Computable Functions and Semicomputable Sets on Many-sorted Algebras

J. V. Tucker and J. I. Zucker

Contents

1 Introduction

The theory of the computable functions is a mathematical theory of total and partial functions of the form

$$f : \mathbb{N}^n \to \mathbb{N},$$

and sets of the form

$$S \subseteq \mathbb{N}^n$$

that can be defined by means of algorithms on the set

$$\mathbb{N} = \{0, 1, 2, \dots\}$$

of natural numbers. The theory establishes what can and cannot be computed in an explicit way using finitely many simple operations on numbers. The set of naturals and a selection of these simple operations together form an algebra. A mathematical objective of the theory is to develop, analyse and compare a variety of models of computation and formal systems for defining functions over a range of algebras of natural numbers.

Computability theory on $\mathbb{N}$ is of importance in science because it establishes the scope and limits of digital computation. The numbers are realised as concrete symbolic objects and the operations on the numbers can be carried out explicitly, in finitely many concrete symbolic steps. More generally, the numbers can be used to represent or code any form of discrete data. However, the question arises:

> *Can we develop theories of functions that can be defined by means of algorithms on other sets of data?*

The obvious examples of numerical data are the integer, rational, real and complex numbers; and associated with these numbers there are data such as matrices, polynomials, power series and various types of functions. In addition, there are geometric objects that are represented using the real and complex numbers, including algebraic curves and manifolds. Examples of syntactic data are finite and infinite strings, terms, formulae, trees and graphs. For each set of data there are many choices for a collection of operations from which to build algorithms.

> *How specific to the set of data and chosen operations are these computability theories? What properties do the computability theories over different sets of data have in common?*

The theory of the computable functions on $\mathbb{N}$ is stable, rich and useful; will the theory of computable functions on the sets of real and complex numbers, and the other data sets also be so?

The theory of computable functions on arbitrary many-sorted algebras will answer these questions. It generalises the theory of functions computable on algebras of natural numbers to a theory of functions computable on any algebra made from any family of sets and operations. The notion of 'computable' here presupposes an algorithm that computes the function in finitely many steps, where a step is an application of a basic operation of the algebra. Since the data are arbitrary, the algorithm's computations are at the same level of abstraction as the data and basic operations of the algebra. For example, this means that computations over the field $\mathbb{R}$ of real numbers are exact rather than approximate. Thus, the algorithms and computations on algebras are intimately connected to their algebraic properties; in particular, the computability theory is invariant under isomorphisms.

Already we can see that, in the general case, there is likely to be a ramification of computability notions. For example, in the case of computable functions on the set $\mathbb{R}$ of real numbers it is also natural to consider computability in terms of computing approximations to the values of a function. The use of approximations recognises the fact that data like the real numbers are infinite objects and can, or must, be algorithmically approximated. This is the approach of computable analysis. We wll present two approaches to computation on the reals: 'algebraic' and 'topological'. In our algebraic approach we are looking for what can be computed exactly, knowing only what the operations reveal about the reals. The operations may have been chosen to reveal properties that are specific to the reals, of course. In the topological approach we are looking for what can be computed with essentially infinite data on the basis of a finite amount of information. Actually, this is again, at bottom, an algebraic approach, for the performance of *approximate* computations.

In this chapter our objective is to show the following:

1. There is a general theory of computable functions on an arbitrary algebra that possesses generalisations of many of the important results in computability theory on natural numbers.
2. The theory provides technical concepts and results that improve our understanding of the foundations of classical computability and definability theory on $\mathbb{N}$.
3. The theory has a wide range of applications in mathematics and computer science.
4. The theory can be developed using many models of computation that are equivalent in that they define the same class of computable functions.
5. The theory possesses a generalisation of the Church–Turing thesis for functions and sets computed by algorithms on any abstract algebraic structure.
6. The theory generalises other less general but still abstract and algebraic, theories of finite computation, including effective algebra and computable analysis.

Computability theories on particular and general classes of algebras address central concerns in mathematics and computer science. Some, such as effective algebra, have a long history and several subfields with deep results, such as the theory of computable rings and fields and the word problem for groups. However, abstract computability theories of the kind we will develop have a short and less eventful history: starting in the late 1950s, with theoretical work on flowcharts, many approaches have been presented that vary in their generality and objectives; indeed, there has been a remarkable amount of reinventing of ideas and results, sometimes with new motivations, such as obtaining results on: the structure of flowcharts; the power of programming constructs; the design of program correctness logics; the development of axiomatic foundations for generalised recursion theories based on ordinals and higher types; and the study of algorithmic aspects of ring and field theory, and of dynamical system theory.

In this section we will introduce in a very informal way the model of computation we will use (in section 1.1) and pose some questions about examples of computable functions (in section 1.2). Then, in section 1.3, we will outline the relationship between our model and other models of computation, especially effective algebra and computable analysis. In section 1.4 the history of the theory of computable functions on abstract algebras is sketched. In sections 1.5 and 1.6 the structure of the chapter and its prerequisites are discussed in more detail.

The chapter is closely linked scientifically with the chapters in this *Handbook* on universal algebra (Volume I), computability (Volume I), and effective algebra (Volume IV); it also connects with other subjects (e.g., topology (Volume I) and those on semantics (Volumes III and IV)). Further information on prerequisites is given in section 1.6.

1.1 Computing in algebras

Let us begin with a basic question:

> *Let S be a non-empty set of data and let $f : S^n \to S$ be a total or partial function. How do we compute f?*

The methods we have in mind start with postulating an algebra A containing the set S. The algebra may consist of a finite family of non-empty sets

$$A_1, \ldots, A_k$$

called the *carriers* of the algebra, one of which is the set S and another is the set $\mathbb{B}$ of Booleans. The algebra is also equipped with a finite family

$$c_1, \ldots, c_p$$

of elements of the sets, called *constants*, and a finite family

$$F_1, \ldots, F_q$$

of functions on the sets called *operations*; these functions are of the form

$$F : \; A_{s_1} \times \ldots \times A_{s_n} \;\to\; A_s$$

and can be total or partial. Among the operations are some standard functions on the Booleans. Such an algebra is called a *standard many-sorted algebra*; we say it is standard because it contains the Booleans and their special operations. An algebra is often written

$$(A_1, \ldots, A_k,\; c_1, \ldots, c_p, F_1, \ldots, F_q).$$

A set Σ of names for the data set, constants and operations (and their arities) of the algebra A is called a *signature*.

For most of the time we will use many-sorted algebras with finitely many constants and total operations, but we will need the case of partial operations to discuss the relationship between our computable functions and continuous functions on topological algebras such as algebras of real numbers, and algebras with infinite data streams.

The problem is to develop and classify models of computation that describe ways of constructing new functions on the set S from the basic operations of the algebra A. In particular, each model of computation $\mathcal{M}$ is a method or technique which we use to define the notion that the function f on the carriers of A is computable from the operations on A by means of method $\mathcal{M}$; and we collect all such functions into the set

$$\mathcal{M}\text{-}\boldsymbol{Comp}(A)$$

of functions $\mathcal{M}$-computable over the algebra A.

There are many useful choices for a model of computation $\mathcal{M}$ with which to develop a computability theory—we list several in a survey in section 8. In this chapter we focus on a theory for computing with a simple imperative model, namely the ***While*** *programming language*.

In this programming language, basic computations on an algebra A are performed by concurrent assignment statements of the form

$$\mathtt{x}_1, \dots, \mathtt{x}_n := t_1, \dots, t_n$$

where $\mathtt{x}_1, \dots, \mathtt{x}_n$ are program variables and $t_1, \dots, t_n$ are terms or expressions built from variables and the operation symbols from the signature of the algebra A; and $\mathtt{x}_i$ and t_i correspond in their types $(1 \le i \le n)$.

The control and sequencing of the basic computations are performed by three constucts that form new programs from given programs S_1, S_2 and S, and Boolean test b:
(i) the *sequential composition* construct

$$S_1; S_2$$

(ii) the *conditional branching* construct

$$\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}$$

(iii) the *iteration* construct

$$\mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}.$$

The set of all programs so constructed over the signature Σ is denoted $\boldsymbol{While}(\Sigma)$.

The operational semantics of a ***While*** program is a function that, given an initial state, enumerates every state of a resulting computation. The input/output (i/o) semantics of a while program is a function that transforms initial states to final states, if they exist. To compute a function on A by means of a ***While*** program we formulate a simple class of function procedures based on ***While*** programs; a function procedure P has the form

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

where a, b, c are lists of input, output and auxiliary variables, respectively, and S is a ***While*** program, satisfying some simple conditions. The semantics of a procedure P is a function $[\![P]\!]^A$ on A whose input and output types are determined by the types of the lists of input and output variables a and b.

A function f is ***While*** *computable* on algebra A if there is a ***While*** function procedure that computes it, i.e., $[\![P]\!]^A = f$. All ***While*** computable functions on A are collected in the set $\boldsymbol{While}(A)$.

A set is defined to be ***While*** *computable*, or *decidable*, if its characteristic function is ***While*** computable. It is ***While*** *semicomputable*, or *semidecidable*, if it is the domain of a partial ***While*** computable function; in other words, if it is the *halting set* of a ***While*** program.

A crucial property of an abstract model of computation is that it is designed to apply to any algebra or class of algebras. Two important consequences are the following.

Firstly, it is easy to explore *uniform computation* where programs generate computations over a class of implementations or representations of data types in a uniform way. For example, think of a ***While*** program that is intended to implement Euclid's algorithm to calculate greatest common divisors in a way that is uniform over a class of algebras. By such a 'class' we could mean, for example, any of the following: (i) the class of all Euclidean domains, (ii) the isomorphism class of all (ideal, infinite) implementations of the ring of integers, (iii) the class of (actual, finite) machine implementations of the integers.

Secondly, it is easy to employ certain forms of type constructions. Since we can compute on any algebra (possessing the Booleans) using the programming model, we can augment an algebra A to form a new algebra A', by adding new types and operations, and apply the programming model to A'. Adding new data sets and operations is a key activity in theory and practice. In particular, three modest expansions of an algebra A that have significant practical effects and (as we shall show) interesting mathematical theories are:

(a) adding the set $\mathbb{N}$ of natural numbers and its standard operations to A to make a new algebra A^N;

(b) adding finite sequences, and appropriate operations, to A to make an algebra A^*.

(c) adding infinite streams, and appropriate operations, to A to make an algebra $\bar{A}$.

We apply the model of computation to form new classes of computable functions, namely:

$$\boldsymbol{While}(A^N), \qquad \boldsymbol{While}(A^*) \qquad \text{and} \qquad \boldsymbol{While}(\bar{A}).$$

By this means it is trivial to add constructs like counters, finite arrays and infinite data streams to the theory of computation, though it is not trivial to chart the consequences.

In summary, what mechanisms are available for computing in an algebra? The methods of computation are merely:

(*i*) basic operations of the algebra; and
(*ii*) sequencing, branching and iterating the operations.

Is equality computable? Do we have available unlimited data storage? Can we search the algebra for data?

We will see that for any many-sorted algebra A with the Booleans, by adding the naturals, we can add

(*iii*) any algorithmic construction on a numerical data representation;

and, by adding A^*, we can add

(*iv*) local search through all elements of the subalgebra generated by given input data;
(*v*) unlimited storage for data in computations.

To obtain equality we have to postulate it as a basic operation of the algebra.

We will study these models of computation. The most important turns out to be the programming language $\boldsymbol{While}^*$, which consists of $\boldsymbol{While}$ programs with the naturals and finite arrays, and is defined simply by

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

This is the fundamental model of imperative programming that yields a full generalisation, to an arbitrary many-sorted algebra A, of the theory of computable functions on the set $\mathbb{N}$ of natural numbers, and for which the generalised Church–Turing thesis for computation on A will be formulated and justified.

1.2 Examples of computable and non-computable functions

First, let us look at the raw material of our theory, namely problems concerning computing functions and sets on specific algebraic structures. We will give a list of questions about computing with $\boldsymbol{While}$ programs on different algebras and invite the reader to speculate on their answers; it is not essential that the reader understand or recognise all the concepts in the examples. The idea is to prepare the reader for the role of algebraic structures in the theory of computable functions and sets, and arouse his or her curiosity.

Let $\mathbb{B}$ be the set of Booleans and let $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$ be the sets of natural, integer, rational, real and complex numbers, respectively.

1. Are the sets of functions $\boldsymbol{While}$ computable over the following algebras the same as those computable over $(\mathbb{N};\ 0,\ n+1)$?
   $(\mathbb{N};\ 0,\ n+1,\ n+m,\ n\cdot m,\ n=m)$
   $(\mathbb{N};\ 0,\ n+1,\ n+m,\ n\cdot m,\ n^m,\ n=m)$
   $(\mathbb{N};\ 0,\ 1, n+m,\ n\cdot m,\ n=m)$
   $(\mathbb{N};\ 0,\ n+m,\ n\cdot m,\ n=m)$
   $(\mathbb{N};\ 0,\ n+m,\ n=m)$
   $(\mathbb{N};\ 0,\ n\cdot m,\ n=m)$
2. Consider each of the following functions:
   $$\begin{aligned} f(n) &= 4\\ f(n) &= n\\ f(n) &= n+1\\ f(n,m) &= n+m\\ f(n,m) &= n-m \end{aligned}$$
   In each case is $f \in \boldsymbol{While}(\mathbb{N};\ 0,\ n-1)$?
3. Let $\mathbb{B}$ be the set of Booleans and $f:\mathbb{B}^k \to \mathbb{B}$. Is $f \in \boldsymbol{While}(\mathbb{B};\mathsf{tt},\mathsf{ff},\mathsf{and},\mathsf{not})$?
4. Let A be a finite set and $f: A\to A$. Is $f \in \boldsymbol{While}(A;\ c_1,\ldots,c_p,\ F_1,\ldots,F_q)$ for any choice of constants c_i and operations F_j on A?
5. Consider the algebra
   $$(\mathbb{B},\mathbb{N},[\mathbb{N}\to\mathbb{B}];\ \mathsf{tt},\mathsf{ff},\ \mathsf{and},\ \mathsf{not}, 0,\ n+1, \mathsf{eval})$$
   of Booleans expanded by adding the set $\mathbb{N}$ of naturals with zero and successor, and the set $[\mathbb{N}\to\mathbb{B}]$ of infinite sequences, or streams, of Booleans, with the evaluation map $\mathsf{eval}:[\mathbb{N}\to\mathbb{B}]\times\mathbb{N}\to\mathbb{B}$ defined by $\mathsf{eval}(b,n)=b(n)$. Are the following functions $\boldsymbol{While}$ computable over this algebra:
   $\boldsymbol{shift}$: $[\mathbb{N}\to\mathbb{B}]\times\mathbb{N}\to\mathbb{B}$ defined by $\boldsymbol{shift}(a,n)=a(n+1)$;
   $\boldsymbol{Shift}$: $[\mathbb{N}\to\mathbb{B}]\to[\mathbb{N}\to\mathbb{B}]$ defined by $\boldsymbol{Shift}(a)(n)=a(n+1)$?
6. Which of the following sets of Boolean streams are
   (i) $\boldsymbol{While}$ computable, and
   (ii) $\boldsymbol{While}$ semicomputable, over the stream algebra in question 5?
   $\{\,a \mid \text{for some } n,\ a(n)=\mathsf{tt}\}$
   $\{\,a \mid \text{for all } n,\ a(n)=\mathsf{tt}\}$
   $\{\,a \mid \text{for infinitely many } n,\ a(n)=\mathsf{tt}\}$
   $\{\,a \mid a(0)=\mathsf{tt},\ \ldots,\ a(n)=\mathsf{tt}\}$ for some fixed n
7. Consider each of the following functions:
   $$\begin{array}{lllllll} f(x) &=& 4 & & f(x) &=& \mathrm{floor}(\sqrt{x})\\ f(x) &=& \sqrt{2} & & f(x) &=& 2^x\\ f(x) &=& \pi & & f(x) &=& \sin(x)\\ f(x) &=& x & & f(x) &=& \cos(x)\\ f(x) &=& x^5 & & f(x) &=& \tan(x)\\ f(x) &=& \sqrt{x} & & f(x) &=& e^x \end{array}$$

In each case, is f ***While*** computable over $(\mathbb{R}; 0, 1, x + y, -x, x \cdot y, x^{-1})$

8. Are $\cos(x)$ and $\tan(x)$ ***While*** computable over $(\mathbb{R}; 0, 1, x + y, -x, x \cdot y, x^{-1}, \sin(x))$?
9. Let $f : \mathbb{R}^2 \to \mathbb{R}$ be the step function

$$f(x, r) = \begin{cases} 0 \text{ if } x < r \\ 1 \text{ if } x \geq r. \end{cases}$$

Is f ***While*** computable over $(\mathbb{R};\ 0,\ 1,\ x + y,\ -x,\ x \cdot y,\ x^{-1})$?

10. Which of the following subsets of $\mathbb{R}$ are
(*a*) ***While*** computable, and
(*b*) ***While*** semicomputable, over the field of real numbers?
(*i*) The rational subfield $\mathbb{Q}$ of the field of reals
(*ii*) The subfield $\mathbb{Q}(\sqrt{2})$ of the field of reals generated by $\mathbb{Q}$ and $\sqrt{2}$
(*iii*) The subfield $\mathbb{Q}(\sqrt{p} \mid p \text{ prime})$ of the field of reals generated by $\mathbb{Q}$ and the set $\{\sqrt{p} \mid p \text{ prime}\}$
(*iv*) The subfield $\mathbb{Q}(r)$ of the field of reals generated by $\mathbb{Q}$ and a non-computable real number r
(*v*) The subfield $\mathbb{A}_{\mathbb{R}}$ of the field of reals containing precisely the real algebraic numbers
11. Is the subalgebra A of $(\mathbb{R};\ 0,\ 1,\ x + y,\ -x,\ x \cdot y,\ x^{-1},\ e^x)$ generated by $\mathbb{Q}$ ***While*** semidecidable?
12. Is there a ***While*** program over $(\mathbb{R};\ 0,\ 1,\ x + y,\ -x,\ x \cdot y,\ x^{-1},\ \sqrt{x})$ that computes all the real roots of all quadratic equations with real coefficients?
13. Consider the polynomial

$$p(X) \equiv a_0 + a_1 X + a_2 X^2 + \ldots + a_n X^n \qquad (a_0, \ldots, a_n \in \mathbb{R}).$$

(*a*) Is the set $\{x \in \mathbb{R} \mid p(x) = 0\}$ of roots of p ***While*** decidable over

$$(\mathbb{R};\ 0,\ 1,\ x + y,\ -x,\ x \cdot y,\ x^{-1},\ a_0, \ldots, a_n)?$$

(*b*) For each n find operations to add to the algebra $(\mathbb{R};\ 0,\ 1,\ x + y,\ -x,\ x \cdot y,\ x^{-1})$ to calculate the n roots of the polynomial as functions of the coefficients.

14. Consider the algebra $(\mathbb{R},\ \mathbb{B};\ 0,\ 1,\ x + y,\ -x,\ x \cdot y,\ x^{-1},\ x = y)$ which adds equality $=: \mathbb{R}^2 \to \mathbb{B}$ to the field of reals. What new functions $f : \mathbb{R}^k \to \mathbb{R}$ can be computed?
15. Consider the algebra $(\mathbb{R},\ \mathbb{B};\ 0,\ 1,\ x + y,\ -x,\ x \cdot y,\ x^{-1},\ x = y,\ x < y)$ which adds ordering $<: \mathbb{R}^2 \to \mathbb{B}$ to the field of reals. What new functions $f : \mathbb{R}^k \to \mathbb{R}$ can be computed?
16. Is every cyclic subgroup $\langle t \rangle$ of the circle group $\mathbf{S}^1$ ***While*** decidable?

17. If $f : \mathbb{R} \to \mathbb{R}$ is ***While*** computable on $(\mathbb{R};\ 0,\ 1,\ x{+}y,\ -x,\ x{\cdot}y,\ x^{-1})$, is f continuous?
18. Can any continuous function $f : \mathbb{R} \to \mathbb{R}$ be approximated (in some suitable metric) by a function ***While*** computable over $(\mathbb{R};\ 0,\ 1,\ x + y,\ -x,\ x \cdot y,\ x^{-1})$?
19. What is the relationship, for functions on $\mathbb{R}$, between computability in the sense of computable analysis, and ***While*** computability on $(\mathbb{R};\ 0,\ 1,\ x + y,\ -x,\ x \cdot y,\ x^{-1})$?
20. Is there an algebra $A(\mathbb{R})$ containing $\mathbb{R}$, for which
$$\boldsymbol{Rec}(\mathbb{R}) \ = \ \boldsymbol{While}(A(\mathbb{R})),$$
where $\boldsymbol{Rec}(\mathbb{R})$ is the set of functions computable on $\mathbb{R}$ in the sense of recursive or computable analysis?
21. Consider the many-sorted algebra
$$(\mathbb{R},\ \mathbb{N},\ [\mathbb{N} \to \mathbb{R}];\ 0_{\mathbb{R}},\ 1_{\mathbb{R}},\ x + y,\ -x,\ 0_{\mathbb{N}},\ n + 1,\ \mathsf{eval})$$
that is an expansion of the additive Abelian group of reals, made by adding the naturals with zero and successor, infinite sequences or streams of real numbers, and the evaluation map $\mathsf{eval} : [\mathbb{N} \to \mathbb{R}] {\times} \mathbb{N} \to \mathbb{R}$ defined by $\mathsf{eval}(a, n) = a(n)$. Are the following functions ***While*** computable on this algebra:
$\boldsymbol{add}$: $[\mathbb{N} \to \mathbb{R}]^2 \times \mathbb{N} \to \mathbb{R}$ defined by $\boldsymbol{add}(a, b, n) = a(n) + b(n)$;
$\boldsymbol{Add}$: $[\mathbb{N} \to \mathbb{R}]^2 \to [\mathbb{N} \to \mathbb{R}]$ defined by $\boldsymbol{Add}(a, b)(n) = a(n) + b(n)$?
22. Are complex conjugation $-z$ and modulus $|z|$ of a complex number z ***While*** computable over the following algebras?
    - (i) $(\mathbb{C};\ 0,\ 1,\ x + y,\ -x,\ x \cdot y,\ x^{-1})$
    - (ii) $(\mathbb{C};\ 0,\ 1,\ i,\ x + y,\ -x,\ x \cdot y,\ x^{-1})$
23. Is the set $\{i\}$ ***While*** decidable over either of the fields listed in question 22?
24. Consider the function $f(x) = 4x(1 - x)$ on the reals. Is the orbit of f, defined by
$$\boldsymbol{orb}(f, x) \ = \ \{f^n(x) \mid n \in \mathbb{N},\ 0 \le x \le 1\}$$
While computable over the field of real numbers?
25. Are the fractal subsets of $\mathbb{C}$, such as the Mandelbrot and Julia sets, ***While*** decidable over the field of complex numbers?
26. Are the following subsets of $\mathbb{C}$ either ***While*** decidable, or ***While*** semidecidable, over the algebra $(\mathbb{C},\ \mathbb{B};\ 0,\ 1,\ i,\ |z|,\ x{+}y,\ -x,\ x{\cdot}y,\ x^{-1},\ =)$?
    - (i) The set $\{i\}$
    - (ii) The set of all roots of unity
    - (iii) The set of all algebraic complex numbers

27. Consider the rings $\mathbb{Z}[X_1, \ldots, X_n]$ of all polynomials in n indeterminates over the integers. Is the ideal membership relation

$$q \in (p_1, \ldots, p_m)$$

(in $q, p_1, \ldots, p_m$) ***While*** decidable over this ring?

28. Consider the rings $F[X_1, \ldots, X_n]$ of all polynomials in n indeterminates over a field F. Is the ideal membership relation ***While*** decidable over this ring?

29. Consider the algebra $T(\Sigma, X)$ of all terms over signature Σ in the finite set X of indeterminates. Let A^X be the set of assignments to X in an algebra A. Define the term evaluation function

$$\boldsymbol{T}\colon\ \boldsymbol{TE}(\Sigma, X) \times A^X \ \to\ A$$

by $\boldsymbol{TE}(t,a) = t(a)$. Is ***TE While*** computable over the algebra formed by simply combining the algebras $T(\Sigma, X)$ and A?

30. (a) Are all first-order definable subsets of natural numbers ***While*** computable with respect to the following algebras?

    (i) $(\mathbb{N};\ 0,\, n+1,\, n+m,\, n\cdot m)$
    (ii) $(\mathbb{N};\ 0,\, n+1,\, n+m)$
    (iii) $(\mathbb{N};\ 0,\, n+1)$

    (b) Are the ***While*** semicomputable sets precisely the Σ_1-definable sets with respect to these algebras?

31. Is any set of complex numbers that is first-order definable over the field of complex numbers ***While*** decidable over this field? Is any set of real numbers that is first-order definable over the ordered field of real numbers ***While*** decidable over this field?

1.3 Relations with effective algebra

In computer science, many-sorted algebras are used to provide a general theory of data and, indeed, of whole computing systems. They have been employed to

specify and analyse many new forms of data types;
classify data representations;
characterise which data types are implementable;
model systems;
analyse the modularisation of computing systems;
formalise the correctness of systems; and
reason about systems.

A many-sorted algebra models a concrete representation of a data type or system; such representations are compared by homomorphisms, axiomatised by equations and conditional equations, and prototyped by term rewriting methods. There is a considerable theoretical and practical literature available which may be accessed through survey works such as Meseguer and Goguen [1985], Wirsing [1991], Wechler [1992] and Meinke and Tucker [1992].

The theory of computable functions and sets on many-sorted algebras is intended to provide an abstract theory of computing to complement this abstract algebraic theory of data. With this in mind we ask the question:

> *How is the theory of **While** computable functions and sets on many-sorted algebras related to other theories of computability on such algebras?*

We have mentioned earlier that there are many models of computation that can be applied to an arbitrary algebra and that turn out to define the same class of functions and sets as the ***While*** language; these equivalent models belong to the computability theory and are the subject of section 8. Here we will discuss an important approach to analysing computability on algebras called *effective algebra*. Effective algebra is concerned with what algebras are computable, or effective, and what functions and sets on these algebras are computable, or effective. The subject is explained in Stoltenberg-Hansen and Tucker [1995; 1999a].

A starting point for the discussion is the theory of the computable functions on the set $\mathbb{N} = \{0, 1, 2, \ldots\}$ of natural numbers. According to the Church–Turing thesis, the class $\boldsymbol{Comp}(\mathbb{N})$ of computable function on $\mathbb{N}$, defined by any one of a number of models of computation, is precisely the class of functions definable by means of algorithms on the natural numbers. As we have noted, the algorithms are often over some algebraic structure on $\mathbb{N}$. In fact, seen from the algebraic theory of data, the algebras used form a *class* of concrete representations of the natural numbers that is parameterised by both the choice of operations and the precise nature of the number representations (e.g.,binary, decimal and roman). The extent to which the theory of computable functions on $\mathbb{N}$ varies over the class of these algebras of numbers is an important question, but one that is not often asked. We expect there to be very little variation in practice (but compare questions 1 and 2 of section 1.2).

In general terms, a *computability theory* consists of

(a) a class of algebraic or relational structures to define data and operations; and

(b) a class of methods, which we call a model of computation, to define algorithms and computations on the data using the operations.

A *generalised computability theory* is one which can be applied to a structure containing the set $\mathbb{N}$ of natural numbers to define the set $\boldsymbol{Comp}(\mathbb{N})$ of computable functions on $\boldsymbol{Comp}(\mathbb{N})$. An abstract computability theory is a computability theory in which the theory is invariant up to isomorphism (in some appropriate sense).

To develop an abstract generalised computability theory for any algebra A, and classify the computable functions and sets on A, one can proceed in either of the following two directions. One can apply computability theory on $\mathbb{N}$ to algebras using maps from sets of natural numbers to algebras called numberings. The long-established theories of decision problems in semigroups, groups, rings and fields, etc. are examples of this approach. Furthermore, the theory of computable functions $\boldsymbol{Comp}(\mathbb{R})$ on the set $\mathbb{R}$ of real numbers in computable analysis uses computability theory on $\boldsymbol{Comp}(\mathbb{N})$ to formalise how real number data and functions are approximated effectively. Theories based on these approaches are parts of what we here call effective algebra.

Alternatively, one can generalise the computability theory on $\mathbb{N}$ to accommodate abstract structures; the theory of computable functions on many-sorted algebras developed in this chapter is an example, of course, and more will be said about equivalent models of computation in section 8. However, there are examples of generalised computation theories that are strictly stronger, such as ordinal recursion theory, set recursion theory, higher type recursion theory and domain theory. Typically these four generalised computability theories allow infinite computations to return outputs. To appreciate the diversity of some of these theories it is necessary to examine closely their original motivations; seen from our simple finitistic algebraic point of view, generalised recursion theories have a surprisingly untidy historical development.

Let us focus on the first direction. Effective algebra is a theory that provides answers for questions such as:

> *When is an algebra A computable? What functions on A are computable? What sets on A are decidable or, at least, semidecidable?*

It attempts to establish the scope and limits of computation by means of algorithms for any set of data, by applying the theory of computation on $\mathbb{N}$ to universal algebras containing the set of data using numberings. Thus, it classifies what data can be represented algorithmically, and what sets and functions can be defined by algorithms, in the same terms as those of the Church–Turing thesis for algorithms on $\mathbb{N}$. Assuming such a thesis, we may then use the theory of the recursive functions on $\mathbb{N}$ to give precise answers to the above questions about algebras, and to the question:

> *What sets of data and functions on those data can be implemented on a computer in principle?*

The numberings capture the scope and limits of digital data representation and, thus, effective algebra is a general theory of the digital view of computation. More specifically, in effective algebra we can investigate the consequences of the fact that

1. an algebra is computable;
2. an algebra is effective in some weaker senses; and

3. a topological algebra can be approximated by a computable or effective algebra.

Among the weaker senses are the concepts of semicomputable, cosemicomputable and, most generally, effective algebras. For an algebra to be effective it must be countable, so that its elements may be enumerated. For an algebra to be effectively approximable it must have a topological structure, so that its elements may be approximated; the bulk of interesting topological algebras are uncountable. A full account of these concepts is given in Stoltenberg-Hansen and Tucker [1995]. At the heart of the theory of effective algebra is the notion of a computable algebra: a computable algebra is an algebra that can be faithfully represented using the natural numbers in a recursive way. Here is the definition for a single-sorted algebra:

Definition 1.1. An algebra $A = (A; c_1, \dots, c_p, F_1, \dots, F_q)$ is *computable* if: (i) the data of A can be computably enumerated—there exists a recursive subset $\Omega_\alpha \subseteq \mathbb{N}$ and a surjection

$$\alpha : \ \Omega_\alpha \ \to \ \mathbb{N}$$

called a *numbering*, that lists or enumerates, possibly with repetitions, all the elements of A; (ii) the operations of A are *computable in the enumeration*—for each operation $F_i : A^{n(i)} \to A$ of A there exists a recursive function

$$F_i : \ {\Omega_\alpha}^{n(i)} \ \to \ \Omega_\alpha$$

that *tracks* the F_i in the set Ω_α of numbers, in the sense that for all $x_1, \dots, x_{n(i)} \in \Omega_\alpha$,

$$F_i(\alpha(x_1), \dots, \alpha(x_{n(i)})) \ = \ \alpha(f_i(x_1, \dots, x_{n(i)}));$$

(iii) the equivalence of numerical representations of data in A is *decidable*—the equivalence relation $\equiv_\alpha$ defined by

$$x_1 \equiv_\alpha x_2 \iff \alpha(x_1) = \alpha(x_2)$$

is recursive.

An equivalent formulation, in the algebraic theory of data, is that A is *computable* if it is the image of a recursive algebra Ω_α of numbers under a homomorphism $\alpha : \Omega_\alpha \to A$ whose kernel $\equiv_\alpha$ is decidable. (This simple algebraic characterisation leads to new methods of generalising computability theories: see Stoltenberg-Hansen and Tucker [1995]).

What mechanisms are available for computing in a computable algebra? Via the enumeration, the methods include:

(i) basic operations of the algebra;
(ii) sequencing, branching and iterating the operations;

(*iii*) any algorithmic construction on the numerical data representation;
(*iv*) global search through all elements of the algebra;
(*v*) unlimited storage for data in computations;
(*vi*) the equality relation on the algebra via the congruence.

Conditions (*i*) and (*ii*) are shared with our ***While*** programming model and, indeed, are necessary for an algebraic theory: recall section 1.1. There are also a number of features that extend the methods of our ***While*** model, including conditions (*iii*) and, more dramatically, (*iv*). Using the properties of the numbers that represent the data we can perform global searches through the data sets (by means of an ordering on the code set), and store data dynamically without limitations on data storage (by means of a pairing on the code set). Note that condition (*vi*) is a defining feature of computable algebras and can be relaxed (as in the case of semicomputable or effective algebras, for instance).

Note that an algebra A is computable if *there exists* some computable numbering α for A. The computability of functions and sets over A may depend on the numbering α; thus, to be more precise, we should say that A, its functions and subsets etc. are α-computable. Let us define the computable subsets and functions for such an algebra.

Definition 1.2 (Sets and maps). Let A be an algebra of signature Σ, computable under the numbering $\alpha : \Omega_\alpha \to A$.

(1) A set $S = A^k$ is *α-decidable*, *α-semidecidable* or *α-cosemidecidable* if the corresponding set

$$\alpha^{-1}(S) \;=\; \{(x_1, \dots, x_k) \in {\Omega_\alpha}^k \mid (\alpha(x_1), \dots, \alpha(x_k) \in S\}$$

of numbers is recursive, recursively enumerable (r.e.) or co-recursively enumerable (co-r.e.) respectively.

(2) A function $\phi : A \to A$ is an *α-computable* map if there exists a recursive function $f : \Omega_\alpha \to \Omega_\alpha$ such that for all $x \in \Omega_\alpha$, $f(\alpha(x)) = \alpha(f(x))$; or, equivalently, f commutes the following diagram:

$$\begin{tikzcd}
A \arrow[r, "\phi"] & A \\
\Omega_\alpha \arrow[u, "\alpha"] \arrow[r, "f"'] & \Omega_\alpha \arrow[u, "\alpha"']
\end{tikzcd}$$

Let $\boldsymbol{Comp}_\alpha(A)$ be the set of all α-computable maps on A.

For any computable algebra, there are many computable numberings, some of which may have desirable properties; for example, it is the case that every computable algebra has a bijective numbering with code set $\mathbb{N}$.

Let $C(A)$ be the set of all computable numberings of the algebra A. The choice of a numbering $\alpha \in C(A)$ suggests that the effectiveness of a subset or function on A may depend on α. To illustrate, let $S \subseteq A$ and consider the following questions:

> *Is S decidable for all computable numberings of A; or decidable for some, and undecidable in others; or undecidable for all computable numberings of A?*

Another question concerns the invariance of computable maps.

> *If A is computable under two numberings α and β then what is the relation between the sets $\boldsymbol{Comp}_\alpha(A)$ and $\boldsymbol{Comp}_\beta(A)$? What is*
> $$\bigcap_{\alpha \in C(A)} \boldsymbol{Comp}_\alpha(A)?$$

Consider our abstract model based on $\boldsymbol{While}$ programs. We have noted that

$$\boldsymbol{While}(\mathbb{N};\ 0, n+1) \ = \ \boldsymbol{Comp}(\mathbb{N}).$$

The question then arises for our algebras:

> *What is the relationship between $\boldsymbol{While}(A)$ and $\boldsymbol{Comp}_\alpha(A)$ for an arbitrary computable representation α?*

We can prove that if A is computable then

$$\boldsymbol{While}(A) \ \subseteq \ \bigcap_{\alpha \in C(A)} \boldsymbol{Comp}_\alpha(A). \tag{1.1}$$

(In fact this inclusion holds for much weaker hypotheses on A.) The converse inclusion does not hold in general. To see why, consider the algebra

$$(\mathbb{N};\ 0, n-1),$$

and the use of a $\boldsymbol{While}$ program to compute a function $f : \mathbb{N}^n \to \mathbb{N}$. It turns out that for any $x_1, \ldots, x_n \in \mathbb{N}$,

$$f(x_1, \ldots, x_n) \ \le \ \max(x_1, \ldots, x_n)$$

because assignments can only reduce the value of the inputs. It follows that

$$\boldsymbol{While}(\mathbb{N};\ 0, n-1) \subsetneq \bigcap_{\alpha \in C(A)} \boldsymbol{Comp}_\alpha(\mathbb{N};\ 0, n-1) \tag{1.2}$$

because in any numbering the successor function $\mathsf{S}(x) = x + 1$ can be computed.

More difficult to answer is the question: *When is*

$$\boldsymbol{While}(A) \;=\; \bigcap_{\alpha \in C(A)} \boldsymbol{Comp}_{\alpha}(A)?$$

Some results in this direction are known.

Inequality (1.2) is not a weakness of the abstract theory. Rather it is an indication of the fact that the abstract models provide a more sensitive analysis of finite computations. For example, the abstract theory reveals the special properties of the algebras of numbers that give computability theory on $\mathbb{N}$ its special characteristics: *the theory of* $\boldsymbol{Comp}_{\alpha}(A)$ *is the same as the theory of* $\boldsymbol{While}^{*}(A)$ *when A is an algebra finitely generated by constants.*

1.4 Historical notes on computable functions on algebras

The generalisation of the theory of computable functions to abstract algebras has a complicated history. On the one hand the connections between computation and algebra are intimate and ancient: algebra grew from problems in computation. However, the fact that it is now necessary to explain how computation theory can be connected or applied to algebra is an aberration, and is the result of interesting intellectual and social mutations in the past. It is a significant task to understand the history of generalisations of computability theory, with questions for research by historians of mathematics, logic and computing, as well as sociologists of science.

The story that underlies this work involves the development of algebra; the development of computability theory; interactions between computability theory and algebra; and applications to computing. Some of the connections between computation theory and algebra have been provided in other *Handbook* chapters: for notes on the histories of

effective algebra, see Stoltenberg-Hansen and Tucker [1995];
computable rings and fields, see Stoltenberg-Hansen and Tucker [1999a];
algebraic methods in computer science, see Meinke and Tucker [1992].

In the following notes we discuss the nature of generalisations and point out the earliest work on abstract computability theory. Section 8 is devoted to a fairly detailed survey of the literature.

We first list some *common-sense reasons for generalising computability theory.* A common view is to say that the purpose of a generalisation of computability theory is one or more of the following:

(*i*) to say something new and useful about the original theory;
(*ii*) to provide new methods of use in computer science and mathematics;
(*iii*) to illuminate and increase our understanding of the nature of computation.

As will be seen, the theory of computable functions on many-sorted algebras is certainly able to meet the goals (*i*)–(*iii*). For a discussion of these and other reasons for generalising computability theory, see Kreisel [1971].

Broadly speaking, it is often the case that a new mathematical generalisation of an old theory focuses on a few basic technical ideas, results or problems in the old theory and makes them primary objects of study in the new theory. If the generalised theory is technically satisfying then a substantial subject can be built on foundations consisting of little more than some modest technical motivations. Recent history records many attempts at generalisations of computability theory that have different, narrower, technical aims than those of the original theory. Indeed, in some cases, *if* the generalisation can be applied to analyse computation on algebras then its aims need not be particularly useful or meaningful.

Generalised computability theories bear witness to the fact that computability theory has several concepts, results and problems that can bear the weighty load of a satisfying generalisation. For instance, on generalising finiteness, and allowing infinite computations, theoretical differences can be found that allow models of computation to cleverly meet goal (*i*), but not (*ii*) or (*iii*).

Computability theory can also support a good axiomatic framework in which deep results can be proved, and generalised computability theories are models. For example, the axiomatic notion of computation theory developed by by Moschovakis and Fenstad elegantly captures basic results and advanced degree theory: see Stoltenberg-Hansen [1979] and Fenstad [1980].

There is usually a good market for general frameworks in theoretical subjects because there is more space in which to seek ideas and show results. Generality is attractive: there are many new technical concepts and the original theory can underwrite their value. Generalisations are developed, gain an audience and reputation, and, like so many other technical discoveries, await an application. Applications can arise in more exotic or commonplace areas than their creators expected. In the case of generalising computability theory, some theories have useful applications (e.g. in set theory), and some languish in the museum of possible models of axiomatic theories of computation.

Abstract computability theory developed rather slowly, and owes much to the development of programming languages.

A starting point is the notion of the *flowchart*. The idea was first seen in examples of programs for the ENIAC from 1946, published in Goldstine and von Neumann [1947]. Flowcharts were adapted and used extensively in practical work. For example, standards were provided by the American Standards Association (see American Standards Association [1963] and Chaplin [1970]).

To define mathematically the informal idea of a flowchart required a number of papers on flow diagrams, graph schemata and other models; some commonly remembered papers are: Ianov [1960], Péter [1958], Voorhes [1958], Asser [1961], Gorn [*1961] and Kaluzhnin [1961]. By the time of the celebrated Böhm and Jacopini [1966] paper on the construction of normal forms for flowcharts, the subject of flowcharts was well established.

In some of these papers the underlying data need not be the natural numbers, strings or bits. In particular, in Kaluzhnin [1961] flowcharts are modelled using finite connected directed graphs. These have vertices either with one edge to which are assigned an operation, for computation, or two exit edges to which are assigned a discriminator, for tests. The graph has one vertex with no incoming edge, for input, and one vertex with no outgoing edge, for output. To interpret a so-called *graph scheme*, a set of functions is used for the operations, and a set of properties is used for the discriminators.

Kaluzhnin's work was used in various studies, such as Elgot's early work, and in Thiele [1966], a major study of programming, in which flow diagrams are presented that are not necessarily connected graphs. The semantics of flow diagrams is defined here formally, in terms of the functions

$El_{\Delta,\xi}(n) =$ *object or data after the nth step in flow diagram Δ starting at state ξ,*
$Kl_{\Delta,\xi}(n) =$ *edge in flow diagram Δ traversed after the n-th step starting at state ξ.*

using simultaneous recursions. Thiele's work influenced the formal development of operational semantics as found in the Vienna Definition Language: see Lauer [1967; 1968] and Lucas *et al.* [1968]. The important point is that predicate calculus with function symbols and equality is extended by adding expressions that correspond with flow diagrams to make an algorithmic language involving graphs.

Thus, in the period 1946–66, some of the basic topics of a theory of computation over any set of data had been recognised, including: equivalence of flowcharts; substitution of flowarts into other flowcharts; transformations and normal forms for flow charts; and logics for reasoning about flow charts.

Flowcharts were not the only abstract model of computation to be developed.

Against the background of early work on the principles of programming by A. A. Lyapunov and theoretical work by Ianov and others in the former Soviet Union, Ershov [1958] considered computation with any set of operations on any set of data. In Ershov [1960; 1962] the concept of operator algorithms is developed. These are imperative commands made from expressions over a set of operations; the algorithms allow self-modification. The model was used in early work on compilation in the former Soviet Union. See Ershov and Shura-Bura [1980] for information on early programming.

Of particular interest is McCarthy [1963], which reviewed the requirements and content of a general mathematical theory of computation. It emphasises the idea that classes of functions can be defined on arbitrary sets of data. Starting with a (finite) collection F of base functions on some collection of sets, we can define a class $\mathrm{C}\{F\}$ of functions computable in terms of F. The mechanism used is that of recursion equations with an informal operational meaning based on term substitution. An abstract computability theory is an aim—not 'merely' a model of programming structure etc.—and McCarthy writes (p. 63):

> *Our characterisation of C{F} as the set of functions computable in terms of the base functions in F cannot be independently verified in general since there is no other concept with which it can be compared. However it is not hard to show that all partial recursive functions in the sense of Church and Kleene are in C{zero, succ}.*

This, of course, falls short of a generalised Church–Turing thesis. The paper also mentions functionals and the construction of new sets of data from old, including a product, union and function space construction for two sets, and recursive definition of strings. McCarthy's paper is eloquent, perceptive and an early milestone in the mathematical development of the subject.

E. Engeler's innovative work on the subject of abstract computability begins in Engeler [1967]. This contains a mathematically clear account of program schemes whose operations and tests are taken from a first-order language over a single-sorted signature. The programs are lists of labelled conditional and operational instructions of the form

```
k :   if φ then goto p else goto q
k :   do ψ then goto p
```

where k, p and q are natural numbers acting as labels for instructions, ϕ is a formula of the language and ψ is an assignment of one of the forms

$$\mathtt{x{:=}c}, \qquad \mathtt{x{:=}y} \quad \text{or} \quad \mathtt{x{:=}f}\,(\mathtt{y}_1,\dots,\mathtt{y}_k)$$

where `x,y,...` are variables, and `c` and `f` are any constant and operation of the signature. Interpretations are given by means of a notion of state, mapping program variables to data in a model. A basic result proved here is this:

> *To each program π one can associate a formula ϕ that is a countable disjunction of open formulae such that for all models,*
>
> $$\pi \text{ terminates on all inputs from } A \iff A \models \phi.$$

Results involving the definability of the halting sets of programs in terms of computable fragments of infinitary languages will be proved in section 5 and applied in section 6, where we refer to them as versions of Engeler's lemma.

In Engeler [1968a] two new models of computation are given: one is based on a new form of Kleene μ-recursion, the other on a deductive system. The functions computable by the programs and these two models are shown to be equivalent. Engeler's development of the subject in the period 1966–76 addresses original and yet basic questions including the computability of geometrical constructs, exact and approximate computation, and a Galois theory for specifications and programs: see Engeler [1993].

The study of program schemes and their interpretation on abstract structures grew in the early 1970s, along with the theoretical computer science community. Problems concerning program equivalence, decidability and the expressive power of constructs were studied and formed a subject called *program schematology* (see, for example, Greibach [1975]). The problem of finding decidable properties, and especially finding decidable equivalence results, is work directly influenced by Ianov [1960]. The subject of program schemes, and the promise of decidability results on abstract structures, was addressed in the unpublished Luckham and Park [1964], and early undecidability results appeared in Luckham *et al.* [1970]. Program schematology was part of the response to the need to develop a comprehensive theory of programming languages, joining early work on programming language semantics, program verification and data abstraction also characteristic of the period. We will look at the subject again in section 8.

The next milestone is that of Friedman [1971a]. Friedman's paper has received a fine exegesis in Shepherdson [1985] which we recommend. Against a backcloth of growing interest in the generalisations of computability theory by mathematical logicians, and inspired by the work of Moschovakis on computation, Friedman considered the mathematical question (in Shepherdson's words):

> *What becomes of the concepts and results of elementary recursion theory if, instead of considering only computations on natural numbers, we consider computations on data objects from any relational structure?*

In this he gave four models of computation for an algebra A. The first two were based on register machines, the programs for which were called *finite algorithmic procedures* (or *faps*). The third was a generalisation of Turing machines. The fourth was a model based a set of r.e. lists of conditional formulae of the form

$$R_{i1} \& \ldots \& R_{ik_i} \ \rightarrow \ t_i$$

for $i = 1, 2, \ldots,$ where the R_{ij} are tests and the t_i is a term, called *effective definitional schemes.* In section 8 we will discuss these models again.

All four models compute the partial recursive functions on the natural numbers. Freidman organised some 22 basic theorems of computability theory on the natural numbers $\mathbb{N}$ into six groups which were defined by the properties of a structure A that are sufficient to prove the theorems on A. Also noteworthy are results which showed some of these models are not equivalent in the abstract setting.

Friedman's paper is technically intense, with several good ideas and results. Looking back at the literature, one wonders why such an indispensable paper had not appeared before.

The theory of computation on natural numbers and strings began in mathematical logic, motivated by questions in the foundations of mathematics. However, the theory of computation on arbitrary algebraic and relational structures began and was sustained in computer science, motivated by the need to model programming language constructs. Mathematical logic plays two roles: firstly, it provides knowledge of abstract structures, formal languages and their semantics; and secondly, it provides a deep theory of computation on natural numbers.

Both the complicated history of algebra and computability theory mentioned earlier (and sketched in the historical notes of other *Handbook* chapters), and the development of abstract computability since the 1950s, have much to offer those interested in the historical development of mathematical theories. In the matter of the development of abstract computability these brief notes, coupled with our survey in section 8, suggest some questions a historical analysis might answer. Why have there been many independent attempts at making models of computation but relatively few attempts to show equivalencies, or undertake sustained programmes of theoretical development and applications? Why have there been so many demonstrations of an ability to ignore earlier work and to reinvent ideas and results? Why was the development of the theory so slow and messy? Why was computability theory on the natural numbers not generalised to rings and fields, or even relational structures, before the Second World War? Why did the subject not find a home in mathematical logic?

It is clear that computer science played an essential role in creating the theory of computable functions on abstract algebras. One is reminded of McCarthy's [1963] commonly quoted words:

> *It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance.*

It also demands patience.

1.5 Objectives and structure of the chapter

Computability theory over algebras can be developed in many directions and can be used in many applications. In this short introduction we have chosen to emphasise computation on general many-sorted algebras.

We see algebra as providing a general theory of data that is theoretically satisfying and practically useful. Therefore, theories of what is computable over algebras are fundamental for a general theory of data.

In section 2 we define the basic algebraic notions we will need: algebras with Booleans and naturals, relative homomorphisms, terms and their evaluation, abstract data types, etc. In particular, we look at expanding an algebra A, by adding new types such as finite sequences to make a new algebra A^* that models arrays, and adding infinite sequences to make a new algebra $\bar{A}$ that models infinite streams of data.

In section 3 we begin the study of computing on algebras with $\boldsymbol{While}$ programs. For a satisfactory theory, the algebras are required to include the Booleans and standard Boolean operations. Such algebras are called *standard algebras*. The semantics of $\boldsymbol{While}$ programs on A is given by a new technique called *algebraic operational semantics (AOS)*. This involves axiomatising a function $\boldsymbol{Comp}^A$ that defines the state $\boldsymbol{Comp}^A(S,\sigma,t)$ at time t in the computation by program S starting in intial state σ. From this we obtain a *state transformer semantics* in which a program S, applied to a state σ, may give rise to a final state $[\![S]\!]^A(\sigma)$.

Simple but important properties of computations are examined. First, the invariance of computations under homomorphisms and isomorphisms: *if algebras A and B are isomorphic, then the semantical interpretations of any $\boldsymbol{While}$ program S on A and B are isomorphic.* This result has many consequences; for example, it comfirms that executing a $\boldsymbol{While}$ program on equivalent implementations of a data type results in equivalent computations.

The second property is that each computation by a $\boldsymbol{While}$ program S takes place in the subalgebra of A generated by the input. This is a key to understanding the nature of abstract computation: in any algebra computations are local to the input in this sense and, for instance, searches are at best local.

Next, in section 4, we consider the universality of computation by $\boldsymbol{While}$ programs. Let A be an algebra with Booleans and naturals. We can code the $\boldsymbol{While}$ programs

$$S_0,\, S_1,\, S_2, \ldots$$

by the natural numbers in A and ask if there exists a universal $\boldsymbol{While}$ program to compute the function Univ^A on A such that

$$\mathsf{Univ}^A(n,\sigma) \;=\; [\![S_n]\!]^A(\sigma).$$

We prove that the universal function is $\boldsymbol{While}$ computable on A if, and only if, the *term evaluation function* is $\boldsymbol{While}$ computable on A.

The evaluation of terms is not always computable. However, it is $\boldsymbol{While}$ computable in several commonly used algebras such as: semigroups, groups, rings, fields, lattices, Boolean algebras; this because such algebras have computationally efficient normal forms for their terms. For any algebra A, the algebra A^* of finite sequences from A has the property that term evaluation is always ***While*** computable on it; hence, the model of ***While***** programs is universal.

In section 5 we turn our attention to sets. We begin with a study of computable and semicomputable sets. We prove Post's theorem in the present setting. We also study the ideas of *projections* of computable and semicomputable sets. It turns out that the classes of computable and semicomputable sets are *not closed* under projection. The notion of projection is very important since it distinguishes clearly between forms of specification and computation. Furthermore, it focuses our attention to the difference between local search and global search in computation.

Projections also lead us to consider the relationship between ***While*** programming and certain non-deterministic constructs on data. These include: *search procedures; initialisation mechanisms*; and *random assignments.*

Next, with each ***While*** program is associated a *computation tree.* With this technique, we prove that every semicomputable set is definable by an effective infinite disjunction of Boolean terms over the signature.

In section 6 we illustrate the core of the theory with a study of its application to computing sets of real and complex numbers over various many sorted algebras. We include some pleasing examples from dynamical systems.

In Section 7 we return to the special properties and problems of computation of the reals. More generally, we study computation on topological algebras. A key consideration is the property that *if a function is computable then it is continuous.* To guarantee a good selection of applications we use partial functions, which raises interesting topological issues. This study of programming over topological algebras contains new material.

We also contrast exact versus approximate computation on the reals. The following fact was observed in Shepherdson [1976]. Let f be a function on the reals. Then f is computable in the sense of computable analysis if, and only if, there is a function g which is ***While*** computable over the algebra $(\mathbb{R}, \mathbb{B}, \mathbb{N};\ 0,\ 1,\ x+y,\ x.y,\ -x, \dots)$ such that

$$|f(x) - g(n,x)| < 2^{-n}$$

for all $n \in \mathbb{N}$ and $x \in \mathbb{R}$. We extend and adapt this result to topological algebras.

In section 8 we survey other models of computation and see their relation with ***While*** programs. We consider briefly: *μ-recursive functions*; *register machines*; *flowcharts*; *axiomatic methods*; *set recursion*; and *equational definability.* A generalised Church–Turing thesis is discussed.

