GENERALIZING COMPUTABILITY THEORY TO ABSTRACT ALGEBRAS

J.V. TUCKER AND J.I. ZUCKER

Abstract. We present a survey of our work over the last few decades on generalizations of computability theory to many-sorted algebras. The following topics are discussed, among others: (1) abstract *v* concrete models of computation for such algebras; (2) computability and continuity, and the use of many-sorted topological partial algebras, containing the reals; (3) comparisons between various equivalent and distinct models of computability; (4) generalized Church-Turing theses.

1. Introduction

Since 1936, most of the development of computability theory has focused on sets and functions of strings and natural numbers – though computability theory immediately found applications, first in logic and algebra and then pretty much everywhere. Indeed, recall that when Turing created his model of computation on strings [Tur36], he applied it to solve a problem involving real numbers. The applications were via codings or representations of other data.

Why would we want to generalize the classical computability theory from strings and natural numbers to arbitrary abstract algebras? How should we do it?

Computability theory is a general theory about what and how we compute. At its heart is the notion of an algorithm that processes data. Today, algorithms and data abound in all parts of our professional, social and personal lives. Data are composed of numbers, texts, video and audio. We know that, being digital, this vast range of data is coded or represented by bitstrings (or strings over a finite alphabet). However, the algorithms that make data useful are created specifically for a high-level, independent model of the data close to the use and users of the data. Therefore, computability theory cannot be content with bitstrings and natural numbers.

Now, the algorithms are designed to schedule sequences of basic operations and tests on the data to accomplish a task. They are naturally high-level and their level of abstraction is defined by the nature of the basic operations and tests. In fact, a fundamental observation is this:

What an algorithm knows of the data it processes is precisely determined by what operations and tests on the data it can employ.

Key words and phrases. generalized computability, computability on abstract structures, computability on the reals, generalized Church-Turing thesis, computability and continuity.

2010 *AMS Mathematics Subject Classification* 03D75, 03D78.

Research supported by a grant from the Natural Sciences and Engineering Research Council (Canada).

This is true of strings and natural numbers, of course. This observation leads to the fundamental idea of an *abstract data type* in computer science. Mathematically, an implementation of a data type is modelled by an algebraic structure with operations and relations, and an abstract data type is modelled by a class of algebraic structures. The signature of an algebra is a syntactic interface to the operations and tests. The abstract data type can be specified by giving a signature and a set of axioms that the operators and relations must satisfy. The algebraic theory of abstract data types is a theory about all data, now and in the future.

What about the models of computation? Thanks to computer science, there is an abundance of computation models, practical and theoretical. Even within the theory of computation, the diversity is daunting: so many motivations, intuitions and technical developments.[1] In this chapter we will give an introduction to *one* abstract model of computation for an arbitrary abstract algebra. Our account will be rather technical and very quick, but it will contain what we think are the key ideas that can launch a whole mathematical theory and sustain its application. The model is a simple form of imperative programming, being an idealised programming language for manipulating data in a store using the constructs of assignments, sequencing, conditionals and iteration. By concentrating on this model the reader will be able to explore and benchmark other models, however complicated or obscure their origins.

To make some sense of the jungle of models of computation, we discuss, in Section 2, two basic types of model, abstract and concrete. The distinction is invaluable when we meet other models, and applications to specific data types, later in Section 8. The first part of this chapter introduces data types modelled by *algebras* (Section 3) and the *imperative* programming model (Section 4), and covers *universality* (Sections 5), and *semicomputability* (Section 6). Here we consider only the case where algebras have *total* operations and tests. Next (Section 7) we look closely at data types with *continuous operations*, such as the real numbers. At this point the theory deepens. New questions arise and there are a number of changes to the imperative model, not least the use of algebras that have *partial* operations and tests and the need for *nondeterministic* constructs. Finally (Section 8) we take a quick look other models and propose some *generalizations of the Church-Turing thesis* to abstract many-sorted algebras.

2. On generalizing computability theory

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 theoretical 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 constructed from any kind of data. Thus, classical computability theory on the set $\mathbb{N}$ of natural numbers is made up of many computability theories (based upon Turing machines, recursive definitions, register machines, etc.).

We divide computability theories into two types:

In an *abstract computability theory* the computations are *independent* of all the representations of the data. Computations are *uniform* over all representations and are necessarily isomorphism invariant. Typical of abstract models of computation

[1]At one time it was possible for us to investigate most of the mathematical models, and compare and classify them [TZ88, TZ00]!

are models based on abstract ideas of *program*, *equation*, *recursion scheme*, or *logical formula*.

In a *concrete computability theory* the computations are *dependent* on *some* data representation. Computations are *not* uniform, and different representations can yield different results. Computations are not automatically isomorphism invariant. Typical of concrete models of computation are those based on concrete ideas of *coding*, *numbering*, or *data representations* using numbers or functions.

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 and tests on these data.

Often we also have in mind the ways these data are

(*iii*) axiomatically specified, and

(*iv*) represented or implemented.

To choose a computation model, we must think carefully 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.

We propose the working principle:

> *Any computability theory should be focused equally on the data types 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 involves 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 and structures. From the beginning there has been a great interest in comparisons between different kinds of numberings, for example in Mal′cev's theory of computability on sets and structures [Mal71]. The study of different notions of reduction and equivalence of numberings, and the space 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[2] [EGNR98, SHT99]. 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 enumerated structures, 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], was not concerned with the choice of numberings.

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 if one represents the reals by infinite decimal expansions then one cannot even compute addition (consider, e.g.,

[2] For example: for all n there exists a computable algebra with precisely n inequivalent computable numberings.

$0.333\cdots+0.666\ldots$). But if one chooses the Cauchy sequence representations then a great deal is computable [BH02].

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

Let $\boldsymbol{AbstComp}_A(D)$ 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 $\boldsymbol{ConcComp}_R(D)$ be the set of functions on D that are computable in a concrete model of computation with representation R.

- **Soundness**: An abstract model of computation $\boldsymbol{AbstComp}_A(D)$ is *sound* for a concrete model of computation $\boldsymbol{ConcComp}_R(D)$ if
$$\boldsymbol{AbstComp}_A(D) \ \subseteq\ \boldsymbol{ConcComp}_R(D).$$
- **Adequacy**: An abstract model of computation $\boldsymbol{AbstComp}_A(D)$ is *adequate* for a concrete model of computation $\boldsymbol{ConcComp}_R(D)$ if
$$\boldsymbol{ConcComp}_R(D) \ \subseteq\ \boldsymbol{AbstComp}_A(D).$$
- **Completeness**: An abstract model of computation $\boldsymbol{AbstComp}_A(D)$ is *complete* for a concrete model of computation $ConcComp_R D$ if it is both sound and adequate, i.e.,
$$\boldsymbol{AbstComp}_A(D) \ =\ \boldsymbol{ConcComp}_R(D).$$

As an example for the l.h.s. here, let us take the data set $D=\mathbb{R}$, the set of reals, the structure $A=\mathcal{R}_{\mathsf{p}}^N$, the partial algebra $\mathcal{R}_{\mathsf{p}}$ of reals defined below in Section 7, with the naturals adjoined, and $\boldsymbol{AbstComp}_A(D) = \boldsymbol{While}^*\,(\mathcal{R}_{\mathsf{p}}^N)$, the set of functions on $\mathbb{R}$ definable by the $\boldsymbol{While}$ programming language with arrays over $\mathcal{R}_{\mathsf{p}}^N$ defined in Section 5. For the r.h.s., take a standard enumeration $\alpha\colon \mathbb{N}\approx\mathbb{Q}$ of the rationals, which generates a representation $\overline{\alpha}$ of the computable reals (as described in §7.1.1 below), and let $\boldsymbol{ConcComp}_{\overline{\alpha}}(\mathbb{R})$ be the corresponding "$\overline{\alpha}$-tracking" model. Then our abstract model is sound, but not adequate, for this concrete model:
$$\boldsymbol{While}^*\,(\mathcal{R}_{\mathsf{p}}) \ \subsetneqq\ \boldsymbol{ConcComp}_{\overline{\alpha}}(\mathbb{R}).$$
On the other hand, if we take for our abstract model over $\mathcal{R}_{\mathsf{p}}$ the non-deterministic $\boldsymbol{WhileCC}^*$ ($\boldsymbol{While}$ + "countable choice" + arrays) language, and further replace "computability" by "approximable computability" [TZ04, TZ05], then we obtain completeness (see §7.3 below):
$$\boldsymbol{WhileCC}^*\text{-approx}(\mathcal{R}_{\mathsf{p}}) \ =\ \boldsymbol{ConcComp}_{\overline{\alpha}}(\mathbb{R}). \tag{1}$$

3. *While* computation on standard many-sorted total algebras

We will study a number of high level imperative programming languages based on the '`while`' construct, applied to a many-sorted signature Σ. We give semantics for these languages relative to a total Σ-algebra A, and define the notions of *computability, semicomputability* and *projective semicomputability* for these languages on A. Much of the material is taken from [TZ00].

We begin by reviewing basic concepts: many-sorted signatures and algebras. Next we define the syntax and semantics of the $\boldsymbol{While}$ programming language. Then we extend this language with special programming constructs to form two new languages: $\boldsymbol{While}^N$ and $\boldsymbol{While}^*$.

3.1. **Basic concepts: Signatures and partial algebras.**

A many-sorted signature Σ is a pair $\langle \boldsymbol{Sort}(\Sigma), \boldsymbol{Func}(\Sigma)\rangle$ where

(a) $\boldsymbol{Sort}(\Sigma)$ is a finite set of basic types called *sorts* $s, s', \ldots$.

(b) $\boldsymbol{Func}(\Sigma)$ is a finite set of basic *function symbols*

$$F\colon s_1 \times \cdots \times s_m \to s \qquad\qquad (m \geq 0)$$

The case $m = 0$ gives a *constant symbol*; we then write $F\colon\ \to s$.

A *product type* has the form $s_1 \times \cdots \times s_m$ $(m \geq 0)$, where $s_1, \ldots, s_m$ are sorts. We write $u, v, \ldots$ for product types. A *function type* has the form $u \to s$, where u is a product type.

