebra A . Definability problems of this kind, first considered in [Mal71a], can provide *some* answers to the Invariance Problem. Not surprisingly, very special properties of A are needed for completeness, and it seems that to progress we must consider either

- (i) particular structures, such as matrix groups, or
- (ii) structures satisfying stringent conditions that are hard to satisfy.

We will discuss one source of illumination of the latter kind.

Now, thanks to soundness, we already have the general observation (Theorem 5.2.4) that, for any countable A , effective infinite disjunctions of existentially quantified Σ -booleans are α -semicomputable under every Σ -effective numbering α of A .

We will look at the Ash-Nerode Theorem which contains strong sufficient conditions on A that imply the converse and, hence, a completeness result. The Ash-Nerode Theorem was first proved in [AN81]; a new account can be found in Chapter 11 of [AK00]. We modify the original definitions, in keeping with the theory of numberings.

First, the kind of invariance Ash-Nerode studied is captured by the following definition.

Definition 6.3.1. Let A be a Σ -algebra computable under a numbering $\alpha: \Omega_\alpha \rightarrow A$. Let $R \subseteq A^k$. Then R is *intrinsically semicomputable* if for any Σ -algebra B computable under $\beta: \Omega_\beta \rightarrow B$, and any Σ -isomorphism $\phi: A \rightarrow B$, $\phi(R)$ is β -semicomputable.

Note that, if R is intrinsically semicomputable, then (taking $B = A$, $\beta = \alpha$, and $\phi = I_A$), it follows that R is α -semicomputable.

It is easy to connect this notion with the Invariance Problem:

Lemma 6.3.2. Let A be an α -computable Σ -algebra, and $R \subseteq A^k$. Then the following are equivalent:

- (1) R is intrinsically semicomputable,
- (2) R is semicomputable in every computable numbering of A , i.e.,

$$R \in \bigcap_{\alpha \in \mathbf{CompNum}(A)} \mathbf{SComp}_\alpha(A).$$

The special condition of the Ash-Nerode Theorem is based on the decidability of this property:

Definition 6.3.3. Let A be a Σ -algebra computable under α , and $R \subseteq A^k$. The *satisfiability problem outside R* is the following decision problem: For any finitary existentially quantified Σ -boolean

$$\exists \mathbf{y} \mathbf{b}(\mathbf{x}, \mathbf{y}) \quad (\mathbf{x} \equiv (\mathbf{x}_1, \dots, \mathbf{x}_k), \quad \mathbf{y} \equiv (\mathbf{y}_1, \dots, \mathbf{y}_l), \quad l > 0)$$

and any $a \in A^k$, is there $b \in A^l$ such that

$$b \notin R \text{ and } A \models \mathbf{b}[a, b]?$$

Note that if the existential diagram of (A, R) is α -decidable, then the satisfiability problem outside R is α -decidable.

The *Ash-Nerode Theorem* says that if the satisfiability problem outside R is α -decidable, then R is intrinsically semicomputable if, and only if, R can be expressed as a **DE**(Σ)-formula (see Definition 5.2.3). Combining this with Lemma 6.3.2, we have

Theorem 6.3.4. *Let A be a Σ -algebra computable under α and let $R \subseteq A^k$. Suppose the satisfiability problem outside R is α -decidable. Then the following are equivalent:*

- (1) R is intrinsically semicomputable;
- (2) R is semicomputable in every computable numbering of A , i.e.,

$$R \in \bigcap_{\alpha \in \mathbf{CompNum}(A)} \mathbf{SComp}_\alpha(A);$$

- (3) R is expressible as **DE**(Σ)-formula:

$$x \in R \iff \bigvee_{i=0}^{\infty} \exists y_i \mathbf{b}_i(\mathbf{x}, y_i).$$

The equivalence of (2) and (3) is a form of completeness theorem.

References

- [AGM92] S. Abramsky, D. Gabbay, and T. Maibaum, editors. *Handbook of Logic in Computer Science*. Oxford University Press, 1992. In 5 volumes, 1992–2000.
- [AK00] C.J. Ash and Julia Knight. *Computable Structures and the Hyperarithmetical Hierarchy*. Elsevier, 2000.
- [AN81] C.J. Ash and A. Nerode. Intrinsically recursive relations. In J.N. Crossley, editor, *Proceedings of the Conference on Aspects of Effective Algebra, Monash University*, pages 26–41. U.D.A. Book Co., Steel’s Creek, Victoria, Australia, 1981.
- [BCSS98] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation*. Springer-Verlag, 1998.
- [Ber93] U. Berger. Total sets and objects in domain theory. *Annals of Pure & Applied Logic*, 60:91–117, 1993.
- [Ber02] U. Berger. Computability and totality in domains. *Mathematical Structures in Computer Science*, 12:455–467, 2002.