There are many subjects that we have omitted from the discussion, for example: the delicate classification of the power of constucts, including types; computations with streams; program verification; connections with proof theory; connections with model theory; degree theory; and generalised complexity theory. There will be good work by many authors that we have neglected to mention, from ignorance or forgetfulness. We will be pleased to receive reminders, information and suggestions. Abstract computability theory is a subject that offers its students considerable theoretical scope, many areas of application, and scientific longevity. We hope this chapter provides a first introduction that is satisfying, stimulating and pleasurable.

We thank Jan Bergstra (Amsterdam), Martin Davis (NYU and Berkeley), Jens Erik Fenstad (Oslo), Dag Norman (Oslo), Viggo Stoltenberg-Hansen (Uppsala) and Karen Stephenson (Swansea) for useful discussions on aspects of this work. We thank Peter Lauer (McMaster) and Ithel Jones (Swansea) for discussions and information on the history of programming language semantics. We also thank our colleagues and students in Swansea and McMaster for their helpful responses to the material in courses and seminars, especially Jeff Koster, Matthew Poole, Dafydd Rees, Kristian Stewart, Anton Wilder and Ian Woodhouse.

Special thanks are due to Sol Feferman, who presented some of the material of this chapter at a graduate course in computation theory at Stanford University in spring 1999, and provided us with valuable feedback.

We are particularly grateful to Jane Spurr for her excellent and essential work in producing the final version of the chapter.

The second author is grateful for funding by a grant from the Natural Science and Engineering Research Council of Canada.

1.6 Prerequisites

First, we assume the reader is familiar with the theory of the recursive functions on the natural numbers. It is treated in many books such as Rogers [1967], Mal'cev [1973], Cutland [1980] and Machtey and Young [1978]. An introduction to the subject is contained in this *Handbook* (see Phillips [1992]) and other handbooks (e.g. Enderton [1977]).

Secondly, we assume the reader is familiar with the basics of universal algebra. Some mathematical text-books are: Burris and Sankappanavar [1981] and McKenzie *et al.* [1987]. An introduction to the subject with the needs of computer science in mind is contained in this *Handbook* (see Meinke and Tucker [1992]) and in Wechler [1992]. The application of universal algebra to the specification of data types is treated in Ehrig and Mahr [1985], Meseguer and Goguen [1985] and Wirsing [1991]. The theory of computable and other effective algebras is covered by Stoltenberg-Hansen and Tucker [1995].

Thirdly, we will need some topology. This is covered in many books, such as Dugundji [1966] and Kelley [1955] and in a chapter in this *Handbook* (see Smyth [1992]).

Finally, we note that the subject connects with other subjects, including term rewriting (see, for example, Klop [1992]) and domain theory (see, for example, Stoltenberg-Hansen *et al.* [1994]).

2 Signatures and algebras

In this section we define some basic algebraic concepts, establish notations and introduce three constructions of many-sorted algebras. We will use many-sorted algebras equipped with Booleans, which we call *standard algebras*. Sometimes we use algebras with the natural numbers as well, which we call *N-standard algebras*. All our algebras have total operations, except in section 7, where we compute on topological partial algebras.

We are particularly interested in the effects on computations of adding and removing operations in algebras. To keep track of these changes, we use expansions and reducts of algebras, and relative homomorphisms.

The constructions of new algebras from old involve adding (i) unspecified elements, (ii) finite arrays, and (iii) infinite streams.

2.1 Signatures

Definition 2.1 (Many-sorted signatures). A *signature* Σ (for a many-sorted algebra) is a pair consisting of (1) a finite set $\boldsymbol{Sort}(\Sigma)$ of *sorts*, and (2) a finite set $\boldsymbol{Func}(\Sigma)$ of (*primitive or basic*) *function symbols*, each symbol F having a *type* $s_1 \times \ldots \times s_m \to s$, where $m \geq 0$ is the *arity* of F, and $s_1, \ldots, s_m \in \boldsymbol{Sort}(\Sigma)$ are the *domain sorts* and $s \in \boldsymbol{Sort}(\Sigma)$ is the *range sort*; in such a case we write

$$F : s_1 \times \ldots \times s_m \to s.$$

The case $m = 0$ corresponds to *constant symbols*; we then write $F : \to s$ or just $F : s$.

Our signatures do not explicitly include relation symbols; relations will be interpreted as Boolean-valued functions.

Definition 2.2 (Product types over Σ). A *product type* over Σ, or Σ-*product type*, is a symbol of the form $s_1 \times \ldots \times s_m$ $(m \geq 0)$, where $s_1, \ldots, s_m$ are sorts of Σ, called its *component sorts*. We define $\boldsymbol{ProdType}(\Sigma)$ to be the set of Σ-product types, with elements $u, v, w, \ldots$.

If $u = s_1 \times \ldots \times s_m$, we put $\boldsymbol{lgth}(u) = m$, the *length* of u. When $\boldsymbol{lgth}(u) = 1$, we identify u with its component sort. When $\boldsymbol{lgth}(u) = 0$, u is the *empty product type*.

For a Σ-product type u and Σ-sort s, let $\boldsymbol{Func}(\Sigma)_{u \to s}$ denote the set of all Σ-function symbols of type $u \to s$.

Definition 2.3 (Σ-algebras). A Σ-*algebra* A has, for each sort s of Σ, a non-empty set A_s, called the *carrier of sort* s, and for each Σ-function symbol $F : s_1 \times \ldots \times s_m \to s$, a function $F^A : A_{s_1} \times \cdots \times A_{s_m} \to A_s$.

For a Σ-product type $u = s_1 \times \ldots \times s_m$, we write

$$A^u \ =_{df} \ A_{s_1} \times \ldots \times A_{s_m}.$$

Thus $x \in A^u$ if, and only if, $x = (x_1, \ldots, x_m)$, where $x_i \in A_{s_i}$ for $i = 1, \ldots, m$. So each Σ-function symbol $F : u \to s$ has an interpretation

$F^A : A^u \to A_s$. If u is empty, i.e., F is a constant symbol, then F^A is an element of A_s.

We will sometimes use the same notation for a function symbol F and its interpretation F^A. The meaning will be clear from the context.

For most of this chapter, we make the following assumption.

Assumption 2.4 (Totality). The algebras A are *total*, i.e., F^A is total for each Σ-function symbol F.

Later (in section 7) we will drop this assumption, in our study of partial algebras.

We will sometimes write $\Sigma(A)$ to denote the signature of an algebra A.

We will also consider classes $\mathbb{K}$ of Σ-algebras. In particular, $\boldsymbol{Alg}(\Sigma)$ denotes the class of all Σ-algebras.

We will use the following perspicuous notation for signatures Σ:

$$\boxed{\begin{array}{lll}
\textsf{signature} & \Sigma & \\
\textsf{sorts} & & \\
 & \vdots & \\
 & s, & (s \in \boldsymbol{Sort}(\Sigma)) \\
 & \vdots & \\
\textsf{functions} & & \\
 & \vdots & \\
 & F : s_1 \times \ldots \times s_m \to s, & (F \in \boldsymbol{Func}(\Sigma)) \\
 & \vdots & \\
\textsf{end} & &
\end{array}}$$

and for Σ-structures A:

$$\boxed{\begin{array}{lll}
\textsf{algebra} & A & \\
\textsf{carriers} & & \\
 & \vdots & \\
 & A_s, & (s \in \boldsymbol{Sort}(\Sigma)) \\
 & \vdots & \\
\textsf{functions} & & \\
 & \vdots & \\
 & F^A : A_{s_1} \times \ldots \times A_{s_m} \to A_s, & (F \in \boldsymbol{Func}(\Sigma)) \\
 & \vdots & \\
\textsf{end} & &
\end{array}}$$

Examples 2.5. (a) The algebra of naturals $\mathcal{N}_0 = (\mathbb{N}; 0, \mathsf{succ})$ has a signature containing the sort nat and the function symbols $0{:} \ \to \mathsf{nat}$ and $\mathsf{succ}{:}\mathsf{nat} \to \mathsf{nat}$. We can display this signature thus:

signature	$\Sigma(\mathcal{N}_0)$
sorts	nat
functions	$0\!: \ \to$nat,
	S:nat$\to$nat
end	

In practice, we can display the algebra thus:

algebra	$\mathcal{N}_0$
carriers	$\mathbb{N}$
functions	$0\!: \ \to \mathbb{N}$,
	S: $\mathbb{N} \to \mathbb{N}$
end	

from which the signature can be inferred. Below, we will often display the algebra instead of the signature.
(b) The ring of reals $\mathcal{R}_0 = (\mathbb{R}; 0, 1, +, \text{-}, \times)$ has a carrier $\mathbb{R}$ of sort real, and can be displayed as follows:

algebra	$\mathcal{R}_0$
carriers	$\mathbb{R}$
functions	0,1: $\to \mathbb{R}$,
	$+, \times : \mathbb{R}^2 \to \mathbb{R}$,
	$- : \mathbb{R} \to \mathbb{R}$
end	

(c) The algebra $\mathcal{C}_0$ of complex numbers has two sorts, complex and real, and hence two carriers, $\mathbb{C}$ and $\mathbb{R}$. It includes the algebra $\mathcal{R}_0$, and therefore has all the operations on $\mathbb{R}$ listed in (b), as well as operations on $\mathbb{C}$, as follows:

algebra	$\mathcal{C}_0$
import	$\mathcal{R}_0$
carriers	$\mathbb{C}$
functions	$0, 1, i : \ \to \mathbb{C}$,
	$+, \times : \mathbb{C}^2 \to \mathbb{C}$,
	$- : \ \mathbb{C} \to \mathbb{C}$,
	re,im: $\mathbb{C} \to \mathbb{R}$,
	$\pi : \mathbb{R}^2 \to \mathbb{C}$
end	

where π is the inverse of re and im.
(d) A group has the form

```
algebra     $\mathcal{G}_0$
carriers    $G$
functions   $1: \ \to G$,
            $*: G^2 \to G$,
            inv: $G \to G$
end
```

where the carrier G has sort grp.

The concepts of *reduct* and *expansion* will be important in our work.

Definition 2.6 (Reducts and expansions). Let Σ and Σ' be signatures.

(a) We write $\Sigma \subseteq \Sigma'$ to mean $\boldsymbol{Sort}(\Sigma) \subseteq \boldsymbol{Sort}(\Sigma')$and $\boldsymbol{Func}(\Sigma) \subseteq \boldsymbol{Func}(\Sigma')$.

(b) Suppose $\Sigma \subseteq \Sigma'$. Let A and A' be algebras with signatures Σ and Σ' respectively.

  (i) The Σ-*reduct* $A'|_\Sigma$ *of* A' is the algebra of signature Σ, consisting of the carriers of A' named by the sorts of Σ and equipped with the functions of A' named by the function symbols of Σ.

  (ii) A' is a Σ'-*expansion* of A iff A is the Σ-reduct of A'.

Example 2.7. The algebra $\mathcal{C}_0$ (see Example 2.5(c)) is an expansion of $\mathcal{R}_0$ to $\Sigma(\mathcal{C}_0)$.

Definition 2.8 (Function types). We collect some definitions and notation. Let A be a Σ-algebra.

(a) A *function type* over Σ, or Σ-*function type*, is a symbol of the form $u \to v$, with *domain type* u and *range type* v, where u and v are Σ-product types.

(b) For any Σ-function type $u \to v$, a *function of type* $u \to v$ *over* A is a function

$$f : A^u \to A^v. \tag{2.1}$$

If $v = s_1 \times \ldots \times s_n$ then the *component functions* of f are $f_1, \ldots, f_n$, where

$$f_j : A^u \to A_{s_j} \tag{2.2}$$

for $j = 1, \ldots, n$, and for all $x \in A^u$,

$$f(x) \ \simeq \ (f_1(x), \ldots, f_n(x)). \tag{2.3}$$

(We will explain the '$\simeq$' in (c) below.) Conversely, given n functions f_j as in (2.2), all with the same domain type u, and with range types (or sorts) $s_1, \ldots, s_n$ respectively, we can form their *vectorisation* as a function f satisfying (2.1) and (2.3).

We will investigate computable *vector-valued functions* (2.1) over A.

(c) Although all the primitive functions of Σ are *total*, the computable functions on the Σ-algebra may very well be *partial*, as we will see.
We use the following notation: if $f : A^u \to A_s$ and $x \in A^u$, then $f(x)\uparrow$ ('$f(x)$ diverges') means that $x \notin \boldsymbol{dom}(f)$; $f(x)\downarrow$ ('$f(x)$ converges') means that $x \in \boldsymbol{dom}(f)$; and $f(x)\downarrow y$ ('$f(x)$ converges to y') means that $x \in \boldsymbol{dom}(f)$ and $f(x) = y$.
We also make the following convention for convergence of vector-valued functions: in the notation of (2.1) and (2.2), for any $x \in A^u$, we say that $f(x)\downarrow$ if, and only if, $f_j(x)\downarrow$ for *every component function* f_j of f, in which case $f(x) = (f_1(x), \dots, f_n(x))$. Otherwise (i.e., if $f_j(x)\uparrow$ for any j with $1 \le j \le n$), we say that $f(x)\uparrow$. (That is the meaning of the symbol '$\simeq$' in (2.3) above.)

Definition 2.9 (Relations; projections of relations). We collect some more definitions and notation.

(a) A relation on A of *type* u is a subset of A^u. We write $R : u$ if R is a relation of type u.
Let R be a relation on A of type $u = s_1 \times \dots \times s_m$.

(b) The *characteristic function* of R is the function $\chi_R : A^u \to \mathbb{B}$ which takes the values $\mathtt{tt}$ on R and $\mathtt{ff}$ off R.

(c) The *complement of* R in A is the relation

$$R^c \;=\; A^u \backslash R \;=\; \{a \in A^u \mid a \notin R\},$$

also of type u.

(d) (*Projections.*) To explain this notion, we begin with an example. Suppose $R : u$ where $u = s_1 \times s_2 \times s_3 \times s_4 \times s_5$. Now let $v = s_1 \times s_2 \times s_3$ and $w = s_4 \times s_5$. Then the *projection of* R *on* v (or *on* A^v), or the A^w*-projection of* R, is the relation $S : v$ defined by existentially quantifying over A^w:

$$S(x_1, x_2, x_3) \iff \exists x_4, x_5 \in A^w : R(x_1, \dots, x_5).$$

More generally (with $R : u$ where $u = s_1 \times \dots \times s_m$) let $\vec{i}$ be any list of numbers $i_1, \dots, i_r$ such that $1 \le i_1 < \dots < i_r \le m$, and let $\vec{j} = j_1, \dots, j_{m-r}$,list $\{1, \dots, m\} \setminus \vec{i}$. Then $u|\vec{i}$ denotes the *restriction of* u to $\vec{i}$, that is, the product type $s_{i_1} \times \dots \times s_{i_r}$; and $\boldsymbol{proj}[u|\vec{i}\,](R)$ is the *projection of* R *on* $\vec{i}$ (or *on* $A^{u|\vec{i}}$), or the $A^{u|\vec{j}}$-*projection of* R, that is, the relation $S : u|\vec{i}$ defined by existentially quantifying over $A^{u|\vec{j}}$:

$$S(x_{i_1}, \dots, x_{i_r}) \iff \exists x_{j_1}, \dots, x_{j_{m-r}} \in A^{u|\vec{j}} : R(x_1, \dots, x_m).$$

2.2 Terms and subalgebras

Definition 2.10 (Closed terms over Σ). We define the class $\boldsymbol{T}(\Sigma)$ of *closed terms over* Σ, denoted $t, t', t_1, \dots$, and for each Σ-sort s, the class $\boldsymbol{T}(\Sigma)_s$of closed terms of sort s. These are generated inductively by the rule: if $F \in \boldsymbol{Func}(\Sigma)_{u \to s}$ and $t_i \in \boldsymbol{T}(\Sigma)_{s_i}$ for $i = 1, \dots, m$, where $u = s_1 \times \dots \times s_m$, then $F(t_1, \dots, t_m) \in \boldsymbol{T}(\Sigma)_s$.

Note that the implicit base case of this inductive definition is that of $m = 0$, which yields: for all constants $c : \to s, c() \in \boldsymbol{T}(\Sigma)_s$. In this case we write c instead of $c()$. Hence if Σ contains no constants, $\boldsymbol{T}(\Sigma)$ is empty.

Definition 2.11 (Valuation of closed terms). For $A \in \boldsymbol{Alg}(\Sigma)$ and $t \in \boldsymbol{T}(\Sigma)_s$, we define the *valuation* $t_A \in A_s$ *of* t *in* A by structural induction on t:

$$F(t_1, \dots, t_m)_A \; = \; F^A((t_1)_A, \dots, (t_m)_A).$$

In particular, for $m = 0$, i.e., for a constant $c : \to s$,

$$c_A = c^A.$$

We want a situation where $\boldsymbol{T}(\Sigma)$ is non-empty, and, in fact, $\boldsymbol{T}(\Sigma)_s$ is non-empty for each $s \in \boldsymbol{Sort}(\Sigma)$. We therefore proceed as follows.

Definition 2.12. The signature Σ is said to be:

(a) *non-void at sort s* if $\boldsymbol{T}(\Sigma)_s \neq \emptyset$;

(b) *non-void* if it is non-void at all Σ-sorts.

Assumption 2.13 (Instantiation). Σ is non-void.

Throughout this paper we will make this assumption, except where explicitly stated: see, for example, Remark 2.31(e). It simplifies the theory of many-sorted algebras (see Meinke and Tucker [1992]).

Definition 2.14 (Default terms; default values).

(a) For each sort s, we pick a closed term of sort s. (There is at least one, by the instantiation assumption.) We call this the *default term of sort* s, written $\boldsymbol{\delta}^s$. Further, for each product type $u = s_1 \times \dots \times s_m$ of Σ, the *default (term) tuple of type* u, written $\boldsymbol{\delta}^u$, is the tuple of default terms $(\boldsymbol{\delta}^{s_1}, \dots, \boldsymbol{\delta}^{s_m})$.

(b) Given a Σ-algebra A, for any sort s, the *default value) of sort s in A* is the valuation $\boldsymbol{\delta}^s_A \in A_s$ of the default term, $\boldsymbol{\delta}^s$; and for any product type $u = s_1 \times \dots \times s_m$, the *default (value) tuple of type u in A* is the tuple of default values $\boldsymbol{\delta}^u A = (\boldsymbol{\delta}^{s_1}_A, \dots, \boldsymbol{\delta}^{s_m}_A) \in A^u$.

Definition 2.15 (Generated subalgebras). Let $X \subseteq \bigcup_{s \in \boldsymbol{Sort}(\Sigma)} A_s$. Then $\langle \mathrm{X} \rangle^A$ is the (Σ-)*subalgebra of A generated by X*, i.e., the smallest subalgebra of A which contains X, and $\langle \mathrm{X} \rangle^A_s$ is the carrier of $\langle \mathrm{X} \rangle^A$ of sort

s. (See Meinke and Tucker [1992, §§3.2.6 *ff.*] for definitions.) Also for a product type $u = s_1 \times \ldots \times s_m$,

$$\langle X \rangle_u^A \; = \; \langle X \rangle_{s_1}^A \times \ldots \times \langle X \rangle_{s_m}^A.$$

Similarly, for a tuple $a \in A^u$, $\langle \mathrm{a} \rangle^A$ is the (Σ-)*subalgebra of A generated by a*, etc.

Remark 2.16.

(a) Using the terminology of sections 3.1–3.3, we can characterise (for all Σ-sorts s and Σ-product types u) the sets $\langle \mathrm{X} \rangle_s^A$ and $\langle \mathrm{X} \rangle_u^A$ by
$\langle X \rangle_s^A = \{ [\![\mathrm{t}]\!]^A \sigma \mid t \in \boldsymbol{Term}_s(\Sigma)$ and for all x $\in \boldsymbol{var}(t), \sigma(\mathrm{x}) \in X \}$
$\langle X \rangle_u^A = \{ [\![\mathrm{t}]\!]^A \sigma \mid t \in \boldsymbol{TermTup}_u(\Sigma)$ and for all x $\in \boldsymbol{var}(t), \sigma(\mathrm{x}) \in X. \}$

(b) The smallest subalgebra of A is its *closed term subalgebra*, given by

$$\langle \emptyset \rangle^A \; = \; \{ t_A \mid t \in \boldsymbol{T}(\Sigma) \}.$$

(c) The instantiation assumption implies that for any X and every sort s, $\langle \mathrm{X} \rangle_s^A \neq \emptyset$.

Definition 2.17 (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 $A_s = \langle \emptyset \rangle_s^A$, i.e., A_s is generated by the closed Σ-terms of sort s.

(b) A is *minimal* if it is minimal at every Σ-sort.

Example 2.18. To take examples from later:

(a) Every N-standard algebra (section 2.5) is minimal at sorts bool and nat.

(b) The ring of reals $\mathcal{R}_0$ (Example 2.5) (or its standardisation (section 2.4) or N-standardisation (section 2.5)) is not minimal at sort real.

2.3 Homomorphisms, isomorphisms and abstract data types

Given a signature Σ, the notions of Σ-*homomorphism* as well as Σ-*epimorphism* (surjective), Σ-*monomorphism* (injective), Σ-*isomorphism* (bijective) and Σ-*automorphism* are defined as usual (see [Meinke and Tucker, 1992, §3.4]). We need a more sophisticated notion, that of *relative homomorphism.*

Definition 2.19 (Relative homomorphism and isomorphism). Let Σ and Σ' be signatures with $\Sigma \subseteq \Sigma'$. Let A and B be two standard Σ'-algebras such that

$$A|_\Sigma \; = \; B|_\Sigma.$$

(*a*) A Σ'-*homomorphism relative to* Σ from A to B, or a Σ'/Σ-*homomorphism* $\phi : A \to B$, is a $\boldsymbol{Sort}(\Sigma')$-indexed family of mappings

$$\phi \;=\; \langle \phi_s : A_s \to B_s \mid s \in \boldsymbol{Sort}(\Sigma') \rangle$$

which is a Σ'-homomorphism from A to B, such that for all $s \in \boldsymbol{Sort}(\Sigma)$, ϕ_s is the *identity* on A_s.

(*b*) A Σ'/Σ-*isomorphism* from A to B is a Σ'/Σ-homomorphism which is also a Σ'-isomorphism from A to B.

(*c*) A and B are Σ'/Σ-*isomorphic*, written $A \cong_{\Sigma'/\Sigma} B$, if there is a Σ'/Σ-isomorphism from A to B.

Definition 2.20 (Abstract data types). An *abstract data type* of signature Σ (Σ-$\boldsymbol{adt}$) is defined to be a class $\mathbb{K}$ of Σ-algebras *closed under* Σ-*isomorphism.* Examples of Σ-$\boldsymbol{adt}$'s are:

(*a*) the class $\boldsymbol{Mod}(\Sigma,\mathrm{T})$ of all models of a first-order Σ-theory T;

(*b*) the isomorphism class of a particular Σ-algebra.

2.4 Adding Booleans: Standard signatures and algebras

An very important signature for our purposes is the signature of *Booleans*:

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

The algebra $\mathcal{B}$ of Booleans, with signature $\Sigma(\mathcal{B})$, has the carrier $\mathbb{B} = \{\mathtt{tt}, \mathtt{ff}\}$ of sort bool, and, as constants and functions, the standard interpretations of the function and constant symbols of $\Sigma(\mathcal{B})$. Thus, for example, $\mathsf{true}^{\mathcal{B}} = \mathtt{tt}$ and $\mathsf{false}^{\mathcal{B}} = \mathtt{ff}$.

Of particular interest to us are those signatures and algebras which contain $\Sigma(\mathcal{B})$ and $\mathcal{B}$.

Definition 2.21 (Standard signatures and algebras).

(*a*) A signature Σ is a *standard signature* if

(*i*) $\Sigma(\mathcal{B}) \subseteq \Sigma$, and

(*ii*) the function symbols of Σ include a *discriminator*

$$\mathtt{if}_s : \mathtt{bool} \times s^2 \to s$$

for all sorts s of Σ other than bool, and an *equality operator*

$$\mathsf{eq}_s : s^2 \to \mathsf{bool}$$

for certain sorts s.

(b) Given a standard signature Σ, a Σ-algebra A is a *standard algebra* if

(i) it is an expansion of $\mathcal{B}$, and

(ii) the discriminators and equality operators have their standard interpretation in A; i.e., for $b \in \mathbb{B}$ and $x, y \in A_s$,

$$\mathtt{if}_s(b,x,y) \;=\; \begin{cases} x \text{ if } b = \mathsf{tt} \\ y \text{ if } b = \mathsf{ff}, \end{cases}$$

and eq_s is interpreted as the *identity* on each equality sort s.

Let $\boldsymbol{EqSort}(\Sigma) \subseteq \boldsymbol{Sort}(\Sigma)$ denote the set of equality sorts of Σ, and let $\boldsymbol{StdAlg}(\Sigma)$ denote the class of standard Σ-algebras.

Remark 2.22.

(a) Strictly speaking, the definition of standardness of a signature Σ or algebra depends on the choice of the set $\boldsymbol{EqSort}(\Sigma)$ of equality sorts of Σ. However, our terminology and notation will not make this dependence explicit.

(b) The exact choice of the set of propositional connectives in $\mathcal{B}$ is not crucial; any complete set would do.

(c) Excluding the sort bool from the sorts of the discriminator is not significant; we can easily define $\mathtt{if}_{\mathtt{bool}}$ from the other Boolean operators. Also, $\mathtt{eq}_{\mathtt{bool}}$ can easily be defined. (*Exercise.*)

(d) Any many-sorted signature Σ can be *standardised* to a signature Σ^B by adjoining the sort bool together with the standard Boolean operations; and, correspondingly, any algebra A can be standardised to an algebra A^B by adjoining the algebra $\mathcal{B}$ and the discriminator and equality operators. Note that both A and $\mathcal{B}$ are reducts of this standardisation A^B. (See the examples below.)

(e) If A and B are two standard Σ-algebras, then any Σ-homorphism from A to B is actually a $\Sigma/\Sigma(\mathcal{B})$-homomorphism, i.e., it fixes the reduct $\mathcal{B}$.

Examples 2.23.

(a) The simplest standard algebra is the algebra $\mathcal{B}$ of the Booleans.

(b) The standard algebra of naturals $\mathcal{N}$ is formed by standardising the algebra $\mathcal{N}_0$ of Example 2.5(a), with nat as an equality sort, and, further, adjoining the order relation[1] $\mathsf{less}_{\mathsf{nat}}$ on $\mathbb{N}$:

[1]The reason for adjoining $\mathsf{less}_{\mathsf{nat}}$ will be clear later: in the proof of Theorem 3.63 (Σ^*/Σ conservativity for terms), we need it for the translation of Σ^*-terms to Σ^N-terms.

$$\boxed{\begin{array}{ll}
\textsf{algebra} & \mathcal{N} \\
\textsf{import} & \mathcal{N}_0, \mathcal{B} \\
\textsf{functions} & \mathtt{if}_{\mathtt{nat}} : \mathbb{B} \times \mathbb{N}^2 \to \mathbb{N}, \\
 & \textsf{eq}_{\textsf{nat}}, \textsf{less}_{\textsf{nat}} : \mathbb{N}^2 \to \mathbb{B} \\
\textsf{end} &
\end{array}}$$

(*c*) The standard algebra $\mathcal{R}$ of reals is formed similarly by standardising the ring $\mathcal{R}_0$ of Example 2.5(*b*), with real as an equality sort:

$$\boxed{\begin{array}{ll}
\textsf{algebra} & \mathcal{R} \\
\textsf{import} & \mathcal{R}_0, \mathcal{B} \\
\textsf{functions} & \textsf{if}_{\textsf{real}} : \mathbb{B} \times \mathbb{R}^2 \to \mathbb{R}, \\
 & \textsf{eq}_{\textsf{real}} : \mathbb{R}^2 \to \mathbb{B} \\
\textsf{end} &
\end{array}}$$

(*d*) We will also be interested (in section 5) in the expansion $\mathcal{R}^{<}$ of $\mathcal{R}$ formed by adjoining the order relation on the reals $\textsf{less}_{\textsf{real}}$: $\mathbb{R}^2 \to \mathbb{B}$, thus:

$$\boxed{\begin{array}{ll}
\textsf{algebra} & \mathcal{R}^{<} \\
\textsf{import} & \mathcal{R} \\
\textsf{functions} & \textsf{less}_{\textsf{real}} : \mathbb{R}^2 \to \mathbb{B} \\
\textsf{end} &
\end{array}}$$

(*e*) The standard algebra $\mathcal{C}$ of complex numbers $\mathbb{C}$ is formed similarly by standardising the algebra $\mathcal{C}_0$ of Example 2.5(*c*), with equality on both $\mathbb{R}$ and $\mathbb{C}$.

(*f*) Again, we will consider the expansion $\mathcal{C}^{<}$ of $\mathcal{C}$ formed by adjoining $\textsf{less}_{\textsf{real}}$.

(*g*) The standard group $\mathcal{G}$ is formed similarly by standardising the group $\mathcal{G}_0$, with equality on $\mathbb{G}$.

Throughout this chapter, we will assume the following, unless otherwise stated.

Assumption 2.24 (Standardness). The signature Σ and the Σ-algebra A are standard.

2.5 Adding counters: N-standard signatures and algebras

Definition 2.25.

(*a*) A standard signature Σ is called *N-standard* if it includes (as well as bool) the *numerical sort* nat, as well as function symbols for the *standard operations* of *zero*, *successor* and *order* on the naturals:

$$\begin{array}{rll}
0: & & \to \textsf{nat} \\
\textsf{S}: & \textsf{nat} & \to \textsf{nat} \\
\textsf{less}_{\textsf{nat}}: & \textsf{nat}^2 & \to \textsf{bool}
\end{array}$$

as well as the *discriminator* $\mathsf{if}_{\mathsf{nat}}$ and the *equality operator* $\mathsf{eq}_{\mathsf{nat}}$ on nat.

(*b*) The corresponding Σ-algebra A is *N-standard* if the carrier A_{nat} is the set of natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$, and the standard operations (listed above) have their *standard interpretations* on $\mathbb{N}$.

Definition 2.26.

(*a*) The *N-standardisation* Σ^N of a standard signature Σ is formed by adjoining the sort nat and the operations $0, S$, $\mathsf{eq}_{\mathsf{nat}}$, $\mathsf{less}_{\mathsf{nat}}$ and $\mathsf{if}_{\mathsf{nat}}$.

(*b*) The *N-standardisation* A^N of a standard Σ-algebra A is the Σ^N-algebra formed by adjoining the carrier $\mathbb{N}$ together with certain standard operations to A, thus:

```
algebra     $A^N$
import      $A$
carriers    $\mathbb{N}$
functions   $0: \to \mathbb{N}$
            $\mathsf{S}: \mathbb{N} \to \mathbb{N}$
            $\mathsf{if}_{\mathsf{nat}}: \mathbb{B} \times \mathbb{N}^2 \to \mathbb{N}$
            $\mathsf{eq}_{\mathsf{nat}}, \mathsf{less}_{\mathsf{nat}}: \mathbb{N}^2 \to \mathbb{B}$
end
```

(*c*) The *N-standardisation* $\mathbb{K}^N$ of a class $\mathbb{K}$ of Σ-algebras is (the closure with respect to Σ^N/Σ-isomorphism of) the class $\{A^N \mid A \in \mathbb{K}\}$.

Examples 2.27.

(*a*) The simplest N-standard algebra is the algebra $\mathcal{N}$ of Example 2.23(*b*).

(*b*) We can N-standardise the real and complex rings $\mathcal{R}$ and $\mathcal{C}$, and the group $\mathcal{G}$ of Examples 2.23, to form the algebras $\mathcal{R}^N$, $\mathcal{C}^N$ and $\mathcal{G}^N$, respectively.

Remark 2.28.

(*a*) For any standard A, both A and $\mathcal{N}$ are Σ-*reducts* of the N-standardisation A^N (cf. Remark 2.22(*d*)).

(*b*) If A and B are two N-standard Σ-algebras, then any Σ-homorphism from A to B is actually a $\Sigma/\Sigma(\mathcal{N})$-homomorphism, i.e., it fixes the reduct $\mathcal{N}$ (cf. Remark 2.22(*e*)).

(*c*) A Σ-homomorphism (or Σ-isomorphism) between two standard Σ-algebras A and B can be *extended* to a Σ-homomorphism (or Σ-isomorphism) between A^N and B^N. (*Exercise.*)

(*d*) If A is already N-standard, then A^N will contain a second copy of $\mathbb{N}$, with (only) the standard operations on it. Further, A^N can be *effectively coded within* A, using a standard coding of $\mathbb{N}^2$ in $\mathbb{N}$. (*Check.*)

(*e*) In particular, $(A^N)^N$ can be *effectively coded within* A^N.

We will occasionally have use of a notion stricter than N-standardness.

Definition 2.29 (Strict N-standardness).

(a) An N-standard signature Σ is said to be *strictly N-standard* if its only function symbols with range sort nat are '0', 'S' and '$\mathsf{if}_{\mathsf{nat}}$'.

(b) An N-standard algebra is *strictly N-standard* if its signature is.

Note that the N-standardisation of any algebra is strictly N-standard.

2.6 Adding the unspecified value ui; Algebras A^{u} of signature Σ^{u}

In this subsection, we need not assume that Σ and A are standard. For each sort s of Σ let ui be a new object, representing an 'unspecified value', and let $A^{\mathsf{u}}_s = A_s \cup \{\mathsf{ui}_s\}$. For each function symbol F of Σ of type $s_1 \times \ldots \times s_m \to s$, extend its interpretation F^A on A to a function

$$F^{A,\mathsf{u}} : A^{\mathsf{u}}_{s_1} \times \ldots \times A^{\mathsf{u}}_{s_m} \longrightarrow A^{\mathsf{u}}_s$$

by *strictness* — i.e. the value is defined as ui whenever any argument is ui. Then the algebra A^{u}, with signature Σ^{u}, contains:

(i) the original carriers A_s of sort s, and functions F^A on them;

(ii) the new carriers A^{u}_s of sort s^{u}, and functions $F^{A,\mathsf{u}}$ on them;

(iii) a constant $\mathtt{unspecs}_s : s^{\mathsf{u}}$ to denote ui_s as a distinguished element of A^{u}_s; and

(iv) an *embedding function* $\mathsf{i}_s : s \to s^{\mathsf{u}}$ to denote the embedding of A_s into A^{u}_s, and the *inverse* function $\mathsf{j}_s : s^{\mathsf{u}} \to s$, mapping ui_s to the default term $\boldsymbol{\delta}^s$ for each sort s.

Further, if A is a standard algebra, we assume A^{u} also includes:

(v) a Boolean-valued function $\mathtt{Unspecs}_s : s^{\mathsf{u}} \to \mathtt{bool}$, the characteristic function of ui_s;

(vi) the *discriminator* on A^{u}_s for each sort s; and

(vii) the *equality* operator on A^{u}_s for each equality sort s.

Thus, if A is standard, A^{u} is constructed from A as follows:

algebra	A^{u}	
import	A	
carriers	A^{u}_s	$(s \in \boldsymbol{S})$
functions	$\mathsf{ui}_s : \ \to A^{\mathsf{u}}_s$	$(s \in \boldsymbol{S})$,
	$F^{A,\mathsf{u}} : A^{\mathsf{u}}_{s_1} \times \ldots \times A^{\mathsf{u}}_{s_m} \to A^{\mathsf{u}}_s$	$(F : s_1 \times \ldots \times s_m \to s \text{ in } \Sigma)$,
	$\mathsf{i}_s : A_s \to A^{\mathsf{u}}_s$	$(s \in \boldsymbol{S})$,
	$\mathsf{j}_s : A^{\mathsf{u}}_s \to A_s$	$(s \in \boldsymbol{S})$,
	$\mathsf{Unspec}_s : A^{\mathsf{u}}_s \to \mathbb{B}$	$(s \in \boldsymbol{S})$,
	$\mathsf{if}_{s^{\mathsf{u}}} : \mathbb{B} \times (A^{\mathsf{u}}_s)^2 \to A^{\mathsf{u}}_s$	$(s \in \boldsymbol{S})$,
	$\mathsf{eq}_{s^{\mathsf{u}}} : (A^{\mathsf{u}}_s)^2 \to \mathbb{B}$	$(s \in \boldsymbol{S}_e)$
end		

where $\boldsymbol{S}$=$\boldsymbol{Sort}(\Sigma)$and $\boldsymbol{S}_e = \boldsymbol{EqSort}(\Sigma)$(and the superscript A has been dropped from the new function symbols).

Also, $\mathbb{K}^{\mathsf{u}}$ is (the closure with respect to $\Sigma^{\mathsf{u}}/\Sigma$-isomorphism of) the class $\{A^{\mathsf{u}} \mid A \in \mathbb{K}\}$.

Remark 2.30.

(*a*) The algebra A^{u} is a Σ^{u}-*expansion* of A. If Σ has r sorts, then Σ^{u} has $2r$ sorts.

(*b*) If A is standard, then so is A^{u}.

(*c*) Suppose A (and hence A^{u}) is standard. Then A^{u} can be *effectively coded within* A. Each element y of A^{u}_s is represented by the *pair* $(b, \mathsf{j}_s(y)) \in \mathbb{B} \times A_s$, where $b = \mathtt{tt}$ if $y \neq \mathsf{ui}_s$ and $b = \mathtt{ff}$ otherwise. This induces, in an obvious way, a coding of the operations on A^{u} by operations on A. (The coding is described in [Tucker and Zucker, 1988] for a slightly different definition of A^{u}—however, it is clear how to modify that for the present context.)

(*d*) (*Two- and three-valued Boolean operations.*) Suppose again that A is standard. Then A^{u} contains the carrier $\mathbb{B}^{\mathsf{u}} = \{\mathtt{tt}, \mathtt{ff}, \mathsf{ui}\}$ as well as $\mathbb{B}$, with associated extensions of the original standard Boolean operations, leading to a *weak three-valued logic* (see [Kleene, 1952; Tucker and Zucker, 1988]). Further, there are *two equality operations* on A^{u}_s for each equality sort s:
(*i*) the extension by strictness of eq^A_s to a three-valued function

$$\mathsf{eq}^{A,\mathsf{u}}_s : A^{\mathsf{u}}_s \times A^{\mathsf{u}}_s \to \mathbb{B}^{\mathsf{u}}$$

which has the value $\mathsf{ui}_{\mathtt{bool}}$ if either argument is ui_s; (*ii*) the standard (two-valued) equality on A^{u}_s,

$$\mathsf{eq}^{A^{\mathsf{u}}}_s : A^{\mathsf{u}}_s \times A^{\mathsf{u}}_s \to \mathbb{B},$$

which we will usually denote by '=' in infix.

(*e*) Some of the functions in A^{u} are *not strict*, namely the (interpretations of) the discriminator $\mathsf{if}_{s}^{\mathsf{u}}$, the function Unspec_s and the two-valued equality operator eq_{u} (see (*d*)(*ii*) above).

(*f*) A Σ-homomorphism (or Σ-isomorphism) between two standard Σ-algebras A and B can be extended to a Σ-homomorphism (or Σ-isomorphism) between A^{u} and B^{u}. (*Exercise.*)

2.7 Adding arrays: Algebras A^* of signature Σ^*

Given a standard signature Σ, and standard Σ-algebra A, we extend Σ and expand A in three stages:

(1°) Construct Σ^{u} and A^{u}, as in section 2.6.

(2°) N-standardise these to form $\Sigma^{\mathsf{u},N}$ and $A^{\mathsf{u},N}$, as in section 2.5.

(3°) Define, for each sort s of Σ, the carrier A^*_s to be the set of pairs

$$a^* = (\alpha, l)$$

where $\alpha : \mathbb{N} \to A_s^{\mathsf{u}}$, $l \in \mathbb{N}$ and, for all $n \geq l$,

$$\alpha(n) = \mathsf{ui}_s.$$

So l is a witness to the "finiteness" of α, or an 'effective upper bound' for a^*. The elements of A_s^* have "starred sort" s^*, and can be considered as finite sequences or *arrays*. The resulting algebras A^* have signature Σ^*, which extends $\Sigma^{\mathsf{u},N}$ by including, for each sort s of Σ, the new starred sorts s^* (in addition to s^{u}), and also the following new function symbols:

(*i*) the null array Null_s of type s^*, where

$$\mathsf{Null}_s^A = (\lambda n \cdot \mathsf{ui}_s,\, 0) \in A_s^*;$$

(*ii*) the application operator Ap_s of type $s^* \times \mathsf{nat} \to s^{\mathsf{u}}$, where

$$\mathsf{Ap}_s^A((\alpha,\, l), n) = \alpha(n);$$

(*iii*) the Update_s operator of type $s^* \times \mathsf{nat} \times s^{\mathsf{u}} \to s^*$, where for $(\alpha, l) \in A_s^*, n \in \mathbb{N}$ and $x \in A_s^{\mathsf{u}}$, $\mathsf{Update}_s^A((\alpha, l), n, x)$ is the array $(\beta, l) \in A_s^*$ such that for all $k \in \mathbb{N}$,

$$\beta(k) = \begin{cases} \alpha(k) & \text{if } \; k < l, k \neq n \\ x & \text{if } \; k < l, k = n \\ \mathsf{ui}_s & \text{otherwise;} \end{cases}$$

(*iv*) the Lgth_s operator, of type $s^* \to \mathsf{nat}$, where

$$\mathsf{Lgth}_s^A((\alpha, l)) = l;$$

(*v*) the $\mathsf{Newlength}_s$ operator of type $s^* \times \mathsf{nat} \to s^*$, where $\mathsf{Newlength}_s^A((\alpha, l), m)$ is the array (β, m) such that for all k,

$$\beta(k) = \begin{cases} \alpha(k) & \text{if } \; k < m \\ \mathsf{ui}_s & \text{if } \; k \geq m; \end{cases}$$

(*vi*) the *discriminator* on A_s^* for each sort s; and
(*vii*) the *equality* operator on A_s^* for each equality sort s.

The justification for (*vii*) is that if a sort s has 'computable' equality, then clearly so has the sort s^*, since it amounts to testing equality of finitely many pairs of objects of sort s, up to a computable length.

For $a^* \in A_s^*$ and $n \in \mathbb{N}$, we write $a^*[n]$ for $\mathsf{j}_s^A(\mathsf{Ap}_s^A(a^*, n))$. Thus $a^*[n]$ is the element of A_s 'corresponding to' $\mathsf{Ap}(a^*, n) \in A_s^{\mathsf{u}}$.

To depict this construction of A^* from a standard A: suppose we have constructed A^{u} as in section 2.6, and then N-standardised it to $A^{\mathsf{u},N}$ as in section 2.5. We now proceed as follows:

algebra	A^*	
import	$A^{\mathsf{u},N}$	
carriers	A^*_s	$(s \in \boldsymbol{S})$
functions	$\mathsf{Null}_s : \ \to A^*_s$	$(s \in \boldsymbol{S})$
	$\mathsf{Ap}_s : A^*_s \times \mathbb{N} \to A^{\mathsf{u}}_s$	$(s \in \boldsymbol{S})$
	$\mathsf{Update}_s : A^*_s \times \mathbb{N} \times A^{\mathsf{u}}_s \to A^*_s$	$(s \in \boldsymbol{S})$
	$\mathsf{Lgth}_s : A^*_s \to \mathbb{N}$	$(s \in \boldsymbol{S})$
	$\mathsf{Newlength}_s : A^*_s \times \mathbb{N} \to A^*_s$	$(s \in \boldsymbol{S})$
	$\mathsf{if}_{s^*} : \mathbb{B} \times (A^*_s)^2 \to A^*_s$	$(s \in \boldsymbol{S})$
	$\mathsf{eq}_{s^*} : (A^*_s)^2 \to \mathbb{B}$	$(s \in \boldsymbol{S}_e)$
end		

where again $\boldsymbol{S} = \boldsymbol{Sort}(\Sigma)$and $\boldsymbol{S}_e = \boldsymbol{EqSort}(\Sigma)$(and the superscript A has been dropped from the new function symbols).

Also, $\mathbb{K}^*$ is (the closure with respect to Σ^*/Σ-isomorphism of) the class $\{A^* \mid A \in \mathbb{K}\}$.

Remark 2.31.

(a) The algebra A^* is a Σ^*-*expansion* of A^{u}, and (hence) of A. If Σ has r sorts, then Σ^* has $3r+1$ sorts, namely s, s^{u} and s^* for each sort s of Σ, and also nat.

(b) Σ^* and A^* are N-standard.

(c) (*Internal versions of* A^* *and* Σ^*.) Suppose A is N-standard. Then A^N has a second copy of $\mathbb{N}$, and, according to our definition above, A^* is constructed on A^N using this second copy of $\mathbb{N}$. Let $A^{*\prime}$ (of sort $\Sigma^{*\prime}$) be an alternative version of A^* constructed on A, using the 'original' copy of $\mathbb{N}$. Then A^* and $A^{*\prime}$ can be effectively coded in each other. (*Check*; cf. Remark 2.28(d).) We call $A^{*\prime}$ and $\Sigma^{*\prime}$ *internal versions* of A^* and Σ^*, respectively.

(d) We may also need to speak of finite sequences of starred sorts. However, we do not have to introduce an algebra $(A^*)^*$ of 'doubly starred' carrier sets containing 'two-dimensional arrays'; such an algebra can be effectively coded in A^*, since we can effectively code a finite sequence of starred objects of a given sort as a single starred object of the same sort, thanks to the explicit Lgth operation. More precisely, a sequence $x^*_0, \ldots, x^*_{k-1}$ of elements of A^*_s (for some sort s) can be coded as a *pair* $(y^*, n^*) \in A^*_s \times \mathbb{N}^*$, where $\mathsf{Lgth}(n^*) = k$, and, for $0 \le j < k, n^*[j] = \mathsf{Lgth}(x^*_j)$, and $\mathsf{Lgth}(y^*) = n^*[0]+\ldots+n^*[k-1]$, and for $1 \le j \le k$ and $0 \le i < n^*[j]$, $y^*[n^*[0] + \ldots + n^*[j-1] + i] = x^*_j[i]$.

(e) A Σ-homomorphism (or Σ-isomorphism) between two standard Σ-algebras A and B can be extended to a Σ-homomorphism (or Σ-isomorphism) between A^* and B^*. (*Exercise*; cf. Remarks 2.28(c)

and 2.30(f).)

(f) The reason for introducing starred sorts is the lack of effective coding of finite sequences within abstract algebras in general.

(g) Starred sorts have significance in programming languages, since starred variables can be used to model arrays, and (hence) *finite but unbounded memory.*

2.8 Adding streams: Algebras $\bar{A}$ of signature $\overline{\Sigma}$

Let, again, Σ be a standard signature, and A a standard Σ-algebra. We define an extension of Σ and a corresponding expansion of A, alternative to Σ^* and A^*.

First we N-standardise Σ and A, to form Σ^N and A^N.

Then we choose a set $\boldsymbol{S} \subseteq \boldsymbol{Sort}(\Sigma)$ of *pre-stream sorts.* We then extend Σ^N to a *stream signature* $\overline{\Sigma}^{\boldsymbol{S}}$ *relative to* $\boldsymbol{S}$, in the following way.

(a) With each $s \in \boldsymbol{S}$, we associate a new *stream sort* $\bar{s}$, also written $\mathsf{nat} \to s$. Then $\boldsymbol{Sort}(\overline{\Sigma}^{\boldsymbol{S}})=\boldsymbol{Sort}(\Sigma)\cup\bar{\boldsymbol{S}}$, where $\bar{\boldsymbol{S}} =_{df} \{\bar{s} \mid s \in \boldsymbol{S}\}$.

(b) $\boldsymbol{Func}$ $(\overline{\Sigma}^{\boldsymbol{S}})$consists of $\boldsymbol{Func}(\Sigma)$, together with the *evaluation function*

$$\mathsf{eval}_s : (\mathsf{nat} \to s) \times \mathsf{nat} \to s,$$

for each $s \in \boldsymbol{S}$.

Now we can *expand* A^N to a $\overline{\Sigma}^{\boldsymbol{S}}$*-stream algebra* $\bar{A}^{\boldsymbol{S}}$ by adding for each pre-stream sort s:

(i) the carrier for $\mathsf{nat} \to s$, which is the set

$$A_{\mathsf{nat}\to s} \ = \ \bar{A}_s \ = \ [\mathbb{N} \to A_s]$$

of all *streams on* A, i.e. functions $\xi : \mathbb{N} \to A$;

(ii) the interpretation of eval_s on A as the function $\mathsf{eval}_s^A : [\mathbb{N} \to A_s]\times\mathbb{N} \to A_s$ which *evaluates* a stream at an index, i.e.,

$$\mathsf{eval}_s^A(\xi, n) = \xi(n);$$

(iii) the discriminator on $\bar{A}_s$, for all $s \in \boldsymbol{S}$.

The algebra $\bar{A}^{\boldsymbol{S}}$ is the *(full) stream algebra over* A *with respect to* $\boldsymbol{S}$.

This construction of $\bar{A}^{\boldsymbol{S}}$ from a standard A is depicted by:

algebra	$\bar{A}^{\boldsymbol{S}}$	
import	A^N	
carriers	$[\mathbb{N} \to A_s]$	$(s \in \boldsymbol{S})$
functions	$\mathsf{eval}_s^A : [\mathbb{N} \to A_s] \times \mathbb{N} \to A_s$	$(s \in \boldsymbol{S})$
	$\mathtt{if}_{\bar{s}}^A : \mathbb{B} \times ([\mathbb{N} \to A_s])^2 \to [\mathbb{N} \to A_s]$	$(s \in \boldsymbol{S})$
end		

where now $\boldsymbol{S}$ is the set of stream sorts.

Also, $\bar{\mathbb{K}}^{\boldsymbol{S}}$ is (the closure with respect to $\overline{\Sigma}^{\boldsymbol{S}}/\Sigma$-isomorphism of) the class $\{\bar{A} \mid A \in \mathbb{K}\}$.

Remark 2.32.

(a) The algebra $\bar{A}^{\boldsymbol{S}}$ is a $\overline{\Sigma}^{\boldsymbol{S}}$-*expansion* of A. If Σ has r sorts, then $\overline{\Sigma}^{\boldsymbol{S}}$ has $r+k+1$ sorts, where k is the cardinality of $\boldsymbol{S}$.

(b) $\overline{\Sigma}^{\boldsymbol{S}}$ and $\bar{A}^{\boldsymbol{S}}$ are N-standard.

(c) Because we have taken $\bar{A}_s$ to be the set of *all* streams on A_s, we call $\bar{A}^{\boldsymbol{S}}$ the *full* stream algebra (with respect to $\boldsymbol{S}$). Note that if A_s has cardinality greater than 1 for some $s \in \boldsymbol{S}$, then $\bar{A}_s$, and hence $\bar{A}$, is uncountable.

(d) A Σ-homomorphism (or Σ-isomorphism) between two standard Σ-algebras A and B can be extended to a Σ-homomorphism (or Σ-isomorphism) between $\bar{A}$ and $\bar{B}$. (*Exercise*; cf. Remarks 2.28(c), 2.30(f) and 2.31(e).)

(e) Note that the instantiation assumption does not hold (in general) on stream algebras.

3 *While* computability on standard algebras

In this section, we begin to study the computation of functions and relations on algebras by means of imperative programming models. We start by defining a simple programming language ***While*** $=$ ***While***(Σ), whose programs are constructed from *concurrent assignments*, *sequential composition*, the *conditional* and the 'while' construct, and may be interpreted on any many-sorted Σ-algebra; this takes up sections 3.1–3.6. We will define in detail the abstract syntax and semantics of this language, and methods by which its programs can compute functions and relations. In sections 3.7 and 3.8 we prove some algebraic properties of computation on algebras, with regard to homomomorphisms and locality.

In sections 3.9–3.13, we will add to the basic language a number of new constructs, namely 'for', procedure calls and arrays, and extend our model of computation accordingly. In section 3.14 we study the concept of a sequence of 'snapshots' of a computation, which will be useful later in investigating the solvability of the halting problem in certain (locally finite) algebras.

We conclude (section 3.15) with a useful syntactic conservativity theorem for Σ^*-terms over Σ-terms.

We illustrate the theory with several examples of computations on the algebras of real and complex numbers.

Throughout section 3, we assume (following Convention 1.4.3) that

Σ *is a standard signature, and A is a standard Σ-algebra.*

3.1 Syntax of $\boldsymbol{While}(\Sigma)$

We begin with the syntax of the language $\boldsymbol{While}(\Sigma)$. First, for each Σ-sort s, there are (*program*) *variables* $\mathtt{a}^s, \mathtt{b}^s, \ldots, \mathtt{x}^s, \mathtt{y}^s \ldots$ of sort s.

We define four syntactic classes: *variables*, *terms*, *statements* and *procedures.*

(a) $\boldsymbol{Var} = \boldsymbol{Var}(\Sigma)$ is the class of Σ-variables, and $\boldsymbol{Var}_s$ is the class of variables of sort s.
For $u = s_1 \times \ldots \times s_m$, we write $\mathtt{x} : u$ to mean that $\mathtt{x}$ is a u-tuple of *distinct variables*, i.e., a tuple of distinct variables of sorts $s_1, \ldots, s_m$, respectively.
Further, we write $\boldsymbol{VarTup}$= $\boldsymbol{VarTup}(\Sigma)$for the class of all tuples of distinct Σ-variables, and $\boldsymbol{VarTup}_u$ for the class of all u-tuples of distinct Σ-variables.

(b) $\boldsymbol{Term} = \boldsymbol{Term}(\Sigma)$ is the class of Σ-terms $t, \ldots$, and for each Σ-sort s, $\boldsymbol{Term}_s$ is the class of terms of sort s. These are generated by the following rules.

  (i) A *variable* $\mathtt{x}$ of sort s is in $\boldsymbol{Term}_s$.