A Σ-algebra A has, for each Σ-sort s, a non-empty set A_s, the carrier of sort s, and for each Σ-function symbol $F\colon s_1 \times \cdots \times s_m \to s$, a (basic) function

$$F^A : A^u \to A_s$$

where $u = s_1 \times \cdots \times s_m$, and $A^u = A_{s_1} \times \cdots \times A_{s_m}$.

We write $\Sigma(A)$ for the signature of an algebra A.

A Σ-algebra is called *total* if all the basic functions are total; it is called *partial* in the absence of such an assumption. Sections 3 to 6 will be devoted to total algebras. In Section 7 we will turn to a more general theory, with partial algebras.

Example 3.1.1 (Booleans)**.** The signature $\Sigma(\mathcal{B})$ of the booleans is

signature	$\Sigma(\mathcal{B})$
sorts	bool
functions	true, false: $\to$ bool,
	not: bool $\to$ bool
	or, and: bool^2 $\to$ bool

The algebra $\mathcal{B}$ of booleans contains the carrier $\mathbb{B} = \{\mathtt{t}, \mathtt{f}\}$ of sort bool, and the standard interpretations of the constant and function symbols of $\Sigma(\mathcal{B})$.

Note that for a structure A to be useful for computational purposes, it should be susceptible to testing, which means it should contain the carrier $\mathbb{B}$ of booleans and the standard boolean operations; in other words, it should contain the algebra $\mathcal{B}$ as a retract. Such an algebra A is called *standard.* All the examples of algebras discussed below will be standard.

Example 3.1.2 (Naturals)**.** The signature $\Sigma(\mathcal{N})$ of the naturals is

signature	$\Sigma(\mathcal{N})$
import	$\Sigma(\mathcal{B})$
sorts	nat
functions	0: $\to$ nat,
	suc: nat $\to$ nat
	eq$_\mathsf{N}$, less$_\mathsf{N}$: nat^2 $\to$ bool

The algebra $\mathcal{N}$ of naturals consists of the carrier $\mathbb{N} = \{0, 1, 2, \ldots\}$ of sort nat, the carrier $\mathbb{B}$ of sort bool, and the standard constants and functions $0_\mathsf{N}\colon\ \to \mathbb{N}$, $\mathsf{suc}_\mathsf{N} : \mathbb{N} \to \mathbb{N}$, and $\mathsf{eq}_\mathsf{N}, \mathsf{less}_\mathsf{N}\colon \mathbb{N}^2 \to \mathbb{B}$ (apart from the standard boolean operations).

We will use the infix notations '$=$' and '$<$' for 'eq$_\mathsf{N}$' and 'less$_\mathsf{N}$'. and also use '$\vee$' and '$\wedge$' for the boolean operations 'or' and 'and'.

Example 3.1.3 (Total algebra of reals)**.** The signature $\Sigma(\mathcal{R}_\mathsf{t})$ of the total algebra of reals is:

signature	$\Sigma(\mathcal{R}_\mathsf{t})$
import	$\Sigma(\mathcal{B})$
sorts	real
functions	$0, 1 : \ \to \mathsf{real}$,
	$+, \times : \mathsf{real}^2 \to \mathsf{real}$,
	$- : \mathsf{real} \to \mathsf{real}$,
	$\mathsf{eq}_\mathsf{R}, \mathsf{less}_\mathsf{R} : \mathsf{real}^2 \to \mathsf{bool}$

(We will study a partial algebra of reals in Section 7.) The algebra $\mathcal{R}_\mathsf{t}$ of reals has the carrier $\mathbb{R}$ of sort real, as well as the imported carrier $\mathbb{B}$ of sort bool with the boolean operations, the real constants and operations $(0, 1, +, \times, -)$, and the (total) boolean-valued functions $\mathsf{eq}_\mathsf{R} : \mathbb{R}^2 \to \mathbb{B}$. and $\mathsf{less}_\mathsf{R} : \mathbb{R}^2 \to \mathbb{B}$. Again, we will use the infix notations '=' and '<' for these.

Definition 3.1.4 (Minimal carriers; minimal algebra)**.**
Let A be a Σ-algebra, and s a Σ-sort.

(a) A is *minimal at* s (or the carrier A_s is *mimimal in* A) if A_s is generated by the closed Σ-terms of sort s.
(b) A is *minimal* if it is minimal at every Σ-sort.

To take two examples:

- Every N-standard algebra (see §3.3) is minimal at sorts bool and nat.
- The algebra $\mathcal{R}_\mathsf{t}$ of reals (Example 3.1.3) is not minimal at sort real.

3.2. **Syntax and semantics of of Σ-terms.**

For a signature Σ, the set $\boldsymbol{Tm}(\Sigma)$ of Σ-terms is defined from Σ-variables $\mathtt{x}^s, \ldots$ of sort s (for all Σ-sorts s) by

$$t^s ::= \mathtt{x}^s | F(t_1^{s_1}, \ldots, t_m^{s_m})$$

where F is a Σ-function symbol of type $s_1 \times \cdots \times s_m \to \ s$.

We write $t : s$ to indicate that t is a Σ-term of sort s, and more generally, $t : u$ to indicate that t is a tuple of terms of product type u. We also write $b, \ldots$ for boolean Σ-terms, i.e. Σ-terms of sort bool.

We turn to the semantics of terms.

A *state* over an algebra A is a family $\langle \sigma_s \mid s \in \boldsymbol{Sort}(\Sigma) \rangle$ of functions $\sigma_s : \boldsymbol{Var}_s \to A_s$ (where $\boldsymbol{Var}_s$ is the set of variables of sort s). Let $\boldsymbol{State}(A)$ be the set of states on A. We will write $\sigma(\mathtt{x})$ for $\sigma_s(\mathtt{x})$ where $\mathtt{x} : s$. Also, for a tuple $\mathtt{x} \equiv (\mathtt{x}_1, \ldots, \mathtt{x}_m)$, we write $\sigma[\mathtt{x}]$ for $(\sigma(\mathtt{x}_1), \ldots, \sigma(\mathtt{x}_m))$.

Let σ be a state over A, and for some Σ-product type u, let $\mathtt{x} \equiv (\mathtt{x}_1, \ldots, \mathtt{x}_n) : u$ and $a = (a_1, \ldots, a_n) \in A^u$ (for $n \geq 1$). We define the *variant* $\sigma\{\mathtt{x}/a\}$ to be the state over A formed from σ by replacing its value at $\mathtt{x}_i$ by a_i for $i = 1, \ldots, n$.

For a term $t : s$, we will define the function

$$[\![t]\!]^A : \ \boldsymbol{State}(A) \to A_s$$

where $[\![t]\!]^A \sigma$ is the value of t in A at state σ.

The definition of $[\![t]\!]^A \sigma$ is by structural induction on Σ-terms t:

$$\begin{aligned} [\![\mathtt{x}]\!]^A\sigma &= \sigma(\mathtt{x}) \\ [\![F(t_1,\ldots,t_m)]\!]^A\sigma &= F^A([\![t_1]\!]^A\sigma,\ldots,[\![t_m]\!]^A\sigma) \end{aligned} \tag{2}$$

3.3. Adding counters: N-standard signatures and algebras.

A signature Σ is *N-standard* if (i) it is standard (see §3.1), and (ii) it contains the standard signature of naturals (Example 3.1.2), i.e., $\Sigma(\mathcal{N}) \subseteq \Sigma$.

Given an N-standard signature Σ, a Σ-algebra A is *N-standard* if it is an expansion of $\mathcal{N}$, i.e., it contains the carrier $\mathbb{N}$ with the standard arithmetic operations.

N-standardness is clearly very useful in computation, with the presence of counters, and the ability for enumerations and numerical coding.

Any standard signature Σ can be "N-standardised" to a signature Σ^N by adjoining the sort nat and the operations 0, suc, $\mathsf{eq_N}$ and $\mathsf{less_N}$. 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 arithmetic and boolean functions on $\mathbb{N}$.

3.4. Adding arrays: Algebras A^* of signature Σ^*.

Given a standard signature Σ, and standard Σ-algebra A, we expand Σ and A in two stages: (1) N-standardise these to form Σ^N and A^N, as in §3.3; and (2) define, for each sort s of Σ, the carrier A^*_s to be the set of *finite sequences* or *arrays* a^* over A_s, of "starred sort" s^*.

The resulting algebras A^* have signature Σ^*, which extends Σ^N by including, for each sort s of Σ, the new starred sorts s^*, and certain new function symbols to read and update arrays. Details are given in [TZ99, TZ00].

We conclude this section with a very useful syntactic conservativity theorem, which says that that every Σ^*-term with sort in Σ is effectively semantically equivalent to a Σ-term. This theorem will be used later for proving universality for ***While**** computations by a ***While**^{N}* procedure (Theorem 3*) and deriving a strong form of Engeler's Lemma (Theorem 8).

Theorem 1 (Σ^*/Σ conservativity for terms). *For every Σ-sort s, every Σ^*-term t of sort s without any variables of starred sort is effectively semantically equivalent*[3] *to a Σ-term.*

4. The ***While*** programming language

Note that we will use '$\equiv$' to denote syntactic identity between two expressions.

We define ***Stmt***(Σ) to be the class of ***While***(Σ)-statements $S,\ldots$ generated by:

$$S ::= \mathsf{skip} \mid \mathtt{x} := t \mid S_1\ ; S_2 \mid \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \mid \mathsf{while}\ b\ \mathsf{do}\ S_0\ \mathsf{od}$$

where the variable $\mathtt{x}$ and term t have the same Σ-sort.

Proc(Σ) is the class of ***While***(Σ)-procedures $P,\ldots$, of the form:

$$P \equiv \mathsf{proc\ in\ a\ out\ b\ aux\ c\ begin}\ S\ \mathsf{end} \tag{3}$$

where S is the body, and and a, b and c are tuples of (distinct) input, output and auxiliary variables respectively.

If $\mathsf{a} : u$ and $\mathsf{b} : v$, then P is said to have type $u \to v$, written $P\colon u \to v$.

We turn to the semantics of statements and procedures.

[3] i.e. we can effectively find a Σ-term t' such that $[\![t']\!]^A\sigma = [\![t]\!]^{A^*}\sigma$ for all Σ-algebras A and states σ over A^* (or A).