- [BH02] V. Brattka and P. Hertling. Topological properties of real number representations. *Theoretical Computer Science*, 284(2):241–257, 2002.
- [Bou01] J.P. Bourguignon. A basis for a new relationship between mathematics and society. In B. Engquist and W. Schmid, editors, *Mathematics Unlimited — 2001 and beyond*, pages 171–188. Springer-Verlag, 2001.
- [Bra96] V. Brattka. Recursive characterisation of computable real-valued functions and relations. *Theoretical Computer Science*, 162:45–77, 1996.
- [Bra97] V. Brattka. Order-free recursion on the real numbers. *Mathematical Logic Quarterly*, 43:216–234, 1997.
- [BSS89] L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bulletin of the American Mathematical Society*, 21:1–46, 1989.
- [BT04] E.J. Beggs and J.V. Tucker. Computations via experiments with kinematic systems. Technical Report 5-2004, Department of Computer Science, University of Wales Swansea, March 2004.
- [Ceï59] G.S. Ceitin. Algebraic operators in constructive complete separable metric spaces. *Doklady Akademii Nauk SSSR*, 128:49–52, 1959.
- [Eda97] A. Edalat. Domains for computation in mathematics, physics and exact real arithmetic. *Bulletin of Symbolic Logic*, 3:401–452, 1997.
- [EGNR98] Y.L. Ershov, S.S. Goncharov, A.S. Nerode, and J.B. Remmel, editors. *Handbook of Recursive Mathematics*. Elsevier, 1998. In 2 volumes.
- [Eng68] E. Engeler. *Formal Languages: Automata and Structures*. Markham Publishing Co, 1968.
- [Esc96] M. Escardó. PCF extended with real numbers. *Theoretical Computer Science*, 162:79–115, 1996.
- [Fen02] J.E. Fenstad. Computability theory: Structure or algorithms. In W. Sieg, R. Sommer, and C. Talcott, editors, *Reflections on the Foundations of Mathematics: Essays in honor of Solomon Feferman*, volume 15 of *Lecture Notes in Logic*, pages 188–213. Association for Symbolic Logic, 2002.
- [Fri71] H. Friedman. Algebraic procedures, generalized Turing algorithms, and elementary recursion theory. In R.O. Gandy and C.M.E. Yates, editors, *Logic Colloquium '69*, pages 361–389. North Holland, 1971.
- [FS56] A. Fröhlich and J. Shepherdson. Effective procedures in field theory. *Philosophical Transactions of the Royal Society London*, (A) 248:407–432, 1956.
- [Gri99] E. Griffor, editor. *Handbook of Computability Theory*. North Holland, 1999.
- [Gro98] M. Gromov. Possible trends in mathematics in the coming decades. *Notices of the AMS*, 45:846–847, 1998.
- [Grz55] A. Grzegorzcyk. Computable functions. *Fundamenta Mathematicae*, 42:168–202, 1955.

- [Grz57] A. Grzegorzcyk. On the definitions of computable real continuous functions. *Fundamenta Mathematicae*, 44:61–71, 1957.
- [HI70] G.T. Herman and S.D. Isard. Computability over arbitrary fields. *Journal of the London Mathematical Society (2)*, 2:73–79, 1970.
- [Jia03] W. Jiang. Universality and semicomputability for non-deterministic programming languages over abstract algebras. M.Sc. Thesis, Department of Computing & Software, McMaster University, 2003. Technical Report CAS-03-03-JZ, Dept of Computing & Software, McMaster University.
- [Kle59] S.C. Kleene. Countable functionals. In A. Heyting, editor, *Constructivity in Mathematics*, pages 81–100. North Holland, 1959.
- [Kre59] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite type. In A. Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North Holland, 1959.
- [Kre71] G. Kreisel. Some reasons for generalizing recursion theory. In R.O. Gandy and C.M.E. Yates, editors, *Logic Colloquium '69*, pages 139–198. North Holland, 1971.
- [Lac55] D. Lacombe. *Extension de la notion de fonction récursive aux fonctions d'une ou plusieurs variables réelles*, I, II, III. *C.R. Acad. Sci. Paris*, 1955. 240:2470–2480, 241:13–14, 151–153.
- [Lon04] J. Longley. Notions of computability at higher types I. In *Proceedings of the ASL Logic Colloquium, Paris, 2000*. A K Peters, 2004. To appear.
- [Mal71a] A.I. Mal'cev. Closely related models and recursively perfect algebras. In *The metamathematics of algebraic systems. A.I. Malcev, Collected papers: 1936–1967*, pages 255–261. North Holland, 1971.
- [Mal71b] A.I. Mal'cev. Constructive algebras I. In *The metamathematics of algebraic systems. A.I. Malcev, Collected papers: 1936–1967*, pages 148–212. North Holland, 1971.
- [MKS76] W. Magnus, A. Karass, and D. Solitar. *Combinatorial Group Theory*. Dover, 1976.
- [Mol77] J. Moldestad. *Computations in higher types*, volume 574 of *Lecture Notes in Mathematics*. Springer-Verlag, 1977.
- [Mon68] R. Montague. Recursion theory as a branch of model theory. In B. van Rootselaar and J.F. Staal, editors, *Logic, Methodology & Philosophy of Science III*, pages 63–86. North Holland, 1968.
- [Mos64] Y.N. Moschovakis. Recursive metric spaces. *Fundamenta Mathematicae*, 55:215–238, 1964.
- [Mos69a] Y.N. Moschovakis. Abstract first order computability I. *Transactions of the American Mathematical Society*, 138:427–464, 1969.
- [Mos69b] Y.N. Moschovakis. Abstract first order computability II. *Transactions of the American Mathematical Society*, 138:465–504, 1969.