  (ii) If $F \in \boldsymbol{Func}(\Sigma)_{u \to s}$ and $t_i \in \boldsymbol{Term}_{s_i}$ for $i = 1, \ldots, m$ where $u = s_1 \times \ldots \times s_m$, then $F(t_1, \ldots, t_m) \in \boldsymbol{Term}_s$.

Note again that Σ-constants are construed as 0-ary functions, and so enter the definition of $\boldsymbol{Term}(\Sigma)$ via clause (ii), with $m = 0$.
The class $\boldsymbol{Term}(\Sigma)$ can also be written (in more customary notation) as $T(\Sigma, \boldsymbol{Var})$, i.e., the set of terms over Σ using the set $\boldsymbol{Var}$ of variables (clause (i) in the definition). Analogously, the set $\boldsymbol{T}(\Sigma)$ of closed terms over Σ (2.10) can be written as $T(\Sigma, \emptyset)$.
We write $\boldsymbol{type}(t) = s$ or $t : s$ to indicate that $t \in \boldsymbol{Term}_s$.
Further, we write $\boldsymbol{TermTup}$=$\boldsymbol{TermTup}(\Sigma)$for the class of all tuples of Σ-terms, and, for $u = s_1 \times \ldots \times s_m$, $\boldsymbol{TermTup}_u$ for the class of u-tuples of terms, i.e.,

$$\boldsymbol{TermTup}_u =_{df} \ Term_{s_1} \times \ldots \times Term_{\ s_m}.$$

We write $\boldsymbol{type}(t) = u$ or $t : u$ to indicate that t is a u-*tuple* of terms, i.e.,, a tuple of terms of sorts $s_1, \ldots, s_m$.
For the sort bool, we have the class of *Boolean terms* or *Booleans* $\boldsymbol{Bool}(\Sigma) =_{df} \boldsymbol{Term}_{\mathsf{bool}}$, denoted either $t^{\mathsf{bool}} \ldots$ (as above) or $b, \ldots$. This class is given (according to the above definition of $\boldsymbol{Term}_s$) by:

$$\begin{array}{rcl} b & ::= & \mathtt{x}^{\mathsf{bool}} | F(t) | \mathsf{eq}_s(t_1^s, t_2^s) | \ \mathsf{true} \ | \ \mathsf{false} \\ & & | \ \mathsf{not}(b) | \ \mathsf{and} \ (b_1, b_2) | \ \mathsf{or}(b_1, b_2) | \ \mathsf{if}(b, b_1, b_2) \end{array}$$

where F is a Σ-function symbol of type $u \to \mathsf{bool}$ (other than one of the standard Boolean operations, which are listed explicitly) and s is an equality sort.

(c) $\boldsymbol{Stmt} = \boldsymbol{Stmt}(\Sigma)$ is the class of statements $S, \ldots$. The *atomic statements* are 'skip' and the *concurrent assignment* $\mathtt{x} := t$ where for some product type u, $\mathtt{x} : u$ and $t : u$.

Statements are then generated by the rules

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

(*d*) $\boldsymbol{Proc} = \boldsymbol{Proc}(\Sigma)$ is the class of procedures $P, Q, \ldots$. These have the form

$$P \equiv \mathsf{proc}\ D\ \mathsf{begin}\ S\ \mathsf{end}$$

where D is the *variable declaration* and S is the *body*. Here D has the form

$$D \equiv \mathsf{in\ a\ out\ b\ aux\ c}$$

where a, b and c are lists of *input variables*, *output variables* and *auxiliary* (or *local) variables*, respectively. Further, we stipulate:

- a, b and c each consist of distinct variables, and they are pairwise disjoint;
- every variable occurring in the body S must be declared in D (among a, b or c);
- the *input variables* a must not occur on the lhs of assignments in S;
- (*initialisation condition*:) S has the form $S_{init}; S'$, where S_{init} is a *concurrent assignment* which *initialises* all the *output* and *auxiliary variables*, i.e., assigns to each of them the default term (section 2.12) of the same sort.

Each variable occurring in the declaration of a procedure *binds* all free occurrences of that variable in the body.
If $\mathsf{a} : u$ and $\mathsf{b} : v$, then P is said to have *type* $u \to v$, written $P : u \to v$. Its *input type* is u.
We write $\boldsymbol{Proc}_{u\to v} = \boldsymbol{Proc}(\Sigma)_{u\to v}$ for the class of Σ-procedures of type $u \to v$.

Notation 3.1.

(*a*) We will often drop the sort superscript or subscript s.
(*b*) We will use $E, E', E_1, \ldots$ to denote syntactic expressions of any of the three classes $\boldsymbol{Term}$, $\boldsymbol{Stmt}$ and $\boldsymbol{Proc}$.
(*c*) For any such expression E, we define $\boldsymbol{var}(E)$ to be the set of variables occurring in E.
(*d*) We use '$\equiv$' to denote syntactic identity between two expressions.

Remark 3.2 (Structural induction; induction on complexity). We will often prove assertions about, or define constructs on, expressions E of a particular syntactic class (such as $\boldsymbol{Term}$, $\boldsymbol{Stmt}$ or $\boldsymbol{Proc}$) by *structural induction* (*or recursion*) on E, following the inductive definition of that class.

Alternatively, we may give such proofs or definitions by course-of-values induction (or recursion) on $\boldsymbol{compl}(E)$, the *structural complexity* of E. One suitable definition of this is the length of the maximum branch of the parse tree of E. Thus, for example, for a program term $t \equiv F(t_1, \dots, t_m)$, $\boldsymbol{compl}(t) = \max_i(\boldsymbol{compl}(t_i) + 1)$. Another possible definition of $\boldsymbol{compl}(E)$, which would in fact be satisfactory for our purposes, is simply the length of E as a string of symbols.

Sections 3.2–3.6 will be devoted to the semantics of $\boldsymbol{While}$.

3.2 States

For each standard Σ-algebra A, a *state* on A is a family $\langle \sigma_s | \ s \in \boldsymbol{Sort}(\Sigma) \rangle$ of functions

$$\sigma_s : \boldsymbol{Var}_s \to A_s. \tag{3.1}$$

Let $\boldsymbol{State}(\mathrm{A})$ be the set of states on A, with elements $\sigma, \dots$. Note that $\boldsymbol{State}(\mathrm{A})$ is the product of the state spaces $\boldsymbol{State}_s(A)$ for all $s \in \boldsymbol{Sort}(\Sigma)$, where each $\boldsymbol{State}_s(A)$ is the set of all functions as in (3.1).

We use the following notation. For $\mathtt{x} \in \boldsymbol{Var}_s$, we often write $\sigma(\mathtt{x})$ for $\sigma_s(\mathtt{x})$. Also, for a tuple $\mathtt{x} \equiv (\mathtt{x}_1, \dots, \mathtt{x}_m)$, we write $\sigma[\mathtt{x}]$ for $(\sigma(\mathtt{x}_1), \dots, \sigma(\mathtt{x}_m))$.

Now we define the *variant of a state.* Let σ be a state over A, $\mathtt{x} \equiv (\mathtt{x}_1, \dots, \mathtt{x}_n) : u$ and $a = (a_1, \dots, a_n) \in A^u$ (for $n \geq 1$). We define $\sigma\{\mathtt{x}/a\}$ to be the state over A formed from σ by replacing its value at $\mathtt{x}_i$ by a_i for $i = 1, \dots, n$. That is, for all variables $\mathtt{y}$:

$$\sigma\{\mathtt{x}/a\}(\mathtt{y}) \;=\; \begin{cases} \sigma(\mathtt{y}) & \text{if } \ \mathtt{y} \not\equiv \mathtt{x}_i \text{ for } i = 1, \dots, n \\ a_i & \text{if } \ \mathtt{y} \equiv \mathtt{x}_i. \end{cases}$$

We can now give the semantics of each of the three syntactic classes: $\boldsymbol{Term}$, $\boldsymbol{Stmt}$ and $\boldsymbol{Proc}$, relative to any $A \in \boldsymbol{StdAlg}(\Sigma)$. For an expression E in each of these classes, we will define a *semantic function* $[\![E]\!]^A$. These three semantic functions are defined in sections 3.3, 3.4–3.5 and 3.6, respectively.

3.3 Semantics of terms

For $t \in \boldsymbol{Term}_s$, we define the function

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

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

The definition is by structural induction on t:

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

Note that this definition of $[\![t]\!]^A\sigma$ extends that of t_A for $t \in \boldsymbol{T}(\Sigma)$ (Definition 2.11). Also the second clause incorporates the cases that (a) F is a constant; (b) F is a standard Boolean operation, e.g. the discriminator:

$$[\![\mathsf{if}(b,t_1,t_2)]\!]^A\sigma \;=\; \begin{cases} [\![t_1]\!]\sigma & \text{if } [\![b^A\sigma = \mathtt{tt} \\ [\![t_2]\!]\sigma & \text{if } [\![b^A\sigma = \mathtt{ff}. \end{cases}$$

For a *tuple* of terms $t = (t_1, \dots, t_m)$, we use the notation

$$[\![t]\!]^A\sigma \;=_{df}\; ([\![t_1]\!]^A\sigma, \dots, [\![t_m]\!]^A\sigma).$$

Definition 3.3. For any $M \subseteq \boldsymbol{Var}_s$, and states σ_1 and σ_2, $\sigma_1 \approx \sigma_2(\text{rel } M)$ means $\sigma_1 \restriction M = \sigma_2 \restriction M$, i.e., $\forall x \in M\big(\sigma_1(x) = \sigma_2(x)\big)$.

Lemma 3.4 (Functionality lemma for terms). *For any term t and states σ_1 and σ_2, if $\sigma_1 \approx \sigma_2$ (rel $\boldsymbol{var}(t)$), then $[\![t]\!]^A\sigma_1 \;=\; [\![t]\!]^A\sigma_2$.*

Proof. By structural induction on t. ∎

3.4 Algebraic operational semantics

In this subsection we will describe a general method for defining the meaning of a statement S, in a wide class of imperative programming languages, as a partial *state transformation*, i.e., a partial function

$$[\![S]\!]^A : \boldsymbol{State}(\mathrm{A}) \to \boldsymbol{State}(\mathrm{A}).$$

We define this via a *computation step* function

$$\boldsymbol{Comp}^A\!: \boldsymbol{Stmt} \times \boldsymbol{State}(\mathrm{A}) \times \mathbb{N} \to \boldsymbol{State}(\mathrm{A}) \cup \{*\}$$

where '$*$' is a new symbol or object. The idea is that

> *$\boldsymbol{Comp}^A(S,\sigma,n)$ is the nth step, or the state at the nth time cycle, in the computation of S on A, starting in state σ.*

The symbol '$*$' indicates that the computation is over. Thus

if for any n, $\boldsymbol{Comp}^A(S,\sigma,n) = *$, then for all $m \ge n$
$\boldsymbol{Comp}^A(S,\sigma,m) = *$.

If we put $\sigma_n = \boldsymbol{Comp}^A(S,\sigma,n)$, then the sequence of states

$$\sigma = \sigma_0,\ \sigma_1,\ \sigma_2,\ \dots,\ \sigma_n,\ \dots$$

is called the *computation sequence generated by S* at σ, written $\boldsymbol{CompSeq}^A$ (S,σ). It is either infinite, or terminates in a final state σ_l, where $\boldsymbol{Comp}^A$ $(S,\, \sigma,\, l+1) = *$.

We will use an algebraic method in which $\boldsymbol{Comp}^A$ is defined equationally. In section 3.5 we will apply this general method to the present programming language $\boldsymbol{While}(\Sigma)$. In later sections we will apply it to other languages.

Assume, *firstly*, that (for the language under consideration) there is a class $\boldsymbol{AtSt} \subset \boldsymbol{Stmt}$ of *atomic statements* for which we have a meaning function

$$\langle\!| S |\!\rangle^A : \boldsymbol{State}(\mathrm{A}) \to \boldsymbol{State}(\mathrm{A}),$$

for $S \in \boldsymbol{AtSt}$, and *secondly*, that we have two functions

$$\begin{aligned} \boldsymbol{First} &: \boldsymbol{Stmt} \to \boldsymbol{AtSt} \\ \boldsymbol{Rest}^A &: \boldsymbol{Stmt} \times \boldsymbol{State}(\mathrm{A}) \to \boldsymbol{Stmt}, \end{aligned}$$

where, for a statement S and state σ,

> $\boldsymbol{First}(S)$ *is an atomic statement which gives the first step in the execution of* S *(in any state), and* $\boldsymbol{Rest}^A(S,\sigma)$ *is a statement which gives the rest of the execution in state* σ*.*

For the languages under consideration here, $\boldsymbol{First}(S)$, unlike $\boldsymbol{Rest}^A(S,\sigma)$, will be independent of A and σ.

In each language we can define these three functions ($\langle\!| \cdot |\!\rangle$, $\boldsymbol{First}$ and Rest^A).

First we define the 'one-step computation of S at σ'

$$\boldsymbol{Comp}_1^A : \boldsymbol{Stmt} \times \boldsymbol{State}(\mathrm{A}) \to \boldsymbol{State}(\mathrm{A})$$

by

$$\boldsymbol{Comp}_1^A(S,\sigma) = \langle\!| \boldsymbol{First}(S) |\!\rangle^A \sigma.$$

The definition of $\boldsymbol{Comp}^A(S,\sigma,n)$ now follows by a simple recursion ('tail recursion') on n:

$$\boldsymbol{Comp}^A(S,\sigma,n+1) = \begin{cases} * & \text{if } n > 0 \text{ and } S \text{ is atomic} \\ \boldsymbol{Comp}^A(\boldsymbol{Rest}^A(S,\sigma), \boldsymbol{Comp}_1^A(S,\sigma), n) & \\ & \text{otherwise.} \end{cases} \tag{3.2}$$

Note that for $n = 1$, this yields

$$\boldsymbol{Comp}^A(S,\sigma,1) = \boldsymbol{Comp}_1^A(S,\sigma).$$

We call this approach *algebraic operational semantics*, first used in Tucker and Zucker [1988], and developed and applied in Stephenson [1996].

From this semantics we can easily derive the *i/o semantics* as follows. First we define the *length of a computation* of a statement S, starting in state σ, as the function

$$\boldsymbol{CompLength}^A : \boldsymbol{Stmt} \times \boldsymbol{State}(\mathrm{A}) \to \mathbb{N} \cup \{\infty\}$$

where

$$\boldsymbol{CompLength}^A(S,\sigma) \;=\; \begin{cases} \text{least } n \text{s.t.} & \boldsymbol{Comp}^A(S,\sigma,n+1) \;=\; * \\ & \text{if such an } n \text{ exists} \\ \infty & \text{otherwise.} \end{cases}$$

Then, putting $l = \boldsymbol{CompLength}^A(S,\sigma)$ and noting that $0 < l \le \infty$, we define

$$[\![S]\!]^A(\sigma) \;\simeq\; \begin{cases} \boldsymbol{Comp}^A(S,\sigma,l) & \text{if } l \ne \infty \\ \uparrow & \text{otherwise.} \end{cases}$$

Remark 3.5 (Tail recursion). Consider the recursive definition (3.2) of $\boldsymbol{Comp}^A$. In the 'recursive call' (the second expression on the right-hand side of the second equation), notice that (1°) $\boldsymbol{Comp}^A$ is on the 'outside', and (2°) the parameter *changes* (from σ to $\boldsymbol{Comp}_1^A(S,\sigma,n)$). Such a definitional scheme is said to be *tail recursive.* Because of (2°), these equations (as they stand) do *not* form a definition by primitive recursion. However, at least in the classical case, where all the arguments and values range over $\mathbb{N}$, it can be shown that such a scheme can be reduced to a primitive recursive definition. (See, for example, Goodstein [1964, §6.1], where this is called 'recursion with parameter substitution', or Péter [1967, §7], where a more general scheme, not satisfying (1°) only, is considered.)

An alternative, primitive recursive, definition of $\boldsymbol{Comp}^A$ is given below in section 3.14.

3.5 Semantics of statements for $\boldsymbol{While}(\Sigma)$

We now apply the above theory to the language $\boldsymbol{While}(\Sigma)$. Here there are two atomic statements: skip and concurrent assignment. We define $\langle\!|\ S\ |\!\rangle^A$ for these:

$$\begin{aligned} \langle\!|\ \mathsf{skip} |\!\rangle^A\sigma &= \sigma \\ \langle\!|\ \mathtt{x}{:=}\, t\ |\!\rangle^A\sigma &= \sigma\{\mathtt{x}/[\![t]\!]^A\sigma\}. \end{aligned}$$

Next we define $\boldsymbol{First}$ and $\boldsymbol{Rest}^A$. The definitions of $\boldsymbol{First}(S)$ and $\boldsymbol{Rest}^A(S,\sigma)$ proceed by structural induction on S.

Case 1. S is atomic.

$$\begin{aligned} \boldsymbol{First}\ (\mathrm{S}) &= \mathrm{S} \\ \boldsymbol{Rest}^A(S,\sigma) &= \mathsf{skip}. \end{aligned}$$

Case 2. $S \equiv S_1; S_2$ (the interesting case!).

$\boldsymbol{First}(S) \;=\; \boldsymbol{First}(S_1)$

$$\boldsymbol{Rest}^A(S,\sigma) \;=\; \begin{cases} S_2 & \text{if } S_1 \text{ is atomic} \\ \boldsymbol{Rest}^A(S_1,\sigma); S_2 & \text{otherwise.} \end{cases}$$

Case 3. $S \equiv \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}.$

$$\begin{aligned} \boldsymbol{First}\ (S) \quad &= \quad \mathsf{skip} \\ \boldsymbol{Rest}^A(S,\sigma) \quad &= \quad \begin{cases} S_1 & \text{if } [\![b]\!]^A\sigma = \mathtt{tt} \\ S_2 & \text{if } [\![b]\!]^A\sigma = \mathtt{ff}. \end{cases} \end{aligned}$$

Case 4. $S \equiv \mathsf{while}\ b\ \mathsf{do}\ S_0\ \mathsf{od}$.

$$\begin{aligned} \boldsymbol{First}\ (S) \quad &= \quad \mathsf{skip} \\ \boldsymbol{Rest}^A(S,\sigma) \quad &= \quad \begin{cases} S_0; S & \text{if } [\![b]\!]^A\sigma = \mathtt{tt} \\ \mathsf{skip} & \text{if } [\![b]\!]^A\sigma = \mathtt{ff}. \end{cases} \end{aligned}$$

This completes the definition of $\boldsymbol{First}$ and $\boldsymbol{Rest}^A$. Note (in cases 3 and 4) that the Boolean test in an 'if' or 'while' statement S is assumed to take up one time cycle; this is modelled by taking $\boldsymbol{First}(S) \equiv \mathsf{skip}$.

The following shows that the i/o semantics, derived from our algebraic operational semantics, satisfies the usual desirable properties.

Theorem 3.6.

(a) For S atomic, $[\![S]\!]^A = \langle\!| S |\!\rangle^A$, i.e.,

$$\begin{aligned} \langle\!| \mathsf{skip} |\!\rangle^A\sigma \quad &= \quad \sigma \\ \langle\!| \mathsf{x} := t |\!\rangle^A\sigma \quad &= \quad \sigma\{\mathsf{x}/\langle\!| t |\!\rangle^A\sigma\}. \end{aligned}$$

(b)

$$[\![S_1; S_2]\!]^A\sigma \;\simeq\; [\![S_2]\!]^A([\![S_1]\!]^A\sigma).$$

(c)

$$[\![\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}]\!]^A\sigma \;\simeq\; \begin{cases} [\![S_1]\!]^A\sigma & \text{if } [\![b]\!]^A\sigma = \mathtt{tt} \\ [\![S_2]\!]^A\sigma & \text{if } [\![b]\!]^A\sigma = \mathtt{ff}. \end{cases}$$

(d)

$$[\![\mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}]\!]^A\sigma \;\simeq\; \begin{cases} [\![S; \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}]\!]^A\sigma & \text{if } [\![b]\!]^A\sigma = \mathtt{tt} \\ \sigma & \text{if } [\![b]\!]^A\sigma = \mathtt{ff}. \end{cases}$$

Proof. Exercise. *Hint*: For part (b), prove the following lemma. Formulate and prove analogous lemmas for parts (a), (c) and (d). ∎

Lemma 3.7. $\boldsymbol{Comp}^A(S_1; S_2, \sigma, n) \;=$

$$\begin{cases} \boldsymbol{Comp}^A(S_1,\sigma,n) & \text{if } \forall k < n \boldsymbol{Comp}^A(S_1,\sigma,k+1) \neq * \\ \boldsymbol{Comp}^A(S_2,\sigma',n-n_0) & \text{if } \exists k < n \boldsymbol{Comp}^A(S_1,\sigma,k+1) = * \\ & \text{where } n_0 \text{ is the least such } k, \text{ and} \\ & \sigma' = \boldsymbol{Comp}^A(S_1,\sigma,n_0). \end{cases}$$

Remark 3.8.

(a) The four suitably formulated lemmas needed to prove parts (a)–(d) of Theorem 3.6 (of which Lemma 3.7 is an example for part (b)) provide an alternative definition of $\boldsymbol{Comp}^A(S,\sigma,n)$, which does not make use of $\boldsymbol{First}$ or $\boldsymbol{Rest}^A$. This definition is by structural induction on S, with a secondary induction on n.

(b) The meaning function $[\![S]\!]^A$ (i.e., our i/o semantics) was derived from our *operational semantics*, i.e., the $\boldsymbol{Comp}^A$ function. We could also give a *denotational i/o semantics* for $\boldsymbol{While}$ statements. Theorem 3.6 would then provide (one direction of) a proof of the equivalence of the two semantics (as in de Bakker [1980]).

(c) The semantics given here is simpler than that given in Tucker and Zucker [1988] where the states have an 'error value' almost everywhere (for uninitialised variables), and there is an 'error state' corresponding to an aborted computation. While such an 'error semantics' is superior (we feel) to the one given here, the semantics given here is simpler, and adequate for our purposes.

For the semantics of procedures, we need the following. Let $M \subseteq \boldsymbol{Var}_s$, and $\sigma, \sigma' \in \boldsymbol{State}(\mathrm{A})$.

Lemma 3.9. *Suppose $\boldsymbol{var}(S) \subseteq M$. If $\sigma_1 \approx \sigma_2$ (rel M), then for all $n \geq 0$,*

$$\boldsymbol{Comp}^A(S,\sigma_1,n) \;\approx\; \boldsymbol{Comp}^A(S,\sigma_2,n) \;\; (\mathrm{rel}\ M).$$

Proof. By induction on n. Use the functionality lemma (3.4) for terms. ∎

Lemma 3.10 (Functionality lemma for statements). *Suppose $\boldsymbol{var}(S) \subseteq M$. If $\sigma_1 \approx \sigma_2$ (rel M), then either*

(i) $[\![S]\!]^A\sigma_1 \downarrow \sigma_1'$ and $[\![S]\!]^A\sigma_2 \downarrow \sigma_2'$ (say), where $\sigma_1' \approx \sigma_2'$ (rel M), or
(ii) $[\![S]\!]^A\sigma_1 \uparrow$ and $[\![S]\!]^A\sigma_2 \uparrow$.

Proof. From Lemma 3.9. ∎

3.6 Semantics of procedures

Now if

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

is a procedure of type $u \to v$, then its meaning is a function

$$[\![P]\!]^A : \; A^u \to A^v$$

defined as follows. For $a \in A^u$, let σ be any state on A such that $\sigma[\mathsf{a}] = a$. Then

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

For $[\![P]\!]^A$ to be well defined, we need the fact that the procedure P is *functional*, as follows.

Lemma 3.11 (Functionality lemma for procedures). *Suppose*

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

If $\sigma_1 \approx \sigma_2$ (rel a), *then either*

(i) $[\![S]\!]^A\sigma_1 \downarrow \sigma_1'$ *and* $[\![S]\!]^A\sigma_2 \downarrow \sigma_2'$ *(say), where* $\sigma_1' \approx \sigma_2'$ (rel b) *or*

(ii) $[\![S]\!]^A\sigma_1 \uparrow$ *and* $[\![S]\!]^A\sigma_2 \uparrow$.

Proof. Suppose $\sigma_1 \approx \sigma_2$ (rel a). We can put $S{\equiv}S_{init};S'$, where S_{init} consists of an initialisation of b and c to closed terms (see section 3.1). Then, putting

$$[\![S_{init}]\!]^A\sigma_1 = \sigma_1'' \qquad \text{and} \qquad [\![S_{init}]\!]^A\sigma_2 = \sigma_2'',$$

it is easy to see that

$$\sigma_1'' \approx \sigma_2'' \ \ (\text{rel}\ \mathsf{a},\mathsf{b},\mathsf{c}).$$

The result then follows from the functionality lemma 3.10 for statements (with S', σ_1'' and σ_2'' in place of S, σ_1 and σ_2, respectively). ∎

Remark 3.12.

(a) Functionality of procedures amounts to saying that there are *no side effects* from the output variables or auxiliary variables.

(b) The initialisation condition (section 3.1) is a *sufficient* (*but not necessary*) *syntactic condition for functionality of procedures.* A more general syntactic condition ensuring functionality was given in Jervis [1988]. A semantic approach to functionality was taken in Tucker and Zucker [1988, §4.3.2].

We can now define:

Definition 3.13 (*While* computable functions).

(a) A function f on A is *computable on A by a **While** procedure P* if $f = [\![P]\!]^A$. It is ***While** computable on A* if it is computable on A by some ***While*** procedure.

(b) A family $f = \langle f_A \mid A \in \mathbb{K}\rangle$ of functions is ***While** computable uniformly over* $\mathbb{K}$ if there is a ***While*** procedure P such that for all $A \in \mathbb{K}$, $f_A = [\![P]\!]^A$.

(c) ***While***(A) is the class of functions ***While*** computable on A.

We will often write P^A for $[\![P]\!]^A$.

Example 3.14.

(a) Recall the standard algebra $\mathcal{N}$ of naturals (Example 2.23(b)). The functions ***While*** computable on $\mathcal{N}$ of type $\mathsf{nat}^k \to \mathsf{nat}$ are precisely the partial recursive functions over $\mathbb{N}$ (Kleene [1952]). This follows from the equivalence of partial recursiveness and ***While*** computability on the naturals (see, for example, McNaughton [1982]), or from the results in section 8. Hence

> *every partial recursive function over $\mathbb{N}$ is* ***While*** *computable on every N-standard algebra.*

(*b*) In the N-standardised group $\mathcal{G}^N$ (Example 2.27(*b*)), the partial function $\boldsymbol{ord}\colon G \to \mathbb{N}$, defined by

$$\boldsymbol{ord}(g) \;\simeq\; \begin{cases} \text{least } n \text{ s.t. } g^n = 1, & \text{if such an } n \text{ exists} \\ \uparrow & \text{otherwise,} \end{cases}$$

which gives the order of group elements, is ***While*** computable, since the 'constructive least number operator' is (see section 8). Alternatively, we can give directly a ***While*** procedure in the signature $\Sigma(\mathcal{G}^N)$:

```
proc in   g:grp
     out  n:nat
     aux  prod:grp      {temporary product}
begin
     prod:=g;
     n:=1;
     while not(prod=1)
          do  prod:=prod* g;
              n:=succ(n)
          od
end
```

We emphasise that this order function is *defined uniformly* over all N-standardised groups (of the given signature $\Sigma(\mathcal{G}^N)$).

The following proposition will be useful.

Proposition 3.15 (Closure of *While* computability under composition). *The class of* ***While*** *computable functions on A is closed under composition. In other words, given (partial) functions $f : A^u \to A^v$ and $g : A^v \to A^w$ (for any Σ-product types u, v, w), if f and g are* ***While*** *computable on A, then so is the composed function $g \circ f : A^u \to A^w$.*

Proof. Exercise. (Construct the appropriate ***While*** procedure for the composed function.) ∎

Remark 3.16. Similarly, we have closure under composition for the related notions of computability still to be considered in this section, namely $\boldsymbol{While}^N$, $\boldsymbol{While}^*$, ***For***, $\boldsymbol{For}^N$ and $\boldsymbol{For}^*$ computability, and the relativised versions of these. The results for ***For*** computability (etc.) can be derived from its equivalence with PR computability (etc.) (cf. section 8).

3.7 Homomorphism invariance theorems

We will investigate how our semantics of $\boldsymbol{While}$ programs interacts with homomorphisms between standard Σ-algebras.

Let A and B be two standard Σ-algebras. Let $\phi = \{\phi_s \mid s \in \boldsymbol{Sort}(\Sigma)\}$ be a Σ-homomorphism from A to B.

For $a \in A_s$, we will write $\phi(a)$ for $\phi_s(a)$; and for a tuple $a = (a_1, \dots, a_m) \in A^u$, we will write $\phi(a)$ for $(\phi(a_1), \dots, \phi(a_m))$.

Lemma 3.17.

(a) ϕ_{bool} *is the identity on* $\mathbb{B}$.
(b) ϕ_s *is injective on all equality sorts* s.

Proof. Exercise. ∎

Definition 3.18. The mapping ϕ induces a mapping

$$\hat{\phi} : \boldsymbol{State}(A) \cup \{*\} \to \boldsymbol{State}(B) \cup \{*\}$$

by

$$\hat{\phi}(\sigma) = \phi \circ \sigma,$$

i.e., if $\sigma = \langle \sigma_s \mid s \in \boldsymbol{Sort}(\Sigma)\rangle$, then $\hat{\phi}(\sigma) = \sigma' = \langle \sigma'_s \mid s \in \boldsymbol{Sort}(\Sigma)\rangle$, where for all $s \in \boldsymbol{Sort}(\Sigma)$ and $\mathtt{x} \in \boldsymbol{Var}_s$, $\sigma'_s(\mathtt{x}) = \phi_s(\sigma_s(\mathtt{x}))$. Further, we stipulate

$$\hat{\phi}(*) = *.$$

Now we state some *homomorphism invariance theorems.*

Theorem 3.19 (Homomorphism invariance for terms). *For* $t \in \boldsymbol{Term}_s$

$$\phi([\![t]\!]^A \sigma) = [\![t]\!]^B \hat{\phi}(\sigma).$$

Proof. By structural induction on t. ∎

Theorem 3.20 (Homomorphism invariance for atomic statements). *For* $S \in \boldsymbol{AtSt}$,

$$\hat{\phi}(\langle\!|\, S \,|\!\rangle^A \sigma) = \langle\!|\, S \,|\!\rangle^B \hat{\phi}(\sigma).$$

Proof. The case where $S \equiv \mathsf{skip}$ is trivial. The case that S is an assignment follows from Theorem 3.19. ∎

Corollary 3.21 (Homomorphism invariance for the $\boldsymbol{Comp}_1$ predicate).

$$\hat{\phi}(\boldsymbol{Comp}_1^A(S, \sigma)) = \boldsymbol{Comp}_1^B(S, \hat{\phi}(\sigma)).$$

Theorem 3.22 (Homomorphism invariance for the $\boldsymbol{Comp}$ predicate).

$$\hat{\phi}(\boldsymbol{Comp}^A(S, \sigma, n)) = \boldsymbol{Comp}^B(S, \hat{\phi}(\sigma), n).$$

Proof. By induction on n. For the base case $n = 1$, use Corollary 3.21. ∎

Theorem 3.23 (Homomorphism invariance for statements). *Either*

(i) $[\![S]\!]^A\sigma \downarrow \sigma'$ *and* $[\![S]\!]^B\hat{\phi}(\sigma) \downarrow \sigma''$ *(say), where* $\hat{\phi}(\sigma') = \sigma''$*, or*

(ii) $[\![S]\!]^A\sigma \uparrow$ *and* $[\![S]\!]^B\hat{\phi}(\sigma) \uparrow$.

Proof. From Theorem 3.22. ∎

Theorem 3.24 (Homomorphism invariance for procedures). *For a procedure* $P : u \to v$ *and* $a \in A^u$,

$$\phi(P^A(a)) \;\simeq\; P^B(\phi(a)).$$

Proof. From Theorem 3.23. ∎

3.8 Locality of computation

We will investigate how the semantics of ***While*** programs relates to the subalgebra generated by the input. (Recall Definition 2.15.)

We want to prove the *locality theorem*: for any ***While*** computable function f on A of type $u \to v$, and any $a \in A^u$,

$$\mathsf{if}\;\; f(a)\downarrow \;\;\text{then}\;\; f(a) \subseteq \langle a\rangle^A.$$

This will follow immediately from Theorem 3.30 below.

Lemma 3.25. *For a term* $t : s$ *with* $\boldsymbol{var}(t) \subseteq \mathtt{x}$,

$$[\![t]\!]^A\sigma \;\in\; \langle\sigma[\mathtt{x}]\rangle^A_s.$$

(Recall the definition of $\sigma[\mathtt{x}]$ in section 3.2.)

Proof. By structural induction on t. ∎

Lemma 3.26. *For an atomic statement* S *with* $\boldsymbol{var}(S) \subseteq \mathtt{x} : u$,

$$\langle\!|\, S \,|\!\rangle^A(\sigma)[\mathtt{x}] \;\subseteq\; \langle\sigma[\mathtt{x}]\rangle^A_u.$$

Proof. There are two cases to consider. If S is an assignment, the result follows from Lemma 3.25. If $S \equiv \mathsf{skip}$, then it is trivial. ∎

Lemma 3.27. *If* $\boldsymbol{var}(S) \subseteq \mathtt{x} : u$, *then*

$$\boldsymbol{Comp}^A_1(S,\sigma)[\mathtt{x}] \;\subseteq\; \langle\sigma[\mathtt{x}]\rangle^A.$$

Proof. From Lemma 3.26. ∎

Lemma 3.28. *If* $\boldsymbol{var}(S) \subseteq \mathtt{x}$ *and* $\boldsymbol{Comp}^A(S,\sigma,n) \neq *$, *then*

$$\boldsymbol{Comp}^A(S,\sigma,n)[\mathtt{x}] \;\subseteq\; \langle\sigma[\mathtt{x}]\rangle^A.$$

Proof. By induction on n (with S and σ varying). For the base cases ($n = 1$), use Lemma 3.27. For the induction step, use the facts that $\boldsymbol{var}(\boldsymbol{Rest}^A(S,\sigma)) \subseteq \boldsymbol{var}(S)$ and that

$$X \subseteq \langle Y\rangle^A \;\Rightarrow\; \langle X\rangle^A \subseteq \langle Y\rangle^A.$$

The details are left as an exercise. ∎

Theorem 3.29 (Locality for statements). *If $\boldsymbol{var}(S) \subseteq \mathtt{x} : u$ and $[\![S]\!]^A(\sigma)\downarrow$ then*

$$[\![S]\!]^A(\sigma)[\mathtt{x}] \;\in\; \langle\sigma[\mathtt{x}]\rangle^A_u.$$

Proof. From Lemma 3.28. ∎

Theorem 3.30 (Locality for procedures). *For a procedure $P : u \to v$ and $a \in A^u$ such that $P^A(a)\downarrow$,*

$$P^A(a) \;\in\; \langle a\rangle^A_u.$$

Proof. Suppose

$$P \equiv \mathsf{proc\ in\ a\ out\ b\ aux\ c\ begin}\ S_{init}; S'\ \mathsf{end}$$

where S_{init} consists of an *initialisation* of b and c to closed terms (see section 3.1). Put $\mathtt{x} \equiv \mathsf{a,b,c}$, and suppose

$$\sigma[\mathsf{a}] = a, \quad [\![S_{init}]\!]^A\sigma = \sigma'' \quad \mathsf{and} \quad [\![S']\!]^A\sigma''\downarrow\sigma'.$$

Then

$$\langle a\rangle^A \;=\; \langle\sigma[\mathsf{a}]\rangle^A \;=\; \langle\sigma''[\mathtt{x}]\rangle^A, \tag{3.3}$$

since S_{init} consists (only) of the initialisation of b and c to the closed terms, the values of which lie in every Σ-subalgebra of A. Also, by the syntax of procedures (section 3.1(c)), $\boldsymbol{var}(S_{init}; S') \subseteq \mathtt{x}$. Hence by Theorem 3.29, applied to S' and σ'',

$$P^A(a) \;=_{df}\; \sigma'[\mathsf{b}] \;\subseteq\; \sigma'[\mathtt{x}] \;\subseteq\; \langle\sigma''[\mathtt{x}]\rangle^A. \tag{3.4}$$

The result follows from (3.3) and (3.4). ∎

Certain useful additions to, or modifications of, the ***While*** language defined in section 3.1, with corresponding notions of computability, will be defined in sections 3.9–3.13.

3.9 The language $\boldsymbol{WhileProc}(\Sigma)$

In the language $\boldsymbol{While}(\Sigma)$, we use procedures not in the construction of statements, but only as a convenient device for defining functions (section 3.6). We can, however, define a language $\boldsymbol{WhileProc}(\Sigma)$ which extends $\boldsymbol{While}(\Sigma)$ by the adjunction of a new kind of atomic statement, the *procedure call*

$$\mathtt{x} := P(t), \tag{3.5}$$

where P is a procedure of type $u \to v$ (say), t is a tuple of terms of type u (the *actual parameters*) and $\mathtt{x} : v$.

The semantics of $\boldsymbol{While}$ is then extended by adding the following clause to the semantics of atomic statements (section 3.4):

$$\langle\!|\ \mathtt{x} := P(t)\ |\!\rangle^A \sigma \ = \ \begin{cases} \sigma\{\mathtt{x}/a\} & \text{if } P^A([\![t]\!]^A\sigma) \downarrow a \text{ (say)} \\ \uparrow & \text{if } P^A([\![t]\!]^A\sigma) \uparrow . \end{cases}$$

Note that the function

$$\langle\!|\ \cdot\ |\!\rangle^A : \boldsymbol{AtSt} \to (\boldsymbol{State}(A) \to \boldsymbol{State}(A))$$

is now partial (compare section 3.4).

However, it is easy to 'eliminate' all such procedure calls from a program statement, i.e., to effectively transform $\boldsymbol{WhileProc}$ statements to $\boldsymbol{While}$ statements with the same semantics, as follows. For any procedure call (3.5), suppose

$$P \equiv \mathsf{proc\ in\ a\ out\ b\ aux\ c\ begin}\ S\ \mathsf{end}. \tag{3.6}$$

Then *replace* (3.5) by the statement

$$S\langle \mathsf{a,b,c}/t, \mathtt{x}, \mathsf{z}\rangle, \tag{3.7}$$

where z is a tuple of distinct 'fresh' variables of the same type as c, and $\langle \dots \rangle$ denotes the *simultaneous substitution* of t,$\mathtt{x}$,z for $\mathsf{a,b,c}$.

Note that the result of this substitution (3.7) is a syntactically correct statement, by the stipulation (section 3.1) that the input variables a not occur on the left-hand side of assignments in S.

Remark 3.31.

(*a*) According to our syntax, in the procedure call (3.5) above, 'P' is not just a name for a procedure but the procedure itself, i.e., the complete text (3.6)! In practice, it is of course much more convenient — and customary — to 'declare' the procedure before its call, introducing an *identifier* for it, and then calling the procedure by means of this identifier.

In any case, our syntax prevents *recursive procedure calls.* The situation with recursive procedures would be quite different from that described above — they cannot be eliminated so simply (de Bakker [1980]).

(*b*) Another way of incorporating procedure calls into statements is by expanding the definition of *terms*, as was done in Tucker and Zucker [1994]. The problem with that approach here is that it would complicate the semantics by leading to *partially defined* terms. In Tucker and Zucker [1994] this problem does not occur, since the procedures, being in the ***For*** language rather than ***While***, produce total functions.

3.10 Relative *While* computability

Let $g = \langle g_A \mid A \in \mathbb{K} \rangle$ be a family of (partial) functions

$$g_A : A^u \to A^v.$$

We define the programming language ***While***(g) which extends the language ***While*** by including a special function symbol g of type $u \to v$. We can think of g as an 'oracle' for g_A.

The atomic statements of ***While***(g) include the *oracle call*

$$\mathtt{x} := \mathtt{g}(t)$$

where $t : u$ and $\mathtt{x} : v$. The semantics of this is given by

$$\langle\!\langle\, \mathtt{x} := \mathtt{g}(t) \,\rangle\!\rangle^A \sigma \;\simeq\; \begin{cases} \sigma\{\mathtt{x}/a\} & \text{if } g_A([\![t]\!]^A\sigma) \downarrow a \text{ (say)} \\ \uparrow & \text{if } g_A([\![t]\!]^A\sigma) \uparrow . \end{cases}$$

Similarly, for a tuple of (families of) functions $g_1, \ldots, g_n$, we can define the programming language ***While***$(\mathtt{g}_1, \ldots, \mathtt{g}_n)$ with oracles $\mathtt{g}_1, \ldots, \mathtt{g}_n$ for $g_1, \ldots, g_n$, or (by abuse of notation) the programming language ***While***$(g_1, \ldots, g_n)$.

In this way we can define the notion of ***While***$(g_1, \ldots, g_n)$ *computability*, or ***While*** *computability relative to* $g_1, \ldots, g_n$, or ***While*** *computability in* $g_1, \ldots, g_n$, of a function on A.

Similarly, we can define the notion of *relative* ***While*** *semicomputability* of a relation on A.

We can also define the notion of *uniform relative* ***While*** *computability* (or *semicomputability*) over a class $\mathbb{K}$.

Lemma 3.32 (Transitivity of relative computability). *If f is* ***While*** *computable in $g_1, \ldots, g_m, h_1, \ldots, h_n$, and $g_1, \ldots, g_m$ are* ***While*** *computable in $h_1, \ldots, h_n$, then f is* ***While*** *computable in $h_1, \ldots, h_n$.*

Proof. Suppose that g_i is computable by a ***While***$(\mathtt{h}_1, \ldots, \mathtt{h}_n)$ procedure P_i for $i = 1, \ldots, m$. Now, given a ***While***$(\mathtt{g}_1, \ldots, \mathtt{g}_m, \mathtt{h}_1, \ldots, \mathtt{h}_n)$

procedure P for f, *replace* each oracle call $\mathtt{x} := \mathtt{g}_i(t)$ in the body of P by the procedure call $\mathtt{x} := P_i(t)$. This results in a $\boldsymbol{While}(\mathtt{h}_1, \dots, \mathtt{h}_n)$ procedure — actually, a $\boldsymbol{WhileProc}(\mathtt{h}_1, \dots, \mathtt{h}_n)$ procedure (section 3.9) — which also computes f. ∎

Note that this result holds over a given algebra A, or uniformly over a class $\mathbb{K}$ of Σ-algebras.

3.11 $\boldsymbol{For}(\Sigma)$ computability

We consider briefly another programming language, $\boldsymbol{For} = \boldsymbol{For}(\Sigma)$, which also plays a role in this paper.

Assume now that Σ is an *N-standard signature*, and A an *N-standard algebra.* The syntax for $\boldsymbol{For}$ is like that for $\boldsymbol{While}$, except that $\boldsymbol{Stmt}(\Sigma)$ is defined by *replacing* the loop statement $\mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}$ by

$$\mathsf{for}\ t\ \mathsf{do}\ S\ \mathsf{od}, \tag{3.8}$$

where $t : \mathsf{nat}$, with the informal semantics: execute S k times, where t evaluates to k. More formally: first we define the notation S^k $(k \geq 0)$ to mean the k-fold iterate of S, i.e.,

$$S^k \equiv \begin{cases} S; \dots ; S \quad (k \text{ times}) & \mathsf{if}\ k > 0 \\ \mathsf{skip} & \mathsf{if}\ k = 0. \end{cases}$$

We now define the semantics of $\boldsymbol{For}$ by modifying the definitions (section 3.5) of the functions $\boldsymbol{First}$ and $\boldsymbol{Rest}^A$, replacing case 4 with:

Case 4′. $S \equiv \mathsf{for}\ t\ \mathsf{do}\ S_0\ \mathsf{od}.$

$$\begin{aligned} \boldsymbol{First}(\mathrm{S}) &= \mathsf{skip} \\ \boldsymbol{Rest}^A(S, \sigma) &= (S_0)^k \end{aligned}$$

where $k = [\![t]\!]^A \sigma$.

Note that t is evaluated (to k) once, upon initial entry into the loop, which is then executed exactly k times (even if the value of t changes in the course of the execution). Thus $[\![S]\!]^A$ is always *total*, and functions computable by $\boldsymbol{For}$ procedures are always total.

We define $\boldsymbol{For}(A)$ to be the class of functions $\boldsymbol{For}$ computable on A.

As in section 3.10, we can define the notion of *relative* $\boldsymbol{For}(\Sigma)$ *computability*, and prove a transitivity lemma for this, analogous to Lemma 3.32.

Example 3.33. The functions $\boldsymbol{For}$ computable on $\mathcal{N}$ of type $\mathsf{nat}^k \to \mathsf{nat}$ are precisely the primitive recursive functions over $\mathbb{N}$.

This follows from the equivalence of primitive recursiveness and $\boldsymbol{For}$ computability on the naturals (proved in Meyer and Ritchie [1967]; see, Davis and Weyuker [1983], for example) or from section 8. Hence *every primitive recursive function over* $\mathbb{N}$ *is* ***For*** *computable on every N-standard algebra.*

(Compare Example 3.14(a).)

Proposition 3.34.

(a) ***For***(Σ) *computability implies* ***While***(Σ) *computability. More precisely, there is an effective translation* $S \to S'$ *of* ***For***(Σ) *statements to* ***While***(Σ) *statements, and (correspondingly) a translation* $P \to P'$ *of* ***For***(Σ) *procedures to* ***While***(Σ) *procedures which is semantics preserving, i.e., for all* ***For***(Σ) *procedures* P *and* N*-standard* Σ*-algebras* A, $[\![P]\!]^A = [\![P']\!]^A$.

(b) *More generally, relative* ***For***(Σ) *computability implies relative* ***While***(Σ) *computability.*

Proof. Simple exercise. ∎

3.12 *While*^{N} and *For*^{N} computability

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

Definition 3.35.

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

(b) ***Proc***$^N(\Sigma)$ is the class of ***While***$^N(\Sigma)$ procedures.

Definition 3.36 (*While*^{N} computable functions).

(a) A function f on A is *computable on A by a* ***While***^{N} *procedure P* if $f = P^A$. It is ***While***^{N} *computable on* A if it is computable on A by some ***While***^{N} procedure.

(b) A family $f = \langle f_A \mid A \in \mathbb{K}\rangle$ of functions is ***While***^{N} *computable uniformly over* $\mathbb{K}$ if there is a ***While***^{N} procedure P such that for all $A \in \mathbb{K}$, $f_A = P^A$.

(c) ***While***$^N(A)$ is the class of functions ***While***^{N} computable on A.

The class of ***For***$^N(\Sigma)$ procedures, and ***For***$^N(\Sigma)$ computability, are defined analogously.

Remark 3.37.

(a) If A is N-standard (so that ***For*** computability is defined on A), then A^N has *two* copies of $\mathbb{N}$, which we can call $\mathbb{N}$ and $\mathbb{N}'$, of sort nat and nat', respectively (each with 0, S and $<$ operations). To avoid technical problems, we assume then that in the for command ((3.8) in section 3.11), the term t can have sort nat *or* nat'. This assumption helps us prove certain desirable results, for example:

(i) There are ***For***(A^N) computable bijections, in both directions, between the two copies of $\mathbb{N}$.

  (*ii*) ***For*** computability implies $\boldsymbol{For}^N$ computability (the seemingly trivial direction '$\Longleftarrow$' of Proposition 3.38).

(*b*) $\boldsymbol{For}^N$ computability implies $\boldsymbol{While}^N$ computability (cf. Proposition 3.34).

(*c*) Relativised versions of $\boldsymbol{While}^N$ and $\boldsymbol{For}^N$ computability can be defined as with ***While*** computability (section 3.10), and corresponding transitivity lemmas (cf. Lemma 3.32) proved. Also, relative $\boldsymbol{For}^N$ computability implies relative $\boldsymbol{While}^N$ computability.

Proposition 3.38. *If A is N-standard, then $\boldsymbol{While}^N$ (or $\boldsymbol{For}^N$) computability coincides with* ***While*** *(or* ***For****) computability on A.*

Proof. For the direction '$\boldsymbol{While}^N$ (or $\boldsymbol{For}^N$) computability $\Longrightarrow$ ***While*** (or ***For***) computability', we can use the coding of A^N in A (see Remark 2.28(*d*)), or, more simply, *represent* the computation over A^N by computation over A, by 'identifying' the two carriers $\mathbb{N}'$ and $\mathbb{N}$ with each other, or (equivalently) 'identifying' the two sorts nat' and nat, renaming variables of these sorts suitably to avoid conflicts. (See also Remark 3.37(*a*).) ∎

3.13 $\boldsymbol{While}^*$ and $\boldsymbol{For}^*$ computability

Recall the algebra A^* of arrays over A, with signature Σ^* (section 2.7). Consider now the ***While*** and ***For*** programming languages over Σ^*.

Definition 3.39.

(*a*) A sort of Σ^* is called *simple*, *augmented* or *starred* according as it has the form s, s^{u} or s^* (respectively), for some $s \in \boldsymbol{Sort}(\Sigma)$.

(*b*) A variable is called *simple*, *augmented* or *starred* according as its sort is simple, augmented or starred.

Note that every sort of Σ^* is simple, augmented, starred or nat.

Definition 3.40.

(*a*) A $\boldsymbol{While}^*(\Sigma)$ procedure is a $\boldsymbol{While}(\Sigma^*)$ procedure in which the *input* and *output* variables are *simple*. (However the auxiliary variables may be augmented or starred or nat.)

(*b*) $\boldsymbol{Proc}^* = \boldsymbol{Proc}^*(\Sigma)$ is the class of $\boldsymbol{While}^*(\Sigma)$ procedures.

(*c*) $\boldsymbol{Proc}^*_{u\to v} = \boldsymbol{Proc}^*(\Sigma)_{u\to v}$ is the class of $\boldsymbol{While}^*(\Sigma)$ procedures of type $u \to v$, for any Σ-product types u and v.

Remark 3.41. We can assume that the auxiliary variables of a $\boldsymbol{While}^*$ procedure are either simple or starred or nat, since a procedure with augmented variables as auxiliary variables can be replaced by one with simple variables, by the device of coding A^{u} in A (see Remark 2.30(*c*)).

Definition 3.42 ($\boldsymbol{While}^*$ computable functions).

(*a*) A function f on A is *computable on A by a $\boldsymbol{While}^*$ procedure P* if $f = P^A$. It is *$\boldsymbol{While}^*$ computable on A* if it is computable on A by some $\boldsymbol{While}^*$ procedure.

(*b*) A family $f = \langle f_A \mid A \in \mathbb{K} \rangle$ of functions is ***While**** *computable uniformly over* $\mathbb{K}$ if there is a ***While**** procedure P such that for all $A \in \mathbb{K}$, $f_A = P^A$.

(*c*) ***While****(A) is the class of functions ***While**** computable on A.

The class of ***For****(Σ) procedures, and ***For****(Σ) computability, are defined analogously.

Remark 3.43.

(*a*) ***While**** computability will be the basis for a generalised Church–Turing thesis, as we will see in section 8.8.

(*b*) ***For****(Σ) computability implies ***While****(Σ) computability (cf. Proposition 3.34).

(*c*) Relativised versions of ***While**** and ***For**** computability can be defined as with ***While*** computability (section 3.10) and corresponding transitivity lemmas (cf. Lemma 3.32) proved. Also, relative ***For**** computability implies relative ***While**** computability.

(*d*) In $\mathcal{N}$, ***While***^{N} and ***While**** computability are equivalent to ***While*** computability, which in turn is equivalent to *partial recursiveness* over $\mathbb{N}$ (Example 3.14(*a*)). Similarly, in $\mathcal{N}$, ***For***, ***For***^{N} and ***For**** computability are all equivalent to *primitive recursiveness* (Example 3.33).

Theorem 3.44 (Locality of computation for *While procedures).** *For a **While*** procedure* $P : u \to v$ *and* $a \in A^u$ *such that* $P^A(a)\downarrow$,

$$P^A(a) \ \in \ \langle a \rangle_v^A.$$

Proof. This follows from the corresponding Theorem 3.30 for ***While*** computability, applied to A^*, together with Σ^*/Σ conservativity of subalgebra generation (to be proved below, in Corollary 3.65). ∎

The following observation will be needed later.

Proposition 3.45. *On* A^*, ***While*** (or **For***) computability coincides with **While** (or **For**) computability.*

This follows from the effective coding of $(A^*)^*$ in A^* (Remark 2.31(*d*)).

Remark 3.46 (Internal versions of *While and *For** computability).** If A is N-standard, we can consider 'internal versions' of ***While**** and ***For**** computability, based on the 'internal version' of A^*, which uses the copy of $\mathbb{N}$ already in A instead of a 'new' copy (see Remark 2.31(*c*)). We can show that these versions provide the same models of computation as our standard ('external') versions.

Proposition 3.47. *Suppose A is N-standard. Let **While***$'$ and **For***$'$ computability on A be the 'internal versions' of **While*** and **For*** (respectively) computability on A (see previous remark). Then **While***$'$ and*

For*$^{*\prime}$* *computability coincide with* ***While****** *and* ***For****** *(respectively) computability on A.*

Proof. Exercise. (Cf. Proposition 3.38.) ∎

3.14 Remainder set of a statement; snapshots

We now return to the operational semantics of section 3.4. The concepts developed here will be useful in investigating the solvability of the halting problem for certain algebras (Section 5.6). First we define the remainder set ***RemSet***(S) of a statement S, which is (roughly) the set of all possible iterations of the ***Rest***^{A} operation on S at any state.