The meaning $[\![S]\!]^A$ of a statement S is a partial state transformer [4] on an algebra A:

$$[\![S]\!]^A : \ \textbf{\textit{State}}(A) \rightharpoonup \textbf{\textit{State}}(A).$$

Its definition is standard [TZ99, TZ00] and lengthy, and so we omit it. Briefly, it is based on defining the *computation sequence* of states from S starting in a state σ, or rather the n-th component of this sequence, by a primary induction on n, and a secondary induction on the size of S.

Next, given a procedure (3) of type $u \to v$, its meaning is a partial function $P^A : A^u \rightharpoonup A^v$ defined as follows. For $a \in A^u$, let σ be any state on A such that $\sigma[\mathsf{a}] = a$, and $\sigma[\mathsf{b}]$ and $\sigma[\mathsf{c}]$ are given preassigned default values. Then

$$P^A(a) \ \simeq \ \begin{cases} \sigma'[b] & \text{if } \ [\![S]\!]^A\sigma \downarrow \sigma' \quad \text{(say)} \\ \uparrow & \text{if } \ [\![S]\!]^A\sigma \ \uparrow. \end{cases}$$

Here '$\simeq$' means that the two sides either both converge to the same value, or both diverge ("Kleene equality" [Kle52, §63]).

We are also using the notation $[\![S]\!]^A\sigma\downarrow$ to mean that evaluation of $[\![S]\!]^A$ at σ halts or converges; $[\![S]\!]^A\sigma\downarrow\sigma'$ that it converges to σ', and $[\![S]\!]^A\sigma\uparrow$ that it diverges.

It is worth noting that the semantics of ***While***(Σ) procedures is invariant under Σ-isomorphism.

Modifications in these semantic definitions ((2),(3)) required for partial algebras will be indicated in Section 7 (Remark 7.3.1).

4.1. *While*, $\textbf{\textit{While}}^N$ and $\textbf{\textit{While}}^*$ computability.

A (partial) function f on A is ***While*** *computable* if $f = P^A$ for some ***While*** procedure P.

Consider now the ***While*** programming language over Σ^N and Σ^*.

A $\textbf{\textit{While}}^N(\Sigma)$ procedure is a $\textbf{\textit{While}}(\Sigma^N)$ procedure in which the input and output variables have sorts in Σ. However the *auxiliary* variables may have sort nat.

Similarly, a $\textbf{\textit{While}}^*(\Sigma)$ procedure is a $\textbf{\textit{While}}(\Sigma^*)$ procedure in which the input and output variables have sorts in Σ. However the *auxiliary* variables may have starred sorts.

A function f on A is $\textbf{\textit{While}}^N$ (or $\textbf{\textit{While}}^*$) *computable* if $f = P^A$ for some $\textbf{\textit{While}}^N$ (or $\textbf{\textit{While}}^*$) procedure P.

We write $\textbf{\textit{While}}(A)$, $\textbf{\textit{While}}^N(A)$ and $\textbf{\textit{While}}^*(A)$ for the classes of functions ***While***, $\textbf{\textit{While}}^N$ and $\textbf{\textit{While}}^*$ computable on A.

Remarks 4.1.1.

(a) Clearly, if A is N-standard, then $\textbf{\textit{While}}^N$ computability coincides with ***While*** computability on A.

(b) Because of the effective enumeration of the set $\mathbb{N}^*$, $\textbf{\textit{While}}^N$ and $\textbf{\textit{While}}^*$ computability coincide with ***While*** computability on $\mathcal{N}$, which is in turn equivalent to classical *partial recursiveness* over $\mathbb{N}$.

(c) $\textbf{\textit{While}}^*$ computability will be the basis for a generalized Church-Turing Thesis, as we will see later (§8.2). On the other hand, $\textbf{\textit{While}}^N$ computability is useful for representing the syntax of ***While*** programming within the formalism, by means of coding (Section 5)).

[4] '$\rightharpoonup$' denotes a partial function

5. Representations of semantic functions; Universality

We examine whether the $\boldsymbol{While}$ programming language is a so-called *universal model* of computation. This means answering questions of the following form. Let A be a standard Σ-algebra.

Is there a universal $\boldsymbol{While}$ *procedure* $\boldsymbol{U}_{\mathsf{proc}} \in \boldsymbol{Proc}(\Sigma)$ *that can compute all the* $\boldsymbol{While}$ *computable functions on* A*?*

To this end we need the techniques of numerical codings (Gödel numberings) and symbolic computations on terms. More accurately, for this to be possible, we need the sort nat, and so we will consider the possibility of representing the syntax of a standard Σ-algebra A (not in A itself, but) in its N-standardisation A^N, or (failing that) in the array algebra A^*. We will see that

For any given Σ*-algebra* A*, there is a universal* $\boldsymbol{While}$ *procedure over* A *if, and only if, there is a* $\boldsymbol{While}$ *program for term evaluation over* A.

Consequently, since term evaluation is always $\boldsymbol{While^*}$ computable on A, we have

For any Σ*-algebra* A *there is a universal* $\boldsymbol{While^*}$ *program and universal* $\boldsymbol{While^*}$ *procedure over* A.

Thus, for any algebra A our $\boldsymbol{While^*}$ model of computation is universal.

Hence, if the Σ-algebra A has a $\boldsymbol{While}$ program to compute term evaluation, then

$$\boldsymbol{While}^*(A) \;=\; \boldsymbol{While}^N(A).$$

5.1. Numbering of syntax.

We assume given families of effective numerical codings of the syntactic classes E with which we deal, i.e. 1-1 maps $\boldsymbol{code}\colon E \hookrightarrow \mathbb{N}$, with $\ulcorner e \urcorner = \boldsymbol{code}(e)$ denoting the code of the expression $e \in E$. Further, we assume standard effective numberings of sets such as $\mathbb{N}^*$, $\mathbb{Q}^2$, etc. Hence we assume that we can primitive recursively simulate all operations involved in processing the syntax of the programming language.

By "effective(ly)", we will mean *effective* in the codes of the syntactic or mathematical objects referred to.

We will use the notation

$$\ulcorner \boldsymbol{Tm} \urcorner \;=\; \{\ulcorner t \urcorner \mid t \in \boldsymbol{Tm}\},$$

etc., for sets of codes of syntactic expressions.

5.2. Representation of states.

We will be interested in the representation of various semantic functions on syntactic classes such as $\boldsymbol{Tm}(\Sigma)$, $\boldsymbol{Stmt}(\Sigma)$ and $\boldsymbol{Proc}(\Sigma)$ by functions on A or A^*, and in the computability of these representing functions. These semantic functions have states as arguments, so we must first define a representation of states.

Let $\mathtt{x}$ be a u-tuple of program variables. A state σ on A is *represented* (relative to $\mathtt{x}$) by a tuple of elements $\boldsymbol{a} \in A^u$ if $\sigma[\mathrm{x}] = \boldsymbol{a}$.

The *state representing function*

$$\boldsymbol{Rep}^A_{\mathtt{x}} :\; \boldsymbol{State}(A) \;\to\; A^u$$

is defined by

$$\boldsymbol{Rep}^A_{\mathtt{x}}(\sigma) \;=\; \sigma[\mathrm{x}].$$

$$\begin{tikzcd}
\boldsymbol{Tm}_{\mathtt{x},s} \times \boldsymbol{State}(A) \arrow[d, "\langle \boldsymbol{code}, \boldsymbol{Rep}^A_{\mathtt{x}} \rangle"'] \arrow[dr, "\boldsymbol{TE}^A_{\mathtt{x},s}"] & \\
\ulcorner \boldsymbol{Tm}_{\mathtt{x},s} \urcorner \times A^u \arrow[r, "\boldsymbol{te}^A_{\mathtt{x},s}"'] & A_s
\end{tikzcd}$$

FIGURE 1. Term evaluation representing function

5.3. **Representation of term evaluation; Term evaluation property.** Let $\mathtt{x}$ be a u-tuple of variables. Let $\boldsymbol{Tm}_{\mathtt{x}} = \boldsymbol{Tm}_{\mathtt{x}}(\Sigma)$ be the class of all Σ-terms with variables among $\mathtt{x}$ only, and for all sorts s of Σ, let $\boldsymbol{Tm}_{\mathtt{x},s} = \boldsymbol{Tm}_{\mathtt{x},s}(\Sigma)$ be the class of such terms of sort s.

The *term evaluation function on A relative to* $\mathtt{x}$

$$\boldsymbol{TE}^A_{\mathtt{x},s} : \ \boldsymbol{Tm}_{\mathtt{x},s} \times \boldsymbol{State}(A) \ \to \ A_s,$$

defined by

$$\boldsymbol{TE}^A_{\mathtt{x},s}(t,\sigma) \ = \ [\![t]\!]^A\sigma,$$

is *represented* by the function

$$\boldsymbol{te}^A_{\mathtt{x},s} : \ \ulcorner \boldsymbol{Tm}_{\mathtt{x},s} \urcorner \times A^u \ \to \ A_s$$

defined by

$$\boldsymbol{te}^A_{\mathtt{x},s}(\ulcorner t \urcorner,\, a) \ = \ [\![t]\!]^A\sigma,$$

where σ is any state on A such that $\sigma[\mathtt{x}] = a$, in the sense that the diagram in Figure 1 commutes.

We will be interested in the computability of this term evaluation representing function.

Definition 5.3.1 (Term evaluation)**.** The algebra A has the *term evaluation property (TEP)* if for all $\mathtt{x}$ and s, the term evaluation representing function $\boldsymbol{te}^A_{\mathtt{x},s}$ is ***While*** computable on A^N.

Many well-known varieties (i.e., equationally axiomatisable classes of algebras) have (uniform versions of) the TEP. Examples are: semigroups, groups, and associative rings with or without unity. This follows from the *effective normalisability* of the terms of these varieties. In the case of rings, this means an effective transformation of arbitrary terms to polynomials.

Thus, for example, the algebra $\mathcal{R}_{\mathrm{t}}$ of reals has the TEP.

Theorem 2.
The term evaluation representing function on A^* is ***While*** *computable on A^*.*

Corollary 5.3.2.
The term evaluation representing function on A is ***While**** *computable on A^N.*

Recall the definition (3.1.4) of minimal carriers.

Corollary 5.3.3.