- [Nor80] D. Normann. *Recursion on the countable functionals*, volume 811 of *Lecture Notes in Mathematics*. Springer-Verlag, 1980.
- [Nor82] D. Normann. External and internal algorithms on the continuous functionals. In G. Metakides, editor, *Patras Logic Symposium*, volume 109 of *Studies in Logic*, pages 137–144. North Holland, 1982.
- [Nor99] D. Normann. The continuous functionals. In E. Griffor, editor, *Handbook of Computability Theory*. North Holland, 1999.
- [Nor00] D. Normann. Computability over the partial continuous functionals. *Journal of Symbolic Logic*, pages 1133–1142, 2000.
- [PER89] M.B. Pour-El and J.I. Richards. *Computability in Analysis and Physics*. Springer-Verlag, 1989.
- [Pla66] R.A. Platek. *Foundations of Recursion Theory*. PhD Thesis, Department of Mathematics, Stanford University, 1966.
- [Plo77] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
- [Rab60] M. Rabin. Computable algebra, general theory and the theory of computable fields. *Transactions of the American Mathematical Society*, 95:341–360, 1960.
- [Ric54] H.G. Rice. Recursive real numbers. *Proceedings of the American Mathematical Society*, 5:784–791, 1954.
- [She73] J.C. Shepherdson. Computations over abstract structures: serial and parallel procedures and friedman’s effective definitional schemes. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium ’73*, pages 445–513. North Holland, 1973.
- [She85] J.C. Shepherdson. Algebraic procedures, generalized turing algorithms, and elementary recursion theory. In L.A. Harrington, M.D. Morley, A. Šcedrov, and S.G. Simpson, editors, *Harvey Friedman’s Research on the Foundations of Mathematics*, pages 285–308. North Holland, 1985.
- [SHLG94] Stoltenberg-Hansen, I. Lindström, and E. Griffor. *Mathematical Theory of Domains*. Cambridge University Press, 1994.
- [SHT88] V. Stoltenberg-Hansen and J.V. Tucker. Complete local rings as domains. *Journal of Symbolic Logic*, 53:603–624, 1988.
- [SHT95] V. Stoltenberg-Hansen and J.V. Tucker. Effective algebras. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 357–526. Oxford University Press, 1995.
- [SHT99a] V. Stoltenberg-Hansen and J.V. Tucker. Computable rings and fields. In E. Griffor, editor, *Handbook of Computability Theory*. North Holland, 1999.
- [SHT99b] V. Stoltenberg-Hansen and J.V. Tucker. Concrete models of computation for topological algebras. *Theoretical Computer Science*, 219:347–378, 1999.

- [Sma98] S. Smale. Mathematical problems for the next century. *The Mathematical Intelligencer*, 20:7–15, 1998.
- [Soa96] R.L. Soare. Computability and recursion. *Bulletin of Symbolic Logic*, 2:284–321, 1996.
- [Spr98] D. Spreen. On effective topological spaces. *Journal of Symbolic Logic*, 63:185–221, 1998.
- [Tai62] W.W. Tait. Continuity properties of partial recursive functionals of finite type. Unpublished notes, 1962.
- [Tur36] A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936. With correction, *ibid.*, 43, 544–546, 1937. Reprinted in *The Undecidable*, M. Davis, ed., Raven Press, 1965.
- [TZ88] J.V. Tucker and J.I. Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics*, volume 6 of *CWI Monographs*. North Holland, 1988.
- [TZ99] J.V. Tucker and J.I. Zucker. Computation by ‘while’ programs on topological partial algebras. *Theoretical Computer Science*, 219:379–420, 1999.
- [TZ00] J.V. Tucker and J.I. Zucker. Computable functions and semicomputable sets on many-sorted algebras. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 5, pages 317–523. Oxford University Press, 2000.
- [TZ04a] J.V. Tucker and J.I. Zucker. Abstract versus concrete computation on metric partial algebras. *ACM Transactions on Computational Logic*, 2004. To appear.
- [TZ04b] J.V. Tucker and J.I. Zucker. Computable total functions, algebraic specifications and dynamical systems. *Journal of Logic and Algebraic Programming*, 2004. To appear.
- [Wei00] K. Weihrauch. *Computable Analysis: An Introduction*. Springer-Verlag, 2000.