Definition 3.48. The *remainder set* ***RemSet***(S) of S is defined by structural induction on S:
Case 1. S is atomic.

$$\boldsymbol{RemSet}(S) \;=\; \{S\}.$$

Case 2. $S \equiv S_1; S_2$.

$$\boldsymbol{RemSet}(S) \;=\; \{S_1'; S_2 \mid S_1' \in \boldsymbol{RemSet}(S_1)\} \cup \boldsymbol{RemSet}(S_2).$$

Case 3. $S \equiv \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}$.

$$\boldsymbol{RemSet}(S) \;=\; \{S\} \cup \boldsymbol{RemSet}(S_1) \cup \boldsymbol{RemSet}(S_2).$$

Case 4. $S \equiv \mathsf{while}\ b\ \mathsf{do}\ S_0\ \mathsf{od}$.

$$\boldsymbol{RemSet}(S) \;=\; \{S\} \cup \{S_0'; S \mid S_0' \in \boldsymbol{RemSet}(S_0)\}.$$

Example 3.49. Consider a statement of the form

$$S \equiv a_1; \mathsf{while}\ b\ \mathsf{do}\ a_2; a_3; a_4\ \mathsf{od}; a_5$$

where the a_i are atomic statements (using *ad hoc* notation) and b is a Boolean test. Then ***RemSet***(S) consists of the following:

$$\begin{array}{l}
S,\\
\mathsf{while}\ b\ \mathsf{do}\ a_2; a_3; a_4\ \mathsf{od}; a_5,\\
a_2; a_3; a_4;\ \mathsf{while}\ b\ \mathsf{do}\ a_2; a_3; a_4\ \mathsf{od}; a_5,\\
a_3; a_4;\ \mathsf{while}\ b\ \mathsf{do}\ a_2; a_3; a_4\ \mathsf{od}; a_5,\\
a_4;\ \mathsf{while}\ b\ \mathsf{do}\ a_2; a_3; a_4\ \mathsf{od}; a_5,\\
a_5.
\end{array}$$

The next proposition says that ***RemSet***(S) contains S, and is closed under the '***Rest***' operation (for any state).

Proposition 3.50.

(a) $S \in \boldsymbol{RemSet}(S)$.

(b) $S' \in \boldsymbol{RemSet}(S) \implies \boldsymbol{Rest}^A(S', \sigma) \in \boldsymbol{RemSet}(S)$ *for any state* σ.

Proof. By structural induction on S. ∎

Proposition 3.51. $\boldsymbol{RemSet}(S)$ *is finite.*

Proof. Structural induction on S. ∎

Definition 3.52. The *statement remainder function*

$$\boldsymbol{Rem}^A : \boldsymbol{Stmt} \times \boldsymbol{State}(\mathrm{A}) \times \mathbb{N} \to \boldsymbol{Stmt}$$

is the function such that $\boldsymbol{Rem}^A(S, \sigma, n)$ is the statement (the 'remainder of S') about to be executed at step n of the computation of S on A, starting in state σ (or skip when the computation is over). This is defined by recursion on n (tail recursion again):

$$\boldsymbol{Rem}^A(S, \sigma, 0) \;=\; S$$

$$\boldsymbol{Rem}^A(S, \sigma, n+1) \;=\; \begin{cases} \mathsf{skip} & \text{if } n > 0 \text{ and } S \text{ is atomic} \\ \boldsymbol{Rem}^A(\boldsymbol{Rest}^A(S, \sigma), \boldsymbol{Comp}_1^A(S, \sigma), n) & \\ & \text{otherwise.} \end{cases}$$

Note the similarity with the tail recursive definition of $\boldsymbol{Comp}^A$ (section 3.4). Note also that for $n = 1$, this yields

$$\boldsymbol{Rem}^A(S, \sigma, 1) \;=\; \boldsymbol{Rest}^A(S, \sigma).$$

The two functions $\boldsymbol{Comp}$ and $\boldsymbol{Rem}$ also satisfy the following pair of relationships, which (together with suitable base cases $n = 0$) could be taken as a (re-)definition of them by *simultaneous primitive recursion*:

Proposition 3.53.

(a) $\boldsymbol{Comp}^A(S, \sigma, n+1) = \boldsymbol{Comp}_1^A(\boldsymbol{Rem}^A(S, \sigma, n), \boldsymbol{Comp}^A(S, \sigma, n))$

(b) $\boldsymbol{Rem}^A(S, \sigma, n+1) = \boldsymbol{Rest}^A(\boldsymbol{Rem}^A(S, \sigma, n), \boldsymbol{Comp}^A(S, \sigma, n))$

provided $\boldsymbol{Comp}^A(S, \sigma, n) \neq *$.

Proof. Exercise. ∎

Proposition 3.54. *For all n,* $\boldsymbol{Rem}^A(S, \sigma, n) \in \boldsymbol{RemSet}(S) \cup \{\mathsf{skip}\}$.

Proof. Induction on n. Use Proposition 3.50. ∎

If we put $S_n = \boldsymbol{Rem}^A(S, \sigma, n)$, then the sequence of statements $S \equiv S_0, S_1, S_2, \ldots$ is called the *remainder sequence generated by S at σ*, written $\boldsymbol{RemSeq}^A(S, \sigma)$.

Corollary 3.55. *For fixed S and σ, the range of $\boldsymbol{RemSeq}^A(S,\sigma)$ is finite.*

Proof. From Propositions 3.51 and 3.54. ∎

Now we introduce the notion of a 'snapshot'.

Definition 3.56.

(*a*) A *snapshot* is an element (σ, S) of $(\boldsymbol{State}(\mathrm{A}) \cup \{*\}) \times \boldsymbol{Stmt}$.

(*b*) The *snapshot function*

$$\boldsymbol{Snap}^A : \boldsymbol{Stmt} \times \boldsymbol{State}(\mathrm{A}) \times \mathbb{N} \to (\boldsymbol{State}(\mathrm{A}) \cup \{*\}) \times \boldsymbol{Stmt}$$

is defined by

$$\boldsymbol{Snap}^A(S,\sigma,n) = (\boldsymbol{Comp}^A(S,\sigma,n), \boldsymbol{Rem}^A(S,\sigma,n)).$$

If we put $\boldsymbol{Snap}^A(S,\sigma,n) = (\sigma_n, S_n)$, so that $\sigma_n = \boldsymbol{Comp}^A(S,\sigma,n)$ and $S_n = \boldsymbol{Rem}^A(S,\sigma,n)$, then the sequence

$$(\sigma, S) = (\sigma_0, S_0), (\sigma_1, S_1), (\sigma_2, S_2), \ldots$$

is called the *snapshot sequence generated by S at σ*, written $\boldsymbol{SnapSeq}^A(S,\sigma)$. It is either infinite, or terminates in a 'final snapshot' (σ_n, S_n), where $\boldsymbol{Comp}^A(S,\sigma,n+1) = *$ and $\boldsymbol{Rem}^A(S,\sigma,n) = \mathsf{skip}$.

Its importance lies in the following:

Proposition 3.57. *If the snapshot sequence generated by S at σ repeats a value at some point, then it is periodic from that point on. In other words, if for some m, n with $m \neq n$*

$$\boldsymbol{Snap}^A(S,\sigma,m) = \boldsymbol{Snap}^A(S,\sigma,n) \neq (*, \mathsf{skip})$$

i.e., $\sigma_m = \sigma_n \neq *$ and $S_m = S_n$, then for all $k > 0$

$$\boldsymbol{Snap}^A(S,\sigma,m+k) = \boldsymbol{Snap}^A(S,\sigma,n+k) \neq (*, \mathsf{skip}).$$

Proof. Exercise. ∎

Corollary 3.58. *If the snapshot sequence generated by S at σ repeats a value, then it is infinite.*

Remark 3.59.

(*a*) The snapshot function will be used later, in considering the solvability of the halting problem for locally finite algebras (section 5.5).

(*b*) The snapshot function is adapted from Davis and Weyuker [1983] (or Davis *et al.* [1994]). There a 'snapshot' or 'instantaneous description' of a program P is defined as a pair (i, σ) consisting of an *instruction number* (or line number) i of P, and the state σ. The reliance on instruction numbers is possible here because programs consist of sequences of elementary instructions, including the conditional jump. However, in the context of our $\boldsymbol{While}$ programming language, the specification of an instantaneous description by a simple 'instruction number' is impossible; we need the more complex notion of a particular 'remainder' of the given program (or statement).

3.15 Σ^*/Σ conservativity for terms

We conclude this section with a very useful syntactic conservativity theorem (Theorem 3.63) which says that every Σ^*-term with sort in Σ is effectively semantically equivalent to a Σ-term. This theorem will be used in sections 4 (universality for $\boldsymbol{While}^*$ computations: Corollary 4.15) and 5 (strengthening Engeler's lemma: Theorem 5.58).

First we review and extend our notation for certain syntactic classes of terms.

Notation 3.60.

(*a*) $\boldsymbol{Term}_{\mathsf{a}} = \boldsymbol{Term}_{\mathsf{a}}(\Sigma)$ is the class of Σ-terms t with $\boldsymbol{var}(t) \subseteq \mathsf{a}$, and $\boldsymbol{Term}_{\mathsf{a},s} = \boldsymbol{Term}_{\mathsf{a},s}(\Sigma)$ is the class of such terms of sort s.

(*b*) Further, we define:

$$\boldsymbol{Term}^*_{\mathsf{a}} = \boldsymbol{Term}_{\mathsf{a}}(\Sigma^*)$$
$$\boldsymbol{Term}^N_{\mathsf{a}} = \boldsymbol{Term}_{\mathsf{a}}(\Sigma^N)$$
$$\boldsymbol{Term}^{u,N}_{\mathsf{a}} = \boldsymbol{Term}_{\mathsf{a}}(\Sigma^{u,N})$$

and similarly,

$$\boldsymbol{Term}^*_{\mathsf{a},s} = \boldsymbol{Term}_{\mathsf{a},s}(\Sigma^*) \quad \text{for any sort } s\text{, etc.}$$

(*c*) For any $\Sigma' \supseteq \Sigma$, we write $\boldsymbol{Term}_{\mathsf{a}}(\Sigma'/\Sigma)$ for the class of Σ'-terms of sort in Σ (but possibly with subterms of sort in $\Sigma' \setminus \Sigma$), and $\boldsymbol{Term}_{\mathsf{a},s}(\Sigma'/\Sigma)$ for the class of such terms of sort s (in Σ).

We will show that for all $s \in \boldsymbol{Sort}(\Sigma)$, every term in $\boldsymbol{Term}^*_{\mathsf{a},s}$ (i.e., Σ^*-term of sort s) is effectively equivalent to a term in $\boldsymbol{Term}_{\mathsf{a},s}$ (i.e., a Σ-term of sort s). We will do this in three stages:

(1°) Define an effective transformation of Σ^*-terms (of sort in $\Sigma^{\mathsf{u},N}$) to $\Sigma^{\mathsf{u},N}$-terms.

(2°) Define an effective transformation of $\Sigma^{\mathsf{u},N}$-terms (of sort in Σ^N) to Σ^N-terms.

(3°) Define an effective transformation of Σ^N-terms (of sort in Σ) to Σ-terms.

here, in all cases, the program variables of the terms are among a.

In preparation for this, we must define the notion of the *maximum value* of a term in $\boldsymbol{Term}^*_{\mathsf{a},\mathsf{nat}}$. This is the maximum possible numerical value that such a term could have, under any assignment to the variables a.

Definition 3.61. For $t \in \boldsymbol{Term}^*_{\mathsf{a},\mathsf{nat}}$, its maximum value $\boldsymbol{maxval}(t) \in \mathbb{N}$ is defined by induction on the complexity of t (which we can take as the length of t as a string of symbols: cf. Remark 3.2). There are four cases:

(a) $t \equiv 0 : \boldsymbol{maxval}(t) = 0$.
(b) $t \equiv \mathsf{S}t_0 : \boldsymbol{maxval}(t) = \boldsymbol{maxval}(t_0) + 1$.
(c) $t \equiv \mathsf{if}(b, t_1, t_2) : \boldsymbol{maxval}(t) = \max(\boldsymbol{maxval}(t_1), \boldsymbol{maxval}(t_2))$.
(d) $t \equiv \mathrm{Lgths}(r)$, where r is of starred sort. There are four subcases, according to the form of r:

(i) $r \equiv \mathsf{Null} : \boldsymbol{maxval}(t) = 0$.

(ii) $r \equiv \mathsf{Update}(r_0, t_1, t_2) : \ \boldsymbol{maxval}(t) = \boldsymbol{maxval}(\mathsf{Lgth}(r_0))$.

(iii) $r \equiv \mathsf{Newlength}(r_0, t_1) : \ \boldsymbol{maxval}(t) = \boldsymbol{maxval}(t_1)$.

(iv) $r \equiv \mathsf{if}(b, r_1, r_2) : \ \boldsymbol{maxval}(t) = \max(\boldsymbol{maxval}(\mathsf{Lgth}(r_1)), \boldsymbol{maxval}(\mathsf{Lgth}(r_2)))$.

Remark 3.62.

(a) This definition, which is used in stage 1 of the syntactic transformation described in Theorem 3.63 below, uses the assumption that the variables of t all have sorts in Σ. If, for example, t (or a subterm of t) was a variable of sort nat, or was of the form $\mathsf{Lgth}(\mathsf{z}^*)$ for a variable z^* of starred sort, we could not define $\boldsymbol{maxval}(t)$.
(b) Suppose (i) Σ is strictly N-standard (and so includes the sort nat), and (ii) the sorts of a do not include nat. Then, with $\boldsymbol{Term}^*_{\mathsf{a},s} = \boldsymbol{Term}_{a,s}(\Sigma^*)$ with the 'internal' version of Σ^* (using this sort nat instead of a 'new' sort, cf. Remark 2.31(c)), we can still give an appropriate definition of $\boldsymbol{maxval}(t)$ for $t \in \boldsymbol{Term}^*_{\mathsf{a},\mathsf{nat}}$. (*Check.*)

Theorem 3.63 (Σ^*/Σ conservativity for terms). *Let* a *be an (arbitary but fixed) tuple of* Σ*-variables. For all* $s \in \boldsymbol{Sort}(\Sigma)$*, every term in* $\boldsymbol{Term}^*_{\mathsf{a},s}$ *is effectively semantically equivalent to a term in* $\boldsymbol{Term}_{\mathsf{a},s}$.

Proof. This construction (or transformation) of terms proceeds in three stages:

Stage 1: from Σ^*-terms (of sort in $\Sigma^{\mathbf{u},N}$) to $\Sigma^{\mathbf{u},N}$-terms;
Stage 2: from $\Sigma^{\mathbf{u},N}$-terms (of sort in Σ^N) to Σ^N-terms;
Stage 3: from Σ^N-terms (of sort in Σ) to Σ-terms.

In all cases, the program variables of the terms are among a.

Stage 1: From $\boldsymbol{Term}_{\mathsf{a}}(\Sigma^*/\Sigma^{\mathbf{u},N})$ to $\boldsymbol{Term}_{\mathsf{a}}(\Sigma^{\mathbf{u},N})$. This amounts to removing subterms of starred sort from a term of unstarred sort.

First notice that if a term of unstarred sort contains a subterm of starred sort, then it must contain a (maximal) subterm r of starred sort in one of the three contexts:

$$r = r', \qquad \mathsf{Ap}(r,t), \qquad \mathsf{Lgth}(r).$$

We will show how to eliminate each of these three contexts in turn.
Step a. Transform all contexts of the form $r_1 = r_2$ (r_i of starred sort) to

$$\mathsf{Lgth}(r_1) = \mathsf{Lgth}(r_2) \ \wedge \ \bigwedge_{k=1}^{M} \big(\mathsf{Ap}(r_1,\bar{k}) = \mathsf{Ap}(r_2,\bar{k})\big),$$

where $M = \boldsymbol{maxval}(\mathsf{Lgth}(r_1))$, and $\bar{k}$ is the numeral for k (that is, '0' preceded by 'S' k times).

Now all (maximal) occurrences of a subterm r of starred sort are in a context of the form *either* $\mathsf{Ap}(r,t)$ *or* $\mathsf{Lgth}(r)$.
Step b. Transform all contexts of the form $\mathsf{Ap}(r,t)$, by structural induction on r. There are four cases, according to the form of r:

(i) $r \equiv \mathsf{Null}$:
$$\mathsf{Ap}(r,t) \longmapsto \mathsf{unspec}.$$
(ii) $r \equiv \mathsf{Update}(r_0,t_0,t_1)$:
$$\mathsf{Ap}(r,t) \longmapsto \mathsf{if}(t = t_0 < \mathsf{Lgth}(r_0),\ t_1,\ \mathsf{Ap}(r_0,t)).$$
(iii) $r \equiv \mathsf{Newlength}(r_0,t_0)$:
$$\mathsf{Ap}(r,t) \longmapsto \mathsf{if}(t < t_0,\ \mathsf{Ap}(r_0,t),\ \mathsf{unspec}).$$
(iv) $r \equiv \mathsf{if}(b,r_1,r_2)$:
$$\mathsf{Ap}(r,t) \longmapsto \mathsf{if}(b,\ \mathsf{Ap}(r_1,t),\ \mathsf{Ap}(r_2,t)).$$

Note the use of the 'if' operator in cases (ii) and (iii). Hence the inclusion of 'if' in the definition of standard algebra (section 2.4). Note also the use of '$<$' in cases (ii) and (iii). Hence the inclusion of '$<$' in the definition of the standard algebra $\mathcal{N}$ (Example 2.23(b)) and N-standardisations (section 2.5).
Step c. Transform all contexts of the form $\mathsf{Lgth}(r)$, by structural induction on r. Again there are four cases, according to the form of r:

(i) $r \equiv \mathsf{Null}$:
$$\mathsf{Lgth}(r) \longmapsto 0.$$
(ii) $r \equiv \mathsf{Update}(r_0,t_0,t_1)$:
$$\mathsf{Lgth}(r) \longmapsto \mathsf{Lgth}(r_0).$$
(iii) $r \equiv \mathsf{Newlength}(r_0,t_0)$:
$$\mathsf{Lgth}(r) \longmapsto t_0.$$
(iv) $r \equiv \mathsf{if}(b,r_1,r_2)$:
$$\mathsf{Lgth}(r) \longmapsto \mathsf{if}(b, \mathsf{Lgth}(r_1),\ \mathsf{Lgth}(r_2)).$$

By these three steps, we transform a starred term (i.e., a term of $\boldsymbol{Term}_{\mathsf{a}}(\Sigma^*)$), into an unstarred term (i.e., a term of $\boldsymbol{Term}_{\mathsf{a}}(\Sigma^{\mathsf{u},N})$), as desired, completing stage 1.

Stage 2: From $\boldsymbol{Term}_{\mathsf{a}}(\Sigma^{\mathsf{u},N}/\Sigma^N)$ to $\boldsymbol{Term}_{\mathsf{a}}(\Sigma^N)$. Let t be a term of $\Sigma^{\mathsf{u},N}$, with sort in Σ^N. We note the two following assertions:

(1°) A maximal subterm r^{u} of t of augmented sort s^{u} must occur in one of the following contexts:
  (*a*) $\mathsf{j}_s(r^{\mathsf{u}})$,
  (*b*) $\mathsf{Unspec}_s(r^{\mathsf{u}})$,
  (*c*) $r^{\mathsf{u}} = r'^{\mathsf{u}}$ or $r'^{\mathsf{u}} = r^{\mathsf{u}}$ (for s an equality sort).

(2°) Any term $r^{\mathsf{u}} \in \boldsymbol{Term}_{\mathsf{a}}(\Sigma^{\mathsf{u},N})$ of sort s^{u} is semantically equivalent to a term having one of the following forms:
  (*i*) $\mathsf{i}_s(r)$, where $r \in \boldsymbol{Term}(\Sigma^N)$,
  (*ii*) unspec_s.

Assertion (1°) is proved by a simple inspection of the possibilities, and (2°) is proved by structural induction on r^{u}. (Details are left to the reader.)

Stage 2 is completed by considering all combinations of cases (*a*), (*b*) and (*c*) in (1°) with cases (*i*) and (*ii*) in (2°), and (writing '$\simeq$' for semantic equivalence over $\Sigma^{\mathsf{u},N}$) noting that

$$\begin{aligned}
&(a)\quad \mathsf{j}_s(\mathsf{i}_s(r)) \;\simeq\; r,\\
&\qquad \mathsf{j}_s(\mathsf{unspec}_s) \;\simeq\; \boldsymbol{\delta}^s \quad \text{(cf. section 2.6)},\\
&(b)\quad \mathsf{Unspec}_s(\mathsf{i}_s(r)) \;\simeq\; \mathsf{false},\\
&\qquad \mathsf{Unspec}_s(\mathsf{unspec}_s)) \;\simeq\; \mathsf{true},\\
&(c)\quad (\mathsf{i}_s(r) = \mathsf{i}_s(r')) \;\simeq\; (r = r'),\\
&\qquad (\mathsf{i}_s(r) = \mathsf{unspec}_s) \;\simeq \mathsf{false},\\
&\qquad (\mathsf{unspec}_s = \mathsf{i}_s(r)) \;\simeq \mathsf{false},\\
&\qquad (\mathsf{unspec}_s = \mathsf{unspec}_s) \;\simeq \mathsf{true}.
\end{aligned}$$

Stage 3: From $\boldsymbol{Term}_{\mathsf{a}}(\Sigma^N/\Sigma)$ to $\boldsymbol{Term}_{\mathsf{a}}(\Sigma)$. Let t be a term of Σ^N, with sort in Σ. We note the two following assertions:

(1°) A maximal subterm r of t of sort nat must occur in one of the contexts

$$r < r', \qquad r' < r, \qquad r = r', \qquad r' = r$$

for some subterm r' of sort nat.
(2°) Any term $r \in \boldsymbol{Term}_{\mathsf{a}}(\Sigma^N)$ of sort nat is semantically equivalent to a *numeral* $\bar{n}$.

Again, assertion (1°) is proved by a simple inspection of the possibilities, and (2°) is proved by structural induction on r. (Details are left to the reader.)

Stage 3, and hence the proof of the lemma, is completed by noting that all four cases listed in (1°) are then equivalent to $\bar{m} < \bar{n}$ or $\bar{m} = \bar{n}$, and hence (depending on m and n) to either true or false. ∎

Remark 3.64.

(*a*) The transformation of terms given by the conservativity theorem is primitive recursive in Gödel numbers.

(*b*) Suppose (*i*) Σ is strictly N-standard (and so includes a sort nat), and (*ii*) the sorts of a do not include nat. Then, with $\boldsymbol{Term}^*_{\mathsf{a},s} = \boldsymbol{Term}_{a,s}(\Sigma^*)$ with the 'internal' version of Σ^* (as in Remark 3.62(*b*)), the conservativity theorem still holds. (*Check.*)

Recall Definition 2.15 on generated subalgebras.

Corollary 3.65 (Σ^*/Σ conservativity of subalgebra generation). *Let $X \subseteq \bigcup_{s \in \boldsymbol{Sort}(\Sigma)} A_s$. Then for any Σ-sort s,*

$$\langle X \rangle_s^{A*} = \langle X \rangle_s^A.$$

We can apply this to strengthen Theorem 3.30:

Theorem 3.66 (Locality for $\boldsymbol{While}$, $\boldsymbol{While}^N$ or $\boldsymbol{While}^*$ computable functions). *Let f be a (partial) function on A of type $u \to v$, let $a \in A^u$, and suppose $f(a)\downarrow$. If f is $\boldsymbol{While}$, $\boldsymbol{While}^N$ or $\boldsymbol{While}^*$ computable, then*

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

4 Representations of semantic functions; universality

In this section we examine whether or not the $\boldsymbol{While}$ programming language is a so-called *universal model* of computation. This means answering questions of the form:

> *Let A be a Σ-algebra. Does there exist a universal $\boldsymbol{While}$ program $\boldsymbol{U}_{prog} \in \boldsymbol{While}(\Sigma)$ that can simulate and perform the computations of all programs in $\boldsymbol{While}(\Sigma)$ on all inputs from A? Is there a universal $\boldsymbol{While}$ procedure $\boldsymbol{U}_{proc} \in \boldsymbol{Proc}(\Sigma)$ that can compute all the $\boldsymbol{While}$ computable functions on A?*

These questions have a number of precise and delicate formulations which involve representing faithfully the syntax and semantics of $\boldsymbol{While}$ computations using functions on A.

To this end we need the techniques of Gödel numbering, symbolic computations on terms, and state localisation. Specifically, for Gödel numbering to be possible, we need the sort nat, and so we will investigate 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^*. Among a number of results, we will show that

> *for any given Σ-algebra A, there is a universal* ***While*** *procedure over A if, and only if, there is a* ***While*** *program for term evaluation over A.*

In consequence, because term evaluation is always ***While*** computable on A^*, we have that

> *for any Σ-algebra A, there is a universal* ***While*** *program and universal* ***While*** *procedure over A^*.*

Thus, for any algebra A our $\boldsymbol{While}^*$ model of computation is universal. In particular, we can enumerate the $\boldsymbol{While}^*$ computable functions $\phi_0, \phi_1, \phi_2, \ldots$ of any type $u \to v$ on A, and evaluate them by a universal function $\boldsymbol{U}_{u\to v} : \mathbb{N} \times A^u \to A^v$ defined by

$$\boldsymbol{U}_{u\to v}(i, a) \;=\; \phi_i(a)$$

which is $\boldsymbol{While}^*$ computable, uniformly in the types u, v.

If the Σ-algebra A has a ***While*** program to compute term evaluation, then

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

We consider also the uniformity of universal programs and procedures over a class $\mathbb{K}$ of algebras. Many familiar classes of algebras, such as groups, rings and fields, have ***While*** programs to compute term evaluation uniformly over these classes.

4.1 Gödel numbering of syntax

We assume given a family of numerical codings, or Gödel numberings, of the classes of syntactic expressions of Σ and Σ^*, i.e., a family $\boldsymbol{gn}$ of effective mappings from expressions E to natural numbers $\ulcorner E \urcorner = \boldsymbol{gn}(E)$, which satisfy certain basic properties:

- $\ulcorner E \urcorner$ increases strictly with $\boldsymbol{compl}(E)$, and in particular, the code of an expression is larger than those of its subexpressions.
- sets of codes of the various syntactic classes, and of their respective subclasses, such as $\{\ulcorner t \urcorner \mid t \in \boldsymbol{Term}\}$, $\{\ulcorner t \urcorner \mid t \in \boldsymbol{Term}_s\}$, $\{\ulcorner S \urcorner \mid S \in \boldsymbol{Stmt}\}$, $\{\ulcorner S \urcorner \mid S$ is an assignment$\}$, etc. are primitive recursive;
- We can go primitive recursively from codes of expressions to codes of their immediate subexpressions, and vice versa; thus, for example, $\ulcorner S_1 \urcorner$ and $\ulcorner S_2 \urcorner$ are primitive recursive in $\ulcorner S_1; S_2 \urcorner$, and conversely, $\ulcorner S_1; S_2 \urcorner$ is primitive recursive in $\ulcorner S_1 \urcorner$ and $\ulcorner S_2 \urcorner$.

In short, *we can primitive recursively simulate all operations involved in processing the syntax of the programming language.* This means that the syntactic classes form a computable (in fact, primitive recursive) algebra, in the sense of Definition 1.1. We will use the notation

$$\ulcorner \boldsymbol{Term} \urcorner =_{df} \ \{\, \ulcorner t \urcorner \mid t \in \boldsymbol{Term}\},$$

etc., for sets of Gödel numbers of syntactic expressions.

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

4.2 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 $a \in A^u$ if $\sigma[\mathtt{x}] = a$. (Recall the definition of $\sigma[\mathtt{x}]$ in section 3.2.)

The *state representing function*

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

is defined by

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

The *modified state representing function*

$$\boldsymbol{Rep}^A_{\mathtt{x}*}\colon \boldsymbol{State}(\mathrm{A}) \cup \{*\} \to \ \mathbb{B} \times A^u$$

is defined by

$$\begin{array}{lcl} \boldsymbol{Rep}^A_{\mathtt{x}*}(\sigma) & = & (\mathtt{tt},\, \sigma[\mathtt{x}]) \\ \boldsymbol{Rep}^A_{\mathtt{x}*}(*) & = & (\mathtt{ff},\, \boldsymbol{\delta}^u_A) \end{array}$$

where $\boldsymbol{\delta}^u_A$ is the default tuple of type u in A (section 2.14).

4.3 Representation of term evaluation

Let $\mathtt{x}$ be a u-tuple of variables. Let $\boldsymbol{Term}_{\mathtt{x}} = \boldsymbol{Term}_{\mathtt{x}}(\Sigma)$ be the class of all Σ-terms with variables among $\mathtt{x}$ only, and for all sorts s of Σ, let $\boldsymbol{Term}_{\mathtt{x},s} = \boldsymbol{Term}_{\mathtt{x},s}(\Sigma)$ be the class of such terms of sort s. Similarly, we write $\boldsymbol{TermTup}_{\mathtt{x}}$ for the class of all term tuples with variables among $\mathtt{x}$ only, and $\boldsymbol{TermTup}_{\mathtt{x},v}$ for the class of all v-tuples of such terms.

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

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

defined by

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

is *represented* by the function

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

defined by

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

where σ is any state on A such that $\sigma[\mathtt{x}] = a$. (This is well defined, by Lemma 3.4.) In other words, the following diagram commutes:

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

Strictly speaking, if $\boldsymbol{gn}$ is not surjective on $\mathbb{N}$, then $\boldsymbol{te}^A_{\mathtt{x},s}$ is not uniquely specified by the above definition, or by the diagram. However, we may assume that for n not a Gödel number (of the required sort), $\boldsymbol{te}^A_{\mathtt{x},s}(n, a)$ takes the default value of sort s (2.12). Similar remarks apply to the other representing functions given below.

Further, for a product type v, we will define a evaluating function for *tuples of terms*

$$\boldsymbol{te}^A_{\mathtt{x},v}\colon\ \ulcorner \boldsymbol{TermTup}_{\mathtt{x},v} \urcorner \times A^u\ \to\ A^v$$

similarly, by

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

We will be interested in the computability of these term evaluation representing functions.

4.4 Representation of the computation step function

Let $\boldsymbol{AtSt}_{\mathtt{x}}$ be the class of atomic statements with variables among $\mathtt{x}$ only. The *atomic statement evaluation function on A relative to* $\mathtt{x}$,

$$\boldsymbol{AE}^A_{\mathtt{x}}\colon\ \boldsymbol{AtSt}_{\mathtt{x}} \times\ \boldsymbol{State}(\mathrm{A})\ \to\ \boldsymbol{State}(\mathrm{A}),$$

defined by

$$\boldsymbol{AE}^A_{\mathtt{x}}(S, \sigma) = [\![\mathrm{S}]\!]^A\sigma$$

is *represented* by the function

$$\boldsymbol{ae}^A_{\mathtt{x}}\colon\ \ulcorner \boldsymbol{AtSt}_{\mathtt{x}} \urcorner \times A^u\ \to\ A^u,$$

defined by

$$\boldsymbol{ae}^A_{\mathtt{x}}(\ \ulcorner S \urcorner,\ a)\ =\ (\langle\!| S |\!\rangle^A\sigma)[\mathtt{x}],$$

where σ is any state on A such that $\sigma[\mathtt{x}] = a$. (Again, this is well defined, by Lemma 3.14.) In other words, the following diagram commutes:

$$\begin{tikzcd} \boldsymbol{AtSt}_{\mathtt{x}} \times \boldsymbol{State}(\mathrm{A}) \arrow[r, "\boldsymbol{AE}^A_{\mathtt{x}}"] & \boldsymbol{State}(\mathrm{A}) \\ \ulcorner \boldsymbol{AtSt}_{\mathtt{x}} \urcorner \times A^u \arrow[u, "\langle \boldsymbol{gn}, \boldsymbol{Rep}^A_{\mathtt{x}} \rangle"] \arrow[r, "\boldsymbol{ae}^A_{\mathtt{x}}"'] & A^u \arrow[u, "\boldsymbol{Rep}^A_{\mathtt{x}}"'] \end{tikzcd}$$

Next, let $\boldsymbol{Stmt}_{\mathtt{x}}$ be the class of statements with variables among $\mathtt{x}$ only, and define

$$\boldsymbol{Rest}^A_{\mathtt{x}} =_{df} \boldsymbol{Rest}^A \restriction (\boldsymbol{Stmt}_{\mathtt{x}} \times \boldsymbol{State}(\mathrm{A})).$$

Then $\boldsymbol{First}$ and $\boldsymbol{Rest}^A_{\mathtt{x}}$ are *represented* by the functions

$$\begin{aligned} \boldsymbol{first}&: \quad \ulcorner \boldsymbol{Stmt} \urcorner \to \ulcorner \boldsymbol{AtSt} \urcorner \\ \boldsymbol{rest}^A_{\mathtt{x}}&: \quad \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \times A^u \to \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \end{aligned}$$

which are defined so as to make the following diagrams commute:

$$\begin{tikzcd} \boldsymbol{Stmt} \arrow[r, "\boldsymbol{First}"] & \boldsymbol{AtSt} \\ \ulcorner \boldsymbol{Stmt} \urcorner \arrow[u, "\boldsymbol{gn}"] \arrow[r, "\boldsymbol{first}"'] & \ulcorner \boldsymbol{AtSt} \urcorner \arrow[u, "\boldsymbol{gn}"'] \end{tikzcd}$$

$$\begin{tikzcd} \boldsymbol{Stmt}_{\mathtt{x}} \times \boldsymbol{State}(\mathrm{A}) \arrow[r, "\boldsymbol{Rest}^A_{\mathtt{x}}"] & \boldsymbol{Stmt}_{\mathtt{x}} \\ \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \times A^u \arrow[u, "\langle \boldsymbol{gn}, \boldsymbol{Rep}^A_{\mathtt{x}} \rangle"] \arrow[r, "\boldsymbol{rest}^A_{\mathtt{x}}"'] & \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \arrow[u, "\boldsymbol{gn}"'] \end{tikzcd}$$

Note that $\boldsymbol{first}$ is a function from $\mathbb{N}$ to $\mathbb{N}$, and (unlike $\boldsymbol{rest}^A_{\mathtt{x}}$ and most of the other representing functions here) does not depend on A or $\mathtt{x}$.

Next, the computation step function (relative to $\mathtt{x}$)
$\boldsymbol{Comp}^A_{\mathtt{x}} = \boldsymbol{Comp}^A \restriction (\boldsymbol{Stmt}_{\mathtt{x}} \times \boldsymbol{State}(\mathrm{A}) \times \mathbb{N})$:

$$\boldsymbol{Stmt}_{\mathtt{x}} \times \boldsymbol{State}(\mathrm{A}) \times \mathbb{N} \ \to \ \boldsymbol{State}(A) \cup \{*\}$$

is *represented* by the function

$$\boldsymbol{comp}^A_{\mathtt{x}}\colon\ \ulcorner \boldsymbol{Stmt}_{\mathtt{x}}\urcorner \times A^u \times \mathbb{N}\ \to\ \mathbb{B}\times A^u$$

which is defined so as to make the following diagram commute:

$$\begin{tikzcd}
\boldsymbol{Stmt}_{\mathtt{x}}\times\boldsymbol{State}(\mathrm{A})\times\mathbb{N} \arrow[r, "\boldsymbol{Comp}^A_{\mathtt{x}}"] & \boldsymbol{State}(\mathrm{A})\cup\{*\} \\
\ulcorner \boldsymbol{Stmt}_{\mathtt{x}}\urcorner\times A^u\times\mathbb{N} \arrow[u, "{\langle \boldsymbol{gn},\boldsymbol{Rep}^A_{\mathtt{x}}\boldsymbol{id}_{\mathbb{N}}\rangle}"] \arrow[r, "\boldsymbol{comp}^A_{\mathtt{x}}"'] & \mathbb{B}\times A^u \arrow[u, "\boldsymbol{Rep}^A_{\mathtt{x}}"']
\end{tikzcd}$$

We put

$$\boldsymbol{comp}^A_{\mathtt{x}}(\ulcorner S\urcorner, a, n)\ = (\boldsymbol{notover}^A_{\mathtt{x}}(\ulcorner S\urcorner, a, n),\ \boldsymbol{state}^A_{\mathtt{x}}(\ulcorner S\urcorner, a, n))$$

with the two 'component functions'

$$\begin{aligned}
\boldsymbol{notover}^A_{\mathtt{x}}\colon&\quad \ulcorner \boldsymbol{Stmt}_{\mathtt{x}}\urcorner\times A^u\times\mathbb{N}\ \to\ \mathbb{B}\\
\boldsymbol{state}^A_{\mathtt{x}}\colon&\quad \ulcorner \boldsymbol{Stmt}_{\mathtt{x}}\urcorner\times A^u\times\mathbb{N}\ \to\ A^u
\end{aligned}$$

where $\boldsymbol{notover}^A_{\mathtt{x}}(\ulcorner S\urcorner, a, n)$ tests whether the computation of $\ulcorner S\urcorner$ at a is over by step n, and $\boldsymbol{state}^A_{\mathtt{x}}(\ulcorner S\urcorner, a, n)$ gives the value of the state (representative) at step n.

4.5 Representation of statement evaluation

Let $\boldsymbol{Stmt}_{\mathtt{x}}$ be the class of $\boldsymbol{While}$ statements with variables among $\mathtt{x}$ only. The *statement evaluation function on A relative to* $\mathtt{x}$,

$$\boldsymbol{SE}^A_{\mathtt{x}}\colon \boldsymbol{Stmt}_{\mathtt{x}}\times\ \boldsymbol{State}(\mathrm{A})\ \to\ \boldsymbol{State}(\mathrm{A}),$$

defined by

$$\boldsymbol{SE}^A_{\mathtt{x}}(S,\sigma)\ =\ [\![S]\!]^A\sigma,$$

is *represented* by the (partial) function

$$\boldsymbol{se}^A_{\mathtt{x}}\colon \ulcorner \boldsymbol{Stmt}_{\mathtt{x}}\urcorner\times A^u\ \to\ A^u,$$

defined by

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

where σ is any state on A such that $\sigma[\mathtt{x}] = a$. (This is also well defined, by the functionality lemma for statements, 3.10.) In other words, the following diagram commutes.

$$\begin{tikzcd}
\boldsymbol{Stmt}_{\mathtt{x}} \times \boldsymbol{State}(\mathrm{A}) \arrow[r, "\boldsymbol{SE}^A_{\mathtt{x}}"] & \boldsymbol{State}(\mathrm{A}) \\
\ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \times A^u \arrow[u, "\langle \boldsymbol{gn}, \boldsymbol{Rep}^A_{\mathtt{x}} \rangle"] \arrow[r, "\boldsymbol{se}^A_{\mathtt{x}}"'] & A^u \arrow[u, "\boldsymbol{Rep}^A_{\mathtt{x}}"']
\end{tikzcd}$$

We will also be interested in the computability of $\boldsymbol{se}^A_{\mathtt{x}}$.

4.6 Representation of procedure evaluation

We will want a representation of the class $\boldsymbol{Proc}_{u\to v}$ of all $\boldsymbol{While}$ procedures of type $u \to v$, in order to construct a universal procedure for that type. This turns out to be a rather subtle matter, since it requires a coding for *arbitrary tuples of auxiliary variables.* We therefore postpone such a representation to section 4.8, and meanwhile consider a local version, for the subclass of $\boldsymbol{Proc}_{u\to v}$ of procedures with *auxiliary variables of a given fixed type*, which is good enough for our present purpose (Lemma 4.2 and Theorem 4.3).

So let $\mathsf{a,b,c}$ be pairwise disjoint lists of variables, with types $\mathsf{a} : u$, $\mathsf{b} : v$ and $\mathsf{c} : w$. Let $\boldsymbol{Proc}_{\mathsf{a,b,c}}$ be the class of $\boldsymbol{While}$ procedures of type $u \to v$, with declaration $\mathsf{in\ a\ out\ b\ aux\ c}$. The *procedure evaluation function on* A *relative to* $\mathsf{a,b,c}$

$$\boldsymbol{PE}^A_{\mathsf{a,b,c}}\colon\ \boldsymbol{Proc}_{\mathsf{a,b,c}} \times A^u \ \to\ A^v$$

defined by

$$\boldsymbol{PE}^A_{\mathsf{a,b,c}}(P, a) \ =\ P^A(a)$$

is *represented* by the function

$$\boldsymbol{pe}^A_{\mathsf{a,b,c}}\colon\ \ulcorner \boldsymbol{Proc}_{\mathsf{a,b,c}} \urcorner \times A^u \ \to\ A^v$$

defined by

$$\boldsymbol{pe}^A_{\mathsf{a,b,c}}(\ulcorner P \urcorner,\, a) \ =\ P^A(a).$$

In other words, the following diagram commutes:

$$\begin{tikzcd}
\boldsymbol{Proc}_{\mathsf{a,b,c}} \times A^u \arrow[d, "\langle \boldsymbol{gn}, \boldsymbol{id}_{A^u}\rangle"'] \arrow[dr, "PE^A_{\mathsf{a,b,c}}"] & \\
\ulcorner \boldsymbol{Term}_{\mathtt{x},\mathtt{s}} \urcorner \times A^u \arrow[r, "\boldsymbol{pe}^A_{\mathsf{a,b,c}}"'] & A^v
\end{tikzcd}$$

We will also be interested in the computability of $\boldsymbol{pe}^A_{\mathsf{a,b,c}}$.

4.7 Computability of semantic representing functions; term evaluation property

By examining the definitions of the various semantic functions in Section 3, we can infer the relative computability of the corresponding representing functions, as follows.

Lemma 4.1. *The function* $\boldsymbol{first}$: $\mathbb{N} \to \mathbb{N}$ *is primitive recursive, and hence* $\boldsymbol{While}$ *computable on* A^N*, for any standard* Σ*-algebra* A.

Lemma 4.2. *Let* $\mathtt{x}$ *be a tuple of program variables and* A *a standard* Σ*-algebra.*

(a) $\boldsymbol{ae}^A_{\mathtt{x}}$ *and* $\boldsymbol{rest}^A_{\mathtt{x}}$ *are* $\boldsymbol{While}$ *computable in* $\langle\, \boldsymbol{te}^A_{\mathsf{a},s} \mid s \in \boldsymbol{Sort}(\Sigma)\rangle$ *on* A^N.

(b) $\boldsymbol{comp}^A_{\mathtt{x}}$*, and its two component functions* $\boldsymbol{notover}^A_{\mathtt{x}}$ *and* $\boldsymbol{state}^A_{\mathtt{x}}$*, are* $\boldsymbol{While}$ *computable in* $\boldsymbol{ae}^A_{\mathtt{x}}$ *and* $\boldsymbol{rest}^A_{\mathtt{x}}$ *on* A^N.

(c) $\boldsymbol{se}^A_{\mathtt{x}}$ *is* $\boldsymbol{While}$ *computable in* $\boldsymbol{comp}^A_{\mathtt{x}}$ *on* A^N.

(d) $\boldsymbol{pe}^A_{\mathsf{a,b,c}}$ *is* $\boldsymbol{While}$ *computable in* $\boldsymbol{se}^A_{\mathtt{x}}$ *on* A^N*, where* $\mathtt{x} \equiv \mathsf{a,b,c}$.

(e) $\boldsymbol{te}^A_{\mathtt{x},s}$ *is* $\boldsymbol{While}$ *computable in* $\boldsymbol{pe}^A_{\mathtt{x},\mathtt{y},\langle\rangle}$ *on* A^N*, where* $\mathtt{y}$ *is a variable of sort* s*, not in* $\mathtt{x}$.

The above relative $\boldsymbol{While}$ computability results all hold uniformly for $A \in \boldsymbol{StdAlg}(\Sigma)$.

Proof. Note first that if a semantic function is defined from others by *structural recursion* on a syntactic class of expressions, then a representing function for the former is definable from representing functions for the latter by *course of values recursion* on the set of Gödel numbers of expressions of this class, which forms a primitive recursive subset of $\mathbb{N}$.

We can then prove parts (a)–(d) by examining the definitions of the semantic functions, and applying Lemma 4.1 and (relativised versions of) the following facts:

(1°) If a function f on A^N is defined by *primitive recursion* or *tail recursion* on nat from functions $g, h, \ldots$ on A^N, then f is $\boldsymbol{For}(g, h, \ldots)$ computable on A^N. (Used in (a) and (b).)

(2°) *Course of value recursion* on nat with *range sort* nat is reducible to *primitive recursion* on nat. (Used in (*a*).)

(3°) The constructive least number operator, used in part (*c*) (cf. the definition of ***CompLength*** in section 3.4), is ***While*** computable on A^N.

References for facts (1°) and (3°) are given later (Theorem 8.5). Fact (2°) can be proved by an analogue of a classical technique for computability on $\mathcal{N}$ which can be found in Péter [1967] or Kleene [1952].

We complete the cycle of relative computability by proving (*e*) as follows: given a term $t \in \boldsymbol{Term}_{\mathtt{x},s}$, consider the procedure

$$P \equiv \ \mathsf{proc\ in\ x\ out\ y\ begin\ y}{:=}t\ \ \mathsf{end}.$$

Then since $\ulcorner P \urcorner$ is primitive recursive in $\ulcorner t \urcorner$ and $\boldsymbol{te}^A_{\mathtt{x},s}(\ulcorner t \urcorner, a) = \boldsymbol{pe}_{\mathtt{x},\mathtt{y},\langle\rangle}(\ulcorner P \urcorner, a)$ (and since ***For*** computability implies ***While*** computability), the result follows from (1°). ∎

Theorem 4.3. *The following are equivalent, uniformly for* $A \in \boldsymbol{StdAlg}(\Sigma)$.

(*i*) *For all* x *and* s, *the term evaluation representing function* $\boldsymbol{te}^A_{\mathtt{x},s}$ *is* ***While*** *computable on* A^N.

(*ii*) *For all* x, *the atomic statement evaluation representing function* $\boldsymbol{ae}^A_{\mathtt{x}}$, *and the representing function* $\boldsymbol{rest}^A_{\mathtt{x}}$, *are* ***While*** *computable on* A^N.

(*iii*) *For all* x, *the computation step representing function* $\boldsymbol{comp}^A_{\mathtt{x}}$, *and its two component functions* $\boldsymbol{notover}^A_{\mathtt{x}}$ *and* $\boldsymbol{state}^A_{\mathtt{x}}$, *are* ***While*** *computable on* A^N.

(*iv*) *For all* x, *the statement evaluation representing function* $\boldsymbol{se}^A_{\mathtt{x}}$ *is* ***While*** *computable on* A^N.

(*v*) *For all* a,b,c, *the procedure evaluation representing function* $\boldsymbol{pe}^A_{\mathsf{a,b,c}}$ *is* ***While*** *computable on* A^N.

Proof. From the transitivity lemma for relative computability (3.32) and Lemma 4.2. ∎

Definition 4.4 (Term evaluation).

(*a*) The algebra A has the *term evaluation property (TEP)* if for all x and s, the term evaluation representing function $\boldsymbol{te}^A_{\mathtt{x},s}$ (or, equivalently, any of the other sets of semantic representing functions listed in Theorem 4.3) is ***While*** computable on A^N.

(*b*) The class $\mathbb{K}$ has the *uniform TEP* if the term evaluation representing function is uniformly ***While*** computable on $\mathbb{K}^N$.

Examples 4.5.

(*a*) 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. Consequently, the unordered and ordered algebras of real and complex numbers ($\mathcal{R}, \mathcal{R}^<, \mathcal{C}$ and $\mathcal{C}^<$, defined in Example 2.23), which we will study in section 6, have the TEP. (See Tucker [1980, §5].)

(*b*) An (artificial) example of an algebra without the TEP is given in Moldestad *et al.* [1980b].

Proposition 4.6. *The term evaluation representing function on A^* is* ***For*** *(and hence* ***While****) computable on A^*, uniformly for $A \in$* ***StdAlg***(Σ). *Hence the class* ***StdAlg***(Σ^*) *has the uniform TEP.*

Proof. (Outline.) The function $\boldsymbol{te}_{\mathtt{x},s}^{A^*}$ is definable by course of values recursion (cf. Remark 8.6) on Gödel numbers of Σ^*-terms, uniformly for $A \in$ ***StdAlg***(Σ). It is therefore uniformly ***For*** computable on A^*, by Theorem 8.7(a). ∎

Corollary 4.7.

(*a*) *The term evaluation representing function on A is* ***For**** *(and hence* ***While*****) computable on A^N, uniformly for $A \in$* ***StdAlg***(Σ).

(*b*) *The other semantic representing functions listed in Theorems 4.3 are* ***While**** *computable on A^N, uniformly for $A \in$* ***StdAlg***(Σ).

Remark 4.8. Suppose Σ and A are N-standard. Then the semantic representing functions listed above (such as $\boldsymbol{te}_{\mathtt{x},s}^{A}$) can all be defined over A instead of A^N. In that case, Lemma 4.2, Theorem 4.3, Definitions 4.4 and Corollary 4.7 can all be restated, replacing 'A^N', 'Σ^N' and '$\mathbb{K}^N$' by 'A', 'Σ' and '$\mathbb{K}$', respectively. Similar remarks apply to the definitions and results in Sections 4.8–4.12.

Recall the definitions of generated subalgebras, and minimal carriers and algebras (Definitions 2.15 and 2.17 and Remark 2.16).

Corollary 4.9 (Effective local enumerability).

(*a*) *Given any Σ-product type u and Σ-sort s, there is a* ***For**** *computable uniform enumeration of the carrier set of sort s of the subalgebra $\langle a\rangle^A$ generated by $a \in A^u$, i.e., a total mapping*

$$\boldsymbol{enum}_{u,s}^{A}: \ A^u \times \mathbb{N} \ \to \ A_s$$

which is ***For**** *computable on A^N, such that for each $a \in A^u$, the mapping*

$$\boldsymbol{enum}_{u,s}^{A}(a, \cdot\,) : \ \ulcorner \boldsymbol{Term}_{\mathtt{x},s} \urcorner \ \to \ \langle a\rangle_s^A$$

(where $\mathtt{x} : u$) is surjective.

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

Proof. Define $\boldsymbol{enum}^A_{u,s}$ simply from the appropriate term evaluation representing function:

$$\boldsymbol{enum}^A_{u,s}(a,n) \ = \boldsymbol{te}^A_{\mathtt{x},s}(n,a).$$

∎

Corollary 4.10 (Effective global enumerability).

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

$$\boldsymbol{enum}^A_s\colon\ \mathbb{N} \ \to \ A_s,$$

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

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

Proof. From Corollary 4.9, using the empty list of generators. ∎

4.8 Universal $\boldsymbol{While}^N$ procedure for $\boldsymbol{While}$

It is important to note that the procedure representing function $\boldsymbol{pe}^A_{\mathsf{a,b,c}}$ of section 4.6 is *not* universal for $\boldsymbol{Proc}(\Sigma)_{u\to v}$ (where $\mathtt{a}:u$ and $\mathtt{b}:v$). It is only 'universal' for $\boldsymbol{While}$ procedures of type $u \to v$ *with auxiliary variables of type* $\boldsymbol{type}(\mathtt{c})$. In this subsection we will construct a universal procedure $\mathsf{Univ}^A_{u,v}(\ulcorner P\urcorner, a)$ for all $P \in \boldsymbol{Proc}_{u\to v}$and $a \in A^u$. This incorporates not the auxiliary variables of P themselves, but *representations of their values* as (Gödel numbers of) *terms in the input variables* $\mathtt{a}$. These can then all be coded by a single number variable.

We will, assuming the TEP for A, construct a *universal procedure* for $\boldsymbol{Proc}_{u\to v}$ on A. For this we need another representation of the computation step function which differs in two ways from $\boldsymbol{comp}^A_{\mathtt{x}}$ in section 4.4:

(1°) it is defined relative to a tuple $\mathtt{a}$ of program variables ('input variables'), which does not necessarily include all the variables in S;

(2°) it has as output not a tuple of *values* in A, but a tuple of *terms* in the input variables — or rather, the Gödel number of such a tuple of terms.

More precisely, given a product type $u = s_1 \times \ldots \times s_m$ and a u-tuple of variables $\mathtt{a} : u$, we define

$$\boldsymbol{compu}^A_{\mathtt{a}}\colon\ \ulcorner \boldsymbol{VarTup}\urcorner \times \ulcorner \boldsymbol{Stmt}\urcorner \times A^u \times \mathbb{N} \ \to \ \mathbb{B} \times \ulcorner \boldsymbol{TermTup}\urcorner$$

as follows: for any product type w extending u, i.e., $w = s_1 \times \ldots \times s_p$ for some $p \geq m$, and for any $\mathtt{x} : w$ extending $\mathtt{a}$ (i.e., $\mathtt{x} \equiv \mathtt{a}, \mathtt{x}_{s_{m+1}}, \ldots, \mathtt{x}_{s_p}$), and for any $S \in \boldsymbol{Stmt}_{\mathtt{x}}$, $a \in A^u$ and $n \in \mathbb{N}$,

$$\boldsymbol{compu}^A_{\mathtt{a}}(\ulcorner\mathtt{x}\urcorner,\ulcorner S\urcorner, a, n) \ = \ (b_n, \ulcorner t_n\urcorner)$$

where

(i) $b_n = \boldsymbol{notover}^A_{\mathtt{x}}(\ulcorner S \urcorner, (a, \boldsymbol{\delta}_A), n)$, and

(ii) $t_n \in \boldsymbol{TermTup}_{\mathtt{x},w}$ and $\boldsymbol{te}^A_{\mathtt{x},w}(\ulcorner t_n \urcorner, (a, \boldsymbol{\delta}_A)) = \boldsymbol{state}^A_{\mathtt{x}}(\ulcorner S \urcorner, (a, \boldsymbol{\delta}_A), n)$,

where $\boldsymbol{\delta}_A$ is the default tuple of type $s_{m+1} \times \ldots \times s_p$. This use of default values follows from the *initialisation condition* for output and auxiliary variables in procedures (section 3.1(d)). (This is also what lies behind the functionality lemma 3.11 for procedures.)