(a) *If A is minimal at s, then there is a* $\boldsymbol{While}^*$ *computable enumeration* (*or listing*) *of the carrier A_s, i.e., a surjective total mapping*

$$\boldsymbol{enum}_s^A : \mathbb{N} \twoheadrightarrow A_s,$$

which is $\boldsymbol{While}^*$ *computable on A^N.*

(b) *If in addition A has the TEP, then* $\boldsymbol{enum}_s^A$ *is also* $\boldsymbol{While}$ *computable on A^N.*

Theorem 3 (Universality characterization theorem for $\boldsymbol{While}$ computations)**.** *The following are equivalent:*

(i) *A has the TEP;*

(ii) *For all Σ-product types u, v, there is a* $\boldsymbol{While}(\Sigma^N)$ *procedure*

$$\mathsf{Univ}_{u \to v} : \mathsf{nat} \times u \to v$$

which is universal for $\boldsymbol{Proc}_{u \to v}$ on A, in the sense that for all $P \in \boldsymbol{Proc}_{u \to v}$ and $\boldsymbol{a} \in A^u$,

$$\mathsf{Univ}^A_{u \to v}(\ulcorner P \urcorner, a) \simeq P^A(a).$$

Using the Σ^*/Σ conservativity theorem (Theorem 1), we can strengthen the above theorem, so as to construct a universal $\boldsymbol{While}(\Sigma^N)$ procedure for $\boldsymbol{While}^*$ computation,

Theorem 3* (Universality characterization theorem for $\boldsymbol{While}^*$ computations)**.** *The following are equivalent.*

(i) *A has the TEP.*

(ii) *For all Σ-product types u, v, there is a* $\boldsymbol{While}(\Sigma^N)$ *procedure*

$$\mathsf{Univ}_{u \to v} : \mathsf{nat} \times u \to v$$

which is universal for $\boldsymbol{Proc}^*_{u \to v}$ on A, in the sense that for all $P \in \boldsymbol{Proc}^*_{u \to v}$ and $\boldsymbol{a} \in A^u$,

$$\mathsf{Univ}^A_{u \to v}(\ulcorner P \urcorner, a) \simeq P^A(a).$$

We conclude that there are universal $\boldsymbol{While}^N$ procedures for $\boldsymbol{While}^*$ computation on $\mathcal{R}_\mathsf{t}$.

6. Concepts of semicomputability

We want to generalize the notion of *recursive enumerability* to many-sorted algebras. There turn out to be many non-equivalent ways to do this.

The primary idea is that a set is $\boldsymbol{While}$ semicomputable if, and only if, it is the domain or halting set of a $\boldsymbol{While}$ procedure; and similarly for $\boldsymbol{While}^N$ and $\boldsymbol{While}^*$ semicomputability.

This concept satisfies the standard closure properties (under finite union and intersection) and also Post's Theorem:

A set is computable if, and only if, it and its complement are semicomputable.

The second idea of importance is that of a *projection* of a computable or semicomputable set. set. In classical computability theory on $\mathbb{N}$, the class of semicomputable sets is closed under projections, but this is not true in the general case of algebras, as we will see. Projective semicomputability is strictly more powerful (and less algorithmic) than semicomputability.

We will also characterize the semicomputable sets as the sets definable by some effective countable disjunction of boolean valued terms. This result, first observed by E. Engeler, has a number of interesting applications.

Definition 6.0.4.

(a) R is ***While*** *computable on* A if its characteristic function is.
(b) R is ***While*** *semicomputable* on A if it is the halting set on A of some ***While*** procedure P, i.e., $R = \{a \in A^u \mid P^A(a)\downarrow\}$.

Examples 6.0.5.
We will have need for the notation $\mathcal{R}_\mathsf{t}^\circ$ to indicate the algebra $\mathcal{R}_\mathsf{t}$ without the '$<$' operation.

(a) On the naturals $\mathcal{N}$ the ***While*** semicomputable sets are precisely the recursively enumerable sets, and the ***While*** computable sets are precisely the recursive sets.
(b) On $\mathcal{R}_\mathsf{t}^\circ$ the set of *naturals* (as a subset of $\mathbb{R}$) is ***While*** semicomputable, being the halting set of the following procedure:

```
is_nat ≡ proc in x : real
             begin while not x = 0
                       do x := x − 1 od
             end
```

(c) Similarly, the set of *integers* is ***While*** semicomputable on $\mathcal{R}_\mathsf{t}^\circ$.
(d) However, the sets of naturals and integers are ***While*** *computable* on $\mathcal{R}_\mathsf{t}$, as can be easily seen.
(e) The set of *rationals* is ***While*** semicomputable on $\mathcal{R}_\mathsf{t}^\circ$. (*Exercise.* Hint: Prove this first for $\mathcal{R}_\mathsf{t}^{\circ N}$.)

6.1. Merging two procedures; Closure theorems.

The classical "merge" theorems generalize:

Theorem 4. *The union and intersection of two* ***While*** *semicomputable relations of the same type are again* ***While*** *semicomputable.*

In the case of union, if we assume that (*i*) A is N-standard, and (*ii*) A has the TEP. then the construction of the "merge" of the two characteristic procedures, i.e., interleaving their steps to form the new procedure, simply follows the classical proof for computation on $\mathbb{N}$. Failing this, the construction of the merge procedure (by structural induction on the pair of statements) is quite challenging. (The tricky case is where both are 'while' statements.)

If R is a relation on A of type u, we write the *complement of* R as $R^\mathsf{c} = A^u \backslash R$.

Theorem 5 (Post's Theorem for ***While*** semicomputability)**.**
For any relation R on A

$$R \text{ is } \textbf{\textit{While}} \text{ computable} \iff R \text{ and } R^\mathsf{c} \text{ are } \textbf{\textit{While}} \text{ semicomputable.}$$

Note that the proofs of the above two theorems depend strongly on the totality of A. (See Remark 7.3.2.)

Another useful closure result, applicable to N-standard structures, is:

Theorem 6 (Closure of ***While*** semicomputability under $\mathbb{N}$-projections)**.** *Suppose A is N-standard. If $R \subseteq A^{u\times\mathsf{nat}}$ is **While** semicomputable on A, then so is its $\mathbb{N}$-projection* $\{x \in A^u \mid \exists n \in \mathbb{N}\, R(x,n)\}$.

To outline the proof: From a procedure P which halts on R, we can effectively construct another procedure which halts on the required projection. Briefly, for input x, we search by "dovetailing" for a number n such that P halts on (x,n).

We can generalize this theorem to the case of an A_s-projection for any *minimal carrier* A_s (recall Definition 3.1.4), provided A has the TEP.

Corollary 6.1.1 (Closure of ***While*** semicomputability under projections off minimal carriers)**.** *Suppose A is N-standard and has the TEP. Let A_s be a minimal carrier of A. If $R \subseteq A^{u\times s}$ is **While** semicomputable on A, then so is its projection off A_s.*

Note that Corollary 6.1.1 is a many-sorted version of (part of) Theorem 2.4 of [Fri71], cited in [She85]. The minimality condition (a version of Friedman's Condition III) means that *search* in A_s is *computable* (or, more strictly, *semicomputable*) provided A has the TEP. Thus in minimal algebras, many of the results of classical recursion theory carry over, e.g.,

- the domains of semicomputable sets are closed under projection (as above)
- a semicomputable relation has a computable selection function
- a function with semicomputable graph is computable [Fri71, Thm 2.4].

If, in addition, there is *computable equality* at the appropriate sorts, other results of classical recursion theory carry over, e.g.,

- the range of a computable function is semicomputable [Fri71, Thm 2.6].

6.2. Projective *While* semicomputability and computability.

A set $R \subseteq A^u$ is *projectively **While** semicomputable* (or *computable*) on A iff R is a projection of a ***While*** semicomputable (or computable) set on A, i.e., for some product types u and v,

$$\forall x \in A^u \; \big[x \in R \iff \exists y \in A^v : (x,y) \in R'\big].$$

where R' is a semicomputable (or computable) subset of $A^{u\times v}$.

We note that although the emphasis in this subsection is on projective *semi*computability, the concept of projective *computability* will be used in our formulation of a generalized Church-Turing thesis for specifiability (in Section 8).

In this connection we note further that

(1) The concepts of projective ***While**** semicomputability and projective ***While**** computability coincide, by the projective equivalence theorem (Theorem 10 below).

(2) Projective ***While***$^{(*)}$ semicomputability is, in general, a broader concept than ***While***$^{(*)}$ semicomputability (§6.7).

We do, however, have closure of semicomputability in the case of $\mathbb{N}$-projections, i.e., existential quantification over $\mathbb{N}$, as we saw in Theorem 6. Further, we have from Corollary 6.1.1:

Theorem 7. *Suppose A is N-standard and minimal and has the TEP. Then on A*

*projective **While** semicomputability = **While** semicomputability.*

6.3. $\boldsymbol{While}^*$ semicomputability.

A relation R on A is $\boldsymbol{While}^*$ *semicomputable* if it is the halting set of some $\boldsymbol{While}(\Sigma^*)$ procedure on A^*.

Again, we have Post's Theorem for $\boldsymbol{While}^*$ computability and semicomputability, and again, we have closure of $\boldsymbol{While}^*$ semicomputability under $\mathbb{N}$-projections, and projections off minimal carriers.

Note that we do not have to assume the TEP for the latter (cf. Corollary 6.1.1), since the term evaluation representing function is always $\boldsymbol{While}^*$ computable.

Example 6.3.1. The subalgebra relation[5] :

$$\boldsymbol{subalg}^A(x,y) \iff x \text{ is in the subalgebra of } A \text{ generated by } y$$

is $\boldsymbol{While}^*$ semicomputable on A. This follows from $\boldsymbol{While}^*$ computability of term evaluation on A^N (Corollary 5.3.2).

6.4. Projective $\boldsymbol{While}^*$ semicomputability.

A relation R on A is said to be *projectively* $\boldsymbol{While}^*$ *computable* (or *semicomputable*) on A if R is a projection of a $\boldsymbol{While}(\Sigma^*)$ computable (or semicomputable) relation on A^*.

Theorem 7 can be re-stated for $\boldsymbol{While}^*$ semicomputability:

Theorem 7*. *Suppose A is a minimal. Then on A*

$$\textit{projective } \boldsymbol{While}^* \textit{ semicomputability } = \boldsymbol{While}^* \textit{ semicomputability.}$$

Note again that the TEP does not have to be assumed here (cf. Theorem 7). Also we are using the fact that if A is minimal then so is A^*.

Example 6.4.1. In $\mathcal{N}$, the various concepts we have listed: $\boldsymbol{While}$, $\boldsymbol{While}^N$ and $\boldsymbol{While}^*$ semicomputability, as well as projective $\boldsymbol{While}$, $\boldsymbol{While}^N$ and $\boldsymbol{While}^*$ semicomputability, all reduce to *recursive enumerability* over $\mathbb{N}$.

In general, however, projective $\boldsymbol{While}^*$ semicomputability is strictly broader than projective $\boldsymbol{While}^N$ semicomputability. In other words, projecting along starred sorts is stronger than projecting along simple sorts or nat. (Intuitively, this corresponds to existentially quantifying over a finite, but unbounded, sequence of elements.) An example to show this will be given below.

We do, however, have the following equivalence:

$$\textit{projective } \boldsymbol{While}^* \textit{ semicomputability } = \textit{ projective } \boldsymbol{While}^* \textit{ computability.}$$

This is the projective equivalence theorem for $\boldsymbol{While}^*$ (Theorem 10).

6.5. Computation trees; Engeler's Lemma.

For any $\boldsymbol{While}$ statement S over Σ, we can define a (possibly infinite) *computation tree* for S. The construction is by strutural induction on S. Details are given in [TZ00].

Using this and the Σ/Σ^* conservativity theorem (Theorem 1), we can prove the following. Let R be a relation on A.

Theorem 8 (Engeler's Lemma for $\boldsymbol{While}^*$ semicomputability)**.** *R is $\boldsymbol{While}^*$ semicomputable over A iff R can be expressed as an effective countable disjunction of booleans over Σ.*

[5] We are suppressing sort superscripts here

If, moreover, A has the TEP, then we can say more:

Theorem 9 (Semicomputability equivalence theorem)**.**
Suppose A has the TEP. Then the following assertions are equivalent:

(i) *R is **While**^{N} semicomputable on A;*
(ii) *R is **While*** semicomputable on A.*
(iii) *R can be expressed as an effective countable disjunction of booleans over Σ.*

The step (i)⇒(ii) is trivial, and (ii)⇒(iii) is just Engeler's Lemma for ***While**** . The new step (iii)⇒(i) follows from an analysis of the coding of an effective infinite disjunction.

Corollary 6.5.1. *Suppose A has the TEP. Then the following are equivalent:*

(i) *R is **While*** computable on A;*
(ii) *R is **While**^{N} computable on A.*

This follows from the above theorem, and Post's Theorem for ***While***^{N} and ***While**** .

6.6. **Projective equivalence theorem for *While** .**

The following theorem uses Engeler's Lemma, and the ***While**** computability of term evaluation.

Theorem 10 (Projective equivalence theorem)**.** *The following are equivalent:*

(i) *R is projectively **While*** semicomputable on A;*
(ii) *R is projectively **While*** computable on A.*

We can strengthen the theorem with a third equivalent clause, if we add an assumption about computability of equality in A.

First we must define certain syntactic classes of formulae over Σ.

Let ***Lang***$^* =$ ***Lang***(Σ^*) be the first order language with equality over Σ^*. We are interested in special classes of formulae of ***Lang****.

Formulae of ***Lang**** are formed from the atomic formulae by means of the propositional connectives and universal and existential quantification over variables of any Σ^*-sort.

Definition 6.6.1 (Classes of formulae of ***Lang***(Σ^*))**.**

(a) An *atomic formula* is an equality between a pair of terms of the same Σ^*-sort.
(b) A *bounded quantifier* has the form '$\forall k < t$' or '$\exists k < t$', where t : nat.
(c) An *elementary formula* is one with only bounded quantifiers.
(d) A Σ_1^* *formula* is formed by prefixing an elementary formula with existential quantifiers only.
(e) An *extended* Σ_1^* *formula* is formed by prefixing an elementary formula with a string of existential quantifiers and bounded universal quantifiers (in any order).

We can show that an extended Σ_1^* formula is equivalent to a Σ_1^* formula over Σ. Hence we will use the term 'Σ_1^*' to denote (possibly) extended Σ_1^* formulae.

We can now re-state the projective equivalence theorem in the presence of equality.

Theorem 10$^=$ (Projective equivalence theorem for Σ^* with equality)**.**
Suppose Σ has an equality operator at all sorts. Then the following are equivalent:

(i) *R is projectively* ***While**** *semicomputable on A;*
(ii) *R is projectively* ***While**** *computable on A;*
(iii) *R is Σ_1^* definable.*

6.7. Semicomputability and projective semicomutability on $\mathcal{R}_\mathsf{t}$.

We apply some of the above ideas and results to the algebra $\mathcal{R}_\mathsf{t}$. Details can be found in [TZ00, §§6.2, 6.3][6]

We begin again with a restatement of the semicomputability equivalence theorem (Theorem 9), for the particular case of $\mathcal{R}_\mathsf{t}$.

Theorem 11 (Semicomputability for $\mathcal{R}_\mathsf{t}$)**.**
Suppose $R \subseteq \mathbb{R}^n$ $(n = 1, 2, \dots)$. Then the following are equivalent:

(i) *R is* ***While***^{N} *semicomputable on $\mathcal{R}_\mathsf{t}$,*
(ii) *R is* ***While**** *semicomputable on $\mathcal{R}_\mathsf{t}$,*
(iii) *R can be expressed as an effective countable disjunction*

$$x \in R \iff \bigvee_i b_i(x)$$

where each $b_i(x)$ is a finite conjunction of equations and inequalities of the form

$$p(x) = 0 \qquad \text{and} \qquad q(x) > 0,$$

where p, q are polynomials in $x \equiv (x_1, \dots, x_n) \in \mathbb{R}^n$, with coefficients in $\mathbb{Z}$.

We also have:

Theorem 12. *In $\mathcal{R}_\mathsf{t}$, the following three concepts coincide for subsets of $\mathbb{R}^n$:*

(i) ***While***^{N} *semicomputability,*
(ii) ***While**** *semicomputability,*
(iii) *projective* ***While***^{N} *semicomputability.*

The proof of equivalence between (iii) and the other two concepts follows from the fact that semialgebraic sets are closed under projection, which in turn follows from Tarski's quantifier-elimination theorem for real closed fields [KK71, Ch. 4].

Interestingly, in the algebra $\mathcal{R}_\mathsf{t}^\circ$ (i.e. $\mathcal{R}_\mathsf{t}$ without the order relation '$<$'), where Tarski's theorem fails, one can find an example of a relation (namely, '$<$' itself!) which is projectively ***While*** semicomputable, but not ***While*** (or ***While****) semicomputable.

On the other hand (returning to $\mathcal{R}_\mathsf{t}$) the three equivalent concepts of semicomputability given in Theorem 12 differ from a fourth:

(iv) *projective* ***While**** *semicomputability*,

as we now show.

Example 6.7.1 (A set which is projectively ***While**** semicomputable, but not projectively ***While***^{N} semicomputable)**.**
In order to prepare for this example, we must first enrich the structure $\mathcal{R}_\mathsf{t}$. Let $E = \{e_0, e_1, e_2, \dots\}$ be a sequence of reals such that

$$\text{for all } i,\ e_i \text{ is transcendental over } \mathbb{Q}(e_0, \dots, e_{i-1}).$$

[6] The notation in [TZ00] is unfortunately not completely consistent with the present notation: $\mathcal{R}$ and $\mathcal{R}^<$ in [TZ00] correspond (resp.) to $\mathcal{R}_\mathsf{t}^\circ$ and $\mathcal{R}_\mathsf{t}$ here.

We define $\mathcal{R}_{\mathsf{t}}^{E}$ to be the algebra $\mathcal{R}_{\mathsf{t}}$ augmented by the set E as a separate sort E, with the embedding $j : E \hookrightarrow \mathbb{R}$ in the signature, thus:

algebra	$\mathcal{R}_{\mathsf{t}}^{E}$
import	$\mathcal{R}_{\mathsf{t}}$
carriers	E
functions	$j : E \hookrightarrow \mathbb{R}$

We write $\overline{E} \subset \mathbb{R}$ for the real algebraic closure of $\mathbb{Q}(E)$.

It is easy to see that $\overline{E}$ is projectively ***While**** semicomputable in $\mathcal{R}_{\mathsf{t}}^{E}$. (In fact, $\overline{E}$ is the projection on $\mathbb{R}$ of a ***While*** semicomputable relation on $\mathbb{R} \times E^*$.) We must show that, on the other hand, $\overline{E}$ is not projectively $\boldsymbol{While}^N$ semicomputable in $\mathcal{R}_{\mathsf{t}}^{E}$.

Briefly, we proceed by showing that if $F \subseteq \overline{E}$ is any projectively $\boldsymbol{While}^N$ semicomputable set in $\mathcal{R}_{\mathsf{t}}^{E}$, then, using Engeler's Lemma and Tarski's theorem, we can show that F cannot equal $\overline{E}$.

The proof further shows that $\overline{E}$ (although a projection on $\mathbb{R}$ of a ***While*** semicomputable relation on $\mathbb{R} \times E^*$) is not a projection of a $\boldsymbol{While}^N$ semicomputable relation in $\mathcal{R}_{\mathsf{t}}^{E}$. In fact, it can be shown (still using Engeler's Lemma) that $\overline{E}$ is not even a projection of a ***While**** semicomputable relation on $\mathbb{R}^n \times E^m$ (for any $n, m > 0$). Thus to define $\overline{E}$, we must project off the *starred sort* E^*, or (in other words) existentially quantify over a *finite, but unbounded* sequence of elements of E.

7. Computation on topological partial algebras

When one considers the relation between abstract and concrete models, a number of intriguing problems appear. We will explain them by considering a series of examples based on the data type of real numbers. Then we formulate our strategy for solving these problems. The picture for topological algebras in general will be clear from our examples with the reals.

7.1. Abstract versus concrete data types of reals; Continuity; Partiality.

7.1.1. ***Abstract and concrete data types of reals.*** To compute on $\mathbb{R}$ with an abstract model of computation, we have only to select an algebra A in which $\mathbb{R}$ is a carrier set. Abstract computability on $\mathbb{R}$ is then computability on A, and we may apply the general theory of computable functions on many-sorted algebras outlined in the previous sections.