(If p is not a Gödel number of a tuple of variables $\mathtt{x}$ which extends $\mathtt{a}$, or if q is not a Gödel number of a statement S with $\boldsymbol{var}(S) \subseteq \mathtt{x}$, then we define $\boldsymbol{compu}^A_{\mathtt{a}}(p, q, a, n) = 0$ (say). This case is decidable primitive recursively in p and q. Similarly for the other functions defined below.)

The function $\boldsymbol{compu}^A_{\mathtt{a}}$ has the two 'component functions'

$$\begin{array}{rcl} \boldsymbol{notoveru}^A_{\mathtt{x}} & : & \boldsymbol{VarTup} \times \ulcorner \boldsymbol{Stmt} \urcorner \times A^u \times \mathbb{N} \to \mathbb{B} \\ \boldsymbol{stateu}^A_{\mathtt{a}} & : & \boldsymbol{VarTup} \times \ulcorner \boldsymbol{Stmt} \urcorner \times A^u \times \mathbb{N} \to \ulcorner \boldsymbol{TermTup} \urcorner \end{array}$$

where, for $\mathtt{x}$ extending $\mathtt{a}$ and $s \in \boldsymbol{Stmt}_{\mathtt{x}}$,

$$\begin{array}{rcl} \boldsymbol{notoveru}^A_{\mathtt{x}}(\ulcorner \mathtt{x} \urcorner, \ulcorner S \urcorner, a, n) & = & b_n \\ \boldsymbol{stateu}^A_{\mathtt{a}}(\ulcorner \mathtt{x} \urcorner, \ulcorner S \urcorner, a, n) & = & \ulcorner t_n \urcorner. \end{array}$$

Compare these functions with $\boldsymbol{comp}^A_{\mathtt{x}}$ and its components $\boldsymbol{notover}^A_{\mathtt{x}}$ and $\boldsymbol{state}^A_{\mathtt{x}}$ (section 4.4). Note that for any $\mathtt{x}$ extending $\mathtt{a}$ and $S \in \boldsymbol{Stmt}_{\mathtt{x}}$,

$$\begin{array}{rcl} \boldsymbol{notover}^A_{\mathtt{x}}(\ulcorner S \urcorner, (a, \boldsymbol{\delta}_A), n) & = & \boldsymbol{notoveru}^A_{\mathtt{x}}(\ulcorner \mathtt{x} \urcorner, \ulcorner S \urcorner, a, n) = b_n \\ \boldsymbol{state}^A_{\mathtt{x}}(\ulcorner S \urcorner, (a, \boldsymbol{\delta}_A), n) & = & \boldsymbol{te}^A_{\mathtt{x},w}(\ulcorner t_n \urcorner, (a, \boldsymbol{\delta}_A)). \end{array}$$

Think of $\boldsymbol{compu}^A_{\mathtt{a}}$ and its component functions as uniform (in $\mathtt{x}$) versions of $\boldsymbol{comp}^A_{\mathtt{x}}$ and its component functions. Only the 'input variables' $\mathtt{a}$ are specified.

We need a syntactic operation on terms and variables.

Definition 4.11. For any term or term tuple t and variable tuple a, $\boldsymbol{subex}(t, a)$ is the result of substituting the default terms $\boldsymbol{\delta}^s$ for all variables $\mathtt{x}^s$ in t *except* for the variables in $\mathtt{a}$.

Remark 4.12.

(a) For all $t \in \boldsymbol{TermTup}$, $\boldsymbol{subex}(t, a) \in \boldsymbol{TermTup}_{\mathtt{a}}$.

(b) $\boldsymbol{subex}$ is primitive recursive in Gödel numbers.

(c) Suppose $t : w$ and $\boldsymbol{var}(t) \subseteq \mathtt{x} \equiv \mathtt{a}, \mathtt{z}$ where $\mathtt{a} : u$. Then for $a \in A^u$,

$$\boldsymbol{te}^A_{\mathtt{a},w}(\ulcorner \boldsymbol{subex}(t, \mathtt{a}) \urcorner, a) = \boldsymbol{te}^A_{\mathtt{x},w}(\ulcorner t \urcorner, (a, \boldsymbol{\delta}_A))$$

where $\boldsymbol{\delta}_A$ is the default tuple of type $\boldsymbol{type}(\mathtt{z})$. This follows from the 'substitution Lemma' in logic; see, for example, Sperschneider and Antoniou [1991].

Lemma 4.13. *The function $\boldsymbol{compu}^A_{\mathtt{a}}$, and its component functions $\boldsymbol{notoveru}^A_{\mathtt{x}}$ and $\boldsymbol{stateu}^A_{\mathtt{a}}$, are* **While** *computable in $\langle \boldsymbol{te}^A_{\mathtt{a},s} \mid s \in \boldsymbol{Sort}(\Sigma) \rangle$ on A^N, uniformly for $A \in \boldsymbol{StdAlg}(\Sigma)$.*

Proof. (Outline.) We essentially redo parts (a) and (b) of Lemma 4.2, using uniform (in $\mathtt{x}$) versions of $\boldsymbol{ae}^A_{\mathtt{x}}$ and $\boldsymbol{rest}^A_{\mathtt{x}}$, i.e., we define (1°) the function

$$\boldsymbol{aeu}^A\colon\ \ulcorner \boldsymbol{VarTup}\urcorner \times \ulcorner \boldsymbol{AtSt}\urcorner \ \to\ \ulcorner \boldsymbol{TermTup}\urcorner$$

where for any $\mathtt{x} : w$ and $S \in \boldsymbol{AtSt}_{\mathtt{x}}$, we have $\boldsymbol{aeu}^A(\ulcorner \mathtt{x}\urcorner, \ulcorner S\urcorner) \in \ulcorner \boldsymbol{TermTup}_{\mathtt{x},w}\urcorner$, such that for any $x \in A^w$,

$$\boldsymbol{te}^A_{\mathtt{x},w}(\boldsymbol{aeu}^A(\ulcorner \mathtt{x}\urcorner \ulcorner S\urcorner), x) \ = \boldsymbol{ae}^A_{\mathtt{x}}(\ulcorner S\urcorner, x);$$

and (2°) the function

$$\boldsymbol{restu}^A_{\mathsf{a}}\colon\ \ulcorner \boldsymbol{VarTup}\urcorner \times \ulcorner \boldsymbol{Stmt}\urcorner \times A^u \ \to\ \ulcorner \boldsymbol{Stmt}\urcorner$$

where for any $\mathtt{x} : w$ extending $\mathsf{a} : u$, $S \in \boldsymbol{AtSt}_0$ and $a \in A^u$,

$$\boldsymbol{restu}^A_{\mathsf{a}}(\ulcorner \mathtt{x}\urcorner, \ulcorner S\urcorner, a) \ = \boldsymbol{rest}^A_{\mathtt{x}}(\ulcorner S\urcorner, (a, \boldsymbol{\delta}_A)).$$

We can then show that

(i) $\boldsymbol{aeu}^A$ is primitive recursive;
(ii) $\boldsymbol{compu}^A_{\mathsf{a}}$ is $\boldsymbol{While}$ computable in $\boldsymbol{restu}^A_{\mathsf{a}}$ on A; and
(iii) $\boldsymbol{restu}^A_{\mathsf{a}}$ is $\boldsymbol{While}$ computable in $\langle \boldsymbol{te}^A_{\mathsf{a},s} \mid s \in \boldsymbol{Sort}(\Sigma)\rangle$.

Combining these three facts gives the result.

Note, in (iii), that the term evaluation functions $\boldsymbol{te}^A_{\mathsf{a},s}$ are used to evaluate Boolean tests in the course of defining $\boldsymbol{restu}^A_{\mathsf{a}}$. The one tricky point is this: how do we evaluate, using $\boldsymbol{te}^A_{\mathsf{a},s}$, a (Gödel number of) a term $t \in \boldsymbol{Term}_{\mathtt{x},s}$, which contains variables in $\mathtt{x}$ other than a? (This is the issue of 'uniformity in $\mathtt{x}$'.) The answer is that by Remark 4.12(c) the evaluation of t is given by $\boldsymbol{te}^A_{\mathsf{a},s}(\ulcorner \boldsymbol{subex}(t,\mathsf{a})\urcorner, a)$. ■

Theorem 4.14 (Universality characterisation theorem for *While*(Σ) computations). *The following are equivalent, uniformly for* $A \in \boldsymbol{StdAlg}(\Sigma)$.

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

$$\mathsf{Univ}_{u,v}\colon \ulcorner \boldsymbol{Proc}_{u\to v}\urcorner \times u \ \to\ v$$

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* $a \in A^u$,

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

Proof.

(i) $\Longrightarrow$ (ii): Assume A has the TEP. We give an informal description of the algorithm represented by the procedure $\mathsf{Univ}_{u,v}$. With *input* ($\ulcorner P\urcorner$, a), where $P \in \boldsymbol{Proc}_{u\to v}$ and $a \in A^u$, suppose

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

where $\mathtt{a} : u$ and $\mathtt{b} : v$. Putting $\mathtt{x}{\equiv}\mathsf{a,b,c}$, evaluate $\boldsymbol{notoveru}^A_{\mathtt{x}}$ $(\ulcorner\mathtt{x}\urcorner, \ulcorner S\urcorner,\ a, n)$ for $n = 0, 1, 2, \ldots,$ until you find the (least) n for which the computation of S at a terminates (if at all), i.e.,, the least $n = n_0$ such that

$$\boldsymbol{notoveru}^A_{\mathtt{x}}(\ulcorner\mathtt{x}\urcorner, \ulcorner S\urcorner, a,\ n_0 + 1)\ =\ \mathsf{ff}.$$

Note that $\boldsymbol{notoveru}^A_{\mathtt{x}}$ is $\boldsymbol{While}$ computable by Lemma 4.13 and assumption. Now let us put

$$\boldsymbol{stateu}^A_{\mathsf{a}}(\ulcorner\mathtt{x}\urcorner, \ulcorner S\urcorner, a,\ n_0)\ = \ulcorner t,\ t',\ t''\urcorner,$$

where the term tuples t, t' and t'' represent the current values of a, b and c, respectively. This is also $\boldsymbol{While}$ computable by Lemma 4.13 and assumption. Finally, the *output* is

$$\boldsymbol{te}^A_{\mathsf{a},v}(\ulcorner \boldsymbol{subex}(t', \mathsf{a})\urcorner, a)$$

(cf. Remark 4.12(c)). By assumption and Remark 4.12(b), this is $\boldsymbol{While}$ computable in $\ulcorner t'\urcorner$ and a, and hence in $\ulcorner P\urcorner$ and a.

(ii) $\Longrightarrow$ (i): Note that for any $\mathsf{a,b,c}$,

$$\boldsymbol{pe}^A_{\mathsf{a,b,c}}\ =\ \mathsf{Univ}^A_{u,v}\restriction(\boldsymbol{Proc}_{\mathsf{a,b,c}}{\times}A^u)$$

where $\mathtt{a} : u$ and $\mathtt{b} : v$. Hence $\boldsymbol{pe}^A_{\mathsf{a,b,c}}$ is $\boldsymbol{While}(\Sigma^N)$ computable if $\mathsf{Univ}^A_{u,v}$ is. The result follows from Theorem 4.3. ∎

Corollary 4.15 (Universality for A^*). *For all Σ-product types u, v, there is a $\boldsymbol{While}^*(\Sigma^N)$ procedure*

$$\mathsf{Univ}^*_{u,v}\colon\ \mathsf{nat}{\times}u\ \to\ v$$

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

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

Proof. $\boldsymbol{StdAlg}(\Sigma^*)$ has the uniform TEP, by Proposition 4.6. ∎

Remark 4.16.

(a) For all u, v, the construction of $\mathsf{Univ}_{u,v}$ (direction (i) $\Rightarrow$ (ii) in the proof of Theorem 4.14) is *uniform* over Σ in the following sense. There is a *relative* $\boldsymbol{While}(\Sigma^N)$ procedure $U_{u,v} : \mathsf{nat} \times u \to v$ containing oracle procedure calls $\langle h_s \mid s \in \boldsymbol{Sort}(\Sigma)\rangle$ (section 3.12) with $h_s : \mathsf{nat}{\times}u \to s$, such that for any $A \in \boldsymbol{StdAlg}(\Sigma)$, if h_s is interpreted as $\boldsymbol{te}^A_{\mathsf{a},s}$ on A (where $\mathtt{a} : u$), then $U_{u,v}$ is universal for $\boldsymbol{Proc}_{u\to v}$ on A. (We ignore the question of whether $\boldsymbol{te}^A_{\mathsf{a},s}$ is computable on A.)

(b) The use of term evaluation occurs at two points in the construction of $\mathsf{Univ}_{u,v}$ (direction (i)$\Longrightarrow$(ii)): (1°) in the evaluation of Boolean tests in the construction of the sequence

$$\boldsymbol{compu}^A_{\mathtt{a}}(\ulcorner \mathtt{x} \urcorner, \ulcorner S \urcorner, a, 0),\ \boldsymbol{compu}^A_{\mathtt{a}}(\ulcorner \mathtt{x} \urcorner, \ulcorner S \urcorner, a, 1),\ \ldots ; \tag{4.1}$$

and (2°) in the evaluation of the output variables t' (see proof of Theorem 4.14). We can separate, and postpone, both these applications of term evaluation by modifying the construction of the universal procedure as follows.

Step 1: Construct from S, not a *computation sequence* as in (4.1) but rather a *computation tree* (section 5.10), specifically $\boldsymbol{comptree}(\ulcorner \mathtt{x} \urcorner, \ulcorner S \urcorner, n)$ (where $\mathtt{x} \equiv \mathtt{a}, \mathtt{b}, \mathtt{c}$), which is the Gödel number of the first n levels of the computation tree from $S \in \boldsymbol{Stmt}_{\mathtt{x}}$ labelled by w-tuples of terms in $\boldsymbol{TermTup}_{\mathtt{x},w}$. Note that $\boldsymbol{comptree}{:}\mathbb{N}^3 \to \mathbb{N}$ is primitive recursive.

Step 2: Select a path in this tree by evaluating Boolean tests (using $\boldsymbol{te}^A_{\mathtt{a},\mathtt{bool}}$ together with the $\boldsymbol{subex}$ operation) until you come (if at all) to a leaf. Evaluate the terms representing the output variables at this leaf (again using $\boldsymbol{te}^A_{\mathtt{a},s}$ with the subex operation).

4.9 Universal $\boldsymbol{While}^N$ procedure for $\boldsymbol{While}^*$

We can strengthen the universal characterisation theorem for $\boldsymbol{While}$ computations (4.14) using the Σ^*/Σ conservativity thorem (3.63).

Theorem 4.17. (Universality characterisation theorem for $\boldsymbol{While}^*$ computations) *The following are equivalent, uniformly for $A \in \boldsymbol{StdAlg}(\Sigma)$.*

(i) *A has the TEP.*

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

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

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 $a \in A^u$,

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

Proof. $(i) \Longrightarrow (ii)$: Modify the proof of Theorem 4.14, following the algorithm of Remark 4.16(b). Construct a computation tree as in 'step 1. Then, in step 2 (term evaluation), *replace* all Boolean terms (in selecting a path) and the output terms (at the leaf) by the corresponding Σ-terms given by Theorem 3.63, and apply $\boldsymbol{te}^A_{\mathtt{a},s}$ (for $s \in \boldsymbol{Sort}(\Sigma)$) to these. Since this transformation of terms is primitive recursive in Gödel numbers (Remark 3.64(a)), the whole algorithm can be formalised as a $\boldsymbol{While}(\Sigma^N)$ procedure.

$(ii) \Longrightarrow (i)$: This follows trivially from Theorem 4.14.

∎

Corollary 4.18. *The following are equivalent, uniformly for* $A \in \boldsymbol{StdAlg}(\Sigma)$.

(i) *A has the TEP.*

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

4.10 Snapshot representing function and sequence

Next we consider the statement remainder and snapshot functions (section 3.14) which will be useful in our investigation of the halting problem (section 5.6). Let `x` : u.

The statement remainder function (relative to `x`)

$$\boldsymbol{Rem}^A_{\mathtt{x}} = \boldsymbol{Rem}^A \restriction (\boldsymbol{Stmt}_{\mathtt{x}} \times \boldsymbol{State}(\mathrm{A}) \times \mathbb{N}) : \\ \boldsymbol{Stmt}_{\mathtt{x}} \times \boldsymbol{State}(\mathrm{A}) \times \mathbb{N} \to \boldsymbol{Stmt}_{\mathtt{x}}$$

(cf. Definition 3.52) is *represented* by the function

$$\boldsymbol{rem}^A_{\mathtt{x}}\colon \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \times A^u \times \mathbb{N} \to \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner$$

which is defined so as to make the following diagram commute:

$$\begin{tikzcd}
\boldsymbol{Stmt}_{\mathtt{x}} \times \boldsymbol{State}(\mathrm{A}) \times \mathbb{N} \arrow[r, "\boldsymbol{Rem}^A_{\mathtt{x}}"] & \boldsymbol{Stmt}_{\mathtt{x}} \\
\ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \times A^u \times \mathbb{N} \arrow[u, "{\langle \boldsymbol{gn}, \boldsymbol{Rep}^A_{\mathtt{x}}, \boldsymbol{id}_{\mathbb{N}} \rangle}"] \arrow[r, "\boldsymbol{rem}^A_{\mathtt{x}}"'] & \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \arrow[u, "\boldsymbol{gn}"']
\end{tikzcd}$$

(Again, this is well defined, by Lemma 3.10.)

The snapshot function (relative to `x`)

$$\boldsymbol{Snap}^A_{\mathtt{x}} = \boldsymbol{Snap}^A \restriction (\boldsymbol{Stmt}_{\mathtt{x}} \times \boldsymbol{State}(\mathrm{A}) \times \mathbb{N}) : \\ \boldsymbol{Stmt}_{\mathtt{x}} \times \boldsymbol{State}(\mathrm{A}) \times \mathbb{N} \to (\boldsymbol{State}(\mathrm{A}) \cup \{*\}) \times \boldsymbol{Stmt}_{\mathtt{x}}$$

(cf. Definition 3.56) is represented by the function

$$\boldsymbol{snap}^A_{\mathtt{x}}\colon \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \times A^u \times \mathbb{N} \to (\mathbb{B} \times A^u) \times \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner$$

which can be defined simply as

$$\begin{aligned}
\boldsymbol{snap}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, n) &= (\boldsymbol{comp}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, n), \boldsymbol{rem}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, n)) \\
&= ((\boldsymbol{notover}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, n), \boldsymbol{state}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, n)), \\
&\quad \boldsymbol{rem}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, n))
\end{aligned}$$

or (equivalently) so as to make the following diagram commute:

$$\begin{tikzcd}
\boldsymbol{Stmt}_{\mathtt{x}} \times \boldsymbol{State}(\mathrm{A}) \times \mathbb{N} \arrow[r, "\boldsymbol{Snap}^A_{\mathtt{x}}"] & (\boldsymbol{State}(A) \cup \{*\}) \times \boldsymbol{Stmt}_{\mathtt{x}} \\
\ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \times A^u \times \mathbb{N} \arrow[r, "\boldsymbol{snap}^A_{\mathtt{x}}"'] \arrow[u, "\langle \boldsymbol{gn}, \boldsymbol{Rep}^A_{\mathtt{x}}, \boldsymbol{id}_{\mathbb{N}} \rangle"] & (\mathbb{B} \times A^u) \times \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \arrow[u, "\langle \boldsymbol{Rep}^A_{\mathtt{x}*}, \boldsymbol{gn} \rangle"']
\end{tikzcd}$$

Fix $\mathtt{x} : u$, $s \in \boldsymbol{Stmt}_{\mathtt{x}}$ and $a \in A^u$. Put $b_n = \boldsymbol{notover}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, n)$, $a_n = \boldsymbol{state}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, n)$ and $\ulcorner S_n \urcorner = \boldsymbol{rem}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, n)$. Then the sequences

$$\begin{aligned}
(\mathtt{tt}, a) &= (b_0, a_0),\ (b_1, a_1),\ (b_2, a_2),\ \ldots \\
\ulcorner S \urcorner &= \ulcorner S_0 \urcorner,\ \ulcorner S_1 \urcorner,\ \ulcorner S_2 \urcorner,\ \ldots \\
((\mathtt{tt}, a), \ulcorner S \urcorner) &= ((b_0, a_0), \ulcorner S_0 \urcorner),\ ((b_1, a_1), \ulcorner S_1 \urcorner),\ ((b_2, a_2), \ulcorner S_2 \urcorner),\ \ldots
\end{aligned}$$

are called, respectively, the *computation representing sequence*, the *remainder representing sequence* and the *snapshot representing sequence* generated by S (or $\ulcorner S \urcorner$) at a (with respect to $\mathtt{x}$), denoted respectively by $\boldsymbol{compseq}^A_{\mathtt{x}}(\ulcorner S \urcorner, a)$, $\boldsymbol{remseq}^A_{\mathtt{x}}(\ulcorner S \urcorner, a)$ and $\boldsymbol{snapseq}^A_{\mathtt{x}}(\ulcorner S \urcorner, a)$. (Compare the sequences $\boldsymbol{CompSeq}^A(S, \sigma)$, $\boldsymbol{RemSeq}^A(S, \sigma)$ and $\boldsymbol{SnapSeq}^A(S, \sigma)$ introduced in section 3.)

The sequences $\boldsymbol{compseq}^A_{\mathtt{x}}(\ulcorner S \urcorner, a)$ and $\boldsymbol{snapseq}^A_{\mathtt{x}}(\ulcorner S \urcorner, a)$ are said to be *non-terminating*, if, for all n, $\boldsymbol{notover}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, n) = \mathtt{tt}$, i.e., for no n is $\boldsymbol{comp}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, n) = (\mathtt{ff}, \boldsymbol{\delta}^u_A)$.

These representing sequences satisfy analogues of the results listed in section 3.14; for example:

Proposition 4.19. *If $\boldsymbol{snapseq}^A_{\mathtt{x}}(\ulcorner S \urcorner, a)$ repeats a value at some point, then it is periodic from that point on, and hence non-terminating. In other words, if for some m, n with $m \neq n$*

$$\boldsymbol{snap}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, m) \ = \ \boldsymbol{snap}^A_{\mathtt{x}}(\ulcorner S \urcorner, a\, n) \ \neq \ ((\mathtt{ff}, \boldsymbol{\delta}^u_A), \mathsf{skip})$$

then, for all $k > 0$,

$$\boldsymbol{snap}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, m + k) \ = \ \boldsymbol{snap}^A_{\mathtt{x}}(\ulcorner S \urcorner, a\, n + k) \ \neq \ ((\mathtt{ff}, \boldsymbol{\delta}^u_A), \mathsf{skip})$$

(Cf. Proposition 3.57 and Corollary 3.58.)

With the function $\boldsymbol{snap}^A_{\mathtt{x}}$, we can extend the list of relative computability results (Lemma 4.2), and add a clause to Theorem 4.3:

Lemma 4.20. *(Cf. Lemma 4.2.) The function $\boldsymbol{snap}^A_{\mathtt{x}}$, and its two component functions $\boldsymbol{comp}^A_{\mathtt{x}}$ and $\boldsymbol{rem}^A_{\mathtt{x}}$, are* $\boldsymbol{While}$ *computable in $\langle \boldsymbol{te}^A_{\mathsf{a},s} \mid s \in \boldsymbol{Sort}(\Sigma) \rangle$ on A^N, uniformly for $A \in \boldsymbol{StdAlg}(\Sigma)$.*

Proof. Simple exercise. ∎

Theorem 4.21. *(Cf. Theorem 4.3.) The following are equivalent, uniformly for $A \in \boldsymbol{StdAlg}(\Sigma)$:*

(i) For all $\mathtt{x}$ *and* s*, the term evaluation representing function* $\boldsymbol{te}^A_{\mathtt{x},s}$ *is* $\boldsymbol{While}$ *computable on* A^N*.*

(ii) For all $\mathtt{x}$*, the snapshot representing function* $\boldsymbol{snap}^A_{\mathtt{x}}$*, and its two component functions* $\boldsymbol{comp}^A_{\mathtt{x}}$ *and* $\boldsymbol{rem}^A_{\mathtt{x}}$*, are* $\boldsymbol{While}$ *computable on* A^N*.*

Proof. As for Theorem 4.3. ∎

A uniform (in $\mathtt{x}$) version of $\boldsymbol{snap}^A_{\mathtt{x}}$ will be used in section 5.6 in our investigation of the 'solvability of the halting problem'.

4.11 Order of a tuple of elements

Let u be a Σ-product type, s a Σ-sort and A a Σ-algebra. The *order function of type* u, s *on* A is the function

$$\boldsymbol{ord}^A_{u,s}\colon A^u \to \mathbb{N}$$

where, for all $x \in A^u$,

$$\boldsymbol{ord}^A_{u,s}(x) \;\simeq\; \boldsymbol{card}(\langle x\rangle^A_s)$$

i.e., the cardinality of the carrier of sort s of the subalgebra of A generated by x. (It is undefined when the cardinality is infinite.)

Note that this is a generalisation of the order operation for single elements of groups (Example 3.14(b)).

Note that for a tuple $x \in A^u$, the subalgebra $\langle x\rangle^A_s$ can be generated in stages as finite sets:

$$\langle x\rangle^A_{s,0} \subseteq \langle x\rangle^A_{s,1} \subseteq \langle x\rangle^A_{s,2} \subseteq \cdots$$

where $\langle x\rangle^A_{s,n}$ is defined by induction on n, simultaneously for all Σ-sorts s (cf. Meinke and Tucker [1992, 3.12.15ff.] for the single-sorted case), and

$$\langle x\rangle^A_s \;=\; \bigcup_n \langle x\rangle^A_{s,n}.$$

Also $\langle x\rangle^A_s$ is finite if, and only if, there exists n such that

$$\langle x\rangle^A_{s,n} \;=\; \langle x\rangle^A_{s,n+1} \tag{4.2}$$

in which case

$$\langle x\rangle^A_{s,n} \;=\; \langle x\rangle^A_{s,n+1} \;=\; \langle x\rangle^A_{s,n+2} \;=\; \cdots \;=\; \langle x\rangle^A_s.$$

Lemma 4.22. *For any tuple of variables* $\mathtt{x} : u$*, there is a primitive recursive function*

$$\boldsymbol{SubAlgStage}_{\mathtt{x},u,s}\colon\; \mathbb{N} \to \mathbb{N}$$

such that $\boldsymbol{SubAlgStage}_{\mathtt{x},u,s}(n)$ *is the Gödel number of a list* $(\ulcorner t_1\urcorner \ldots \ulcorner t_{k_n}\urcorner)$ *of Gödel numbers of the set of terms generated by stage* n*, i.e.,,*

$$\langle x\rangle^A_{s,n} \;=\; \{\, \boldsymbol{te}^A_{\mathtt{x},s}(t_i,x) \mid i=1,\dots,k_n \,\}.$$

Example 4.23. Suppose s is an equality sort.

(*a*) The order function $\boldsymbol{ord}^A_{u,s}$ is $\boldsymbol{While}$ computable in $\boldsymbol{te}^A_{\mathtt{x},s}$ (where $\mathtt{x} : u$) on A^N, uniformly for $A \in \boldsymbol{StdAlg}(\Sigma)$.

(*b*) Hence if A has the TEP, then $\boldsymbol{ord}^A_{u,s}$ is $\boldsymbol{While}$ computable on A^N.

Proof. The algorithm to compute $\boldsymbol{ord}^A_{u,s}$ is (briefly) as follows. Suppose given an input $x \in A^u$. With the help of the functions $\boldsymbol{SubAlgStage}_{\mathtt{x},u,s}$ and $\boldsymbol{te}^A_{\mathtt{x},s}$ and the equality operator on A_s, test for $n = 0,1,2,\dots$ whether (4.2) holds. If and when such an n is found, determine $\boldsymbol{card}(\langle x\rangle^A_{s,n})$, again using the equality operator on A_s (this time to determine repetitions in the list $\langle x\rangle^A_{s,n}$). ∎

4.12 Locally finite algebras

Definition 4.24. An algebra A is *locally finite* if every finitely generated subalgebra of A is finite, i.e., if for every finite $X \subseteq \bigcup_{s\in \boldsymbol{Sort}(\Sigma)} A_s$ and every sort s, $\langle \mathrm{X}\rangle^A_s$ is finite.

Note that A is locally finite if, and only if, $\boldsymbol{ord}^A_{u,s}$ (section 4.11) is total for all u and s.

Example 4.25. Consider the algebra

$$\mathcal{N}_0^- \;=\; (\mathbb{N}^-;\; 0, \mathsf{pred})$$

where $\mathbb{N}^-$ is just (a copy of) $\mathbb{N}$, and 'pred' is the predecessor operation on this: $\mathsf{pred}(n+1) = n$ and $\mathsf{pred}(0) = 0$. We write '$\mathbb{N}^-$' to distinguish this carrier from the 'standard' naturals $\mathbb{N}$, which we can adjoin to form the N-standardised algebra. We also write the sort of $\mathbb{N}^-$ as nat^-. Let

$$\mathcal{N}^- \;=\; (\mathbb{N}^-, \mathbb{B};\; 0, \mathsf{pred}, \mathsf{eq}_{\mathsf{nat}^-},\; \dots\;)$$

be the standardised version of $\mathcal{N}_0^-$ (with nat^- an equality sort). Then both $\mathcal{N}_0^-$ and $\mathcal{N}^-$ are locally finite; in fact for any $k_1,\dots,k_m \in \mathbb{N}^-$,

$$\langle\{k_1,\dots,k_m\}\rangle^{\mathcal{N}^-}_{\mathsf{nat}^-} \;=\; \{0,1,2,\dots,k\}$$

where $k = \max(k_1,\dots,k_m)$. (*Check.*) Hence

$$\boldsymbol{ord}^{\mathcal{N}^-}_{(\mathsf{nat}^-)^m,\,\mathsf{nat}^-}(k_1,\dots,k_m) \;=\; \max(k_1,\dots,k_m)+1.$$

Theorem 4.26. *Suppose A is locally finite. Then for any $\mathtt{x} : u$, $s \in \boldsymbol{Stmt}_{\mathtt{x}}$ and $a \in A^u$:*

(a) $\boldsymbol{snapseq}^A_{\mathtt{x}}(\ulcorner S\urcorner, a)$ has finite range.

(b) $\boldsymbol{snapseq}^A_{\mathtt{x}}(\ulcorner S\urcorner, a)$ (or, equivalently, $\boldsymbol{compseq}^A_{\mathtt{x}}(\ulcorner S\urcorner, a)$) is non-terminating $\Longleftrightarrow$ $\boldsymbol{snapseq}^A_{\mathtt{x}}(\ulcorner S\urcorner, a)$ repeats a value (other than $((\mathsf{ff}, \boldsymbol{\delta}^u_A), \mathsf{skip})$).

Proof.

(a) Consider a typical element of the snapshot representing sequence generated by S at a:

$$(a_n, \ulcorner S_n \urcorner) \tag{4.3}$$

where $a_n = \boldsymbol{comp}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, n)$ and $\ulcorner S_n \urcorner = \boldsymbol{rem}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, n)$ for some n. By Lemma 3.28, a_n must be in $\langle a \rangle^A_u$, which is finite by assumption. Also, by Proposition 3.54, S_n must be in $\boldsymbol{RemSet}(S) \cup \{\mathsf{skip}\}$, which is finite, by Proposition 3.51. Hence the pair (4.3) must be in the product set $\langle a \rangle^A_u \times \ulcorner \boldsymbol{RemSet}(S) \urcorner$, which is also finite.

(b) The direction '$\Longrightarrow$' follows from (a). The direction '$\Longleftarrow$' follows from Corollary 3.58 or (equivalently) Proposition 4.19. ∎

Local finiteness will be used later, in considering 'solvability of the halting problem' (Section 5.6).

4.13 Representing functions for specific terms or programs

The representing functions that we considered in sections 4.3–4.6 and 4.10 have as arguments (typically)

(i) Gödel numbers of terms, statements or procedures, and

(ii) representations of states.

Computability of all these functions is equivalent to the TEP (Theorems 4.3 and 4.21).

Another form of representation which will be useful is to use (i) the term, statement, etc. as a *parameter*, not an *argument*, and just have (ii) the state representation as an argument.

More precisely, we define (for $\mathtt{x} : u$, $t \in \boldsymbol{Term}_{\mathtt{x},s}$, $s \in \boldsymbol{Stmt}_{\mathtt{x}}$, $\mathtt{a} : u$, $\mathtt{b} : v$ and $P \in \boldsymbol{Proc}_{\mathtt{a},\mathtt{b},\mathtt{c}}$) the functions

$$\begin{array}{rcl}
\boldsymbol{te}^A_{\mathtt{x},s,t} & : & A^u \to A_s \\
\boldsymbol{ae}^A_{\mathtt{x},S} & : & A^u \to A^u \\
\boldsymbol{rest}^A_{\mathtt{x},S} & : & A^u \to \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \\
\boldsymbol{notover}^A_{\mathtt{x},S} & : & A^u \times \mathbb{N} \to \mathbb{B} \\
\boldsymbol{state}^A_{\mathtt{x},S} & : & A^u \times \mathbb{N} \to A^u \\
\boldsymbol{comp}^A_{\mathtt{x},S} & : & A^u \times \mathbb{N} \to \mathbb{B} \times A^u
\end{array} \tag{4.4}$$

$$\begin{array}{rcl}
\boldsymbol{rem}^A_{\mathtt{x},S} & : & A^u \times \mathbb{N} \to \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \\
\boldsymbol{snap}^A_{\mathtt{x},S} & : & A^u \times \mathbb{N} \to (\mathbb{B} \times A^u) \times \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \\
\boldsymbol{se}^A_{\mathtt{x},S} & : & A^u \to A^u \\
\boldsymbol{pe}^A_{\mathtt{a},\mathtt{b},\mathtt{c},P} & : & A^u \to A^v
\end{array}$$

such that

$$\begin{array}{rcl}
\boldsymbol{te}^A_{\mathtt{x},s,t}(a) & = & \boldsymbol{te}^A_{\mathtt{x},s}(\ulcorner t \urcorner, a), \\
\boldsymbol{comp}^A_{\mathtt{x},S}(a,n) & = & \boldsymbol{comp}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, n),
\end{array}$$

and similarly for the other functions listed in (4.4). We then have:

Theorem 4.27.

(*a*) *The functions* $\boldsymbol{te}^A_{\mathtt{x},s,t}$ *and* $\boldsymbol{ae}^A_{\mathtt{x},S}$ *are* $\boldsymbol{While}$ *computable on* A. *The functions* $\boldsymbol{rest}^A_{\mathtt{x},S}$, $\boldsymbol{notover}^A_{\mathtt{x},S}$, $\boldsymbol{state}^A_{\mathtt{x},S}$, $\boldsymbol{comp}^A_{\mathtt{x},S}$, $\boldsymbol{rem}^A_{\mathtt{x},S}$ *and* $\boldsymbol{snap}^A_{\mathtt{x},S}$ *are* $\boldsymbol{While}$ *computable on* A^N. *The functions* $\boldsymbol{se}^A_{\mathtt{x},S}$ *and* $\boldsymbol{pe}^A_{\mathtt{a},\mathtt{b},\mathtt{c},P}$ *are* $\boldsymbol{While}^N$ *computable on* A.

(*b*) *Suppose* A *is* N*-standard. Then all the functions listed in (4.4) are* $\boldsymbol{While}$ *computable on* A.

Proof. For (*a*): computability of $\boldsymbol{te}^A_{\mathtt{x},s,t}$ is proved by structural induction on $t \in \boldsymbol{Term}_{\mathtt{x}}$. To prove computability of $\boldsymbol{rest}^A_{\mathtt{x},S}$ on A^N, put $S \equiv S_0; S_1$, where S_0 does *not* have the form $S'; S''$ (and '$;S_1$' may be empty), and rewrite the definition of $\boldsymbol{Rest}^A$ in section 3.5 as an explicit definition by cases, according to the different forms of S_0. For computability of $\boldsymbol{comp}^A_{\mathtt{x},S}$ on A^N, show that the family of functions $\langle \boldsymbol{comp}^A_{\mathtt{x},S'} | S' \in \boldsymbol{RemSet}(S)\rangle$ is definable by *simultaneous primitive recursion.* (Compare the definition of $\boldsymbol{Comp}^A$ in section 3.4.) Use the fact that this family is finite, by Proposition 3.51.

Part (*b*) follows immediately from (*a*). ∎

5 Notions of semicomputability

We want to generalise 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. There are many useful applications of these concepts, and they satisfy closure properties and 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 semicomputable set. In computability theory on the set $\mathbb{N}$ of natural numbers, the class of semicomputable sets is closed under taking projections, but this is not true in the general case of algebras, even with $\boldsymbol{While}^*$ computability. (A reason is the restricted form of computable local search available in our models of computation.) Projective semicomputability is strictly more powerful (and less algorithmic) than semicomputability.

In this section we will study the two notions of semicomputability and projective semicomputability in some detail. We will consider the invariance of the properties under homomorphisms. We will prove equivalences, such as

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

In the course of the section, we also consider extensions of the $\boldsymbol{While}$ language by non-deterministic constructs, including allowing:

(i) arbitrary initialisations of some auxiliary variables in programs;
(ii) random assignments in programs.

We prove that in these non-deterministic languages, semicomputability is equivalent to the corresponding notion of projective semicomputability. We also show an equivalence between projective semicomputability and

(iii) definability in a weak second-order language.

We characterise the semicomputable sets as the sets definable by some effective countable disjunction

$$\bigvee_{k=0}^{\infty} \boldsymbol{b}_k$$

of Boolean-valued terms. This result was first observed by E. Engeler. There are a number of attractive applications, e.g. in classifying the semicomputable sets over rings and fields, where Boolean terms can be replaced by polynomial identities; we consider this topic in section 6.

These concepts and results are developed for computations with the three languages based on the $\boldsymbol{While}$, $\boldsymbol{While}^N$ and $\boldsymbol{While}^*$ constructs; and their uniformity over classes of algebras is discussed.

We assume throughout this section that Σ is a standard signature, and A a standard Σ-algebra.

5.1 *While* semicomputability

Definition 5.1. The *halting set* of a procedure $P : u \to v$ on A is the relation

$$\boldsymbol{Halt}^A(P) =_{df} \{a \in A^u \mid P^A(a)\downarrow\}.$$

Now let R be a relation on A.

Definition 5.2.

(*a*) R is $\boldsymbol{While}$ *computable on* A if its characteristic function is.

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

(*c*) A family $R = \langle R_A \mid A \in \mathbb{K} \rangle$ of relations is $\boldsymbol{While}$ *semicomputable uniformly over* $\mathbb{K}$ if there is a $\boldsymbol{While}$ procedure P such that for all $A \in \mathbb{K}$, R_A is the halting set of P on A.

It follows from the definition that R is $\boldsymbol{While}$ semicomputable on A if, and only if, R is the domain of a $\boldsymbol{While}$ computable (partial) function on A.

Remark 5.3. As far as defining relations by procedures is concerned, we can ignore output variables. More precisely, if $R = \boldsymbol{Halt}^A(P)$, then we may assume that P has no output variables, since otherwise we can remove all output variables from P simply by reclassifying them as auxiliary variables. We will call any procedure without output variables a *relational procedure.*

Definition 5.4 (Relative $\boldsymbol{While}$ semicomputability). Given a tuple $g_1, \ldots, g_n$ of functions on A, a relation R on A is $\boldsymbol{While}$ *semicomputable in* $g_1, \ldots, g_n$ if it is the halting set on A of a $\boldsymbol{While}(g_1, \ldots, g_n)$ procedure, or (equivalently) the domain of a function $\boldsymbol{While}$ computable in $g_1, \ldots, g_n$ (cf. section 3.10).

Example 5.5.

(*a*) On the naturals $\mathcal{N}$ (Example 2.23(*b*)), the $\boldsymbol{While}$ semicomputable sets are precisely the recursively enumerable sets, and the $\boldsymbol{While}$ computable sets are precisely the recursive sets.

(*b*) Consider the standard algebra $\mathcal{R}$ of reals (Example 2.23(*c*)). The set of *naturals* (as a subset of $\mathbb{R}$) is $\boldsymbol{While}$ semicomputable on $\mathcal{R}$, 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 $\boldsymbol{While}$ semicomputable on $\mathcal{R}$. (*Exercise.*)

(*d*) However, the sets of naturals and integers are $\boldsymbol{While}$ *computable* on $\mathcal{R}^<$ (section 2.23(*d*)). (*Exercise.*)

(*e*) The set of *rationals* is $\boldsymbol{While}$ semicomputable on $\mathcal{R}$. (*Exercise.* Hint: Prove this first for $\mathcal{R}^N$.)

5.2 Merging two procedures: Closure theorems

In order to prove certain important results for $\boldsymbol{While}$ semicomputable sets, namely (*a*) closure under finite unions, and (*b*) Post's theorem, we need to develop an operation of *merging* two procedures, i.e., interleaving their steps to form a new procedure. In the context of N-standard structures, and assuming the TEP, this is a simple construction (as in the classical case over $\mathbb{N}$). In general, however, the merge construction is quite non-trivial, as we shall now see.

Lemma 5.6. *Given two relational* $\boldsymbol{While}(\Sigma)$ *procedures* P_1 *and* P_2*, of input type* u*, we can construct a* $\boldsymbol{While}(\Sigma)$ *procedure*

$$Q \equiv \boldsymbol{mg}(P_1, P_2): \ u \to \mathsf{bool},$$

the merge of P_1 *and* P_2*, with Boolean output values (written '1' and '2' for clarity) such that for all* $A \in \boldsymbol{StdAlg}(\Sigma)$ *and* $a \in A^u$*:*

$$\begin{array}{lcl} Q^A(a)\downarrow & \Longleftrightarrow & P_1^A(a)\downarrow \ \ or \ \ P_2^A(a)\downarrow, \\ Q^A(a)\downarrow 1 & \Longrightarrow & P_1^A(a)\downarrow, \\ Q^A(a)\downarrow 2 & \Longrightarrow & P_2^A(a)\downarrow. \end{array}$$

Proof. We may assume without loss of generality that

$$\begin{array}{lcl} P_1 & \equiv & \mathsf{proc\ in\ x\ aux\ z_1\ begin}\ S_1\ \ \mathsf{end} \\ P_2 & \equiv & \mathsf{proc\ in\ x\ aux\ z_2\ begin}\ S_2\ \ \mathsf{end} \end{array}$$

where $\mathtt{x} : u$ and $\mathsf{z}_1 \cap \mathsf{z}_2 = \emptyset$. We can construct a procedure

$$Q \equiv \mathsf{proc\ in\ x\ aux\ z_1, z_2, \ldots\ \ out\ which\ begin\ S\ \ end}$$

where $S \equiv \boldsymbol{mg}(S_1, S_2)$, the 'merge' of S_1 and S_2, and 'which' is a Boolean variable with values written as '1' and '2'. The operation $\boldsymbol{mg}(S_1, S_2)$ is actually defined for all pairs S_1, S_2 such that $\boldsymbol{var}(S_1) \cap \boldsymbol{var}(S_2) \subseteq \mathtt{x}$, and none of the $\mathtt{x}$ occur on the lhs of any assignment in S_1 or S_2. It has the semantic property that for all A, σ:

$$[\![\boldsymbol{mg}(S_1, S_2)]\!]^A \sigma \downarrow \ \Longleftrightarrow \ [\![S_1]\!]^A \sigma \downarrow \ \text{ or } \ [\![S_2]\!]^A \sigma \downarrow,$$

and if $[\![\boldsymbol{mg}(S_1, S_2)]\!]^A \sigma \downarrow \sigma'$ then

$$\begin{array}{l} \sigma'(\mathsf{which}) = 1 \implies [\![S_1]\!]\sigma \downarrow \text{ and } [\![\boldsymbol{mg}(S_1, S_2)]\!]^A \sigma \approx [\![S_1]\!]^A \sigma \ (\text{rel } \boldsymbol{var} S_1), \\ \sigma'(\mathsf{which}) = 2 \implies [\![S_2]\!]\sigma \downarrow \text{ and } [\![\boldsymbol{mg}(S_1, S_2)]\!]^A \sigma \approx [\![S_2]\!]^A \sigma \ (\text{rel } \boldsymbol{var} S_2). \end{array}$$

The definition of $\boldsymbol{mg}(S_1, S_2)$ is by course of values recursion on the sum of $\boldsymbol{compl}(S_1)$ and $\boldsymbol{compl}(S_2)$. Details are left as a (challenging) exercise. (*Hint:* The tricky case is when both S_1 and S_2 have the form $S_i \equiv \mathsf{while}\ b_i\ \mathsf{do}\ S_i^0\ \mathsf{od}; S_i'$ $(i = 1, 2)$, where '$;S_i'$' may be empty). ■

Remark 5.7. The construction of $\boldsymbol{mg}(P_1, P_2)$ for A is much simpler if we can assume that (*i*) A is N-standard, and (*ii*) A has the TEP. In that case, by Theorem 4.3, the computation step representing function $\boldsymbol{comp}_{\mathtt{x}}^A$ is $\boldsymbol{While}$ computable on A. Using this, we can construct a $\boldsymbol{While}$ procedure which interleaves the computation steps of S_1 and S_2, tests at each step whether either computation has halted, and (accordingly) gives an output of 1 or 2.

Theorem 5.8 (Closure of *While* semicomputability under union and intersection). *The union and intersection of two* ***While*** *semicomputable relations of the same type are again* ***While*** *semicomputable. Moreover, this result is uniformly effective over* $\boldsymbol{StdAlg}(\Sigma)$*, in the sense that given two* ***While*** *procedures* P_1 *and* P_2 *of the same input type* u*, there are two other procedures* $P_{1\cup 2}$ *and* $P_{1\cap 2}$ *of input type* u*, effectively constructible from* P_1 *and* P_2*, such that on any standard* Σ*-algebra* A*,*

$$(a) \quad \boldsymbol{Halt}^A(P_{1\cup 2}) \;=\; \boldsymbol{Halt}^A(P_1) \cup \boldsymbol{Halt}^A(P_2);$$
$$(b) \quad \boldsymbol{Halt}^A(P_{1\cap 2}) \;=\; \boldsymbol{Halt}^A(P_1) \cap \boldsymbol{Halt}^A(P_2).$$

Proof. Suppose again without loss of generality that

$$\begin{array}{lcl} P_1 & \equiv & \mathsf{proc\ in\ x\ aux\ z_1\ begin}\ S_1\ \mathsf{end} \\ P_2 & \equiv & \mathsf{proc\ in\ x\ aux\ z_2\ begin}\ S_2\ \mathsf{end} \end{array}$$

where $\mathsf{z}_1 \cap \mathsf{z}_2 = \emptyset$.

(a) $P_{1\cup 2}$ can be defined as $\boldsymbol{mg}(P_1, P_2)$, as in Lemma 5.6. (We ignore its output here.)

(b) $P_{1\cap 2}$ can be defined, more simply, as in the classical case:

$$P_{1\cap 2} \equiv \mathsf{proc\ in\ x\ aux\ z_1,\ z_2\ begin}\ S_1; S_2\ \mathsf{end}.$$ ■

If R is a relation on A of type u, we write the *complement of* A as

$$R^c \;=_{df}\; A^u \backslash R.$$

Theorem 5.9 (Post's theorem for *While* semicomputability). *For any relation* R *on* A

$$R \text{ is } \boldsymbol{While} \text{ computable} \quad \Longleftrightarrow \quad R \text{ and } R^c \text{ are } \boldsymbol{While} \text{ semicomputable.}$$

Moreover, this equivalence is uniformly effective over $\boldsymbol{StdAlg}(\Sigma)$*, i.e., (considering the reverse direction) given any procedures* P_1 *and* P_2 *of the same input type* u*, there is a procedure* $P_3 : u \to \mathsf{bool}$*, effectively constructible from* P_1 *and* P_2*, such that on any standard* Σ*-algebra* A*, if the halting sets of* P_1 *and* P_2 *are* R_A *and* R_A^c *respectively, then* P_3 *computes the characteristic function of* R_A*.*

Proof.

($\Longrightarrow$) This follows, as in the classical case, by modifying a procedure which computes the characteristic function of R into two procedures which have R and R^c respectively as halting sets.

($\Longleftarrow$) Again, we can just take $P_3 \equiv \boldsymbol{mg}(P_1, P_2)$, as in Lemma 5.6. ■

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

Theorem 5.10 (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* $\{x \in A^u \mid \exists n \in \mathbb{N} R(x,n)\}$*.*

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). In other words, the algorithm proceeds in *stages* (1,2, ...), given by the iterations of a 'while' loop. At *stage* n, test whether P halts in at most n steps, with input (x, k), for some $k < n$. This can be done by computing $\boldsymbol{notover}^A_{\mathtt{x},S}(x, k)$ for all $k < n$ (see section 4.13). The algorithm halts if and when we get an output ff. ∎

Note that if A has the TEP, we could just as well use the computation step representing function $\boldsymbol{comp}^A_{\mathtt{x}}$ in the above proof instead of $\boldsymbol{comp}^A_{\mathtt{x},S}$. (Cf. Remark 5.7.)

We can generalise Theorem 5.10 to the case of an A_s-projection for any *minimal carrier* A_s (recall Definition 2.17), provided A has the TEP:

Corollary 5.11 (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 $\{x \in A^u \mid \exists y \in A_s\, R(x, y)\}$.*

Proof. Recall that by Corollary 4.10, there is a total ***While*** computable enumeration of A_s,

$$\boldsymbol{enum}^A_s \colon\ \mathbb{N}\ \to\ A_s.$$

So for all $x \in A^u$,

$$\exists y \in A_s R(x, y) \iff \exists n R(x, \boldsymbol{enum}^A_s(n)) \iff \exists n R'(x, n)$$

where (as is easily seen) the relation

$$R'(x, n)\ =_{df}\ R(x, \boldsymbol{enum}^A_s(n))$$

is ***While*** semicomputable. The result follows from Theorem 5.10. ∎

Note that there are *relativised versions* (cf. Definition 5.4) of all the results of this subsection so far.

Discussion 5.12 (Minimality and search). Corollary 5.11 is a many-sorted version of (part of) Theorem 2.4 of [Friedman, 1971a], cited in [Shepherdson, 1985]. 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.

(Cf.Theorem 2.4 of Friedman [1971a].) 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.

(Cf. Theorem 2.6 of Friedman [1971a].)

5.3 Projective *While* semicomputability: semicomputability with search

We introduce and compare two new notions of semicomputability: (1°) projective ***While*** semicomputability and (2°) ***While*** semicomputability with search. First, for (1°):

Definition 5.13.

(a) R is *projectively* ***While*** *computable* on A if, and only if, R is a *projection* of a ***While*** computable relation on A (see Definition 2.9(d)).

(b) R is *projective* ***While*** *semicomputable* on A if, and only if, R is a *projection* of a ***While*** semicomputable relation on A.

The notions of *uniform projective* ***While*** *computability and semicomputability over* $\mathbb{K}$ of a family of relations, are defined analogously (cf. Definition 5.2(c)).

Note that projective ***While*** semicomputability is, in general, weaker than ***While*** semicomputability. Example 6.15 will show this, using Engeler's lemma.

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 5.10. Further, we have from Corollary 5.11:

Proposition 5.14. *Suppose A is N-standard and minimal and has the TEP. Then on A*

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

Now, for (2°), we introduce a new feature: definability with the possibility of *arbitrary initialisation of search variables.* For this, we define a new type of procedure.

Definition 5.15. A *search procedure* has the form

$$P_{srch} \equiv \textsf{proc in a out b aux c srch d begin } S \textsf{ end}, \tag{5.1}$$

with *search variables* d as well as input, output and auxiliary variables, and with the stipulations (compare section 3.1(d)):

- a, b, c and d each consist of distinct variables, and they are pairwise disjoint;
- every variable in S is included among a, b, c or d;
- the *input and search variables* a,d can occur only on the right-hand side of an assignment in S;
- (*initialisation condition*): S has the form $S_{init}; S'$, where S_{init} consists of an *initialisation* of the output and auxiliary variables, but *not of the search variables* d.

Again, we may assume in (5.1) that P_{srch} has no output variables, i.e., that `b` is empty. (See Remark 5.3.)

Definition 5.16. The *halting set* of a search procedure as in (5.1) on A (assuming $\mathtt{a} : u$ and $\mathtt{d} : w$) is the set

$$\boldsymbol{Halt}^A(P_{srch}) \ =_{df} \ \{a \in A^u \mid \text{for some } \sigma \text{ with } \sigma[\mathtt{a}] = a,\ [\![S]\!]\sigma \downarrow\}.$$

In other words, it is the set of tuples $a \in A^u$ such that when `a` is initialised to a, then *for some (non-deterministic) initialisation of* `d`, *S halts.*

Note that this reduces to Definition 5.1 when P_{srch} has no search variables.

Now let R be a relation on A.

Definition 5.17. R is ***While*** *semicomputable with search on A* if R is the halting set on A of some ***While*** search procedure.

Again, the notion of *uniform* ***While*** *semicomputability with search over* $\mathbb{K}$ of a family of relations, is defined analogously.

Now we compare the two notions introduced above.

Theorem 5.18.