By contrast, to compute on $\mathbb{R}$ with a concrete model of computation (say the tracking model), we first take a standard enumeration of the rationals $\alpha\colon \mathbb{N} \approx \mathbb{Q}$, which in turn yields a representation $\overline{\alpha}\colon C \twoheadrightarrow \mathbb{R}_{\mathsf{c}}$ that maps the set $C \subset \mathbb{N}$ of codes of effective fast Cauchy sequences of rationals onto the computable reals $\mathbb{R}_{\mathsf{c}} \subset \mathbb{R}$. With this natural number representation, computable functions on $\mathbb{R}$ are investigated by means of their (classically computable) $\overline{\alpha}$-*tracking functions* on $\mathbb{N}$ [TZ04, TZ05].

7.1.2. ***Continuity.*** Computations with real numbers involve infinite data. Computations are finite processes that approximate in some way infinite processes. The topology of $\mathbb{R}$ defines a process of approximation for infinite data; the functions on the data that are *continuous* in the topology are exactly the functions that can be approximated to any desired degree of precision. This suggests a *continuity principle*:

$$\textit{computability} \implies \textit{continuity}. \tag{4}$$

For abstract models, we assume the algebra A that contains $\mathbb{R}$ is a *topological algebra*, i.e., one in which the basic operations are continuous in its topologies. This implies, in turn, that *all* computable functions will be continuous. As it turns out, the class of functions that can be exactly abstractly computed is, in general, quite limited — "approximate" computations are also necessary [TZ99].

In the concrete models, on the other hand, continuity of computable functions is a consequence of the the Kreisel-Lacombe-Tseitin Theorem [KLS59, Tse59, Tse62].

Thus, in both abstract and concrete approaches, an analysis of basic concepts leads to the continuity principle.

7.1.3. ***Partiality.*** In computing with an abstract model on A we assume A has some boolean-valued functions to test data. For example, in computing on $\mathbb{R}$ we need the functions

$$\mathsf{eq}_{\mathsf{R}} : \mathbb{R}^2 \ \to\ \mathbb{B} \qquad \text{and} \qquad \mathsf{less}_{\mathsf{R}} : \mathbb{R}^2 \ \to\ \mathbb{B}. \tag{5}$$

This presents a problem, since total continuous boolean-valued functions on the reals, being continuous functions from a connected space $\mathbb{R}^n$ to a discrete space $\mathbb{B}$, must be constant. Furthermore, in consequence, we can show that the 'while' and 'while'-array computable functions on connected total topological algebras are precisely the functions explicitly definable by terms over the algebra [TZ99]. This demands the use of partiality for such functions.

To study the full range of real number computations, we must therefore redefine these tests as *partial* boolean-valued functions. This has interesting effects on the theory of computable functions in the areas of nondeterminism and many-valuedness, as we will see.

We turn to some examples to illustrate these features.

7.2. Examples of nondeterminism and many-valuedness.

We look at two examples of computing functions on $\mathbb{R}$.

Example 7.2.1 (Nonzero selection function)**.** Define the function

$$\boldsymbol{piv} : \mathbb{R}^n \ \rightharpoonup\ \{1, \ldots, n\}$$

by

$$\boldsymbol{piv}(x_1, \ldots, x_n) \ = \ \begin{cases} \text{some } i : \ x_i \neq 0 & \text{if such an } i \text{ exists} \\ \uparrow & \text{otherwise.} \end{cases} \tag{6}$$

Computation of this nondeterministic ("pivot") function is a crucial step in the Gaussian elimination algorithm for inverting matrices.

Note that (depending on the precise semantics for the phrase "some i" in (6)) $\boldsymbol{piv}$ is *nondeterministic* or (alternatively) *many-valued* on $\boldsymbol{dom}(\boldsymbol{piv}) = \mathbb{R}^n \backslash \{0\}$. Further:

(a) There is no *single-valued* function which satisfies the definition (2) and is *continuous* on $\mathbb{R}^n$ (as can be easily seen).

(b) However there *is* a computable (and hence continuous!) single-valued function

$$\boldsymbol{piv}\colon \mathbf{CS}^n \rightharpoonup \{\,1,\dots,n\,\}$$

(where $\mathbf{CS}$ is the space of fast Cauchy sequences of rationals) with a simple algorithm. Note however that $\boldsymbol{piv}$ is *not extensional* on $\mathbf{CS}^n$, in the sense that it cannot be factored through $\mathbb{R}^n$:

$$\begin{tikzcd}
\mathbf{CS}^n \arrow[d, "\kappa^n"'] \arrow[drr, "\boldsymbol{piv}"] & & \\
\mathbb{R}^n \arrow[rr, dashed, "?"] & & \{\,1,\dots,n\,\}
\end{tikzcd}$$

where κ is the map from Cauchy sequences to their limits.

In effect, we can regain continuity (for a single-valued function), by foregoing extensionality.

(c) Alternatively, we can maintain continuity *and* extensionality by giving up single-valuedness. For the *many-valued* function

$$\boldsymbol{piv}_{\mathsf{m}}\colon \mathbb{R}^n \;\to\; \wp(\{1,\dots,n\})$$

defined by

$$k \,\in\, \boldsymbol{piv}_{\mathsf{m}}(x_1,\dots,x_n) \quad\Longleftrightarrow\quad x_k \neq 0 \quad \text{for } k = 1,\dots,n$$

is *extensional* and *continuous*, where a function $f\colon A \to \mathcal{P}(B)$ is defined to be continuous iff for all open $Y \subseteq B$, $f^{-1}[Y]$ $(=_{df}\ \{\, x \in A \mid f(x) \cap Y \neq \emptyset \,\})$ is open in A.

Note that the complete Gaussian algorithm for inverting matrices is *continuous* and *deterministic* (hence *single-valued*) and *extensional*, even though it contains $\boldsymbol{piv}$ as an essential component!

Example 7.2.2 (Finding the root of a function — adapted from [Wei00])**.** Consider the function f_a (Figure 2)[7], with real parameter a, defined by

$$f_a(x) \;=\; \begin{cases} x+a+2 & \text{if } \ x \le -1 \\ a-x & \text{if } \ -1 \le x \le 1 \\ x+a-2 & \text{if } \ 1 \le x. \end{cases}$$

This function has either 1 or 3 roots, depending on the size of a. For $a < -1$, f_a has a single (large positive) root; for $a > 1$, f_a has a single (large negative) root; and for $-1 < a < 1$, f_a has three roots, two of which become equal when $a = \pm 1$.

Let g be the (many-valued) function, such that $g(a)$ gives all the non-repeated roots of f_a (Figure 3). Again we have the situation of the previous examples:

(a) We cannot choose a (single) root of f_a continuously as a function of a.

(b) However, one can easily choose and compute a root of f_a continuously as a function of a *Cauchy sequence representation* of a, i.e., non-extensionally in a.

[7] Figures 2 and 3 are taken by kind permission from [TZ04], © 2004 ACM Inc.

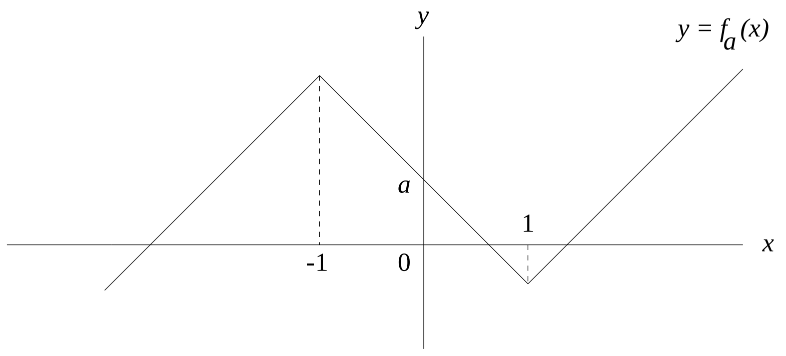

FIGURE 2

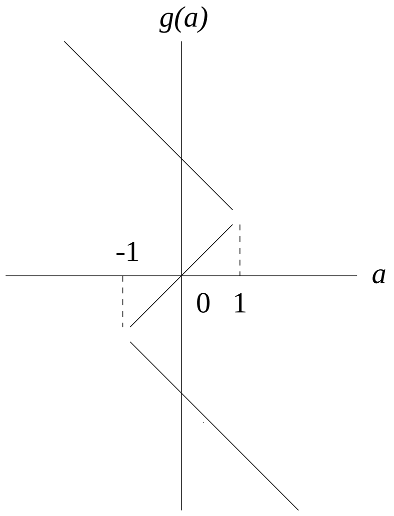

FIGURE 3

(c) Finally, $g(a)$, as a *many-valued* function of a, is continuous. (Note that in order to have continuity, we must exclude the repeated roots of f_a, at $a = \pm 1$.)

Other examples of a similar nature abound, and can be handled similarly; for example, the problem of finding, for a given real number x, an integer $n > x$.

7.3. **Partial algebra of reals; Completeness for the abstract model.**

At the level of *concrete models* of computation, there is no real problem with the issues raised by these examples, since concrete models work only by computations on *representations* of the reals (say by Cauchy sequences).

The real problem arises with the construction of *abstract models* of computation on the reals which should model the phenomena illustrated by these examples, and also correspond, in some sense, to the concrete models.

An immmediate problem in this regard is that the total boolean-valued functions eq_R and less_R are not continuous, and hence also (by the continuity principle, §7.1.2) not (concretely) computable.

We therefore define an N-standard *partial algebra* $\mathcal{R}_{\mathsf{p}}$ on the reals, formed from the total algebra $\mathcal{R}_{\mathsf{t}}$ (Example 3.1.3) by *replacing* the total boolean-valued functions eq_{R} and $\mathsf{less}_{\mathsf{R}}$ (§7.1.3, (5)) by the partial functions

$$\mathsf{eq}_{\mathsf{R,p}}(x,y) \;\simeq\; \begin{cases} \uparrow & \text{if } \; x = y \\ \mathbb{f} & \text{otherwise,} \end{cases}$$

$$\mathsf{less}_{\mathsf{R,p}}(x,y) \;\simeq\; \begin{cases} \mathbb{t} & \text{if } \; x < y \\ \mathbb{f} & \text{if } \; x > y \\ \uparrow & \text{if } \; x = y. \end{cases}$$

These partial functions (unlike the total versions), *are* continuous, and hence $\mathcal{R}_{\mathsf{p}}$ (unlike $\mathcal{R}_{\mathsf{t}}$) is a topological partial algebra. Moreover, these partial functions are concretely computable (by e.g. the tracking model, cf. §7.1.1).