(a) *R is* ***While*** *semicomputable with search on A $\Longleftrightarrow$ R is projectively* ***While*** *semicomputable on A.*

(b) *This equivalence is uniform over* $\boldsymbol{StdAlg}(\Sigma)$*, in the sense that there are effective transformations $P_{srch} \mapsto P$ and $P \mapsto P_{srch}$ between search procedures P_{srch} and ordinary procedures P, such that for all $A \in \boldsymbol{StdAlg}(\Sigma)$, $\boldsymbol{Halt}^A(P_{srch})$ is a projection of $\boldsymbol{Halt}^A P$.*

Proof. The equivalence follows easily from the definitions. Suppose R is the halting set on A by a search procedure P_{srch} with input variables $\mathtt{a} : u$ and search variables $\mathtt{d} : w$. Let P be the procedure formed from P_{srch} simply by relabelling `d` as additional input variables. (So the input type of P is $u \times w$.) Then R is the projection onto A^u of the halting set of P. The opposite direction is just as easy. ∎

5.4 $\boldsymbol{While}^N$ semicomputability

Let R be a relation on A.

Definition 5.19.

(a) R is $\boldsymbol{While}^N$ *computable* on A if its characteristic function is (section 3.12).

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

Again, we may assume that P has no output variables. (See Remark 5.3.)

From Proposition 3.38 we have:

Proposition 5.20. *If A is N-standard, then $\boldsymbol{While}^N$ semicomputability on A coincides with $\boldsymbol{While}$ semicomputability on A.*

Theorem 5.21 (Closure of $\boldsymbol{While}^N$ semicomputability under union and intersection). *The union and intersection of two $\boldsymbol{While}^N$ semicomputable relations of the same input type are again $\boldsymbol{While}^N$ semicomputable, uniformly over $\boldsymbol{StdAlg}(\Sigma)$.*

Proof. From Theorem 5.8, applied to A^N. ∎

Theorem 5.22 (Post's theorem for $\boldsymbol{While}^N$ semicomputability). *For any relation R on A*

$$R \text{ is } \boldsymbol{While}^N \text{ computable} \iff$$
$$R \text{ and } R^c \text{ are } \boldsymbol{While}^N \text{ semicomputable,}$$

uniformly for $A \in \boldsymbol{StdAlg}(\Sigma)$.

Proof. From Theorem 5.9, applied to A^N. ∎

Note that if A has the TEP, then the construction of a 'merged' $\boldsymbol{While}^N$ procedure $\boldsymbol{mg}(P_1, P_2)$ from two $\boldsymbol{While}^N$ procedures P_1 and P_2, used in the above two theorems, is much simpler than the construction given in Lemma 5.6 (cf. Remark 5.7).

Also Theorem 5.10 and Corollary 5.11 can respectively be restated for $\boldsymbol{While}^N$ semicomputability:

Theorem 5.23 (Closure of $\boldsymbol{While}^N$ semicomputability under $\mathbb{N}$-projections). *Suppose $R \subseteq A^{u\times \mathsf{nat}}$, where $u \in \boldsymbol{ProdType}(\Sigma)$, and R is $\boldsymbol{While}$ semicomputable on A^N. Then its $\mathbb{N}$-projection $\{x \mid \exists n \in \mathbb{N} R(x,n)\}$ is $\boldsymbol{While}^N$ semicomputable on A.*

Corollary 5.24 (Closure of $\boldsymbol{While}^N$ semicomputability under projections off minimal carriers). *Suppose A has the TEP. Let A_s be a minimal carrier of A. If $R \subseteq A^{u\times s}$ is $\boldsymbol{While}^N$ semicomputable on A, then so is its projection $\{x \in A^u \mid \exists y \in A_s\, R(x,y)\}$.*

Example 5.25.

(*a*) ($\boldsymbol{While}^N$ *semicomputability of the subalgebra relation.*) For a standard signature Σ, equality sort s, product type u and standard Σ-algebra A, the *subalgebra relation*

$$\{(y,x) \in A_s \times A^u \mid y \in \langle x\rangle_s^A\}$$

(where $\langle x\rangle_s^A$ is the carrier of sort s of the subalgebra of A generated by $x \in A^u$) is $\boldsymbol{While}$ semicomputable (on A^N) in the term evaluation representing function $\boldsymbol{te}^A_{\mathtt{x},s}$, where $\mathtt{x} : u$ (section 4.3). To show this, we note that

$$y \in \langle x\rangle_s^A \iff \exists z \in \mathbb{N}(\boldsymbol{te}^A_{\mathtt{x},s}(z,x) = y)$$

(cf. Remark 2.16) and apply (a relativised version of) Theorem 5.23. Hence if A has the TEP, this relation is $\boldsymbol{While}^N$ semicomputable on A.

(*b*) On the standard group $\mathcal{G}$ (Examaple 2.23(*g*)) the set $\{g \in G \mid \exists n(g^n = 1)\}$ of *elements of finite order* is $\boldsymbol{While}^N$ semicomputable, being the domain of the order function on $\mathcal{G}^N$, which is $\boldsymbol{While}$ computable on $\mathcal{G}^N$ (Example 3.14(*b*)). In fact, this set is even $\boldsymbol{While}$ semicomputable on $\mathcal{G}$. (*Exercise.*)

(*c*) More generally, for any Σ-product type u and Σ-equality sort s, the set

$$\{x \in A^u \mid \langle x \rangle_s^A \text{ is finite}\}$$

is $\boldsymbol{While}^N$ semicomputable in $\boldsymbol{te}^A_{\mathtt{x},s}$. This follows from the fact that it is the domain of the function $\boldsymbol{ord}^A_{u,s}$, which is $\boldsymbol{While}$ computable in $\boldsymbol{te}^A_{\mathtt{x},s}$ on A^N (Example 4.23). Hence if A has the TEP, then this set is $\boldsymbol{While}^N$ semicomputable on A.

5.5 Projective $\boldsymbol{While}^N$ semicomputability

Let R be a relation on A.

Definition 5.26.

(*a*) R is *projectively* $\boldsymbol{While}^N$ *computable* on A if, and only if, R is a *projection* of a $\boldsymbol{While}(\Sigma^N)$ computable relation on A^N.

(*b*) R is *projectively* $\boldsymbol{While}^N$ *semicomputable* on A if R is a *projection* of a $\boldsymbol{While}(\Sigma^N)$ semicomputable relation on A^N.

Proposition 5.14 can be restated for $\boldsymbol{While}^N$ semicomputability:

Proposition 5.27. *Suppose A is a minimal and has the TEP. Then on A*

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

Definition 5.28. R is $\boldsymbol{While}^N$ *semicomputable with search* on A if R is the halting set of a $\boldsymbol{While}(\Sigma^N)$ search procedure on A^N.

Note that the $\boldsymbol{While}(\Sigma^N)$ search procedure in this definition has *simple input variables.* However, the auxiliary, search and output variables may be simple or nat.

Again, the following equivalence follows easily from the definitions. (Cf. Theorem 5.18.)

Theorem 5.29.

(a) R is $\boldsymbol{While}^N$ *semicomputable with search on A* $\Longleftrightarrow$ *R is projectively* $\boldsymbol{While}^N$ *semicomputable on A.*

(b) This equivalence is uniform over $\boldsymbol{StdAlg}(\Sigma)$.

5.6 Solvability of the halting problem

The classical question of the solvability of the halting problem ([Davis, 1958]; implicit in [Turing, 1936]) applies to the algebra of naturals $\mathcal{N}_0$ (Example 2.5(*a*)) or its standardised version $\mathcal{N}$ (Example 2.23(*b*)). We want to generalise this question to any standard signature Σ and standard Σ-algebra A. We will find that the problem can only be formulated in N-standard algebras.

Definition 5.30. Suppose $\Sigma \subseteq \Sigma'$, where Σ is standard and Σ' is N-standard, and suppose A is a standard Σ-algebra and A' is a Σ'-expansion of A. Then we say *the halting problem (HP) for* $\boldsymbol{While}(\Sigma)$ *computation on* A *is* $\boldsymbol{While}(\Sigma')$*-solvable on* A', or *the HP for* $\boldsymbol{While}(A)$ *is solvable in* $\boldsymbol{While}(A')$, if for every Σ-product type u there is a $\boldsymbol{While}(\Sigma')$ procedure

$$\mathsf{HaltTest}_u\colon\ \mathsf{nat}\times u \to \mathsf{bool},$$

such that $\mathsf{HaltTest}_u^A$ is total, and for every $\boldsymbol{While}(\Sigma)$ procedure P of input type u, and all $a \in A^u$,

$$\mathsf{HaltTest}_u^A(\ulcorner P\urcorner, a) \;=\; \begin{cases} \mathtt{tt} & \text{if } P^A(a)\downarrow \\ \mathtt{ff} & \text{otherwise.} \end{cases}$$

The procedure $\mathsf{HaltTest}_u$ in the above definition is called a *universal halting test for type* u *on* $\boldsymbol{While}(A)$.

Proposition 5.31. *If the HP for* $\boldsymbol{While}(A)$ *is solvable in* $\boldsymbol{While}(A')$, *then every* $\boldsymbol{While}(\Sigma)$ *semicomputable set in* A *is* $\boldsymbol{While}(\Sigma')$ *computable in* A'.

Proof. Simple exercise. ∎

The two typical situations are:

(i) $\Sigma' = \Sigma^N$ and $A' = A^N$;

(ii) Σ is N-standard, $\Sigma' = \Sigma$ and $A' = A$.

Example 5.32. For the algebra $\mathcal{N}$ (Example 2.23(b)), the HP is not solvable in $\boldsymbol{While}(\mathcal{N})$. This is a version of the classical result of [Kleene, 1952].

Theorem 5.34 makes use of the concept of local finiteness (section 4.12). In preparation for it, we define *uniform* (in $\mathtt{x}$) representations of the *statement remainder function* and the *snapshot function* (cf. section 4.10). Namely, given a product type $u = s_1 \times \ldots \times s_m$ and a u-tuple of variables $\mathsf{a} : u$ (which we think of as input variables), we define

$$\begin{aligned} \boldsymbol{remu}_{\mathsf{a}}^A\colon &\quad \ulcorner \boldsymbol{VarTup}\urcorner \times \ulcorner \boldsymbol{Stmt}\urcorner \times A^u \times \mathbb{N} \to \ulcorner \boldsymbol{Stmt}\urcorner \\ \boldsymbol{snapu}_{\mathsf{a}}^A\colon &\quad \ulcorner \boldsymbol{VarTup}\urcorner \times \ulcorner \boldsymbol{Stmt}\urcorner \times A^u \times \mathbb{N} \to \\ &\quad\quad (\mathbb{B} \times \ulcorner \boldsymbol{TermTup}_{\mathsf{a}}\urcorner) \times \ulcorner \boldsymbol{Stmt}\urcorner \end{aligned}$$

as follows: for any product type w extending u, i.e.,
$w = s_1 \times \ldots \times s_p$ for some $p \geq m$, for any $\mathtt{x} : w$ extending a (i.e., $\mathtt{x} \equiv \mathsf{a}, \mathtt{x}_{s_{m+1}}, \ldots, \mathtt{x}_{s_p}$), and for any $S \in \boldsymbol{Stmt}_{\mathtt{x}}$, $a \in A^u$ and $n \in \mathbb{N}$,

$$\boldsymbol{remu}_{\mathsf{a}}^A(\ulcorner x\urcorner, \ulcorner S\urcorner, a, n,) \;=\; \boldsymbol{rem}_{\mathtt{x}}^A(\ulcorner S\urcorner, a, \boldsymbol{\delta}_A), n)$$

where $\boldsymbol{\delta}_A$ is the default value of type $s_{m+1} \times \ldots \times s_p$, and

$$\begin{aligned} \boldsymbol{snapu}^A_{\mathsf{a}}(\ulcorner \mathtt{x} \urcorner, \ulcorner S \urcorner, a, n) &= (\boldsymbol{compu}^A_{\mathsf{a}}(\ulcorner \mathtt{x} \urcorner, \ulcorner S \urcorner, a, n), \\ &\qquad \boldsymbol{remu}^A_{\mathsf{a}}(\ulcorner \mathtt{x} \urcorner, \ulcorner S \urcorner, a, n)) \\ &= ((b_n, \ulcorner t_n \urcorner), \ulcorner S_n \urcorner) \end{aligned} \tag{5.2}$$

where

$$\begin{aligned} b_n &= \boldsymbol{notoveru}^A_{\mathtt{x}}(\ulcorner \mathtt{x} \urcorner, \ulcorner S \urcorner, a, n) \\ \ulcorner t_n \urcorner &= \boldsymbol{stateu}^A_{\mathsf{a}}(\ulcorner \mathtt{x} \urcorner, \ulcorner S \urcorner, a, n) \\ \ulcorner S_n \urcorner &= \boldsymbol{remu}^A_{\mathsf{a}}(\ulcorner \mathtt{x} \urcorner, \ulcorner S \urcorner, a, n). \end{aligned}$$

Lemma 5.33. *The function* $\boldsymbol{snapu}^A_{\mathsf{a}}$*, and its components* $\boldsymbol{compu}^A_{\mathsf{a}}$ *and* $\boldsymbol{remu}^A_{\mathsf{a}}$*, are* $\boldsymbol{While}$ *computable in* $\langle \boldsymbol{te}^A_{\mathsf{a},s} \mid s \in \boldsymbol{Sort}(\Sigma) \rangle$ *on* A^N*, uniformly for* $A \in \boldsymbol{StdAlg}(\Sigma)$*.*

Proof. Similar to Lemma 4.20. ∎

Theorem 5.34. *Suppose*

(1°) *Σ has equality at all sorts,*

(2°) *A has the TEP, and*

(3°) *A is locally finite.*

Then the HP for $\boldsymbol{While}(A)$ *is solvable in* $\boldsymbol{While}(A^N)$*.*

Proof. Given a Σ-product type $u = s_1 \times \ldots \times s_m$, we will give an informal description of an algorithm over A^N for a universal halting test for type u on $\boldsymbol{While}(A)$. (Compare the construction of the universal procedure in section 4.8.)

With input $(\ulcorner P \urcorner, a)$, where P has input type u, and $a \in A^u$, suppose

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

where $\mathsf{a} : u$. Put $\mathtt{x} \equiv \mathsf{a}, \mathsf{b}, \mathsf{c}$. Then for $n = 0, 1, 2, \ldots$

$$\boldsymbol{snapu}^A_{\mathsf{a}}(\ulcorner \mathtt{x} \urcorner, \ulcorner S \urcorner, a, n) = ((b_n, \ulcorner t_n \urcorner), \ulcorner S_n \urcorner)$$

as in (5.2) above. Now put

$$x_n = \boldsymbol{te}^A_{\mathsf{a},w}(\ulcorner t_n \urcorner, a) = \boldsymbol{state}^A_{\mathtt{x}}(\ulcorner S \urcorner, a, \boldsymbol{\delta}_A), n).$$

By Lemma 5.33 and assumption (2°), $\boldsymbol{snapu}^A_{\mathsf{a}}$ is $\boldsymbol{While}$ computable on A^N. In other words, its three components b_n, $\ulcorner t_n \urcorner$ and $\ulcorner S_n \urcorner$ (as functions of n) are $\boldsymbol{While}$ computable on A^N. Hence by assumption (2°) again, the components of the w-tuple x_n are $\boldsymbol{While}$ computable on A^N (as functions of n).

Now for $n = 0, 1, 2, \ldots,$ evaluate the w-tuple x_n, and compare it (componentwise) to x_m for all $m < n$ (which is possible by assumption (1°)), until *either*

(a) $b_n = \mathsf{ff}$, which means that the computation of S (and hence of P at a) terminates; *or* (*b*) for some distinct m and n, $b_m = b_n = \mathtt{tt}$, $x_m = x_n$ and $\ulcorner S_m \urcorner = \ulcorner S_n \urcorner$, which means that the computation of S never terminates, by Proposition 3.34.

Exactly one of these two cases must happen, by Theorem 4.26 and assumption (3°). In case (*a*) halt with output $\mathtt{tt}$, and in case (*b*) halt with output $\mathtt{ff}$. ∎

Note that Assumption (1°) can be weakened to:

(1°′) Equality on A_s is $\boldsymbol{While}^N$ computable, for all Σ-sorts s.

Also, for all u, the above construction of $\mathsf{HaltTest}_u$ is *uniform* over Σ in the following sense: t'here is a *relative* $\boldsymbol{While}(\Sigma^N)$ procedure $H_u : \mathsf{nat} \times u \to \mathsf{bool}$ containing oracle procedure calls $\langle g_s \mid s \in \boldsymbol{Sort}(\Sigma)\rangle$ and $\langle h_s \mid s \in \boldsymbol{Sort}(\Sigma)\rangle$ with $g_s : s^2 \to \mathsf{bool}$ and $h_s : \mathsf{nat} \times u \to s$, such that for any $A \in \boldsymbol{StdAlg}(\Sigma)$, if A is *locally finite*, then, interpreting g_s and h_s as eq_s^A and $\boldsymbol{te}_{\mathtt{x},s}^A$ respectively on A (where $\mathtt{x} : u$), H_u is a universal halting test for type u on A. (Cf. Remark 4.16(*a*).)

Corollary 5.35. *Suppose*

(1°) Σ has equality at all sorts,
(2°) A has the TEP,
(3°) A is locally finite, and
(4°) A is N-standard.

Then the HP for $\boldsymbol{While}(A)$ is solvable in $\boldsymbol{While}(A)$.

Example 5.36 (A set which is $\boldsymbol{While}^N$ but not $\boldsymbol{While}$ semicomputable). The above theory allows us to produce an example to distinguish between $\boldsymbol{While}$ and $\boldsymbol{While}^N$ semicomputability. Let A be the algebra $\mathcal{N}^-$ (Example 4.25). We present an outline of the argument. Check each of the following points in turn.

(*i*) In A^N there is a computable bijection $(n \mapsto n)$ from $\mathbb{N}^-$ to $\mathbb{N}$.

(*ii*) Hence the $\boldsymbol{While}^N$ computable subsets of $\mathbb{N}^-$ are precisely the *recursive* sets of natural numbers (cf. Remark 3.43(*d*)).

(*iii*) Similarly the $\boldsymbol{While}^N$ semicomputable subsets of $\mathbb{N}^-$ are precisely the *recursively enumerable* sets of natural numbers (cf. Example 5.5).

(*iv*) Since $\mathcal{N}^-$ is locally finite (4.25) and has the TEP, the HP for $\boldsymbol{While}(A)$ is solvable in $\boldsymbol{While}(A^N)$. Therefore, by Proposition 5.31, every $\boldsymbol{While}$ semicomputable subset of $\mathbb{N}^-$ is $\boldsymbol{While}^N$ computable, and hence recursive.

The result follows from (*iii*) and (*iv*).

The same algebra, $\mathcal{N}^-$, can be used to distinguish between $\boldsymbol{While}$ and $\boldsymbol{While}^N$ computable functions. (*Exercise. Hint*: There is a universal $\boldsymbol{While}^N(\mathcal{N}^-)$ procedure for all total $\boldsymbol{While}(\mathcal{N}^-)$ functions of type $\mathsf{nat}^- \to \mathsf{nat}^-$.)

5.7 $\boldsymbol{While}^*$ semicomputability

Let R be a relation on A.

Definition 5.37.

(*a*) R is $\boldsymbol{While}^*$ *computable* on A if, and only if, its characteristic function is.

(*b*) R is $\boldsymbol{While}^*$ *semicomputable* if, and only if, it is the halting set of some $\boldsymbol{While}$ procedure P on A^*.

Again, we may assume that P has no output variables. (See Remark 5.3.)

From Proposition 3.45 we have:

Proposition 5.38. *On A^*, $\boldsymbol{While}^*$ semicomputability coincides with $\boldsymbol{While}$ semicomputability.*

Theorem 5.39 (Closure of $\boldsymbol{While}^*$ semicomputability under union and intersection). *The union and intersection of two $\boldsymbol{While}^*$ semicomputable relations of the same type are again $\boldsymbol{While}^*$ semicomputable, uniformly over $\boldsymbol{StdAlg}(\Sigma)$.*

Proof. From Theorem 5.8, applied to A^* ∎

Theorem 5.40 (Post's theorem for $\boldsymbol{While}^*$ semicomputability). *For any relation R on A*

$$R \text{ is } \boldsymbol{While}^* \text{ computable} \iff \text{both } R \text{ and } R^c \text{ are } \boldsymbol{While}^* \text{ semicomputable,}$$

uniformly for $A \in \boldsymbol{StdAlg}(\Sigma)$.

Proof. From Theorem 5.9, applied to A^*. ∎

Note that if A has the TEP, then the construction of a 'merged' $\boldsymbol{While}^N$ procedure $\boldsymbol{mg}(P_1, P_2)$ from two $\boldsymbol{While}^N$ procedures P_1 and P_2, used in the above two theorems, is much simpler than the construction given in Lemma 5.6 (cf. Remark 5.7).

Note that since A^* has the TEP for all $A \in \boldsymbol{StdAlg}(\Sigma)$, there is a uniform construction of a 'merged' $\boldsymbol{While}^*$ procedure $\boldsymbol{mg}(P_1, P_2)$ from two $\boldsymbol{While}^*$ procedures P_1 and P_2, used in the above two theorems, which is much simpler than the construction given in Lemma 5.6 (cf. Remark 5.7).

Also Theorem 5.10 (and 5.23) and Corollary 5.11 (and 5.24) can be restated for $\boldsymbol{While}^*$ semicomputability:

Theorem 5.41. *Suppose $R \subseteq A^{u \times \mathsf{nat}}$, where $u \in \boldsymbol{ProdType}(\Sigma)$, and R is $\boldsymbol{While}^*$ semicomputable on A^N. Then its $\mathbb{N}$-projection $\{x \mid \exists n \in \mathbb{N} R(x,n)\}$ is $\boldsymbol{While}^*$ semicomputable on A.*

Corollary 5.42 (Closure of $\boldsymbol{While}^*$ semicomputability under projections off minimal carriers). *Let A_s be a minimal carrier of A, and let $u \in \boldsymbol{ProdType}(\Sigma)$.*

(*a*) *If* $R \subseteq A^{u \times s}$ *is* $\boldsymbol{While}^*$ *semicomputable on* A*, then so is its projection* $\{x \in A^u \mid \exists y \in A_s\, R(x,y)\}$.

(*b*) *If* $R \subseteq A^{u \times s^*}$ *is* $\boldsymbol{While}$ *semicomputable on* A^**, then its projection* $\{x \in Au \mid \exists y^* \in A_s^*\, R(x,y^*)\}$ *is* $\boldsymbol{While}^*$ *semicomputable on* A.

Proof. In (b) we use the fact that if A_s is minimal in A, then A_s^* is minimal in A^*. (*Exercise.*) ∎

Remark 5.43. Unlike the case with Corollaries 5.11 and 5.24, we do not have to assume the TEP here, since the term evaluation representing function is always $\boldsymbol{While}^*$ computable.

Example 5.44. The *subalgebra relation* is $\boldsymbol{While}^*$ semicomputable on A. This follows from its $\boldsymbol{While}^N$ semicomputability in term evaluation (Example 5.25(a)), and $\boldsymbol{While}^*$ computability of the latter (Corollary 4.7.)

The *semicomputability equivalence theorem*, which we prove later (Theorem 5.61), states that for algebras with the TEP, $\boldsymbol{While}^*$ semicomputability coincides with $\boldsymbol{While}^N$ semicomputability.

5.8 Projective $\boldsymbol{While}^*$ semicomputability

Let R be a relation on A.

Definition 5.45.

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

(b) R is *projectively* $\boldsymbol{While}^*$ *semicomputable* on A if R is a *projection* of a $\boldsymbol{While}(\Sigma^*)$ semicomputable relation on A^*.

Proposition 5.14 (or 5.27) can be restated for $\boldsymbol{While}^*$ semicomputability:

Proposition 5.46. *Suppose* A *is a minimal. Then on* A

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

Note again that the TEP does not have to be assumed here (cf. Remark 5.43). Also we are using the fact that if A is minimal then so is A^*.

Definition 5.47. R is $\boldsymbol{While}^*$ *semicomputable with search* on A if R is the halting set of a $\boldsymbol{While}(\Sigma^*)$ search procedure on A^*.

Note that the $\boldsymbol{While}(\Sigma^*)$ search procedure in this definition has *simple input variables.* However the auxiliary, search and output variables may be simple, nat or starred.

Again, we have (cf. Theorems 5.18 and 5.29):

Theorem 5.48.

(*a*) R *is* $\boldsymbol{While}^*$ *semicomputable with search on* $A \iff R$ *is projectively* $\boldsymbol{While}^*$ *semicomputable on* A.

(*b*) *This equivalence is uniform over* $\boldsymbol{StdAlg}(\Sigma)$.

Example 5.49. 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 $\mathbb{N}$ (cf. 5.5(*a*)).

In general, however, projective $\boldsymbol{While}^*$ semicomputability is strictly stronger than projective $\boldsymbol{While}$ or $\boldsymbol{While}^N$ 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 in section 6.4.

We do, however, have the following equivalence:

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

This is the *projective equivalence theorem*, which will be proved in section 5.14.

Projective $\boldsymbol{While}^*$ semicomputability is the model of *specifiability* which will be the basis for a second generalised Church–Turing thesis (section 8.9).

5.9 Homomorphism invariance for semicomputable sets

For a Σ-homomorphism $\phi : A \to B$ and a relation $R : u$ on A, we write

$$\phi[R] \;=_{df}\; \{\phi(x) \mid x \in R\}$$

which is a relation of type u on B.

Theorem 5.50 (Epimorphism invariance for halting sets). *For any Σ-epimorphism $\phi : A \to B$,*

$$\phi[\boldsymbol{Halt}^A(P)] \;=\; \boldsymbol{Halt}^B(P).$$

Proof. From Theorem 3.24. ∎

Notice that the above result holds for a given procedure P, and *any* epimorphism $\phi : A \to B$. In particular, taking the case $B = A$, we obtain:

Corollary 5.51 (Automorphism invariance for semicomputability).

(a) If R is $\boldsymbol{While}$ semicomputable on A, then for any Σ-automorphism ϕ of A, $\phi[R] = R$.

(b) Similarly for $\boldsymbol{While}^*$ semicomputable sets.

Corollary 5.52 (Automorphism invariance for projective semicomputability).

(a) If R is projectively $\boldsymbol{While}$ semicomputable on A, then for any Σ-automorphism ϕ of A, $\phi[R] = R$.

(b) Similarly for projectively $\boldsymbol{While}^*$ semicomputable sets.

Example 5.53. In the algebra $\mathcal{C}^-$ of complex numbers (Example 2.23(e) *without the constant* i, the singleton set $\{i\}$ is not ***While*** semicomputable, or even projectively ***While***** semicomputable. This is because there is an automorphism of $\mathcal{C}^-$ with itself which maps i to $-i$. However the set $\{-i, i\}$ is ***While*** semicomputable, and in fact computable, in $\mathcal{C}^-$, by the procedure

```
proc in x:complex out b:bool
    begin
        b:= x×x= -1
    end.
```

5.10 The computation tree of a *While* statement

We will define, for any ***While*** statement S over Σ, and any tuple of distinct program variables $\mathtt{x} \equiv \mathtt{x}_1, \ldots, \mathtt{x}_n$ of type $u = s_1 \times \ldots \times s_n$ such that $\boldsymbol{var}(S) \subseteq \mathtt{x}$, the *computation tree* $\mathcal{T}[S, \mathtt{x}]$, which is like an 'unfolded flow chart' of S.

The root of the tree $\mathcal{T}[S, \mathtt{x}]$ is labelled 's' (for 'start'), and the leaves are labelled 'e' (for 'end'). The internal nodes are labelled with assignment statements and Boolean tests.

Furthermore, each edge of $\mathcal{T}[S, \mathtt{x}]$ is labeled with a *syntactic state*, i.e., a tuple of terms $t : u$, where $t \equiv t_1, \ldots, t_n$, with $t_i \in \boldsymbol{Term}_{\mathtt{x}, s_i}$. Intuitively, t gives the current state, assuming execution of S starts in the initial state (represented by) $\mathtt{x}$.

In the course of the following definition we will make use of the *restricted tree* $\mathcal{T}^-[S, \mathtt{x}]$, which is just $\mathcal{T}[S, \mathtt{x}]$ without the 's' node.

We also use the notation $\mathcal{T}[S, t]$ for the tree formed from $\mathcal{T}[S, \mathtt{x}]$ by replacing all edge labels t' by $t'\langle \mathtt{x}/t \rangle$.

The definition is by structural induction on S.

(i) $S \equiv$ skip. Then $\mathcal{T}[S, \mathtt{x}]$ is as in Fig. 1.

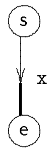

Fig. 1.

(ii) $S \equiv \mathtt{y} := r$, where $\mathtt{y} \equiv \mathtt{y}_1, \ldots, \mathtt{y}_m$ and $r \equiv r_1, \ldots, r_m$, with each $\mathtt{y}_j$ in $\mathtt{x}$. Then $\mathcal{T}[S, \mathtt{x}]$ is as in Fig. 2, where $t \equiv t_1, \ldots, t_n$ is defined by:

$$t_i \equiv \begin{cases} r_j \text{ if } \mathtt{x}_i \equiv \mathtt{y}_j & \text{for some } j \\ \mathtt{x}_i & \text{otherwise.} \end{cases}$$

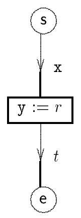

Fig. 2.

(*iii*) $T \equiv S_1; S_2$. Then $\mathcal{T}[S, \mathtt{x}]$ is formed from $\mathcal{T}[S_1, \mathtt{x}]$ by *replacing* each leaf (Fig. 3) by the tree in Fig. 4.

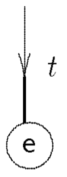

Fig. 3.

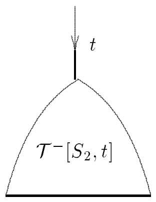

Fig. 4.

(*iv*) $S \equiv$ if b then S_1 else S_2 fi. Then $\mathcal{T}[S, \mathtt{x}]$ is as in Fig. 5.

(*v*) $S \equiv$ while b do S_1 od. For the sake of this case, we temporarily adjoin another kind of leaf to our tree formalism, labelled 'i' (for 'incomplete computation'), in addition to the e-leaf (representing an end to the computation). Then $\mathcal{T}[S, \mathtt{x}]$ is defined as the 'limit' of the sequence of trees $\mathcal{T}_n$, where $\mathcal{T}_0$ is as in Fig. 6, and $\mathcal{T}_{n+1}$ is formed from $\mathcal{T}_n$ by *replacing* each i-leaf (Fig. 7) by the tree in Fig. 8, where $\mathcal{T}_{\mathsf{i}}^-[S_1, t]$ is formed from $\mathcal{T}^-[S_1, t]$ by replacing all e-leaves in the latter by i-leaves. Note that the Boolean test b shown in Fig. 8 is evaluated at the 'current syntactic state' t (which amounts to evaluating $b\langle \mathtt{x}/t \rangle$ at 'the initial state' $\mathtt{x}$). Note also that the 'limiting tree' $\mathcal{T}[S, \mathtt{x}]$ does not contain any i-leaves. (*Exercise.*)

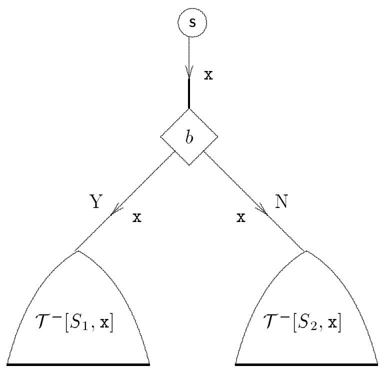
s
x
b
Y
x
x
N
$\mathcal{T}^-[S_1, \mathtt{x}]$
$\mathcal{T}^-[S_2, \mathtt{x}]$

Fig. 5.

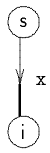

Fig. 6.

Remark 5.54.

(*a*) In case (*v*) the sequence $\mathcal{T}_n[S, \mathtt{x}]$ is defined by ***primitive recursion*** on n. An equivalent definition by *tail recursion* is possible (*Exercise*; compare the two definitions of $\boldsymbol{Comp}^A(S, \sigma, n)$ in sections 3.4 and 3.14; see also Remark 3.5).

(*b*) The construction of $\mathcal{T}[S, \mathtt{x}]$ is *effective* in S and $\mathtt{x}$. More precisely: $\mathcal{T}[S, \mathtt{x}]$ can be coded as a *recursive set* of numbers, with index primitive recursive in $\ulcorner S \urcorner$ and $\ulcorner \mathtt{x} \urcorner$.

Example 5.55. Let $S \equiv \mathsf{while}\ \mathtt{x} > 0\ \mathsf{do}\ \mathtt{x} := \mathtt{x} - 1\ \mathsf{od}$, where $\mathtt{x}$ is a natural number variable. Then (in the notation of case (*v*)) $\mathcal{T}_0$, $\mathcal{T}_1$ and $\mathcal{T}_2$ are, respectively, as shown in Figs. 9, 10 and 11, and $\mathcal{T}[S, \mathtt{x}]$ is the infinite tree shown in Fig. 12.

Notice that each tree in the sequence of approximations is obtained from the previous tree by replacing each i-leaf by one more iteration of the 'while' loop.

5.11 Engeler's lemma

Using the computation tree for a ***While*** statement constructed in the previous subsection, we will prove an important structure theorem for ***While*** semicomputabilty due to Engeler [1968a]. One of the consequences of this result will be the semicomputability equivalence thorem (5.61).

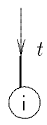

Fig. 7.

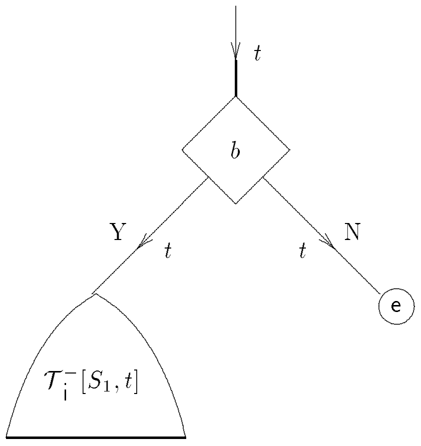

Fig. 8.

For each leaf λ of the computation tree $\mathcal{T}[S, \mathtt{x}]$, there is a *Boolean* $\boldsymbol{b}_{S,\lambda}$, with variables among $\mathtt{x}$, which expresses the conjunction of results of all the successive tests, that (the current values of) the variables $\mathtt{x}$ must satisfy in order for the computation to 'follow' the finite path from the root s to λ.

Consider, for example, a test node in $\mathcal{T}[S, \mathtt{x}]$:
If the path goes to the right here (say), then it contributes to $\boldsymbol{b}_{S,\lambda}$ the conjunct

$$\ldots \wedge \neg b\langle \mathtt{x}/\mathtt{t}\rangle \wedge \ldots$$

Next, let $(\lambda_0, \lambda_1, \lambda_2, \ldots)$ be some *effective enumeration* of leaves of $\mathcal{T}[S, \mathtt{x}]$ (e.g., in increasing depth, and, at a given depth, from left to right). Then, writing $\boldsymbol{b}_{S,k} \equiv \boldsymbol{b}_{S,\lambda_k}$, we can express the *halting formula for* S as the countable disjunction

$$\boldsymbol{halt}_S \equiv_{df} \bigvee_{k=0}^{\infty} \boldsymbol{b}_{S,k} \tag{5.3}$$

which expresses the conditions under which execution of S eventually *halts*, if started in the initial state (represented by) $\mathtt{x}$.

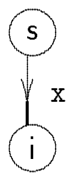

Fig. 9.

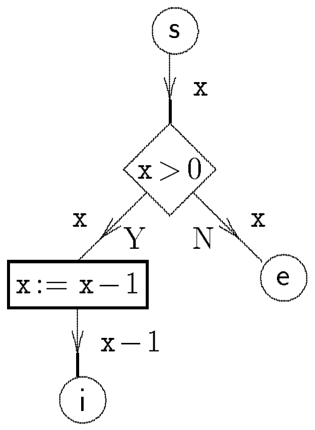

Fig. 10.

Note that although the Booleans $\boldsymbol{b}_{S,k}$, and (hence) the formula $\boldsymbol{halt}_S$, are constructed from a computation tree $\mathcal{T}[S, \mathtt{x}]$ for some tuple $\mathtt{x}$ containing $\boldsymbol{var}(S)$, their construction is independent of the choice of $\mathtt{x}$.

Remark 5.56.

(a) The Booleans $\boldsymbol{b}_{S,k}$ are effective in S and k. More precisely, $\ulcorner \boldsymbol{b}_{S,k} \urcorner$ is *partial recursive* in $\ulcorner S \urcorner$ and k.

(b) Further, by a standard technique of classical recursion theory, for a fixed S, if $\mathcal{T}[S, \mathtt{x}]$ has at least one leaf, then the enumeration

$$\boldsymbol{b}_{S,0}, \boldsymbol{b}_{S,1}, \boldsymbol{b}_{S,2}, \ \ldots$$

can be constructed (with repetitions) so that $\boldsymbol{b}_{S,k}$ is a total function of k, and, in fact, *primitive recursive* in k.

Now consider a relational procedure

$$P \equiv \mathsf{proc\ in\ a\ aux\ c\ begin}\ S\ \mathsf{end}$$

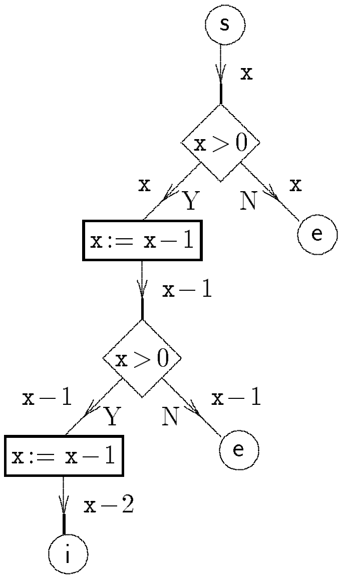

Fig. 11.

with input variables $\mathsf{a} : u$ and auxiliary variables $\mathsf{c} : w$. Then $S \equiv S_{init}; S'$, where S_{init} is an initialisation of the auxiliary variables c to the default tuple $\boldsymbol{\delta}^w$. The *computation tree for P* is defined to be

$$\mathcal{T}(P) \ =_{df} \ \mathcal{T}[S', \mathsf{a}, \mathsf{c}]$$

with a corresponding *halting formula*

$$\boldsymbol{halt}_{S'} \equiv \bigvee_{k=0}^{\infty} \boldsymbol{b}_{S',k} \tag{5.4}$$

(cf. (5.3)). Now, for $k = 0, 1, \ldots$, let $\boldsymbol{b}_k$ be the Boolean which results from substituting $\boldsymbol{\delta}^w$ for c in $\boldsymbol{b}_{S',k}$. Note that $\boldsymbol{var}(\boldsymbol{b}_k) \subseteq \mathsf{a}$. Let $\boldsymbol{b}_k[a] \in \mathbb{B}$ be the evaluation of $\boldsymbol{b}_k$ when $a \in A^u$ is assigned to a. Then by (5.4), the *halting set of P* (5.1) is characterised as an *effective countable disjunction*

$$a \in \boldsymbol{Halt}^A(P) \Longleftrightarrow \bigvee_{k=0}^{\infty} \boldsymbol{b}_k[a] \tag{5.5}$$

for all $a \in A^u$. Now suppose R is a $\boldsymbol{While}$ semicomputable relation on A. That means, by definition, that $R = \boldsymbol{Halt}^A(P)$ for a suitable $\boldsymbol{While}$ procedure P. Hence, by (5.5), we immediately derive the following theorem

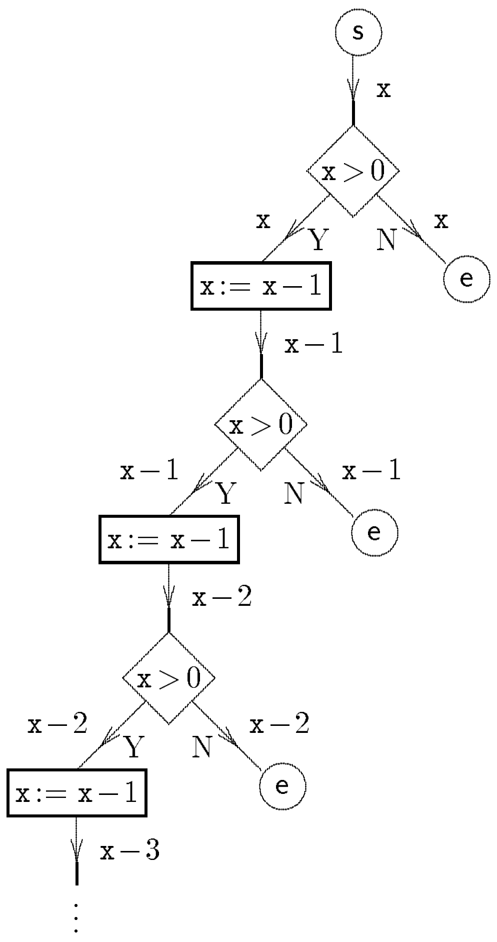

Fig. 12.

due to Engeler [1968a]:

Theorem 5.57 (Engeler's lemma). *Let R be a **While** semicomputable relation on a standard Σ-structure A. Then R can be expressed as an effective countable disjunction of Booleans over Σ.*

Actually, we need a stronger version of Engeler's lemma, applied to ***While**** programs, which we will derive next.

5.12 Engeler's lemma for *While** semicomputability

We will use the results of the previous subsection, applied to ***While**** computation, together with the Σ^*/Σ conservativity theorem for terms (Theorem 3.63), to prove a strengthened version of Engeler's lemma.

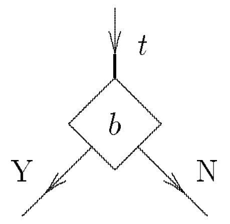

Fig. 13.

Theorem 5.58 (Engeler's lemma for $\boldsymbol{While}^*$ semicomputability). *Let R be a $\boldsymbol{While}^*$ semicomputable relation on a standard Σ-structure A. Then R can be expressed as an effective countable disjunction of Booleans over Σ.*

Proof. Suppose R has type u (over Σ). By assumption, R is the halting set $\boldsymbol{Halt}^A(P)$ for a $\boldsymbol{While}^*$ procedure P. By (5.5) of the previous subsection,

$$R(a) \iff \bigvee_{k=0}^{\infty} \boldsymbol{b}_k[a] \tag{5.6}$$

for all $a \in A^u$, where (now) $\boldsymbol{b}_k \in \boldsymbol{Term}^*_{\mathtt{a},\mathtt{bool}}$, i.e., the Booleans $\boldsymbol{b}_k$, though not of starred sort, may contain subterms of starred sort — for example, they may be equations or inequalities between terms of starred sort. As before, $\boldsymbol{b}_k[a]$ is the evaluation of $\boldsymbol{b}_k$ when $a \in A^u$ is assigned to $\mathtt{a}$.

Now for any Boolean $b \in \boldsymbol{Term}^*_{\mathtt{a},\mathtt{bool}}$, let b' be the Boolean in $\boldsymbol{Term}_{\mathtt{a},\mathtt{bool}}$ associated with b by the conservativity theorem (3.63). Then from (5.6), for all $a \in A^u$,

$$R(a) \iff \bigvee_{k=0}^{\infty} \boldsymbol{b}'_k[a]. \tag{5.7}$$

Because the disjunction in (5.6) and the transformation $b \mapsto b'$ are both effective, the disjunction in (5.7) is also effective. ∎

For the converse direction:

Lemma 5.59. *Let R be a relation on a standard Σ-algebra A. If R can be expressed as an effective countable disjunction of Booleans over Σ, then R is $\boldsymbol{While}^*$ semicomputable.*

Proof. Suppose R is expressed by an effective disjunction

$$R(a) \iff \bigvee_{k=0}^{\infty} \boldsymbol{b}_k[a]$$

for all $a \in A^u$, where $\boldsymbol{b}_k \in \boldsymbol{Term}_{\mathtt{a,bool}}$. Then for all $a \in A^u$,

$$R(a) \iff \exists k \left(\boldsymbol{te}^A_{\mathtt{a,bool}}(\ulcorner \boldsymbol{b}_k \urcorner, a) = \mathtt{tt}\right) \tag{5.8}$$

where $\ulcorner \boldsymbol{b}_k \urcorner$ is (total) recursive in k. Hence, by Corollary 4.7, Remark 3.16 and Theorem 5.41, R is $\boldsymbol{While}^*$ semicomputable. ∎

Combining Engeler's lemma for $\boldsymbol{While}^*$ semicomputability with this lemma gives the following 'structural' characterisation of $\boldsymbol{While}^*$ semicomputable relations.

Corollary 5.60. *Let R be a relation on a standard Σ-algebra. Then R can be expressed as an effective countable disjunction of Booleans over Σ if, and only if, R is $\boldsymbol{While}^*$ semicomputable.*

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

Theorem 5.61 (Semicomputability equivalence theorem). *Suppose that A is a standard Σ-algebra with the TEP, and that R is a relation on A. Then the following assertions are equivalent:*

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

Proof. The step $(i) \Rightarrow (ii)$ is trivial, and $(ii) \Rightarrow (iii)$ is just Engeler's lemma for $\boldsymbol{While}^*$. The new step here ($(iii) \Rightarrow (i)$) follows from (5.8) and Theorem 5.23. ∎

Corollary 5.62. *Suppose that A is a standard Σ-algebra with the TEP, and that R is a relation on A. Then*

$$R \text{ is } \boldsymbol{While}^N \text{ computable on } A \iff R \text{ is } \boldsymbol{While}^* \text{ computable on } A.$$

Proof. From Theorem 5.61, or from Corollary 4.18. ∎

5.13 Σ_1^* definability: Input/output and halting formulae

For any standard signature Σ, let $\boldsymbol{Lang}^* = \boldsymbol{Lang}(\Sigma^*)$ be the first-order language with equality over Σ^*.

The *atomic formulae* of $\boldsymbol{Lang}^*$ are equalities between pairs of terms of the same sort, for any sort of Σ^*, i.e., sort s, s^{u} and s^*, for all sorts s of Σ (whether equality sorts or not), and for the sort `nat`.

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

We are more interested in special classes of formulae of $\boldsymbol{Lang}^*$.

Definition 5.63 (Classes of formulae of $\boldsymbol{Lang}^*$).

(a) A *bounded quantifier* has the form '$\forall k < t$' or '$\exists k < t$', where $t : \mathsf{nat}$.
(b) An *elementary formula* is one with only bounded quantifiers.
(c) A $\boldsymbol{\Sigma}_1^*$ *formula* is formed by prefixing an elementary formula with existential quantifiers only.
(d) An *extended* $\boldsymbol{\Sigma}_1^*$ *formula* is formed by prefixing an elementary formula with a string of existential quantifiers and bounded universal quantifiers (in any order).

Proposition 5.64. *For any extended $\boldsymbol{\Sigma}_1^*$ formula P, there is a $\boldsymbol{\Sigma}_1^*$ formula Q which is equivalent to P over Σ, in the sense that*

$$\boldsymbol{StdAlg}(\Sigma^*) \models P \leftrightarrow Q.$$

Proof. The construction of Q from the P is given in Tucker and Zucker [1993]. (In that paper, the equivalence is actually shown relative to a formal system over $\mathbb{K}$ with $\boldsymbol{\Sigma}_1^*$ induction. However, we are not concerned with issues of provability in this chapter.) ∎

Because of this result, we will use the term '$\boldsymbol{\Sigma}_1^*$' to denote (possibly) extended $\boldsymbol{\Sigma}_1^*$ formulae.

Proposition 5.65. *If P is an elementary formula all of whose variables are of equality sort, then the predicate defined by P is $\boldsymbol{For}^*$ computable.*

Proof. By structural induction on P. Equations between variables of equality sort, and Boolean operations, are trivially computable. Bounded quantification uses the 'for' loop. ∎

In general, formulae over the structure A^* (or rather, over the signature Σ^*) may have simple, augmented, starred or nat variables (Definition 3.39). We are interested in formulae with the property that all free variables are *simple*, since such formulae define relations on A. For such formulae, all bound augmented variables may be replaced by bound simple variables, by the effective coding of A^{u} in A (see Remark 2.30(c)).

Theorem 5.66 (The $\boldsymbol{\Sigma}_1^*$ i/o formula for a procedure). *Given a $\boldsymbol{While}^*$ procedure $P : u \to v$*

$$P \equiv \mathsf{proc\ in\ a\ out\ b\ aux\ c^*\ begin}\ S\ \mathsf{end} \tag{5.9}$$

where $\mathsf{a} : u$, $\mathsf{b} : v$ and $\mathsf{c}^* : w^*$, we can effectively construct a $\boldsymbol{\Sigma}_1^*$ formula $\boldsymbol{IO}_P \equiv \boldsymbol{IO}_P(\mathsf{a}, \mathsf{b})$, with free variables among a,b, called the input/output (or i/o) formula for P, which satisfies: for all $A \in \boldsymbol{StdAlg}(\Sigma)$, $a \in A^u$ and $b \in A^v$,

$$A \models \boldsymbol{IO}_P[a, b] \iff P^A(a) \downarrow b.$$

Note: The left-hand side means that A satisfies $\boldsymbol{IO}_P$ at any state which assigns a to a and b to b.

Proof. First we construct an elementary formula

$$\boldsymbol{Compu}_S(\mathtt{x},\mathtt{y},\mathtt{z}^*)$$

(where $\boldsymbol{var}(S) \subseteq \mathtt{x}$), as in Tucker and Zucker [1988, §2.6.11], by structural induction on S, with the meaning: '$\mathtt{z}^*$ represents a computation sequence generated by statement S, starting in a state in which *all the variables* of S have values (represented by) $\mathtt{x}$, and ending in a state in which these variables have value $\mathtt{y}$'. From this it is easy to construct a $\mathbf{\Sigma}_1^*$ formula

$$\boldsymbol{Compu}_P(\mathtt{a},\ \mathtt{b},\ \mathtt{z}^*)$$

with the meaning: '$\mathtt{z}^*$ represents a computation sequence generated by procedure P, starting in a state in which the *input variables* have values (represented by) $\mathtt{a}$, and ending in a state in which the *output variables* have values $\mathtt{b}$'.

Finally we obtain the $\mathbf{\Sigma}_1^*$ i/o formula

$$\boldsymbol{IO}_P(\mathtt{a},\mathtt{b}) \ =_{df} \ \exists z^* \ \boldsymbol{Compu}_P(\mathtt{a},\ \mathtt{b},\ \mathtt{z}^*)$$

as required. ∎

Remark 5.67 (Quantification over $\mathbb{N}$ sufficient). We can construct a $\mathbf{\Sigma}_1^*$ i/o formula in which there is only existential quantification over $\mathbb{N}$. Briefly, a formula similar to $\boldsymbol{Compu}_S$ (in the above proof) is constructed in which the variable $\mathtt{z}^*$ representing the computation sequence is replaced by a Gödel number. (Cf. point (2°) in section 4.8, concerning the replacement of the function $\boldsymbol{comp}_{\mathtt{x}}^A$ by $\boldsymbol{compu}_{\mathtt{a}}^A$.)

Remark 5.68 (Alternative construction of $\boldsymbol{IO}_P$). Let α be the $\boldsymbol{\mu PR}^*$ scheme (section 8.1) which corresponds to P according to the construction given by the proof of Theorem 8.5(d). By structural induction on α, we construct the formula $\boldsymbol{IO}_\alpha(\equiv \boldsymbol{IO}_P)$, as in Tucker *et al.* [1990] or Tucker and Zucker [1993, §5] (where it is called P_α).

Corollary 5.69 (The $\mathbf{\Sigma}_1^*$ halting formula for a procedure). *Given a* $\boldsymbol{While}^*$ *procedure* $P : u \to v$ *as in (5.9), we can construct a* $\mathbf{\Sigma}_1^*$ *definition for the halting formula* $\boldsymbol{halt}_P \equiv \boldsymbol{halt}_P(\mathtt{a})$ *for* P.

Proof. We define

$$\boldsymbol{halt}_P(\mathtt{a}) \equiv \exists \mathtt{b}\, \boldsymbol{IO}_P(\mathtt{a},\mathtt{b})$$

Alternatively, recalling (Remark 5.3) that in the context of semicomputability we may assume that P has no output variables, we can put, more simply,

$$\boldsymbol{halt}_P(\mathtt{a}) \equiv \boldsymbol{IO}_P(\mathtt{a}).$$

Since $\boldsymbol{IO}_P$ is $\mathbf{\Sigma}_1^*$, so is $\boldsymbol{halt}_P$ (in either case). ∎

Remark 5.70 (Quantification over $\mathbb{N}$ sufficient). Again, by the use of Gödel numbers, we can construct a $\mathbf{\Sigma}_1^*$ halting formula in which there is only existential quantification over $\mathbb{N}$. (Cf. Remark 5.67.)

Note finally that by Corollary 5.69, since for all $A \in \boldsymbol{StdAlg}(\Sigma)$ and all $a \in A^u$,

$$a \in \boldsymbol{Halt}^A(P) \iff A \models \boldsymbol{halt}_P(a),$$

it follows that

the halting set for $\boldsymbol{While}^*$ *procedures is* $\boldsymbol{\Sigma}_1^*$ *definable, uniformly over* $\boldsymbol{StdAlg}(\Sigma)$.

5.14 The projective equivalence theorem

Theorem 5.71. *Let R be a relation on a standard Σ-algebra A. Then*

R is projectively $\boldsymbol{While}^*$ semicomputable $\iff$
R is projectively $\boldsymbol{For}^*$ computable.

We present two proofs of this theorem. The first uses $\boldsymbol{\Sigma}_1^*$ definability of the halting set (and the assumption that Σ has equality at all sorts) while the second uses Engeler's lemma (without any assumption about equality sorts).

First proof. First we restate the theorem suitably.

Suppose Σ has an equality operator at all sorts. Let R be a relation on A. Then the following are equivalent:

(i) *R is projectively $\boldsymbol{While}^*$ semicomputable;*
(ii) *R is $\boldsymbol{\Sigma}_1^*$ definable;*
(iii) *R is projectively $\boldsymbol{For}^*$ computable.*

(i)$\Longrightarrow$*(ii)*: Suppose $R : u$, and for all $x \in A^u$,

$$R(x) \Longleftrightarrow \exists y^* \in A^{v^*} R_1(x, y^*) \tag{5.10}$$

where v^* is a product type of Σ^*, and $R_1 : u \times v^*$ is $\boldsymbol{While}$ semicomputable on A^*. Then R_1 is the halting set of a $\boldsymbol{While}(\Sigma^*)$ procedure P on A^*. By Corollary 5.69, R_1 is $\boldsymbol{\Sigma}_1^*$ definable on A^*, say

$$R_1(x, y^*) \Longleftrightarrow \exists z^{**} \in A^{w^{**}} R_0(x, y^*, z^{**}), \tag{5.11}$$

where w^{**} is a product type of Σ^{**}, and $R_0(\dots)$ is given by an elementary formula over Σ^{**}. Combining (5.10) and (5.11):

$$R(x) \Longleftrightarrow \exists y^* \in A^{v^*} \exists z^{**} \in A^{w^{**}} R_0(x, y^*, z^{**}). \tag{5.12}$$

Finally, by the coding of $(A^*)^*$ in A^* (Remark 2.31(*d*)), we can rewrite the existential quantification over $(A^*)^*$ in ('5.12) ('$\exists z^{**} \in A^{w^{**}}$') as existential quantification over A^*, yielding a $\boldsymbol{\Sigma}_1^*$ definition of R on A.

(ii)$\Longrightarrow$*(iii)*: Suppose R is defined by the formula $\exists \mathtt{z}^* P(\mathtt{x}, \mathtt{z}^*)$, where $\mathtt{z}^*$ is a tuple of starred and unstarred variables, and P is elementary. Then R is a projection of P, which, by Proposition 5.65, is $\boldsymbol{For}^*$ computable. (Here we use the assumption about equality sorts.)

(iii)$\Longrightarrow$*(i)*: Trivial (using Proposition 3.34). ■

Second proof. (Here we make no assumption about equality sorts.) Suppose $R : u$ is projectively ***While***** semicomputable on A. Then (as before) for some product type v^* of Σ^*,

$$R(x) \iff \exists y^* \in A^{v^*} R_1(x, y^*)$$

where $R_1 : u \times v^*$ is ***While*** semicomputable on A^*.

By Engeler's lemma (Theorem 5.57) applied to A^*, there is an effective sequence $\boldsymbol{b}_k{}^*(\mathtt{x}, \mathtt{y}^*)$ $(k = 0, 1, 2, \ldots)$ of Booleans over Σ^* such that $R_1(x, y^*)$ is equivalent over A to the disjunction of $\boldsymbol{b}_k{}^*[x, y^*]$. Further, by Remark 5.56(*b*), this sequence can be defined so that $\ulcorner \boldsymbol{b}_k{}^* \urcorner$ is primitive recursive in k. (Assume here that R is non-empty, otherwise the theorem is trivial.) Then

$$R_1(x, y^*) \iff \text{for some } k,\ \boldsymbol{te}^{A^*}_{\mathtt{x},\mathtt{y}^*,\mathtt{bool}}(\ulcorner \boldsymbol{b}_k{}^* \urcorner, x, y^*) = \mathtt{tt}.$$

Further, $\boldsymbol{te}^{A^*}_{\mathtt{x},\mathtt{y}^*,\mathtt{bool}}$ is ***For*** computable on A^* (by Proposition 4.6). Hence the function g defined on A^* by

$$g(k, x, y^*) \ =_{df}\ \boldsymbol{te}^{A^*}_{\mathtt{x},\mathtt{y}^*,\mathtt{bool}}(\ulcorner \boldsymbol{b}_k{}^* \urcorner,\ x, y^*)$$

is ***For*** computable on A^* (by Equation 3.8 and Remark 3.16). Hence the relation

$$R_0(k, x, y^*) \iff_{df} g(k, x, y^*) = \mathtt{tt}$$

is ***For*** computable on A^* (composing g with equality on bool), and so the relation

$$R(x) \iff \exists\, y^* \exists k\, R_0(k, x, y^*)$$

is projectively ***For***** computable on A.

The other direction is trivial. ■

5.15 Halting sets of *While* procedures with random assignments

We now consider the ***While*** programming language over Σ, extended by the *random assignment*

$$\mathtt{x} := ?$$

for variables $\mathtt{x}$ of every sort of Σ. This is an example of *non-deterministic computation.*

The semantics of the ***While****-random* programming language can be obtained by a modification of the semantics of the ***While*** language given in section 3, by taking the meaning of a statement S to be a function $[\![\mathrm{S}]\!]^A$ from states to *sets of* states. In the case that S is a random assignment $\mathtt{x} := ?$, $[\![\mathrm{S}]\!]^A\sigma$ is the set of *all states* which agree with σ on all variables *except* $\mathtt{x}$.

However we are only interested here in the ***While***-random language for defining relations, not functions, as the following definition clarifies.

Definition 5.72. Let P be a ***While***-random procedure, with input variables $\mathtt{a} : u$ (and, we may assume, no output variables). The *halting set* of P on A is the set of tuples $a \in A^u$ such that when $\mathtt{a}$ is initialised to a, then *for some values* of the random assignments, execution of P halts.

Definition 5.73. Let R be a relation on A.

(a) R is ***While**-random semicomputable* on A if R is the halting set of a ***While***-random procedure on A.

(b) R is ***While***-random semicomputable* on A if R is the halting set of a ***While***(Σ^*)-random procedure on A^*.

Note that in (b), there could be random assignments to starred (auxiliary) variables.

Remark 5.74. Clearly, semicomputability with random assignments can be viewed as a *generalisation* of the notion of semicomputability with search, i.e., initialisation of search variables (section 5.3), since *initialisation amounts to random assignments at the beginning of the program.* We may ask how these two notions of semicomputability compare. We will show that, at least over A^*, they coincide: both are equivalent to projective ***While**** semicomputability.

Theorem 5.75. *Let R be a relation on A. Then*

$$R \text{ is } \boldsymbol{While}^*\text{-random semicomputable} \Longleftrightarrow$$
$$R \text{ is projectively } \boldsymbol{While}^* \text{ semicomputable.}$$

Proof. The direction ($\Longleftarrow$) follows from Remark 5.74 and Theorem 5.18. We turn to the direction ($\Longrightarrow$). For ease of exposition, we will assume first that R is ***While**-random semicomputable.*

We will define a *computation tree* $\mathcal{T}[S, \mathtt{x}]$ for ***While***-random statements S with $\boldsymbol{var} S \subseteq \mathtt{x} \equiv \mathtt{x}_1, \dots, \mathtt{x}_n$, extending the definition for ***While*** statements given in Section 5.10. There is one new case:

(vi) $S \equiv \mathtt{x}_i := ?$. Then $\mathcal{T}[S, \mathtt{x}]$ is as in Fig. 14.

So $\mathtt{x}_i$ is replaced by a *new variable* $\mathtt{x}_i'$ of the same sort.

Notice that for a '?'-assignment S, and for terms $t \equiv t_1, \dots, t_n$, the tree $\mathcal{T}[S, t]$ is asn in Fig. 15.
where $\mathtt{x}_i'$ does not occur in $\mathtt{x}$ or t. The intuition here is that there is nothing we can say about the 'new' value of $\mathtt{x}_i$, so we can only represent it by a brand new variable $\mathtt{x}_i'$. If this assignment is followed by another assignment $\mathtt{x}_i := ?$, we introduce *another* new variable $\mathtt{x}_i''$, and so on.

In this way *the variables proliferate*, and the tree contains (possibly) infinitely many variables. Hence we cannot simply construct a halting formula as an (infinite) disjunction of Booleans in a *fixed finite number of variables over* Σ, as we did in section 5.11.

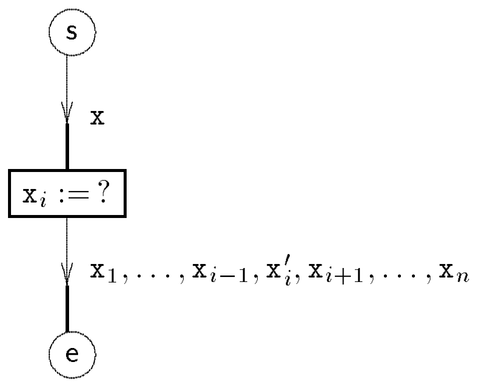

Fig. 14.

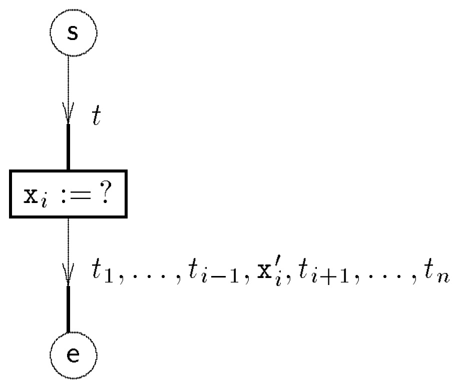

Fig. 15.

The solution is to represent all the variables $\mathtt{x}_i, \mathtt{x}_i', \mathtt{x}_i'', \ldots$ which arise in this way for each i $(1 \leq i \leq n)$ by a *single starred variable* $\mathtt{x}_i^*$ (with $\mathtt{x}_i^*[0], \mathtt{x}_i^*[1], \mathtt{x}_i^*[2], \ldots$ representing $\mathtt{x}_i, \mathtt{x}_i', \mathtt{x}_i'', \ldots$). Then to each leaf λ of $\mathcal{T}[S, \mathtt{x}]$ there corresponds (as in section 5.11) a Boolean $\boldsymbol{b}_{S,\lambda}$, but now in the *starred variables* $\mathtt{x}^* \equiv \mathtt{x}_1^*, \ldots, \mathtt{x}_n^*$.

Again, as in section 5.11, we can define the *halting formula* for S as a countable disjunction of Booleans:

$$\boldsymbol{halt}_S \equiv_{df} \bigvee_{k=0}^{\infty} \boldsymbol{b}_{S,k}$$

where the $\boldsymbol{b}_{S,k}$ are effective, in fact primitive recursive, in S and k. Note however that the program variables in $\boldsymbol{halt}_S$ are now among $\mathtt{x}^*$, not $\mathtt{x}$.

Now suppose that the relation $R : u$ is the halting set of a ***While***-

random procedure on A,

$$P \equiv \textsf{proc in a aux c begin } S_{init}; S' \textsf{ end}$$

where $\mathsf{a} : u = s_1 \times \ldots \times s_m$ and $\mathsf{c} : w$.

As in section 5.11, let $\boldsymbol{b}_k$ be the Boolean which results from substituting the default tuple $\boldsymbol{\delta}^w$ for c in $\boldsymbol{b}_{S',k}$. Note that $\boldsymbol{var}(\boldsymbol{b}_k) \subseteq \mathsf{a}, \mathsf{c}^* : u \times w^*$. Then for all $a \in A^u$:

$$a \in R \Longleftrightarrow \exists c^* \in A^{w^*} \exists k \left[\bigwedge_{i=1}^{m} (\boldsymbol{\delta}_i = c_i^*[0]) \wedge \boldsymbol{te}^{A^*}_{\mathsf{a},\mathsf{c}^*,\mathsf{bool}}(\ulcorner \boldsymbol{b}_k \urcorner, a, c^*) = \mathtt{tt} \right]$$

(cf. (5.8) in section 5.12) which (by Proposition 4.6) is projectively ***For**** computable on A, and hence projectively ***While**** semicomputable on A.

This proves the theorem for the case that R is ***While***-random semicomputable on A.

Assume, finally, that R is ***While****-random semicomputable, i.e., the procedure for R may contain *starred auxiliary variables*, and there may be random assignments to these. Now we can represent a sequence of random assignments to a starred variable by a single *doubly starred variable*, or two-dimensional array, which can then be effectively coded in A^* (Remark 2.31(d)), and proceed as before. ∎

6 Examples of semicomputable sets of real and complex numbers

In this section we look at the various notions of semicomputability in the case of algebras based on the set $\mathbb{R}$ of real numbers and the set $\mathbb{C}$ of complex numbers. By doing so, we will find examples proving the inequivalence of the following notions:

(i) ***While*** computability;
(ii) ***While*** semicomputability;
(iii) projective ***While*** semicomputability;
(iv) projective ***While**** semicomputability.

We will also find interesting examples of sets of real and complex numbers which are semicomputable but not computable. Some of these sets belong to dynamical system theory: orbits and periodic sets of chaotic systems turn out to be semicomputable but not computable.

Finally we will also reconsider an example of a semicomputable, non-computable set of complex numbers described in Blum *et al.* [1989]. The effective content of their work can be obtained from the general theory.

Our main tool will be Engeler's lemma.

We will concentrate on the following algebras introduced in Example 2.23: the standard algebras

$$\mathcal{R} \;=\; (\mathcal{B};\ \mathbb{R};\ 0,\ 1,\ +,\ -,\ \times, \mathsf{if}_{\mathsf{real}}, \mathsf{eq}_{\mathsf{real}})$$

of reals, and

$$\mathcal{C} \;=\; (\mathcal{R}; \mathbb{C};\ 0,\ 1,\ i, +,\ -,\ \times,\ \mathsf{re},\ \mathsf{im},\ \pi)$$

of complex numbers, and their expansions

$$\mathcal{R}^{<} \;=\; (\mathcal{R}; \mathsf{less}_{\mathsf{real}}) \qquad \text{and} \qquad \mathcal{C}^{<} \;=\; (\mathcal{C}; \mathsf{less}_{\mathsf{real}})$$

formed by adjoining the order relation on the reals $\mathsf{less}_{\mathsf{real}}$: $\mathbb{R}^2 \to \mathbb{B}$ (which we will write as infix '$<$'). We will show that:

(a) the order relation on $\mathbb{R}$ is projectively ***While*** semicomputable, but not ***While*** semicomputable, on $\mathcal{R}$; and

(b) a certain real closed subfield of $\mathbb{R}$ is projectively ***While**** semicomputable, but not projectively ***While*** semicomputable, on $\mathcal{R}^{<}$.

6.1 Computability on $\mathbb{R}$ and $\mathbb{C}$

Based on the general theory of computability developed so far, we can see that each of these four algebras has a computability theory with several standard properties (e.g., universality, section 4.9). First, we will list some preliminary results for computability on the real and complex numbers that will entail simplicity and elegance of computation on these structures, but will also show that the analogy with the classical case of computation on $\mathbb{N}$ often breaks down. To begin with, we have:

Proposition 6.1. *For* $A = \mathcal{R}, \mathcal{R}^{<}, \mathcal{C}$ *or* $\mathcal{C}^{<}$,

$$\textbf{\textit{While}}^N(A) \;=\; \textbf{\textit{While}}(A). \tag{6.1}$$

This is proved essentially by simulating the algebra $\mathcal{N}$, i.e., the carrier $\mathbb{N}$, with zero, successor, etc., in the carrier $\mathbb{R}$, using the non-negative integers together with 0, the operation $+1$, etc.

As an exercise, the reader should formulate a theorem expressing a sufficient condition on an algebra A for (6.1) to hold, from which the above proposition will follow as a simple corollary.

This situation should be contrasted with that in Example 5.36.

Recall Definition 4.4 and Examples 4.5:

Lemma 6.2 (TEP). *The algebras* $\mathcal{R}, \mathcal{R}^{<}, \mathcal{C}$ *and* $\mathcal{C}^{<}$ *all have the TEP.*

The TEP has a profound impact on the computability theory for an algebra. For example, from Corollary 4.18 we know that on $\mathcal{R}, \mathcal{R}^{<}, \mathcal{C}$ and $\mathcal{C}^{<}$:

$$\textbf{\textit{While}}^* \textit{ computability} \;=\; \textbf{\textit{While}}^N \textit{ computability}$$

and hence (or from the semicomputability equivalence theorem (5.61))

$$\textbf{\textit{While}}^* \textit{ semicomputability} \;=\; \textbf{\textit{While}}^N \textit{ semicomputability}$$

for these four algebras. We will give more detailed formulations of these facts for each of these algebras shortly.

Example 6.3 (Non-computable functions). Recall Theorem 3.66 which says that the output of a $\boldsymbol{While}$, $\boldsymbol{While}^N$ or $\boldsymbol{While}^*$ computable function is contained in the subalgebra generated by its inputs. From this we can derive some negative computability results for these algebras:

(a) The square root function is not $\boldsymbol{While}^*$ computable on $\mathcal{R}$ or $\mathcal{R}^<$. This follows from the fact that the subset of $\mathbb{R}$ generated from the empty set by the constants and operations of $\mathcal{R}$ or $\mathcal{R}^<$ is the set $\mathbb{Z}$ of integers. But $\sqrt{2}$ is not in this set. (For computability in ordered Euclidean fields incorporating the square root operation, see Engeler [1975a]).

(b) The mod function $(z \mapsto |z|)$ is not $\boldsymbol{While}^*$ computable on $\mathcal{C}$ or $\mathcal{C}^<$. This follows from the fact that the subset of $\mathbb{R}$ generated from the empty set by the constants and operations of $\mathcal{C}$ or $\mathcal{C}^<$ is again $\mathbb{Z}$. But again, $|1+i| = \sqrt{2}$ is not in this set.

(c) The mod function *would* be computable in $\mathcal{C}$ if we adjoined the square root function to the algebra $\mathcal{R}$ (as a reduct of $\mathcal{C}$).

In the rest of this subsection, we will apply Engeler's lemma for $\boldsymbol{While}^*$ semicomputability (section 5.12) to the algebras $\mathcal{R}, \mathcal{R}^<, \mathcal{C}$ and $\mathcal{C}^<$.

From the semicomputability equivalence theorem (5.61) (which follows from Engeler's lemma) and from the TEP lemma (6.2), we get:

Theorem 6.4 (Semicomputability equivalences for $\mathcal{R}, \mathcal{R}^<, \mathcal{C}, \mathcal{C}^<$). *Suppose A is $\mathcal{R}$ or $\mathcal{R}^<$, and $R \subseteq \mathbb{R}^n$; or A is $\mathcal{C}$ or $\mathcal{C}^<$, and $R \subseteq \mathbb{C}^n$. Then the following are equivalent:*

(i) R is $\boldsymbol{While}^N$ semicomputable on A;

(ii) R is $\boldsymbol{While}^*$ semicomputable on A;

(iii) R can be expressed as an effective countable disjunction of Booleans over A.

For applications of this theorem, we need the following normal form lemmas for Booleans over $\mathcal{R}$ and $\mathcal{R}^<$.

Lemma 6.5 (Normal form for Booleans over $\mathcal{R}$). *A Boolean over $\mathcal{R}$, with variables $x = x_1, \dots, x_n$ of sort* real *only, is effectively equivalent over $\mathcal{R}$ to a finite disjunction of finite conjunctions of equations and negations of equations of the form*

$$p(x) = 0 \qquad \textit{and} \qquad q(x) \neq 0,$$

where p and q are polynomials in x with coefficients in $\mathbb{Z}$.

Lemma 6.6 (Normal form for Booleans over $\mathcal{R}^<$). *A Boolean over $\mathcal{R}^<$, with variables $x = x_1, \dots, x_n$ of sort* real *only, is effectively equivalent over $\mathcal{R}^<$ to a finite disjunction of finite conjunctions of equations and* *inequalities of the form*

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

where p and q are polynomials in x with coefficients in $\mathbb{Z}$.

The proofs of these are left as exercises.

6.2 The algebra of reals; a set which is projectively *While* semicomputable but not *While** semicomputable

In this subsection we obtain results distinguishing various notions of semicomputability, using the algebra $\mathcal{R}$ of reals. In the next subsection we will obtain other results in a similar vein, using the ordered algebra $\mathcal{R}^<$ of reals. We begin with a restatement of the semicomputability equivalence thorem (6.4) for the particular case of $\mathcal{R}$.

Theorem 6.7 (Semicomputability for $\mathcal{R}$). *Suppose $R \subseteq \mathbb{R}^n$. Then the following are equivalent:*

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

$$x \in R \iff \bigvee_i \boldsymbol{b}_i(x) \tag{6.2}$$

where each $\boldsymbol{b}_i(x)$ is a finite conjunction of equations and negations of equations of the form

$$p(x) = 0 \qquad \text{and} \qquad q(x) \neq 0, \tag{6.3}$$

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

Proof. From Theorem 6.4 and Lemma 6.5. ∎

Thus we see that there is an intimate connection between computability, polynomials and algebraic field extensions on $\mathbb{R}$.

Definition 6.8 (Algebraic and transcendental points). Let us define a point $\alpha = (\alpha_1, \dots, \alpha_n) \in \mathbb{R}^n$ to be (*i*) *algebraic* if it is the root of a polynomial in n variables with coefficients in $\mathbb{Z}$; and (*ii*) *transcendental* if it is not algebraic, or, equivalently, if for each $i = 1, \dots, n$, α_i is transcendental (in the usual sense) over $\mathbb{Q}(\alpha_1, \dots, \alpha_{i-1})$.

The following corollary was stated for $n = 1$ in Herman and Isard [1970].

Corollary 6.9. *If $R \subseteq \mathbb{R}^n$ is $\boldsymbol{While}^*$ semicomputable on $\mathcal{R}$, and contains a transcendental point α, then R contains some open neighbourhood of α.*

Proof. In the notation of (6.2): α satisfies $\boldsymbol{b}_i(x)$ for some i. Then (for this i) $\boldsymbol{b}_i(x)$ cannot contain any equations (as in (6.3)) since α is transcendental, and so it must contain negations of equations only. The result follows from the continuity of polynomial functions. ∎

An immediate consequence of this is:

Corollary 6.10 (Density/codensity condition). *Any subset of $\mathbb{R}^n$ which is both dense and co-dense in $\mathbb{R}^n$ (or in any non-empty open subset of $\mathbb{R}^n$) cannot be $\boldsymbol{While}^*$ computable on $\mathcal{R}$.*

Example 6.11. The following subsets of $\mathbb{R}^n$ are easily seen to be $\boldsymbol{While}^N$ semicomputable on $\mathcal{R}$ — in fact $\boldsymbol{While}$ semicomputable (by Proposition 6.1). However, they are not $\boldsymbol{While}$ (= $\boldsymbol{While}^N$= $\boldsymbol{While}^*$) computable, by the density/codensity condition:

(a) the set of points with *rational coordinates*;
(b) the set of points with *algebraic coordinates*;
(c) the set of *algebraic points*.

Of course, a standard example of a $\boldsymbol{While}$ semicomputable but not $\boldsymbol{While}^*$ computable set can be found in $\mathcal{N}$, namely any recursively enumerable, non-recursive set of naturals (Example 5.49).

Next, specialising to $n = 1$:

Corollary 6.12 (Countability/cofiniteness condition). *If $R \subseteq \mathbb{R}$ is $\boldsymbol{While}^*$ semicomputable on $\mathcal{R}$, then R is either countable or cofinite (i.e., the complement of a fine set) in $\mathbb{R}$.*

Proof. By the *fundamental theorem of algebra*, each polynomial equation with coefficients in $\mathbb{Z}$ has *at most finitely many roots* in $\mathbb{R}$. Hence, regarding the disjunction in (6.2), there are two cases:
Case 1. For some i, $\boldsymbol{b}_i(x)$ contains *only negations of equations*. Then (for this i) $\boldsymbol{b}_i(x)$ holds for *all but finitely many* $x \in \mathbb{R}$. Hence R is *co-finite* in $\mathbb{R}$.
Case 2. For all i, $\boldsymbol{b}_i(x)$ contains *at least one equation*. Then (for all i) $\boldsymbol{b}_i(x)$ holds for *at most finitely many* $x \in \mathbb{R}$. Hence R is *countable*. ∎

Hence we have:

Corollary 6.13. *A subset of $\mathbb{R}$ which is ($\boldsymbol{While}$or $\boldsymbol{While}^N$or $\boldsymbol{While}^*$) computable on $\mathcal{R}$ it is either finite or cofinite.*

Example 6.14. From Corollary 6.13 we have another example of a subset of $\mathbb{R}$ which is $\boldsymbol{While}$ semicomputable but not $\boldsymbol{While}^*$ computable, namely the integers (Example 5.5(c)).

Example 6.15 (Projectively $\boldsymbol{While}$ semicomputable, not $\boldsymbol{While}^*$ semicomputable set). The order relation on $\mathbb{R}$ is a primitive operation in $\mathcal{R}^<$, but, as we shall see, is not even semicomputable in $\mathcal{R}$. Consider first the relation

$$R_0(x,y) \iff_{df} x = y^2$$

on $\mathbb{R}$. R_0 is clearly ***While*** computable on $\mathcal{R}$, and so its projection on the first argument,

$$R =_{df} \{x \mid \exists y(x = y^2)\}$$

i.e., the set $\{x \mid x \geq 0\}$ of all *non-negative reals*, is projectively ***While*** semicomputable. From the countability/cofiniteness condition (6.12) however, it is not (even ***While****) semicomputable on $\mathcal{R}$. From this it is easy to see that the order relation

$$x < y \iff (y - x) \in R \text{ and } x \neq y$$

is also projectively ***While*** semicomputable, but not ***While**** semicomputable, on $\mathcal{R}$.

6.3 The ordered algebra of reals; sets of reals which are *While* semicomputable but not *While** computable

In the previous subsection we saw that the order relation on $\mathbb{R}$ is not (even) ***While**** semicomputable on the algebra $\mathcal{R}$. Let us add it now to $\mathcal{R}$, to form the algebra $\mathcal{R}^<$, and see how this affects the computability theory.

We begin again with a restatement of the semicomputability equivalence theorem (6.4), this time for the algebra $\mathcal{R}^<$.

Theorem 6.16 (Semicomputability for $\mathcal{R}^<$). *Suppose $R \subseteq \mathbb{R}^n$. Then the following are equivalent:*

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

$$x \in R \iff \bigvee_i \boldsymbol{b}_i(x)$$

where each $\boldsymbol{b}_i(x)$ is a finite conjunction of equations and inequalities of the form

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

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

Proof. From Theorem 6.4 and Lemma 6.5. ∎

To proceed further, we need some definitions and lemmas about points and sets defined by polynomials. (Background information on algebraic geometry can be found, for example, in Shafarevich [1977] or Bröcker and

Lander [1975, Chapter 12]. For the application below (section 6.4), we relativise our concepts to an arbitrary subset D of $\mathbb{R}$.

Definition 6.17.

(a) An interval in $\mathbb{R}$ (open, half-open or closed) is *algebraic in D* if, and only if, its end-points are.

(b) A *patch* in $\mathbb{R}$ is a finite union of points and intervals.

(c) A *D-algebraic patch* in $\mathbb{R}$ is a finite union of points and intervals algebraic in D.

Definition 6.18. A set in $\mathbb{R}^n$ is *D-semialgebraic* if, and only if, it can be defined as a finite disjunction of finite conjunctions of equations and inequalities of the form

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

where p and q are polynomials in x with coefficients in $\mathbb{Z}[D]$.

We will drop the 'D' when it denotes the empty set.

Note that clause (iii) of Theorem 6.16 can, by Definition 6.18, be written equivalently in the form:

(iii') *R can be expressed as an effective countable union of semialgebraic sets.*

Lemma 6.19. *A semialgebraic set in $\mathbb{R}^n$ has a finite number of connected components.*

(See Becker [1986]. This will be used in section 6.6.) It follows that a semialgebraic subset of $\mathbb{R}$ is a patch. However, for $n = 1$ we need a stronger result:

Lemma 6.20. *A subset of $\mathbb{R}$ is D-semialgebraic if, and only if, it is a D-algebraic patch.*

Lemma 6.21. *A projection of a D-semialgebraic set in $\mathbb{R}^n$ on $\mathbb{R}^m$ ($m < n$) is again D-semialgebraic.*

This follows from Tarski's quantifier-elimination theorem for real closed fields (see, for example, Kreisel and Krivine [1971, Chapter 4]). From this and Lemma 6.20:

Corollary 6.22. *A projection of a D-semialgebraic set in $\mathbb{R}^n$ on $\mathbb{R}$ is a D-algebraic patch.*

Remark 6.23. Corollaries 6.9 and 6.10 (the density/codensity condition) hold for $\mathcal{R}^<$ as well as $\mathcal{R}$, leading to the same examples (6.11) of subsets of $\mathbb{R}^n$ which are $\boldsymbol{While}^N$ semicomputable, but not $\boldsymbol{While}^N$ ($= \boldsymbol{While}^*$) computable, on $\mathcal{R}^<$. Another example is given below (6.26).

The following corollary, however, points out a contrast with $\mathcal{R}$.

Corollary 6.24. *In $\mathcal{R}^<$, the following three notions coincide for subsets of $\mathbb{R}^n$:*

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

This follows from Theorem 6.16 and Lemma 6.21. (This corollary fails in the structure $\mathcal{R}$, since in that structure, Lemma 6.21, depending on Tarski's quantifier-elimination theorem, does not hold.)

However, the above three notions of semicomputability differ in $\mathcal{R}^<$ from a fourth:

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

as we will see in section 6.4. But first we need:

Corollary 6.25 (Countable connectivity condition).

(a) If $R \subseteq \mathbb{R}$ is ***While**** *semicomputable on $\mathcal{R}^<$, then R consists of countably many connected components.*

(b) If $R \subseteq \mathbb{R}$ is ***While**** *semicomputable on $\mathcal{R}^<$, then either R is countable or R contains an interval.*

(This is a refinement of the countability/cofiniteness condition (6.12).) This follows from Theorem 6.16 and Lemma 6.19, since the connected subsets of $\mathbb{R}$ are precisely the singletons and the intervals.

Example 6.26.

(*a*) The Cantor set in $[0, 1]$ is not ***While**** semicomputable on $\mathcal{R}^<$, by the countable connectivity condition.

(*b*) The complement of the Cantor set in $[0, 1]$ is ***While*** semicomputable on $\mathcal{R}^<$ (*Exercise*), but (by (*a*)) it is not (even ***While****) computable on $\mathcal{R}^<$.

Other interesting examples of semicomputable, non-computable sets are given in sections 6.5 and 6.6.

6.4 A set which is projectively *While** semicomputable but not projectively *While*^{N} semicomputable

First we must enrich the structure $\mathcal{R}^<$. Let $E = \{e_0, e_1, e_2, \ldots\}$ be a sequence of reals such that

$$\text{for all } i,\ e_i \text{ is transcendental over } \mathbb{Q}(e_0, \ldots, e_{i-1}). \tag{6.4}$$

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

algebra	$\mathcal{R}^{<,E}$
import	$\mathcal{R}^<$
carriers	E
functions	$j : E \hookrightarrow \mathbb{R}$
end	

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

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

Theorem 6.27. *Let $F \subseteq \bar{E}$ be projectively $\boldsymbol{While}^N$ semicomputable in $\mathcal{R}^{<,E}$. Then $F \neq \bar{E}$. Specifically, suppose for some $\boldsymbol{While}$ computable function φ on $\mathcal{R}^{<,E,N}$:*

$$F = \{x \in \mathbb{R} \mid \ (\exists y \in E^r)(\exists z \in \mathbb{R}^s)(\exists u \in \mathbb{N}^k)(\exists v \in \mathbb{B}^l)\, \varphi(x,y,z,u,v) \downarrow\} \tag{6.5}$$

(with existential quantification over all four sorts in $\mathcal{R}^{<,E,N}$). Then for all $x \in F$, x is algebraic over some subset of E of cardinality r (= the number of arguments of φ of sort E in (6.5)).

The rest of this subsection is a sketch of the proof.

Lemma 6.28. *(In the notation of the Theorem 6.27,) F can be represented as a countable union of the form $F = \bigcup_{i=0}^{\infty} F_i$, where*

$$F_i = \{x \mid (\exists y \in E^r)(\exists z \in \mathbb{R}^s)\boldsymbol{b}_i(x,y,z)\}$$

and $\boldsymbol{b}_i$ is a finite conjunction of equations and inequalities of the form

$$p(x,y,z,) = 0 \qquad \textit{and} \qquad q(x,y,z) > 0$$

where p and q are polynomials in x, y, z with coefficients in $\mathbb{Z}$.

Proof. Apply Engeler's lemma. Also replace existential quantification over nat and bool by countable disjunctions. ∎

Lemma 6.29. *(In the notation of Lemma 6.28,) for any r-tuple $e = (e_{i_1}, \dots, e_{i_r})$ of elements of E, put*

$$F_i[e] =_{df} \{x \mid (\exists z \in \mathbb{R}^s)\boldsymbol{b}_i(x,e,z)\}.$$

Then for all $e \in E^r$, $F_i[e]$ is a (finite) set of points, all algebraic in e.

Proof. Note that $F_i[e]$ is a projection on R of an e-semialgebraic set in $\mathbb{R}^{s+1}$. Hence, by Corollary 6.22, it is an e-algebraic patch. Since by assumption

$$F_i[e] \subseteq F \subseteq \bar{E},$$

$F_i[e]$ is countable, and hence cannot contain any (non-degenerate) interval. The result follows from the definition of e-algebraic patch. ∎

Since F is the union of $F_i[e]$ over all i, and all r-tuples e from E, the theorem follows from Lemma 6.29 and the following:

Lemma 6.30. *For all n, there exists a real which is algebraic over E but not over any subset of E of cardinality n.*

Proof. Take $x = e_0 + e_1 + \ldots + e_n$ (more strictly, $j(e_0) + \ldots + j(e_n)$). The result follows from the construction (6.4) of E. ∎

We have shown that $\bar{E}$ (although a projection on $\mathbb{R}$ of a $\boldsymbol{While}$ semicomputable relation on $\mathbb{R} \times E^*$) is not a projection of a $\boldsymbol{While}^N$ semicomputable relation in $\mathcal{R}^{<,E}$. In fact, we can see (still using Engeler's lemma) that $\bar{E}$ is not even a projection of a $\boldsymbol{While}^*$ semicomputable relation on $\mathbb{R}^n \times E^m$ (for any $n, m > 0$). Thus to define $\bar{E}$, we must project off the *starred sort* E^*, or (in other words) existentially quantify over a *finite, but unbounded* sequence of elements of E.

6.5 Dynamical systems and chaotic systems on $\mathbb{R}$; sets which are $\boldsymbol{While}^N$ semicomputable but not $\boldsymbol{While}^*$ computable

We will examine algorithmic aspects of certain dynamical systems. Many physical, biological and computing systems are deterministic and share a common mathematical form.

Consider a deterministic system (S, F) modelled by means of a set S of states, whose dynamical behaviour in discrete time is given by a *system function*

$$F : \mathbb{T} \times S \to S$$

where $\mathbb{T} = \mathbb{N} = \{0, 1, 2, \ldots\}$ and for $t \in \mathbb{T}$ and $s \in S$, $F(t, s)$ is the state of the system at time t given initial state s.

The *orbit* of F at state s is the set

$$\boldsymbol{Orb}(F, s) \;=\; \{F(t, s) \mid t \in \mathbb{T}\}.$$

The set of *periodic points of* F is

$$\boldsymbol{Per}(F) \;=\; \{s \in S \mid \exists t \in \mathbb{T}(F(t, s) = s)\}.$$

In modelling a dynamical system (S, F), the computability of the F and of sets such as the orbits and periodic points is of immediate interest and importance.

Now suppose, more specifically, that the evolution of the system in time is determined by a *next state function*

$$f : S \to S$$

through the equations

$$\begin{aligned} F(0, s) &= s \\ F(t+1, s) &= f(F(t, s)) \end{aligned}$$

which have the solution

$$F(t,s) = f^t(s)$$

for $t \in \mathbb{T}$ and $s \in S$. We call such systems *iterated maps.* In this case, we write

$$\begin{array}{lclcl} \boldsymbol{orb}(f,s) & = & \boldsymbol{Orb}(F,s) & = & \{f^t(s) \mid t \in \mathbb{T}\} \\ \boldsymbol{per}(f) & = & \boldsymbol{Per}(F) & = & \{s \in S \mid \exists t > 0(F(t,s) = s)\}. \end{array}$$

Theorem 6.31. *Let A be an N-standard algebra (with $\mathbb{N} = \mathbb{T}$), and containing the state space S. If the next state function f is **While** computable on A then so is the system function F. Furthermore, the orbits **orb**(f,s) and the set of periodic points **per**(f) are **While** semicomputable on A.*

Proof. By computability of primitive recursion (Theorem 8.5) and closure of semicomputability under existential quantification over $\mathbb{N}$ (Theorem 5.23). ∎

Now we will consider the computability of some simple dynamical systems with one-dimensional state spaces. More specifically, suppose the state space is an interval

$$S \;=\; I \;=\; [a,b] \;\subseteq\; \mathbb{R}$$

and so the next state function and system function have the form

$$\begin{array}{l} f: I \to I \\ F: \mathbb{T} \times I \to I. \end{array}$$

F is called an *iterated map on the interval I.* Dynamical systems based on such maps have a wide range of uses and a beautiful theory. For example, such systems will under certain circumstances exhibit 'chaos'. The following discussion is taken from Devaney [1989]. Let (I,F) be a dynamical system based on the iterated map F.

Definition 6.32.

(a) (I,F) is *sensitive to initial conditions* if there exists $\delta > 0$ such that for all $x \in I$ and any neighbourhood U of x, there exist $y \in U$ and $t \in \mathbb{T}$ such that
$$|F(t,x) - F(t,y)| > \delta.$$

(b) (I,F) is *topologically transitive* if for any open sets U_1 and U_2 there exist $x \in U_1$ and $t \in \mathbb{T}$ such that $F(t,x) \in U_2$.

Note that if I is compact then (I,F) is topologically transitive if, and only if, $\boldsymbol{Orb}(F,x)$ is dense in I for some $x \in I$. (The direction '$\Leftarrow$' is clear. The proof of '$\Rightarrow$' depends on the Baire category theorem.)

Definition 6.33. The system (I, F) is *chaotic* if:

(a) it is sensitive to initial conditions;

(b) it is topologically transitive;

(c) the set $\boldsymbol{Per}(F)$ of periodic points of F is dense in I.

Consider the *quadratic family* of dynamical systems (I, F_μ) for μ real, where $I = [0, 1]$ and the next state function is

$$f_\mu(x) \ = \ \mu x(1-x).$$

For $\mu = 4$ we have:

Theorem 6.34. *The system (I, F_4) is chaotic. Thus, for the algebras $\mathcal{R}$ and $\mathcal{R}^<$:*

(a) for some $x \in [0,1]$, the set $\boldsymbol{Orb}(F_4, x)$ is $\boldsymbol{While}^N$ semicomputable but not $\boldsymbol{While}^N$ ($= \boldsymbol{While}^*$) computable;

(b) the set $\boldsymbol{Per}(F)$ is $\boldsymbol{While}^N$ semicomputable, but not $\boldsymbol{While}^N$ ($= \boldsymbol{While}^*$) computable.

Proof. That (I, F_4) is chaotic is proved in Devaney [1989]. The semicomputability of f_4 is clear. Semicomputability of $\boldsymbol{Orb}(F_4, x)$ and $\boldsymbol{Per}(F)$ follows from Theorem 6.31. Further, it can be shown that $\boldsymbol{Orb}(F_4, x)$ and $\boldsymbol{Per}(F)$ are both dense and codense in I. (*Exercise.*) Non-computability then follows from the density/codensity condition (see Remark 6.23). ∎

6.6 Dynamical systems and Julia sets on $\mathbb{C}$; sets which are $\boldsymbol{While}^N$ semicomputable but not $\boldsymbol{While}^*$ computable

We reconsider an example from Blum *et al.* [1989], and show how it follows from our general theory of semicomputability. We work from now on in $\mathcal{C}^<$. First, we must relate computability in the complex and real algebras. We consider the algebras $\mathcal{C}$ and $\mathcal{C}^<$.

Notation 6.35. If $S \subseteq \mathbb{C}^n$, we write

$$\hat{S} \ =_{df} \ \{(\mathsf{re}(z_1), \mathsf{im}(z_1), \ldots, \mathsf{re}(z_n), \mathsf{im}(z_n)) \mid (z_1, \ldots, z_n) \in \mathbb{C}^n\} \ \subseteq \ \mathbb{R}^{2n}.$$

Lemma 6.36 (Reduction lemma). *Let $S \subseteq \mathbb{C}^n$.*

(a) S is $\boldsymbol{While}$ (or $\boldsymbol{While}^N$ or $\boldsymbol{While}^*$) semicomputable in $\mathcal{C}$ if, and only if, $\hat{S}$ is $\boldsymbol{While}$ (or $\boldsymbol{While}^N$ or $\boldsymbol{While}^*$) semicomputable in $\mathcal{R}$.

(b) S is $\boldsymbol{While}$ (or $\boldsymbol{While}^N$ or $\boldsymbol{While}^*$) semicomputable in $\mathcal{C}^<$ if, and only if, $\hat{S}$ is $\boldsymbol{While}$ (or $\boldsymbol{While}^N$ or $\boldsymbol{While}^*$) semicomputable in $\mathcal{R}^<$.

This lemma will enable us to reduce problems of semicomputability in the algebras $\mathcal{C}$ or $\mathcal{C}^<$ to those in the corresponding real algebras. For example, from this lemma and Corollary 6.24 we have:

Corollary 6.37. *In $\mathcal{C}^<$, the notions of $\boldsymbol{While}^N$, $\boldsymbol{While}^*$ and projective $\boldsymbol{While}^N$ semicomputability all coincide for subsets of $\mathbb{C}^n$.*

Note that the reduction lemma would not be true if we included the mod function ($z \mapsto |z|$) in $\mathcal{C}$ or $\mathcal{C}^<$, by Example 6.3(b).

We work from now on in $\mathcal{C}^<$.

Let $g : \mathbb{C} \to \mathbb{C}$ be a function. For $z \in \mathbb{C}$, the *orbit of g at z* (as in section 6.5) is the set

$$\boldsymbol{orb}(g, z) \;=\; \{g^n(z) \mid n = 0, 1, 2, \ldots\}.$$

Let

$$U(g) \;=\; \{z \in \mathbb{C} \mid \boldsymbol{orb}(g, z) \text{ is unbounded}\}$$

and

$$\begin{aligned} F(g) \quad &= \{z \in \mathbb{C} \mid \boldsymbol{orb}(g, z) \text{ is bounded}\} \\ &= \mathbb{C} \backslash U(g). \end{aligned}$$

The set $F(g)$ is the *filled Julia set* of g; the boundary $J(g)$ of $F(g)$ is the *Julia set* of g.

For any $r \in \mathbb{R}$ define

$$V_r(g) \;=\; \{z \in \mathbb{C} \mid \exists n(|g^n(z)| > r)\}.$$

Clearly, $U(g) \subseteq V_r(g)$ for all r.

Theorem 6.38. *For $g(z) = z^2 - c$, with $|c| > 4$, we have $U(g)$ is $\boldsymbol{While}$ semicomputable but not (even $\boldsymbol{While}^*$) computable. Thus, $F(g)$ is not $\boldsymbol{While}^*$ semicomputable.*

Proof. Assume for now that $|c| \geq 1$, and choose $r = 2|c|$. Then for $|z| > r$,

$$|g(z)| \;=\; |z^2 - c| \;\geq\; |z|^2 - |c| \;\geq\; \frac{3}{2}|z|.$$

Hence for all n,

$$|g^n(z)| \;\geq\; \left(\frac{3}{2}\right)^n |z|,$$

and so

$$g^n(z) \to \infty \;\text{ as }\; n \to \infty.$$

Hence for such r, $V_r(g) \subseteq U(g)$, and so

$$U(g) \;=\; V_r(g) \;=\; \{z \in \mathbb{C} \mid \exists n(|g^n(z)| > r)\}.$$

To show that $U(g)$ is semicomputable is routine; for example, as the halting set of the procedure

```
proc in   a: complex
     aux b: complex
begin
     b:= a;
     while  | b |$^2 \leq 4|c|^2$  do  b := b$^2 - c$  od
end.
```

(Note that although the function $z \mapsto |z|$ is not computable, the function $z \mapsto |z|^2 = \mathsf{re}(z)^2 + \mathsf{im}(z)^2$ is.)

To conclude the proof we must show that $F(g)$ is not ***While**** semicomputable. Suppose it was, then (by the countable connectivity condition and the reduction lemma) it would consist of countably many connected components. But if we choose $|c| > 4$ it can be shown that $F(g)$ is compact, totally disconnected and perfect, i.e., homeomorphic to the Cantor set (see, for example, Hocking and Young [1961]), and so we have a contradiction (cf. Example 6.26(a)). ∎

7 Computation on topological partial algebras

We have considered ***While*** computations on algebras of reals in section 6. Connections were made between notions of semicomputability and familiar rational polynomial definability; we also made some observations on connections between projective semicomputability and field extensions of $\mathbb{Q}$.

There is thus a close relationship between *computability* properties, and *algebraic* properties of sets of reals. (Of course many of these properties can be reformulated for arbitrary rings and fields.)

In this section we explore the relationship between *computability* properties and *topological* properties of sets of reals. We will analyse ***While*** computations on general topological algebras, and using these general concepts and results, we will be able to give a quick guide to the primary case of computation on $\mathbb{R}$.

The outline of this section is as follows. In section 7.1 we indicate the basic problem: although computability implies *continuity* (to be proved later), total Boolean-valued functions on $\mathbb{R}$ such as equality and order are *discontinuous*. The solution is to work with *partial functions*. We therefore define *partial algebras* in section 7.2, and *topological partial algebras* in section 7.3. In section 7.4 we compare the two approaches to computation on the reals: the *algebraic model* of section 6, and the *stream model* which lies behind the models studied in this section. In section 7.5 we prove that computable functions are continuous, from which it follows that semicomputable or projectively semicomutable sets are open, and hence computable sets are clopen (= closed and open). In section 7.6 we infer a converse of this last statement in the case of compact algebras with open subbases of semicomputable sets. In section 7.7 we specialise to metric partial algebras, and in section 7.8 show the equivalence between computability and explicit definability in the case of a connected domain. This result is used in the study of *approximable computability* in section 7.9, in which *effective Weierstrass computability* (generalising the classical notion of Weierstrass approximability) is shown to be equivalent on connected domains (under certain broad assumptions) to *effective uniform* $\boldsymbol{While}$ *(or* $\boldsymbol{While}^*$*) approximability.* Finally, in section 7.10, we discuss the relationship between abstract and concrete models of computability, with particular reference to computation on the reals.

The material of this section is based on Tucker and Zucker [1999]. Background information on topology can be found in any standard text, such as Kelley [1955], Hocking and Young [1961], Simmons [1963] or Dugundji [1966].

7.1 The problem

Consider again the standard algebras

$$\mathcal{R} \;=\; (\mathbb{R},\, \mathbb{B};\, 0,\, 1,\, +,\, -,\, \times,\, \mathsf{if}_{\mathsf{real}},\, \mathsf{eq}_{\mathsf{real}}, \ldots)$$

and $\mathcal{R}^<$, which extends $\mathcal{R}$ with $\mathsf{less}_{\mathsf{real}}$ (or '$<$').

Not all the functions in $\boldsymbol{While}(\mathcal{R})$ and $\boldsymbol{While}(\mathcal{R}^<)$ are continuous. This is obvious, because both algebras contain certain basic operations, namely $\mathsf{eq}_{\mathsf{real}}$ and $\mathsf{less}_{\mathsf{real}}$ ('$=$' and '$<$'), that are not continuous (with respect to the usual topology on $\mathbb{R}$).

> *If A is an algebra built on $\mathbb{R}$ such that all its basic operations are continuous, then is every function in $\boldsymbol{While}(A)$ continuous?*

Let us immediately consider this question more generally.

Definition 7.1.

(1) A *topological (total) Σ-algebra* is a pair $(A, \mathcal{T})$, where A is a Σ-algebra and $\mathcal{T}$ is a family $\langle \mathcal{T}_s \mid s \in \boldsymbol{Sort}(\Sigma)\rangle$, where for each $s \in \boldsymbol{Sort}(\Sigma)$, $\mathcal{T}_s$ is a topology on A_s, such that for each basic function symbol $F : u \to s$ of Σ, the function $F^A : A^u \to A_s$ is continuous.

(2) A *standard total topological algebra* $(A, \mathcal{T})$ is a total topological algebra in which A is standard, and the carrier $A_{\mathtt{bool}} = \mathbb{B}$ has the discrete topology.

We will often speak of a 'topological algebra A', without stating the topology explicitly.

Remark 7.2. In a topological algebra, the carriers of all *equality sorts* must be discrete, in order for the equality operation on them to be continuous. In particular, if A is N-standard, then the carrier $A_{\mathtt{nat}} = \mathbb{N}$ must be discrete.

To provide motivation, we state the following theorem here. (It will be formulated and proved later in a more general context.)

Theorem 7.3. *Let A be a standard topological algebra.*

(a) If $f \in \boldsymbol{While}(A)$ then f is continuous on A.
(b) If $f \in \boldsymbol{While}^N(A)$ then f is continuous on A.
(c) If $f \in \boldsymbol{While}^*(A)$ then f is continuous on A.

At first sight this gives a satisfactory answer to the above question about continuity of ***While*** computable functions. However, a standard total topological algebra based on $\mathbb{R}$ has the following problem. There can be no non-constant basic operations of the form $F : \mathbb{R}^q \to \mathbb{B}$ such as '$<$' or even '$=$'. This is because if $f : \mathbb{R}^q \to \mathbb{B}$ is continuous, then $f^{-1}[\mathsf{tt}]$ and $f^{-1}[\mathsf{ff}]$ are disjoint open sets whose union is $\mathbb{R}^q$. So one must be $\mathbb{R}^q$, and the other $\emptyset$, by the connectness of $\mathbb{R}$. (We investigate connectedness in topological algebras in section 7.8.)

Hence the problem with the above theorem is the paucity of useful applications. In fact, the only continuous equality test is on a *discrete space*.

However, equality and order on $\mathbb{R}$ do have some properties close to continuity. For example, given two points x and y with $x < y$, there are disjoint neighbourhoods U_x and U_y of x and y respectively such that for all $u \in U_x$ and $v \in U_y$, $u < v$. (Similarly for inequality '$\neq$'.)

We will develop notions that allow us to express these 'continuity' properties as follows. Define *partial functions*

$$\begin{aligned} \mathsf{less}_p &: \mathbb{R}^2 \to \mathbb{B} \\ \mathsf{eq}_p &: \mathbb{R}^2 \to \mathbb{B} \end{aligned}$$

so that

$$\mathsf{less}_p(x,y) \;=\; \begin{cases} \mathsf{tt} \text{ if } x < y \\ \mathsf{ff} \text{ if } x > y \\ \uparrow \text{ if } x = y, \end{cases}$$

and

$$\mathsf{eq}_p(x,y) \;=\; \begin{cases} \uparrow \text{ if } x = y \\ \mathsf{ff} \text{ if } x \neq y. \end{cases}$$

These partial functions are continuous, in the sense that the inverse images of $\{\mathsf{tt}\}$ and $\{\mathsf{ff}\}$ are always open subsets of $\mathbb{R}^2$.

We will exploit these observations about '$<$' and '$=$' to the full by studying *topological partial algebras.* We will also prove a more general version of Theorem 7.3 for such partial algebras (Theorem 7.12).

7.2 Partial algebras and *While* computation

A *partial* Σ*-algebra* is defined in the same way as a Σ-algebra (section 2.3), except that for each $F : u \to s$ in $\boldsymbol{Func}(\Sigma)$, the function $F^A : A^u \to A_s$ may be partial.

Standard and *N-standard* partial Σ-algebras are defined analogously, as are the *standardisation* and *N-standardisation* of partial Σ-algebras (cf. sections 2.4, 2.5).

Suppose Σ is a standard signature, and A is a standard partial Σ-algebra.

The *error partial algebra* A^{u}, of signature Σ^{u}, is constructed as before (cf. section 2.6). In particular, for each $F \in \boldsymbol{Func}(\Sigma)$, its interpretation F^A on A is extended by *strictness* to a partial function $F^{A,\mathsf{u}}$ on A^{u}.

The *array partial algebra* A^*, of signature Σ^*, is constructed as before (cf. section 2.7).

The *stream partial algebra* $\bar{A}$, of signature $\overline{\Sigma}$, is constructed as before (cf. section 2.8).

The *semantics* of ***While*** programs on A is similar to that for total algebras (cf. sections 3.3–3.8), except that many of the semantic functions are now *partial*, namely: the term evaluation function $[\![t]\!]^A$ (section 3.3), the functions $\langle\!|\mathrm{S}|\!\rangle^A$, $\boldsymbol{Comp}_1^A$, $\boldsymbol{Comp}^A$, $\boldsymbol{CompLength}^A$(section 3.4), and (as before) the statement evaluation function $[\![S]\!]^A$ (section 3.5) and procedure evaluation function P^A (section 3.6). For example, the definition of $[\![t]\!]^A$becomes (cf. section 3.3):

$$\begin{aligned} [\![\mathtt{x}]\!]^A\sigma &= \sigma(\mathtt{x}) \\ [\![F(t_1,\dots,t_m)]\!]^A\sigma &\simeq F^A([\![t_1]\!]^A\sigma,\dots,[\![t_m]\!]^A\sigma). \end{aligned}$$

Here the second clause is interpreted as

$$[\![F(t_1,\dots,t_m)]\!]^A\sigma \simeq \begin{cases} F^A([\![t_1]\!]^A\sigma,\dots,[\![t_m]\!]^A\sigma) & \text{if } [\![t_i]\!]^A\sigma\downarrow \\ & \text{for } i=1,\dots,m \\ \uparrow & \text{otherwise.} \end{cases}$$

except for the case that $F(\dots)$ is the discriminator $\mathsf{if}(b,t_1,t_2)$, in which case we have a 'non-strict' computation of *either* $[\![t_1]\!]^A\sigma$ *or* $[\![t_2]\!]^A\sigma$, depending on the value of $[\![b]\!]^A\sigma$:

$$[\![\mathsf{if}(b,t_1,t_2)]\!]^A\sigma \simeq \begin{cases} [\![t_1]\!]\sigma & \text{if } [\![b]\!]^A\sigma\downarrow \mathtt{tt} \\ [\![t_2]\!]\sigma & \text{if } [\![b]\!]^A\sigma\downarrow \mathtt{ff} \\ \uparrow & \text{if } [\![b]\!]^A\sigma\uparrow. \end{cases}$$

The results in sections 3.3–3.10 (*functionality* lemmas, *homomorphism invariance* and *locality* theorems) still hold, with certain obvious modifications related to divergence. For example, the functionality lemma for temrs (3.4) becomes:

Lemma 7.4 (Functionality lemma for terms). *For any term t and states σ_1 and σ_2, if $\sigma_1 \approx \sigma_2$* (rel $\boldsymbol{vart}$), *then either*

(i) $[\![t]\!]^A\sigma_1\downarrow$ *and* $[\![t]\!]^A\sigma_2\downarrow$ *and* $[\![t]\!]^A\sigma_1 = [\![t]\!]^A\sigma_2$, *or*

(ii) $[\![t]\!]^A\sigma_1\uparrow$ *and* $[\![t]\!]^A\sigma_2\uparrow$.