Then we have the question:

Can such continuous many-valued functions be computed on the abstract data type A containing $\mathbb{R}$ using new abstract models of computation?
If so, are the concrete and abstract models equivalent?

The solution presented in [TZ04] was to take $A = \mathcal{R}_{\mathsf{p}}^{N}$, the N-standard extension of $\mathcal{R}_{\mathsf{p}}$, and then extend the ***While**** programming language over A [TZ00] with a nondeterministic "countable choice" programming construct, so that in the rules of program term formation,

$$\mathsf{choose}\ \mathsf{z} : b$$

is a new term of type nat, where z is a variable of type nat and b a term of type bool. In addition (calling the resulting language ***WhileCC**** for ***While**** computability with countable choice), ***WhileCC**** *computability* is replaced by ***WhileCC**** *approximability* [TZ04, TZ05]. We then obtain a *completeness theorem* for abstract/concrete computation, i.e. the equivalence (1) shown at the end of Section 2. Actually (1) was proved in [TZ04] for N-standard metric algebras satisfying some general conditions.

The above considerations lead us to propose the topological partial algebra $\mathcal{R}_{\mathsf{p}}$ as a better basis for abstract models of computation on $\mathbb{R}$ than the (total) algebra $\mathcal{R}_{\mathsf{t}}$ — better in the sense of being more faithful to the intuition of computing on the reals.[8]

Remark 7.3.1 (Semantics of partial algebras)**.** We briefly indicate the semantics for terms and statements over partial algebras, or rather indicate how the semantics for total algebras given in §3.2 and Section 4 can be adapted to partial algebras.

First, the semantics of terms is as given by the equations (2) in §3.2 (with the second '$=$' replaced by '$\simeq$'), using strict evaluation for partial functions (i.e., divergence of any subterm entailing divergence of the term).[9]

Secondly, the semantics of statements is as given in Section 4; i.e., the value $[\![S]\!]^{A}\sigma$ of a statement S at a state σ is the last state in a computation sequence (i.e. a sequence of states) generated by S at σ, provided that the sequence is (well defined and) finite. Otherwise (with an infinite computation sequence) the value

[8] For another perspective on computing with total algebras on the reals, see [BCSS98].
[9] As a general rule. For a case where boolean operators with non-strict semantics are appropriate, see [XFZ15, §3].

diverges. The case of partial algebras is similar, except that there are now *two* cases where the value of $[\![S]\!]^A\sigma$ diverges: (i) (as before, *global divergence*) where the computation sequence is infinite, and (ii) (a new case, *local divergence*) where the computation sequence is finite, but the last item diverges (instead of converging to a state) because of a divergent term on the right of an assignment statement or a divergent boolean test.

Remark 7.3.2 (Comparison of formal results for $\mathcal{R}_\mathsf{p}$ and $\mathcal{R}_\mathsf{t}$)**.** It would be interesting to see to what extent the results concerning abstract computing on the reals with $\mathcal{R}_\mathsf{t}$ detailed in Sections 3 to 6 (for example, the merging and closure theorems (§6.1) and comparisons of various notions of semicomputability and projective semicomputability in $\mathcal{R}_\mathsf{t}$ (§6.7) hold, or fail to hold, in $\mathcal{R}_\mathsf{p}$.[10]

It should be noted, in this regard, that the merging procedure used in our proofs of the closure theorems (Theorems 3, 4 and 5) depend heavily on the totality of the algebra A.

8. Comparing models and generalizing the Church-Turing Thesis

To conclude, we will mention several other abstract approaches to computability on abstract algebras, comment on their comparison, and discuss how to generalize the Church-Turing thesis. These other methods have a variety of technical intuitions and objectives, though they share the abstract setting of an algebraic structure. So let us suppose their common purpose to be the characterization of those functions that are computable in an abstract setting.

8.1. Abstract models of computation.

The computable functions on an abstract algebra can also be characterized by approaches based upon

(i) machine models;
(ii) high-level programming constructs;
(iii) recursion schemes;
(iv) axiomatic methods;
(v) equational calculi;
(vi) fixed-point methods for inductive definitions;
(vii) set-theoretic methods;
(viii) logical languages.

We consider only a couple of these; a fuller survey can be found in [TZ00].

Recursion schemes. Kleene's recursion schemes suggest that we create the class $\boldsymbol{\mu PR}(A)$ of functions $\boldsymbol{\mu PR}$ computable on a standard algebra A, namely those functions definable from the basic operations of A by the application of *composition*, *simultaneous primitive recursion* and *least number search*. We can also extend this to the class $\boldsymbol{\mu PR}^*(A)$ of functions on A definable by the $\boldsymbol{\mu PR}$ oprations on A^* (analogous to the definition of the class ***While****(A) from ***While***(A^*) in §4.1).

Alternatively, we can define the class $\boldsymbol{\mu CR}(A)$ of functions on A in which simultaneous primitive recursion is replaced by simultaneous *course-of-values recursive schemes*. (Simultaneous recursions are needed because the structures are many-sorted.) Then we have:

[10] This is currently being investigated by Mark Armstrong [Arm15].

Theorem 13 (Recursive equivalence theorem)**.** *For any N-standard Σ-algebra A,*

$$\boldsymbol{\mu CR}(A) \;=\; \boldsymbol{\mu PR}^*(A) \;=\; \boldsymbol{While}^*(A).$$

The question of unbounded memory – A^* versus A – re-appears here in the difference between primitive and course-of-values recursion. This model of computation was created [TZ88] with the needs of equational and logical definability in mind.

Axiomatic methods. In an axiomatic method one defines the concept of a *computation theory* as a set $\boldsymbol{\Theta}(A)$ of partial functions on an algebra A having some of the essential properties of the set of partial recursive functions on $\mathbb{N}$. To take an example, $\boldsymbol{\Theta}(A)$ can be required to contain the basic algebraic operators and tests of A; be closed under operations such as *composition*; and, in particular, possess an enumeration for which appropriate *universality* and *s-m-n properties* are true. Thus in Section 5 we saw that $\boldsymbol{While}^*(A)$ is a computation theory in this sense.

The definition of a computation theory used here is due to Fenstad [Fen75, Fen80] who takes up ideas from Moschovakis [Mos71]. Computation theory definitions typically require a code set (such as $\mathbb{N}$) to be part of the underlying structure A for the indexing of functions.

The following fact is easily derived from [MSHT80] (where register machines are used); see also Fenstad [Fen80, Ch. 0].

Theorem 14 (Minimal Computation Theory)**.** *The set $\boldsymbol{While}^*(A)$ of $\boldsymbol{While}^*$ computable functions on an N-standard algebra A is the smallest set of partial functions on A to satisfy the axioms of a computation theory; in consequence, $\boldsymbol{While}^*(A)$ is a subset of every computation theory $\boldsymbol{\Theta}(A)$ on A.*

8.2. **Generalizing the Church-Turing Thesis.**

The $\boldsymbol{While}^*$ computable functions are a mathematically interesting and useful generalization of the partial recursive functions on $\mathbb{N}$ to abstract many-sorted algebras A and classes $\mathbb{K}$ of such algebras. Do they also give rise to an interesting and useful generalization to A and $\mathbb{K}$ of the Church-Turing thesis, concerning effective computability on $\mathbb{N}$? They do; though this answer is difficult to explain fully and briefly. In this section we will only *sketch* some reasons. The issues are discussed in more detail in [TZ88, TZ00], as well as in the chapter by Feferman in this volume [Fef15].

First, consider the following naive attempt at a generalization of the Church-Turing thesis.

Thesis 1 (A naive generalized Church-Turing thesis)**.** *The functions "effectively computable" on a many-sorted algebra A are precisely the functions $\boldsymbol{While}^*$ computable on A.*

Consider now: what can be meant by "effective computability" on an abstract algebra?

The idea of effective computability is inspired by a variety of distinct philosophical and mathematical ideas about the nature of finite computation with finite elements. There are many ways to analyse and formalize the notion of effective calculability, by thinking about concepts such as algorithm; deterministic procedure; mechanical procedure; computer program; programming language; formal system; machine; device; and, of course, the functions definable by these entities.

The idea of effective computability is invaluable because of the close relationships that exist between its constituent concepts. However, only a few of these constituent concepts make sense in an abstract setting. Therefore *the general concept of "effective computability" does not belong in a generalization of the Church-Turing thesis.* We propose to use the term "effective computation" only to talk about finite computation on finite data.

In seeking a generalization of the Church-Turing thesis, we are trying to make explicit certain primary informal concepts that are formalized by the technical definitions, and hence to clarify the nature and use of the computable functions.

We will start by trying to clarify the nature and use of abstract structures. There are three points of view from which to consider the step from concrete to abstract structures, and hence three points of view from which to consider ***While***** computable functions.

(1) There is *abstract algebra*, which is a theory of calculation based upon the "behaviour" of elements in calculations without reference to their "nature". This abstraction is achieved through the concept of isomorphism between concrete structures; an abstract algebra A can be viewed as "a concrete algebra considered unique only up to isomorphism".

(2) There is the viewpoint of *formal logic*, concerned with the scope and limits of axiomatizations and formal reasonings. Here structures are used to discuss formal systems and axiomatic theories in terms of consistency, soundness, completeness, and so on.

(3) There is *data type theory*, an offshoot of programming language theory, which is about data types that the user may care to define and that arise independently of programming languages. Here structures are employed to discuss the semantics of data types, and isomorphisms are employed to make the semantics independent of implementations. In addition, axiomatic theories are employed to discuss their specifications and implementation.

Data type theory is built upon and developed from the first two subjects: it is our main point of view.

Computation in each of the three cases is thought of slightly differently. In algebra, it is natural to think informally of algorithms built from the basic operations that compute functions and sets in algebras, or over classes of algebras uniformly. In formal logic, it is natural to think of formulae that define functions and sets, and their manipulation by algorithms. In data type theory, we use programming languages to define computations. We return to a consideration of each of these approaches, which, because of its special concerns and technical emphasis, leads to a distinctive theory of computability on abstract structures:

Going first to (1): suppose the ***While***** computable functions are considered with the needs of doing algebra in mind. Then the context of studying algorithms and decision problems for algebraic structures (groups, rings, fields, etc.) leads to a formalization of a generalized Church-Turing thesis tailored to the language and use of algebraists:

Thesis 2 (Generalized Church-Turing thesis for algebraic computability)**.** *The functions computable by finite deterministic algebraic algorithms on a many-sorted algebra A are precisely the functions* ***While***** *computable on A.*

An account of computability on abstract structures from this algebraic point of view is given in [Tuc80].

We agree with Feferman [Fef15] who argues that this should be termed a Church-Turing thesis for *algorithms* on abstract structures, rather than *computations*. He prefers, in this context, to reserve the term "computation" for *calculations* on concrete structures composed of finite symbolic configurations.[11]

Now (jumping to viewpoint (3)) suppose that the ***While****** computable functions are considered with the needs of computation in mind. This context of studying data types, programming and specification constructs, etc., leads to a formulation tailored to the language and use of computer scientists:

Thesis 3 (Generalized Church-Turing thesis for programming languages)**.** *Consider a deterministic programming language over an abstract data type* ***dt****. The functions that can be programmed in the language on an algebra A which represents an implementation of* ***dt****, are the same as the functions* ***While****** *programmable on A.*

This thesis has been discussed in [TZ88].

Finally, returning to approach (2): we note that "logical" and non-deterministic languages are suitable for specifying problems. These can be considered as languages for *specification* rather than computation. Here *projectively computable* relations (§6.2) and the use of selection functions for these, play a central role.

We define a specification language to be *adequate* for an abstract data type ***dt*** if all computations on any algebra A implementing ***dt*** can be specified in A. We then formulate a generalized Church-Turing thesis for specifiability on abstract data types:

Thesis 4 (Generalized Church-Turing thesis for specifiability)**.** *Consider an adequate specification language S over an abstract data type* ***dt****. The relations on a many-sorted algebra A implementing* ***dt*** *that can be specified by S are precisely the projectively* ***While****** *computable relations on A.*

8.3. **Concluding remarks.**

We have sketched the elements of our work over four decades on generalizing computability theory to abstract structures. A thorough exposition is to be found in our survey paper [TZ00]. In [TZ02] we have had the opportunity to recall the diverse origins of, and influences on, our research programme.

Since [TZ00], our research has emphasized computation on many-sorted topological partial algebras (the focus of Section 7 here) and its diverse applications:

- computable analysis, especially on the reals [TZ99, TZ05, FZ15],
- classical analog systems [TZ07, TZ11, TZ14, JZ13],
- analog networks of discrete and continuous processors [TZ07, TTZ09],
- generalized stream processing in discrete and continuous time [TZ11, TZ14].

These applications bring us close to an investigation of the physical foundation of computability. In this regard, considerations of *continuity* are central (cf. the discussion in §7.1.2). This is related to the issue of stability of analog systems, and more broadly, to Hadamard's principle [Had52] which, as (re-)formulated by Courant and Hilbert [CH53, Had64], states that for a scientific problem to be well posed, the solution must exist, be unique and depend continuously on the data. To this we might add: it must also be computable.

[11] In Feferman's memorable slogan: "No calculation without representation."

Acknowledgements. We are grateful to Mark Armstrong, Sol Feferman, Diogo Poças and an anonymous referee for very helpful comments on earlier drafts of this chapter. We also thank the editors, Giovanni Sommaruga and Thomas Strahm, for the opportunity to participate in this volume, and for their helpfulness during the preparation of this chapter.

References

[Arm15] Mark Armstrong. Abstract semicomputability in topological algebras over the reals. MSc Thesis, Department of Computing & Software, McMaster University, 2015. In progress.

[BCSS98] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation.* Springer, 1998.

[BH02] V. Brattka and P. Hertling. Topological properties of real number representations. *Theoretical Computer Science*, 284(2):241–257, 2002.

[CH53] R. Courant and D. Hilbert. *Methods of Mathematical Physics, Vol. II.* Interscience, 1953. Translated and revised from the German edition [1937].

[EGNR98] Y.L. Ershov, S.S. Goncharov, A.S. Nerode, and J.B. Remmel, editors. *Handbook of Recursive Mathematics.* Elsevier, 1998. In 2 volumes.

[Fef15] S. Feferman. Theses for computation and recursion on abstract structure. In G. Sommaruga and T. Strahm, editors, *Turing's Revolution. The Impact of his Ideas about Computability.* Birkhäuser/Springer Basel, 2015.

[Fen75] J.E. Fenstad. Computation theories: an axiomatic approach to recursion on general structures. In G. Muller, A. Oberschelp, and K. Potthoff, editors, *Logic Conference, Kiel 1974*, pages 143–168. Springer-Verlag, 1975.

[Fen80] J.E. Fenstad. *Recursion Theory: an Axiomatic Approach.* Springer-Verlag, 1980.

[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.

[FZ15] M.Q. Fu and J.I. Zucker. Models of computation for partial functions on the reals. *Journal of Logical and Algebraic Methods in Programming*, 2015.

[Had52] Jacques Hadamard. *Lectures on Cauchy's Problem in Linear Partial Differential Equations.* Dover, 1952. Translated from the French edition [1922].

[Had64] J. Hadamard. *La Théorie des Équations aux Dérivées Partielles.* Éditions Scientifiques, 1964.

[JZ13] N.D. James and J.I. Zucker. A class of contracting stream operators. *The Computer Journal*, 56:15–33, 2013.

[KK71] G. Kreisel and J.L. Krivine. *Elements of Mathematical Logic.* North Holland, 1971.

[Kle52] S.C. Kleene. *Introduction to Metamathematics.* North Holland, 1952.

[KLS59] G. Kreisel, D. Lacombe, and J. Shoenfield. Partial recursive functions and effective operations. In A. Heyting, editor, *Constructivity in Mathematics: Proceedings of the Colloqium in Amsterdam, 1957*, pages 290–297. North Holland, 1959.

[Mal71] A.I. Mal'cev. Constructive algebras I. In *The Metamathematics of Algebraic Systems. A.I. Mal'cev, Collected papers: 1936–1967*, pages 148–212. North Holland, 1971.

[MKS76] W. Magnus, A. Karass, and D. Solitar. *Combinatorial Group Theory.* Dover, 1976.

[Mos71] Y.N. Moschovakis. Axioms for computation theories — first draft. In R.O. Gandy and C.E.M. Yates, editors, *Logic Colloquium '69*, pages 199–255. North Holland, 1971.

[MSHT80] J. Moldestat, V. Stoltenberg-Hansen, and J.V. Tucker. Finite algorithmic procedures and computation theories. *Mathematica Scandinavica*, 46:77–94, 1980.

[Rab60] M. Rabin. Computable algebra, general theory and the theory of computable fields. *Transactions of the American Mathematical Society*, 95:341–360, 1960.

[She85] J.C. Shepherdson. Algebraic procedures, generalized Turing algorithms, and elementary recursion theory. In L.A. Harrington, M.D. Morley, A. Ščedrov, and S.G. Simpson, editors, *Harvey Friedman's Research on the Foundations of Mathematics*, pages 285–308. North Holland, 1985.

[SHT99] V. Stoltenberg-Hansen and J.V. Tucker. Computable rings and fields. In E. Griffor, editor, *Handbook of Computability Theory*. Elsevier, 1999.

[Tse59] G.S. Tseitin. Algebraic operators in constructive complete separable metric spaces. *Doklady Akademii Nauk SSSR*, 128:49–52, 1959. In Russian.

[Tse62] G.S. Tseitin. Algebraic operators in constructive metric spaces. *Tr. Mat. Inst. Steklov*, 67:295–361, 1962. In Russian. Translated in AMS Translations (2) 64:1–80. MR 27#2406.

[TTZ09] B.C. Thompson, J.V. Tucker, and J.I. Zucker. Unifying computers and dynamical systems using the theory of synchronous concurrent algorithms. *Applied Mathematics and Computation*, 215:1386–1403, 2009.

[Tuc80] J.V. Tucker. Computing in algebraic systems. In F.R. Drake and S.S. Wainer, editors, *Recursion Theory, its Generalisations and Applications*, volume 45 of *London Mathematical Society Lecture Note Series*, pages 215–235. Cambridge University Press, 1980.

[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.

[TZ02] J.V. Tucker and J.I. Zucker. Origins of our theory of computation on abstract data types at the Mathematical Centre, Amsterdam, 1979–80. In F. de Boer, M. van der Heijden, P. Klint, and J.J.M.M. Rutten, editors, *Liber Amicorum: Jaco de Bakker*, pages 197–221. *Centrum Wiskunde & Informatica*, Amsterdam, 2002.

[TZ04] J.V. Tucker and J.I. Zucker. Abstract versus concrete computation on metric partial algebras. *ACM Transactions on Computational Logic*, 5:611–668, 2004.

[TZ05] J.V. Tucker and J.I. Zucker. Computable total functions, algebraic specifications and dynamical systems. *Journal of Logic and Algebraic Programming*, 62:71–108, 2005.

[TZ07] J.V. Tucker and J.I. Zucker. Computability of analog networks. *Theoretical Computer Science*, 371:115–146, 2007.

[TZ11] J.V. Tucker and J.I. Zucker. Continuity of operators on continuous and discrete time streams. *Theoretical Computer Science*, 412:3378–3403, 2011.

[TZ14] J.V. Tucker and J.I. Zucker. Computability of operators on continuous and discrete time streams. *Computability*, 3:9–44, 2014. DOI: 10.3233/COM-14024.

[Wei00] K. Weihrauch. *Computable Analysis: An Introduction*. Springer, 2000.

[XFZ15] Bo Xie, Ming Quan Fu, and Jeffery Zucker. Characterizations of semicomputable sets of real numbers. *Journal of Logic and Algebraic Programming*, 84:124–154, 2015.

(J.V. Tucker) Department of Computer Science, Swansea University, Singleton Park, Swansea SA2 8PP, Wales, UK

E-mail address, J.V. Tucker: `J.V.Tucker@swansea.ac.uk`

(J.I. Zucker) Department of Computing and Software, McMaster University, Hamilton, Ontario L8S 4K1, Canada

E-mail address, J.I. Zucker: `zucker@mcmaster.ca`