7.3 Topological partial algebras

Note that in, this section, by 'function' we generally mean *partial function.*

Definition 7.5. Given two topological spaces X and Y, a function $f : X \to Y$ is *continuous* if for every open $V \subseteq Y$, $f^{-1}[V] =_{df} \{x \in X \mid x \in \boldsymbol{dom}(f) \text{ and } f(x) \in Y\}$ is open in X.

Definition 7.6.

(1) A *topological partial* Σ*-algebra* is a partial Σ-algebra with topologies on the carriers such that each of the basic functions is continuous.

(2) A *standard topological partial algebra* is a topological partial algebra which is also a standard partial algebra, such that the carrier $\mathbb{B}$ has the discrete topology. (Cf. Definition 7.1.)

Examples 7.7.

(a) (*Real algebra.*) An important standard topological partial algebra for our purpose is the algebra

$$\mathcal{R}_p \;=\; (\mathbb{R},\, \mathbb{B};\; 0,\, 1,\, +,\, -,\, \times, \mathsf{if}_{\mathsf{real}},\, \mathsf{eq}_p,\, \mathsf{less}_p, \ldots)$$

which is formed from $\mathcal{R}^{<}$ by the replacement of $\mathsf{eq}_{\mathsf{real}}$ and $\mathsf{less}_{\mathsf{real}}$ by the partial operations eq_p and less_p (defined in section 7.1). It becomes a *topological partial algebra* by giving $\mathbb{R}$ its usual topology, and $\mathbb{B}$ the discrete topology. An open base for the standard topology on $\mathbb{R}$ is given by the collection of open intervals with rational endpoints. These intervals are all ***While*** semicomputable on $\mathcal{R}_p$. (*Exercise.*)

(b) (*Interval algebras.*) Another useful class of topological partial algebras are of the form

```
algebra     $\mathcal{I}_p$
import      $\mathcal{R}_p$
carriers    $I$
functions   $\mathsf{i}_I \;:\; I \to \mathbb{R}$,
            $F_1 \;:\; I^{m_1} \to I$,
               $\ldots$
            $F_k \;:\; I^{m_k} \to I$
end
```

where I is the closed interval $[0,1]$ (with its usual topology), i_I is the embedding of I into $\mathbb{R}$, and $F_i \;:\; I^{m_i} \to I$ are continuous partial functions. These are called *(partial) interval algebras on* I.

Example 7.8 (*While* computable functions on $\mathcal{R}_p$). We give two examples of functions computable by ***While*** programs, using the above Boolean-valued functions (eq_p and less_p) as tests. (In both cases, the inputs are taken to be positive reals to simplify the programs, although the programs could easily be modified to apply to all reals, positive and non-positive).

(*a*) The characteristic function of $\mathbb{Z}$ on $\mathbb{R}$, $\mathsf{is\_int} : \mathbb{R}^+ \to \mathbb{B}$, where

$$\mathsf{is\_int}(x) = \begin{cases} \uparrow & \text{if } x \text{ is an integer} \\ \mathsf{ff} & \text{otherwise.} \end{cases}$$

This is defined by the procedure

```
proc in   x : pos-real
     out  b : bool
begin
     b : =false;
     while  x>0   {if x = 0, test diverges!}
          do  x := x-1
          od
end
```

(*b*) The truncation function $\mathsf{trunc} : \mathbb{R}^+ \to \mathbb{Z}$, where

$$\mathsf{trunc}(x) = \begin{cases} \lfloor \mathtt{x} \rfloor & \text{if } x \text{ is not an integer} \\ \uparrow & \text{otherwise.} \end{cases}$$

The procedure for this is similar:

```
proc in   x : pos-real
     out  c : int
begin
     c := 0;
     while  x > 1   {if x = 1, test diverges}
          do  x := x−1;
              c := c+1
          od
end
```

Until further notice (section 7.8) let A be a standard topological partial Σ-algebra.

Definition 7.9 (Expansions of topological partial algebra).

(*a*) The topological partial algebra A^{u}, of signature Σ^{u}, is constructed from A by giving each new carrier A^{u}_s the *disjoint union* topology of A_s and $\{\mathsf{ui}\}$. (This makes ui an isolated point of A^{u}_s.)

(*b*) The topological partial algebra A^N, of signature Σ^N, is constructed from A by giving the new carrier $\mathbb{N}$ the discrete topology.

(c) The topological partial algebra A^*, of signature Σ^*, is constructed from A^N as follows. Viewing the elements of each new carrier A^*_s as (essentially) *infinite sequences* of elements of A^{u}_s, which take the value ui for all indices greater than $\mathsf{Lgth}(a^*)$, we give A^*_s the subspace topology of the set $(A^{\mathsf{u}}_s)^{\mathbb{N}}$ of all infinite sequences from A^{u}_s, with the *product topology* over A^{u}_s. Equivalently, viewing the elements of A^*_s as (essentially) arrays of elements of A^{u}_s of finite length, we can give A^*_s the *disjoint union* topology of the sets $(A^{\mathsf{u}}_s)^n$ of arrays of length n, for all $n \geq 0$, where each set $(A^{\mathsf{u}}_s)^n$ is given the *product topology* of its components A^{u}_s. It is easy to check that A^* is indeed a topological algebra, i.e., all the new functions of A^* are continuous.

(d) The topological partial algebra $\bar{A}$, of signature $\overline{\Sigma}$, is constructed by giving each new carrier $\bar{A}_s$ the *product topology* over A_s. Note that, if A_s is *compact* for any sort s, then so is $\bar{A}_s$, by Tychonoff's theorem (see Remark 7.28).

Definition 7.10. A is *Hausdorff* if each carrier of A is Hausdorff (i.e., any distinct pair of points can be separated by disjoint neighbourhoods.)

Proposition 7.11. *If A is Hausdorff, then so are the expansions A^{u}, A^N, A^* and $\bar{A}$.*

Theorem 7.12. *Let A be a standard topological partial algebra.*

*(a) If $f \in$ **While**(A) then f is continuous on A.*

*(b) If $f \in$ **While**$^N(A)$ then f is continuous on A.*

*(c) If $f \in$ **While**$^*(A)$ then f is continuous on A.*

The proof will be given in section 7.5. For now we observe that this theorem implies the following.

Theorem 7.13. *If R is*

*(a) **While*** semicomputable on A, or*

*(b) projectively **While*** semicomputable on A,*

then R is open in A.

Proof.

(a) Suppose R is the halting set of a ***While**** computable function $f : A^u \to A_s$. By Theorem 7.12, f is continuous. Hence $R = f^{-1}[A_s]$ is open.

(b) From (a) and since a projection of an open set is open. (*Check.*) ∎

Note that in the above proof, we used the fact that a projection of an open set is open. (*Check.*)

Corollary 7.14. *A **While*** computable relation on A is clopen in A.*

Proof. By Post's theorems (5.40) and Theorem 7.13. ∎

7.4 Discussion: Two models of computation on the reals

The purpose of this subsection is to explain the conceptual background for our models of computation on the reals.

There are two types of models of reals, and computations on them:

(1) *The algebraic model.* Here we work with a many-sorted algebra like

$$\mathcal{R} \;=\; (\mathbb{R}, \mathbb{N}, \mathbb{B};\; 0, 1, +, \times, \ldots),$$

This was the approach in section 6.

(2) *The stream model.* A real number input or output is given as a *stream* of

(i) digits (representing a decimal expansion), or
(ii) rationals (representing a Cauchy sequence), or
(iii) integers (representing a continued fraction).

This idea lies behind the partial algebras of reals $\mathcal{R}_p$ and $\mathcal{I}_p$ studied in this section. For convenience, we concentrate on (i). (The decimal representation may be to any base.)

Then a procedure for a computable real-valued function $f : \mathbb{R}^n \to \mathbb{R}$ has as *input* n streams of digits, and as *output* a stream of digits. Similarly a procedure for a computable relation on the reals, or Boolean-valued function, $R : \mathbb{R}^n \to \mathbb{B}$ has as *input* n streams of digits, and as *output* a Boolean value (or bit).

In the algebraic approach, the input and output reals are just ‘points’ (elements of $\mathbb{R}$) given in one step. Continuity of the computable functions, or even of the primitive functions (with respect to the usual topology on $\mathbb{R}$), is *not* forced on us — and our models in section 6 violated it.

In the stream model, however, the reals form *infinite* data; at any finite time, only a finite part has been processed (written or read). *Continuity* of the computable functions (which we will prove formally in the next subsection) is then forced on us conceptually by computability requirements, i.e.,:

(a) For $f : \mathbb{R}^n \to \mathbb{R}$ to be computable, we must be able to get the output real (= stream of digits) to any desired degree of accuracy (= length) by inputting sufficiently long input streams. (Briefly: the longer the inputs, the longer the output.)

(b) For $R : \mathbb{R}^n \to \mathbb{B}$ to be computable, we must be able to get an output bit after finite (suficiently long) input streams.

We often work with the algebraic model, because of its simplicity. It is a good source of examples to distinguish various notions of abstract computability and semicomputability (as we saw in section 6). However, the stream model is more satisfying conceptually, conforming to our intuition of reals as they occur to us, e.g., in physical measurements and calculations. So we can use the stream model as a source of insights for our requirements or assumptions regarding the algebraic model, notably the *continuity requirement* for computable functions.

Recall the problem discussed in Section 7.1 concerning the continuity requirement for computable relations, i.e., Boolean-valued functions $R : \mathbb{R}^n \to \mathbb{B}$: the only continuous total functions from $\mathbb{R}^n$ to any discrete space such as $\mathbb{B}$ are the *constant* funtions.

The *solution*, we saw, was to work with *partial algebras*, i.e., to interpret the function symbols in the signature by partial functions. We use the stream model for insight. Consider, for example, the *equality* and *order* relations on the reals. Suppose we have two input reals (between 0 and 1) defined by streams of decimal digits, which we read, one digit at a time:

$$\begin{array}{rcl}\alpha & = & 0.a_0a_1a_2\ldots \\ \beta & = & 0.b_0b_1b_2\ldots\end{array}$$

Consider the various possibilities:

(a) $\alpha < \beta$. Then for some n, this will be determined by the initial segments $a_0 \ldots a_n$ and $b_0 \ldots b_n$.

(b) $\alpha > \beta$. Similarly, this will be determined by some pair of initial segments.

(c) $\alpha \neq \beta$. This is the *disjunction* of cases (a) and (c), so again it will be determined by some pair of initial segments.

(d) $\alpha = \beta$. This case, however, *cannot* be determined by initial segments of any length! (Note that this analysis is not affected by the double decimal representation of rationals.)

This analysis suggests the following definitions for partial functions in the signature of $\mathbb{R}$:

$$\text{(i)} \quad \mathsf{eq}_p(x,y) \;=\; \begin{cases} \uparrow & \text{if } x = y \\ \mathsf{ff} & \text{if } x \neq y; \end{cases}$$

$$\text{(ii)} \quad \mathsf{uneq}_p(x,y) \;=\; \begin{cases} \mathsf{tt} & \text{if } x \neq y \\ \uparrow & \text{if } x = y \end{cases} \qquad \text{(compare } (i));$$

$$\text{(iii)} \quad \mathsf{less}_p(x,y) \;=\; \begin{cases} \mathsf{tt} & \text{if } x < y \\ \uparrow & \text{if } x = y \\ \mathsf{ff} & \text{if } x > y; \end{cases}$$

$$\text{(iv)} \quad \mathsf{lseq}_p(x,y) \;=\; \begin{cases} \mathsf{tt} & \text{if } x < y \\ \uparrow & \text{if } x = y \\ \mathsf{ff} & \text{if } x > y \end{cases} \qquad \text{(same as } (iii)!).$$

Note that examples (i) and (iii) were incorporated as basic operations in the topological partial algebras $\mathcal{R}_p$ and $\mathcal{I}_p$ in section 7.3.

Further, we can add real-valued functions such as division:

$$\text{(v)}\quad x \,\mathsf{div}_{\mathsf{R}}\, y \;=\; \begin{cases} x/y & \text{if } y \neq 0 \\ \uparrow & \text{if } y = 0. \end{cases}$$

Note that in the above definitions, '$\uparrow$' (*undefinedness or divergence*) must not be confused with 'ϵ' (*error*), which occurs in *integer* division:

$$\text{(vi)}\quad x \,\mathsf{div}_{\mathsf{Z}}\, y \;=\; \begin{cases} \llcorner \mathrm{x}/\mathrm{y} \lrcorner & \text{if } y \neq 0 \\ \epsilon & \text{if } y = 0. \end{cases}$$

In the integer case, we can *effectively test* whether the input y is 0, and so (if $y = 0$) give an output, namely an error message (or default value, if we prefer). In the real case, if $y = 0$, this cannot be effectively decided, and so *no output* (error or other) is possible. (Suppose the first n digits of the input y are $00\ldots0$. The $(n+1)$th digit may or may not also be 0.)

We remark that the concept of reals as streams is reminiscent of Brouwer's notion of reals defined by *lawless sequences* or *choice sequences.* In fact, for Brouwer, a function *was* a *constructively defined* function, and he 'proved' that every function on $\mathbb{R}$ is continuous! (See, for example, the discussion in Troelstra and van Dalen [1988, Chapter 12]).

We conclude this discussion by pointing out a related, intensional, approach by Feferman to computation on the reals, based on Bishop's constructive approach to higher analysis (Bishop [1967], Bishop and Bridges [1985]). This is outlined in Feferman [1992a; 1992b].

7.5 Continuity of computable functions

In this section we will prove that computational processes associated with ***While***** programs over topological partial algebras are continuous. More precisely, we will prove Theorem 7.12:

(a) If $f \in$ ***While****(A) then f is continuous on A.*
(b) If $f \in$ ***While****$^N(A)$ then f is continuous on A.*
(c) If $f \in$ ***While****$^*(A)$ then f is continuous on A.*

Clearly, part (a) follows trivially from parts (b) and (c). Note that, conversely, parts (b) and (c) follow easily from (a). For example, if $f \in$ ***While***$^*(A)$ then $f \in$ ***While***(A^*), therefore f is continuous on A^*, and hence on A. We will prove part (a) by demonstrating the continuity of the operational semantics developed in section 3 (as modified for partial algebras). We will see the advantage of the algebraic approach to operational semantics used there, since these functions are built up from simpler functions using composition, thus preserving continuity.

We proceed with a series of lemmas. Let $X, Y, \ldots$ be topological spaces. Remember, functions are (in general) partial.

Lemma 7.15 (Basic lemmas on continuity).

(a) A composition of continuous functions is continuous.

(b) Let $f : X \to Y_1 \times \ldots \times Y_n$ have component functions $f_i : X \to Y_i$ for $i = 1, \ldots, n$, i.e., $f(x) \simeq (f_1(x), \ldots, f_n(x))$ for all $x \in X$. Then f is continuous if, and only if, all the f_i are continuous for $i = 1, \ldots, n$.

(c) If D is a discrete space, then a function $f : X \times D \to Y$ is continuous if, and only if, $f(\cdot, d) : X \to Y$ is continuous for all $d \in D$.

Proof. *Exercise.* ∎

Corollary 7.16. *The discriminator $f : \mathbb{B} \times X^2 \to X$, defined by $f(\mathtt{tt}, x, y) = x$ and $f(\mathtt{ff}, x, y) = y$, is continuous.*

Proof. *Exercise.* ∎

Corollary 7.17. *Let $f : X \to Y$ be defined by*

$$f(x) \simeq \begin{cases} g_1(x) & \text{if } h(x) \downarrow \mathtt{tt} \\ g_2(x) & \text{if } h(x) \downarrow \mathtt{ff} \\ \uparrow & \text{otherwise,} \end{cases}$$

where $g_1, g_2 : X \to Y$ and $h : X \to \mathbb{B}$ are continuous. Then f is continuous.

Proof. From Corollary 7.16 and Lemma 7.15(a). ∎

Lemma 7.18 (Least number operator). *Let $g : X \times \mathbb{N} \to Y$ be continuous, and let $y_0 \in Y$ be such that $\{y_0\}$ is clopen in Y. Let $f : X \to \mathbb{N}$ be defined by*

$$f(x) \ \simeq \ \mu k[g(x,k) \downarrow y_0],$$

i.e.,

$$f(x) \downarrow k \iff \forall i < k(g(x,i) \downarrow \neq y_0) \wedge (g(x,k) \downarrow y_0)$$

Then f is continuous.

Proof. Since $\mathbb{N}$ has the discrete topology, it is sufficient to show that for any $k \in \mathbb{N}$, $f^{-1}(\{k\})$ is open. We have

$$\begin{aligned} f^{-1}(\{k\}) &= \bigcap_{i=0}^{k-1} \{x \mid g(x,i) \downarrow \neq y_0\} \cap \{x \mid g(x,k) \downarrow y_0\} \\ &= \bigcap_{i=0}^{k-1} g(\cdot, i)^{-1}(Y \backslash \{y_0\}) \cap g(\cdot, k)^{-1}(\{y_0\}) \end{aligned}$$

which is a finite intersection of open sets, since by assumption both $\{y_0\}$ and $Y \backslash \{y_0\}$ are open. ∎

The rest of the proof consists of showing the continuity of the various semantic operations defined in section 3.

First we must specify topologies on the various spaces involved in the operational semantics.

For a product type $u = s_1 \times \ldots \times s_m$, the space $A^u =_{df} A_{s_1} \times \ldots \times A_{s_m}$ has (of course) the product topology of the A_{s_i}'s.

The state space $\boldsymbol{State}(\mathrm{A})$ is the (finite) product of the state spaces $\boldsymbol{State}_s(A)$ for all $s \in \boldsymbol{Sort}(\Sigma)$ (section 3.2), where each $\boldsymbol{State}_s(A)$ is an (infinite) product of the carriers A_s (indexed by $\boldsymbol{Var}_s$). Thus $\boldsymbol{State}(\mathrm{A})$ is an infinite product of all the carriers A_s, and takes the product topology of the A_s's. The space $\boldsymbol{State}(\mathrm{A})\cup\{*\}$ is formed as the union of $\boldsymbol{State}(\mathrm{A})$ and the singleton space $\{*\}$. Note that this makes the point $*$ clopen in $\boldsymbol{State}(\mathrm{A})\cup\{*\}$.

The syntactic sets $\boldsymbol{Stmt}$ and $\boldsymbol{AtSt}$ have the discrete topology, as do the sets $\mathbb{B}$ and $\mathbb{N}$ of Booleans and naturals.

Lemma 7.19. *For $t \in \boldsymbol{Term}_s$, the function $[\![t]\!]^A$: $\boldsymbol{State}(A) \to A_s$ (section 3.3) is continuous.*

Proof. By structural induction on t. Use the facts that the basic functions of Σ are continuous, and that continuity is preserved by composition (Lemma 7.15(a)). ∎

Recall that $[\![t]\!]^A$ and the other semantic functions considered below are actually the *partial algebra* analogues of the functions defined in section 3 (as discussed in section 7.2).

Lemma 7.20. *The state variant function*

$$\boldsymbol{\lambda}\,\sigma, a \cdot \sigma\{\mathtt{x}/a\} : \boldsymbol{State}(A) \times A^u \to \boldsymbol{State}(A)$$

(for some product type u and fixed tuple of variables $\mathtt{x} : u$) is continuous.

Proof. *Exercise.* ∎

Lemma 7.21. *For $S \in \boldsymbol{AtSt}$, the function $\langle\!|S|\!\rangle^A$: $\boldsymbol{State}(A) \to \boldsymbol{State}(A)$ (section 3.5) is continuous.*

Proof. For $S\equiv$skip, this is trivial. For $S\equiv\mathtt{x} := t$, use Lemmas 7.19, 7.20 and 7.15(b). ∎

Lemma 7.22. *The functions $\boldsymbol{First}$ and $\boldsymbol{Rest}^A$ (section 3.6) are continuous.*

Proof. For $\boldsymbol{First}$, this is trivial (a mapping with discrete domain space). For $\boldsymbol{Rest}^A$, it is sufficient, by Lemma 7.15(c), to show that for any fixed $S \in \boldsymbol{Stmt}$, the function

$$\boldsymbol{Rest}^A(S, \cdot\,) : \boldsymbol{State}(\mathrm{A}) \to \boldsymbol{Stmt}$$

is continuous. This is proved by structural induction on S, making use (in the case that S is a conditional or 'while' statement) of Corollary 7.17. ∎

Lemma 7.23. *The one-step computation function*

$$\boldsymbol{Comp}_1^A : \boldsymbol{Stmt} \times \boldsymbol{State}(A) \to \boldsymbol{State}(A)$$

(section 3.4) is continuous.

Proof. Again, by Lemma 7.15(c), it is sufficient to show that for any fixed $S \in \boldsymbol{Stmt}$, the function $\boldsymbol{Comp}_1^A(S, \cdot)$ is continuous. But by definition, this is $\langle\!|\boldsymbol{First}(S)|\!\rangle^A$, which is continuous by Lemma 7.22. ∎

Lemma 7.24. *The computation step function* $\boldsymbol{Comp}^A$ *(section 3.4) is continuous.*

Proof. Again, it is sufficient to show that for any fixed $S \in \boldsymbol{Stmt}$ and $n \in \mathbb{N}$, the function

$$\boldsymbol{Comp}^A(S, \cdot, n) : \boldsymbol{State}(\mathrm{A}) \to \boldsymbol{State}(\mathrm{A}) \cup \{*\}$$

is continuous. This is proved by induction on n, using (in the base case) Lemma 7.23 and (in the induction step) Lemmas 7.22 and 7.23. ∎

Lemma 7.25. *The computation length function* $\boldsymbol{CompLength}^A$ *(section 3.4) is continuous.*

Proof. This function is defined by

$$\boldsymbol{CompLength}^A(S, \sigma) \ \simeq \ \mu n[\boldsymbol{Comp}^A(S, \sigma, n+1) \downarrow *].$$

Its continuity follows from Lemma 7.18, since $\{*\}$ is clopen in $\boldsymbol{State}(\mathrm{A}) \cup \{*\}$, and by Lemma 7.24. ∎

Lemma 7.26. *For* $S \in \boldsymbol{Stmt}$*, the function* $[\![S]\!]^A$*:* $\boldsymbol{State}(A) \to \boldsymbol{State}(A)$ *(section 3.5) is continuous.*

Proof. Since

$$[\![\mathrm{S}]\!]^A(\sigma) \ \simeq \boldsymbol{Comp}^A(S, \sigma, \boldsymbol{CompLength}^A(S, \sigma)),$$

the result follows from Lemmas 7.24 and 7.25. ∎

Lemma 7.27. *For any* **While** *procedure* P*, the function* P^A *(section 3.6) is continuous.*

Proof. Suppose $P \equiv$ proc in a out b aux cbegin S end, where a : u and b : v, so that $P^A : A^u \to A^v$. Fix any state $\sigma_0 \in \boldsymbol{State}(\mathrm{A})$. The imbedding and projection functions

$$\iota_{\mathsf{a}} : A^u \to \boldsymbol{State}(\mathrm{A}) \qquad \text{and} \qquad \pi_{\mathsf{b}} : \boldsymbol{State}(\mathrm{A}) \to A^v$$

defined by

$$\iota_{\mathsf{a}}(x) \ = \ \sigma_0\{\mathsf{a}/x\} \qquad \text{and} \qquad \pi_{\mathsf{b}}(\sigma) \ = \ \sigma[\mathsf{b}]$$

are continuous. (*Exercise.*) Hence the composition

$$\pi_{\mathsf{b}} \circ [\![\mathrm{S}]\!]^A \circ \iota_{\mathsf{a}} : A^u \to A^v.$$

is continuous. But this is just P^A, independent of the choice of σ_0, by the functionality lemma (3.11) for procedures. ∎

Theorem 7.12(a) follows from this.

7.6 Topological characterisation of computable sets in compact algebras

For background on compactness, see any of the books listed at the beginning of section 7.

Remark 7.28 (Compactness).

(a) By Tychonoff's theorem, the product of compact spaces is compact.

(b) The unit interval I is compact. Hence so is the product space I^q for any q.

Now we have seen (Theorem 7.13 and Corollary 7.14) that for sets:

$$\begin{array}{rcl} \text{semicomputable} & \Longrightarrow & \text{open} \\ \text{computable} & \Longrightarrow & \text{clopen.} \end{array}$$

We can reverse the direction of the implication in the second of these assertions, under the assumption of *compactness.*

Theorem 7.29. *Let A be a topological partial algebra, and let $u = s_1 \times \ldots \times s_m \in$ **ProdType**(Σ), where, for $i = 1, \ldots, n$,*

(a) A_{s_i} is compact, and

*(b) A_{s_i} has an open subbase of **While** semicomputable sets.*

Then for any relation $R \subseteq A^u$, the following are equivalent:

*(i) R is **While** computable;*

*(ii) R is **While*** computable;*

(iii) R is clopen in A^u.

Proof. $(i) \Longrightarrow (ii)$ is trivial.

$(ii) \Longrightarrow (iii)$ follows from Corollary 7.14.

$(iii) \Longrightarrow (i)$**:** Note first that from assumptions (a) and (b), the product space A^u (with the product topology) is compact, and has an open subbase of ***While*** semicomputable sets. Suppose now that R is clopen in A^u. Since R is *open*, we can write

$$R \;=\; \bigcup_{i \in I} B_i$$

where the B_i are basic open sets. Each B_i is a finite intersection of subbasic open sets, and hence semicomputable, by Theorem 5.8.

Since R is *closed*, R is compact, and hence R is the union of *finitely many* of the B_i's, and so R is semicomputable, by Theorem 5.8.

Repeating the above argument for R^c, we infer by Post's theorem (5.9) that R is computable. ∎

7.7 Metric partial algebra

A particular type of topological partial algebra is a *metric partial algebra.* This is a pair (A, d) where d is a family of metrics $\langle d_s \mid s \in \boldsymbol{Sort}(\Sigma)\rangle$, and for each $s \in \boldsymbol{Sort}(\Sigma)$, d_s is a metric on A_s, such that for each basic function symbol $F : u \to s$ of Σ, the function $F^A : A^u \to A_s$ is continuous (where continuity of a partial function is as per Definition 7.5).

This induces or defines a *topological partial algebra* in the standard way.

Note that if A is *standard*, then the carrier $\mathbb{B}$, as well as the carriers of all equality sorts, will have the *discrete metric*, defined by

$$d(x,y) \;=\; \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y, \end{cases}$$

which induces the discrete topology (see Remark 7.2).

Again, we will often speak of a 'metric algebra A', without stating the metric explicitly.

Example 7.30. The real algebra $\mathcal{R}_p$ and interval algebras $\mathcal{I}_p$ (Examples 7.7) can be viewed (or recast) as metric algebras in an obvious way.

Remark 7.31. If A is a metric partial algebra, then for each product sort $u = s_1 \times \ldots \times s_m$, we can define a metric d_u on A^u, which induces the product topology on A^u, by

$$d_u((x_1, \ldots, x_m), (y_1, \ldots, y_m)) \;=\; \max_{i=1}^{m}\bigl(d_{s_i}(x_i, y_i)\bigr)$$

or more generally, by the ℓ_p metric

$$d_u((x_1, \ldots, x_m), (y_1, \ldots, y_m)) \;=\; \Bigl(\sum_{i=1}^{m}(d_{s_i}(x_i, y_i))^p\Bigr)^{1/p} \qquad (1 \leq p \leq \infty)$$

Metric algebras will be used in our study of approximable computability (section 7.9).

7.8 Connected domains: computability and explicit definability

In this subsection we investigate the relationship between computability and explicit definability for a function on a connected domain.

First we review the concept of connectedness.

Remark 7.32 (Connectedness).

(*a*) A topological space X is said to be *connected* if the only clopen subsets of X are X and $\emptyset$.

(*b*) It is easy to see that X is connected if, and only if, the only continuous total functions from X to $\mathbb{B}$ (or to any discrete space) are the constant functions. (*Exercise.*)

(c) A finite product of connected spaces is connected. (See any of the references listed at the beginning of section 7.) Hence in a topological Σ-algebra A, if $u = s_1 \times \ldots \times s_m \in \boldsymbol{ProdType}(\Sigma)$, and A_{s_i} is connected for $i = 1, \ldots, m$, then so is A^u.

(d) The space $\mathbb{R}$ of the reals, with its usual topology, is connected. Therefore, so is the product space $\mathbb{R}^q$ for any q. Hence, by Corollary 7.14, for any topological partial algebra over $\mathbb{R}$, such as the algebra $\mathcal{R}_p$ (Example 7.7(a)), the only ***While*** or ***While***** computable subsets of $\mathbb{R}^q$ are $\mathbb{R}^q$ itself and $\emptyset$.

(e) Similarly, by the connectedness of the unit interval I (and hence of I^q), the only ***While*** or ***While***** computable subsets of I^q in any interval algebra over I (Example 7.7(b)) are I^q itself and $\emptyset$, ... , regardless of the choice of (continuous) functions $F_1, \ldots, F_k$ as basic operations!

We will only develop the theory in this section for *total functions* on *total algebras*. The essential idea is that if f is a computable total function on A, then f is continuous, and so, by Remark 7.32(b), its definition cannot depend non-trivially on any Boolean tests involving variables of sort s if A_s is connected. (We will make this precise below, in the proof of Lemma 7.40.)

Note that many of these results can be extended to the case of *total* functions f on connected domains in *partial* algebras. We intend to investigate this more fully in future work. However, for now we assume in this subsection:

Assumption 7.33. A is a total topological algebra.

Examples 7.34 (Topological total algebras on the reals). Two important total topological algebras based on the reals which will be important for our purposes are:

(a) The algebra $\mathcal{R}_t^N$ ('t' for 'total topological'), defined by

algebra	$\mathcal{R}_t^N$
import	$\mathcal{R}_0, \mathcal{N}, \mathcal{B}$
functions	$\mathsf{if}_{\mathsf{real}} : \mathbb{B} \times \mathbb{R}^2 \to \mathbb{R}$,
	$\mathsf{div}_{\mathsf{nat}} : \mathbb{R} \times \mathbb{N} \to \mathbb{R}$,
end	

Here $\mathcal{R}_0$ is the ring of reals $(\mathbb{R};\ 0,\ 1,\ +,\ -,\ \times)$ (Example 2.5(b)), $\mathcal{N}$ is the standard algebra of naturals (Example 2.23(b)), $\mathsf{div}_{\mathsf{nat}}$ is division of reals by naturals (where division by zero is defined as zero), and $\mathbb{R}$ has its usual topology.

Note that $\mathcal{R}_t^N$ does not contain the (total) Boolean-valued functions $\mathsf{eq}_{\mathsf{real}}$ or $\mathsf{less}_{\mathsf{real}}$, since they are not continuous (cf. the partial functions

eq_p and less_p of $\mathcal{R}_p$). It is therefore not an expansion of the standard algebra $\mathcal{R}$ of reals (Example 2.23(c)) which contains $\mathsf{eq}_{\mathsf{real}}$. (Compare the N-standardisation $\mathcal{R}^N$ of $\mathcal{R}$ (Example 2.27(b)) which does contain $\mathsf{eq}_{\mathsf{real}}$.)

(b) (*The interval algebra of reals.*) Here the unit interval $I = [0,1]$ is included as a separate carrier of sort 'intvl', again with the usual topology. This is useful for studying real continuous functions with compact domain. (We could also choose $I = [-1,1]$, etc.) The total topological algebra $\mathcal{I}_t^N$ is defined by

```
algebra     I_t^N
import      R_t^N
carriers    I
functions   i_I : I → ℝ
end
```

Here i_I is the embedding of I into $\mathbb{R}$.

Remark 7.35. Note that both algebras $\mathcal{R}_t^N$ and $\mathcal{I}_t^N$ are *strictly N-standard.* The reason why $\mathbb{N}$, and the function $\mathsf{div}_{\mathsf{nat}}$, are included in these total algebras (unlike the partial algebras $\mathcal{R}_p$ and $\mathcal{I}_p$ of 7.7) is because of their applicability in the theory of approximable computability in section 7.9.

Definition 7.36. Let f be a function on A.

(a) f is Σ-*explicitly definable on* A if f is definable on A by a Σ-term.

(b) f is Σ^*-*explicitly definable on* A if f is definable on A by a Σ^*-term.

By the Σ^*/Σ conservativity theorem (3.63)', the two concepts defined above are equivalent:

Proposition 7.37. *A function on A is Σ-explicitly definable if, and only if, it is Σ^*-explicitly definable.*

Remarks 7.38.

(a) Suppose (i) A is strictly N-standard (e.g., $\mathcal{R}_t^N$ and $\mathcal{I}_t^N$), and (ii) the domain and range types of f do not include nat (e.g., $f : \mathbb{R}^q \to \mathbb{R}$ or $f : I^q \to \mathbb{R}$ in these algebra, respectively). Then this proposition also holds with the 'internal' version of Σ^* (Remark 2.31(c)), by Remark 3.64(b).

(b) Because of Proposition 7.37, we shall usually use 'explicit definability' over an algebra to mean either Σ- or Σ^*-explicit definability.

In the following lemma, A is any total algebra, not necessarily topological.

Lemma 7.39. *Explicit definability on A $\Longrightarrow$ **While** computability on A.*

Proof. Simple exercise. ∎

In preparation for the converse direction, we need the following:

Lemma 7.40. *Suppose A^u is connected. Let $P : u \to v$ be a (**While** or **While***) procedure which defines a total function on A, i.e., $\boldsymbol{Halt}^A(P) = A^u$. Then the computation tree $\mathcal{T}(P)$ for P is essentially finite, or (more accurately) semantically equivalent to a finite, unbranching tree.*

(The computation tree for a procedure was defined in section 5.11.)

Proof. Put

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

where $\mathsf{a} : u$, $\mathsf{b} : v$ and $\mathsf{c} : w$, and $S \equiv S_{init}; S'$, where S_{init} is an initialisation of the variables b,c to their default values. Let $\mathcal{T} = \mathcal{T}(P)$. First, we show that all branches in $\mathcal{T}$ can be eliminated. Consider a branch at a test node in $\mathcal{T}$ (Fig. 16).

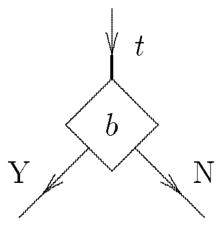

Fig. 16.

This Boolean test defines a function

$$f_{b,t} : A^u \to \mathbb{B}$$

where (putting $\mathtt{x} \equiv \mathsf{a}, \mathsf{b}, \mathsf{c}$)

$$f_{b,t}(a) = b\langle \mathtt{x}/t \rangle [a, \boldsymbol{\delta}^v_A, \boldsymbol{\delta}^w_A]$$

i.e., $f_{b,t}(a)$ is the evaluation of $b\langle \mathtt{x}/\mathtt{t}\rangle$ when a is assigned to a and the default tuples $\boldsymbol{\delta}^v, \boldsymbol{\delta}^w$ are assigned to b,c respectively. The function $f_{b,t}$ is clearly (***While*** or ***While****) computable, by Lemma 7.39, and hence continuous, by Theorem 7.12. It is also total, since A is total by assumption (7.33). By Remark 7.32(b) it must therefore be *constant* on A^u. If it is constantly $\mathtt{tt}$, we can replace this test node by its left branch (i.e., delete the node and the right branch), and if it is constantly $\mathtt{ff}$, we can similarly replace the node by its right branch only.

By repeating this process, we can replace $\mathcal{T}$ by a semantically equivalent tree $\mathcal{T}'$ without any Boolean tests, and (hence) without any branching. The tree $\mathcal{T}'$ consists of a single path, which must be finite, since P^A is total by assumption. ∎

Remarks 7.41.

(a) Examples of the application of this lemma are the total topological algebras $\mathcal{R}_t^N$ and $\mathcal{I}_t^N$, and procedures of type $\mathsf{real}^q \to \mathsf{real}$ and $\mathsf{intvl}^q \to \mathsf{real}$, respectively. Note that the result also holds with the 'internal' version of ***While**** computability, by Proposition 3.47.

(b) Without the assumption that A^u be connected, Lemma 7.40 is false, i.e., it is possible for P^A to be total, but $\mathcal{T}(P)$ to be infinite. (*Exercise.*)

(c) Note that any computation tree $\mathcal{T}$ is finitely branching; therefore, by König's lemma, $\mathcal{T}$ is finite if, and only if, all its paths are finite. Hence any counterexample to demonstrate (b) would be an example of a computation tree for a procedure which defines a total function, but nevertheless has infinite paths!

(d) The lemma also holds without the assumption that A be total, as long as P^A is total (and A^u is connected). (*Exercise.*)

(e) In general, this transformation of $\mathcal{T}(P)$ to a finite unbranching tree given by the proof of Lemma 7.40 is *not effective in* P, since it depends on the evaluation of (constant) Boolean tests. If we want it to be effective in P (as we will in the next subsection, dealing with approximable computability), we will need a further condition on A, such as the Boolean computability property (Definition 7.56).

Lemma 7.42. *If a computation tree $\mathcal{T}(P)$ for a (**While** or **While***) procedure P is finite and unbranching, then P^A is (Σ-)explicitly definable on A.*

Proof. *Exercise.* ∎

Remark 7.43. More generally, Lemma 7.42 holds if $\mathcal{T}(P)$ is finite but (possibly) branching. (Use the discriminator in constructing the defining term.)

Combining Lemmas 7.39, 7.40 and 7.42, we have conditions for an equivalence between explicit definability and ***While*** computability:

Theorem 7.44. *Let A be a total topological algebra, and suppose A^u is connected. Let $f : A^u \to A^v$ be a total function. Then the following are equivalent:*

*(i) f is **While** computable on A;*
*(ii) f is **While*** computable on A;*
(iii) f is explicitly definable on A.

Example 7.45. This theorem holds for the total topological algebras $\mathcal{R}_t^N$ and $\mathcal{I}_t^N$, and total functions $f : \mathbb{R}^q \to \mathbb{R}$ and $f : I^q \to I$, respectively.

Note that by Remarks 7.38(a) and 7.41(a), the theorem also holds in these algebras with 'internal' versions of ***While**** computability and Σ^*-explicit definability.

7.9 Approximable computability

It is often the case that functions are computed approximately, by a sequence of 'polynomial approximations'. In this way we extend the class of computable functions to that of *approximably computable* functions. This theory will build on the work of section 7.8.

First we review some basic notions on convergence of sequences of functions.

Definition 7.46 (Effective uniform convergence). Given a set X, a metric space Y, a total function $f : X \to Y$ and a sequence of total functions $g_n : X \to Y$ $(n = 0, 1, 2, \dots)$, we say that g_n *converges effectively uniformly to* f *on* X (or *approximates* f *effectively uniformly on* X) if, and only if, there is a total recursive function $e : \mathbb{N} \to \mathbb{N}$ such that for all n, k and all $x \in X$,

$$k \geq e(n) \implies d_Y(g_k(x), f(x)) < 2^{-n}.$$

Remark 7.47. Let $M : \mathbb{N} \to \mathbb{N}$ be any total recursive function which is increasing and unbounded. Then (in the notation of Definition 7.46) the sequence g_n converges effectively uniformly to f on X if, and only if, there is a total recursive function $e : \mathbb{N} \to \mathbb{N}$ such that for all n, k and all $x \in X$,

$$k \geq e(n) \implies d_Y(g_k(x), f(x)) < 1/M(n).$$

(*Exercise.*)

The theory here will be developed for *total functions* on *metric total algebras* (defined in section 7.5). We therefore assume in this subsection:

Assumption 7.48. A is a *metric total algebra.*

Example 7.49 (Metric total algebras on the reals). The two total topological algebras based on the reals given in Example 7.34 can be viewed as metric algebras in an obvious way. The second of these, the interval algebra $\mathcal{I}_t^N$, will be particularly useful here.

We will present, and compare, two notions of approximable computability on metric total algebras: effective uniform ***While*** (or ***While****) approximability (Definition 7.50) and effective Weierstrass approximability (Definition 7.54).

So suppose A is a metric total Σ-algebra. Let $u, v \in \boldsymbol{ProdType}(\Sigma)$ and $s \in \boldsymbol{Sort}(\Sigma)$.

Definition 7.50. A total function $f : A^u \to A^v$ is *effectively uniformly* ***While*** (or ***While****) *approximable on* A if there is a ***While*** (or ***While****) procedure

$$P : \mathsf{nat} \times u \to v$$

on A^N such that P^{A^N} is total on A^N and, putting $g_n(x) =_{df} P^{A^N}(n, x)$, the sequence g_n converges to f effectively uniformly on A^u.

Remark 7.51. If A is N-standard, we can replace 'A^N' by 'A' in the above definition (by Proposition 3.38).

Lemma 7.52. *If A^u is compact, and $f : A^u \to A_s$ is effectively uniformly* ***While***** *approximable on A, then f is continuous.*

Proof. By Theorem 7.12, the approximating functions for f are continuous. The theorem follows by a standard result for uniform convergence on compact spaces. ∎

Remark 7.53. Note that a function from $\mathbb{R}^q$ to $\mathbb{R}$ is explicitly definable over $\mathcal{R}_t^N$ if, and only if, it is definable by a *polynomial* in q variables over $\mathbb{R}$ with rational coefficients. Similarly, a function from I^q to $\mathbb{R}$ is explicitly definable over $\mathcal{I}_t^N$ if, and only if, it is definable by a polynomial in q variables over I with rational coefficients. This explains the following terminology, since Weierstrass-type theorems deal typically with approximations of real functions by polynomial functions (uniformly on compact domains).

Definition 7.54 (Effective Weierstrass approximability).

(*a*) A total function $f : A^u \to A_s$ is *effectively Σ-Weierstrass approximable over A* if, for some $\mathtt{x} : u$, there is a total computable function

$$h : \mathbb{N} \to \ulcorner \boldsymbol{Term}_{\mathtt{x},s}(\Sigma) \urcorner$$

such that, putting $g_n(x) =_{df} \boldsymbol{te}^A_{\mathtt{x},s}(h(n), x)$, the sequence g_n converges to f effectively uniformly on A^u.

(*b*) *Effective Σ^*-Weierstrass approximability* is defined similarly, by replacing 'Σ' by 'Σ^*' and '$\boldsymbol{te}^A_{\mathtt{x},s}$' by '$\boldsymbol{te}^{A^*}_{\mathtt{x},s}$'.

(The term evaluation representing function $\boldsymbol{te}^A_{\mathtt{x},s}$ was defined in section 4.3.)

Proposition 7.55. *A function on A is effectively Σ-Weierstrass approximable if, and only if, it is effectively Σ^*-Weierstrass approximable.*

Proof. From a computable function

$$h^* : \mathbb{N} \to \ulcorner \boldsymbol{Term}_{\mathtt{x},s}(\Sigma^*) \urcorner$$

we can construct a computable function

$$h : \mathbb{N} \to \ulcorner \boldsymbol{Term}_{\mathtt{x},s}(\Sigma) \urcorner$$

where, for each n, $h(n)$ and $h^*(n)$ are Gödel numbers for semantically equivalent terms, using the fact that the transformation of Σ^*-terms to Σ-terms in the conservativity theorem (the2.15.4) is effective. ∎

We shall therefore usually speak of 'effective Weierstrass approximability' over an algebra to mean effective Weierstrass approximability in either sense.

We now investigate the connection between effective uniform ***While*** approximability and effective Weierstrass approximability. We are looking for a uniform version of Theorem 7.44 (i.e., uniform over $\mathbb{N}$-sequences of functions).

To attain this uniformity, we need an extra condition in each direction: for 'effective Weierstrass $\Rightarrow$ effective uniform ***While***' (i.e., a uniform version of Lemma 7.39) we need the TEP (section 4.7), and for 'effective uniform ***While*** $\Rightarrow$ effective Weierstrass' (i.e., a uniform version of Lemma 7.40) we need a new condition, the Boolean computability property (cf. Remark 7.41(e)), which we now define.

Definition 7.56. A Σ-algebra A has the *Boolean computability property (BCP)* if for any closed Σ-Boolean term b, its valuation b_A ($=$ tt or ff, cf. Definition 2.11) can be effectively computed, i.e., (equivalently) there is a recursive function

$$f\ :\ulcorner \boldsymbol{T}(\Sigma)_{\mathtt{bool}}\urcorner \ \to\ \mathbb{B}$$

with $f(\ulcorner b \urcorner) \ =\ b_A$.

Remark 7.57. To avoid confusion: the BCP is *not* a special case of the TEP, for closed terms of sort bool. It requires the function f in Definition 7.56 to be recursive, i.e., computable over $\mathbb{N}$ (and $\mathbb{B}$) in the sense of *classical recursion theory.* The TEP entails only that f be computable over A — a weaker assumption (in general).

Example 7.58. Both $\mathcal{R}_t^N$ and $\mathcal{I}_t^N$ have the TEP and the BCP. (*Exercise.*)

We will see how these two conditions (TEP and BCP) are applied in opposite directions to obtain a uniform version of Theorem 7.44.

In the following lemma, A is any total algebra, not necessarily metric or even topological (cf. Lemma 7.39).

Lemma 7.59. *Suppose A has the TEP. Given variables $\mathtt{x} : u$, let*

$$h : \mathbb{N} \to \ulcorner \boldsymbol{Term}_{\mathtt{x},s}(\Sigma)\urcorner$$

be a total computable function. Then there is a $\boldsymbol{While}(\Sigma^N)$ *procedure* $P\ :\ \mathsf{nat} \times u \to s$ *such that for all* $x \in A^u$ *and* $n \in \mathbb{N}$,

$$P^{A^N}(n, x) \ =\ \boldsymbol{te}^A_{\mathtt{x},s}(h(n), x).$$

Proof. Simple exercise. ∎

For the converse direction:

Lemma 7.60. *Suppose A^u is connected and A has the BCP. Let* $P\ :\ \mathsf{nat} \times u \to v$ *be a (**While** or **While***) procedure over A^N which defines* *a total function on A^N. Then there is a computable function $h : \mathbb{N} \to \ulcorner \boldsymbol{Term}_{\mathtt{x},s}(\Sigma)\urcorner$ such that for all $x \in A^u$ and $n \in \mathbb{N}$,*

$$\boldsymbol{te}^A_{\mathtt{x},s}(h(n), x) \;=\; P^{A^N}(n, x).$$

Proof. Suppose

$$P \equiv \textsf{proc in } \mathtt{n}, \textsf{a out b aux c begin } S \textsf{ end}$$

where $\mathtt{n}$: $\textsf{nat}$. Consider the $\boldsymbol{While}^N(\Sigma)$ procedures $P_n : u \to v$ $(n = 0, 1, 2, \dots)$ defined by

$$P_n \equiv \textsf{proc in a out b aux } \mathtt{n}, \textsf{c begin } \mathtt{n} := \bar{n}; S \textsf{ end}$$

where $\bar{n}$ is the numeral for n. It is clear that for all $n \in \mathbb{N}$ and $x \in A^u$,

$$P_n^A(x) \;=\; P^{A^N}(n, x).$$

By Lemmas 7.40 and 7.42, P_n^A is definable by a Σ-term t_n. Moreover, the sequence (t_n) is *computable in n*, by use of the BCP to effectivise the transformation of the tree $\mathcal{T}$ to $\mathcal{T}'$ in the construction given by the proof of Lemma 7.40. (Note that the evaluation of a *constant Boolean test* can be effected by the computation of any *closed instance* of the Boolean term, which exists by the instantiation assumption.) Hence the function h defined by

$$h(n) \;=\; \ulcorner t_n \urcorner$$

is computable. ∎

We now have a uniform version of Theorem 7.44:

Theorem 7.61. *Suppose A^u is connected and A has the TEP and BCP. Let $f : A^u \to A_s$ be a total function. Then the following are equivalent:*

*(i) f is effectively uniformly **While** approximable on A;*
*(ii) f is effectively uniformly **While*** approximable on A;*
(iii) f is effectively Weierstass approximable on A.

Proof. From Lemmas 7.59 and 7.60. ∎

The requirement in the above theorem that f be *total* derives from the application of Lemma 7.60, which in turn used Lemma 7.40, where totality was required.

Remark 7.62. The equivalence of (i) and (iii) was noted for the special case $A = \mathcal{I}_t^N$, $A^u = I^q$ and $A_s = \mathbb{R}$ in Shepherdson [1976], in the course of proving the equivalence of these with another notion of computability on the reals (Theorem 7.64).

We are especially interested in computability on the reals, and, in particular, a notion of computability of functions from I^q to $\mathbb{R}$, developed in Grzegorczyk [1955; 1957] and Lacombe [1955]. We repeat the version given in Pour-El and Richards [1989], giving also, for completeness, the definitions of computable sequences of rationals and computable reals. Finally (Theorem 7.64),we state the equivalence of this notion with the others listed in Theorem 7.61.

Definition 7.63.

(a) A sequence (r_k) of rationals is *computable* if there exist recursive functions $a, b, s : \mathbb{N} \to \mathbb{N}$ such that, for all k, $b(k) \neq 0$ and

$$r_k \;=\; (-1)^{s(k)} \frac{a(k)}{b(k)}.$$

A *double sequence* of rationals is *computable* if it is mapped onto a computable sequence of rationals by one of the standard recursive pairing functions from $\mathbb{N}^2$ onto $\mathbb{N}$.

(b) A sequence (x_k) of reals is *computable* if there is a computable double sequence of rationals (r_{nk}) such that

$$|r_{nk} - x_n| \leq 2^{-k} \qquad \text{for all } k \text{ and } n.$$

(c) A total function $f : I^q \to \mathbb{R}$ is *GL* (or *Grzegorczyk–Lacombe*) *computable* if:

(i) f is *sequentially computable*, i.e., f maps every computable sequence of points in I^q into a computable sequence of points in $\mathbb{R}$;

(ii) f is *effectively uniformly continuous*, i.e., there is a recursive function $\delta : \mathbb{N} \to \mathbb{N}$ such that, for all $x, y \in I^q$ and all $n \in \mathbb{N}$,

$$|x - y| < 2^{-\delta(n)} \Longrightarrow |f(x) - f(y)| < 2^{-n}.$$

Theorem 7.64. *Let $f : I^q \to \mathbb{R}$ be a total function. Then the following are equivalent:*

(i) f is effectively uniformly ***While*** *approximable on $\mathcal{I}_t^N$;*

(ii) f is effectively uniformly ***While**** *approximable on $\mathcal{I}_t^N$;*

(iii f is effectively Weierstrass approximable on $\mathcal{I}_t^N$;

(iv) f is GL computable.

Proof. As we have noted, I^q is connected and $\mathcal{I}_t^N$ has the TEP and BCP. Hence the equivalence of the first three assertions is a special case of Theorem 7.61. The equivalence of (iii) and (iv) is proved in detail in Pour-El and Richards [1989]. ∎

Remark 7.65 (Historical). The equivalence (iii)$\Leftrightarrow$(iv) was proved in Pour-El and Caldwell [1975]. An exposition of this proof is given in Pour-El and Richards [1989]. Shepherdson [1976] gave a proof of $(i)\Leftrightarrow(iv)$ by (essentially) noting the equivalence $(i)\Leftrightarrow(iii)$ and reproving $(iii)\Leftrightarrow(iv)$. The new features in the present treatment are: (a) the equivalence $(i)\Leftrightarrow(iii)$ in a more general context (Theorem 7.61), and (b) the equivalence of (ii) with the rest (Theorems 7.61 and 7.64).

7.10 Abstract versus concrete models for computing on the real numbers

Our models of computation can be applied to any algebraic structure. Furthermore, our models of computation are abstract: the computable sets and functions on an algebra are *isomorphism invariant.* Thus to compute on the real numbers we have only to choose an algebra A in which (any one of the representations of) the set $\mathbb{R}$ of reals is a carrier set. There are infinitely many such algebras of representations or implementations of the reals, all with decent theories resembling the theory of the computable functions on the naturals. However, unlike the case of the natural numbers, it is easy to list different algebras of reals with different classes of ***While*** computable functions (see below).

In sections 6 and 7, we have let the abstract theory dictate our development of computation on the reals. The goal of making an attractive and useful connection with continuity led us to use partial algebras in section 7. Because of the fundamental role of continuity, this partial algebra approach is important since it enables us to relate abstract computation on the reals with *concrete computation* on representations of the reals (via the natural numbers). This we saw in section 7.4 and, especially, in section 7.9. Here we will reflect further on the distinction between concrete and abstract, following Tucker and Zucker [1999].

The real numbers can be built from the rational numbers, and hence the natural numbers, in a variety of equivalent ways, such as Dedekind cuts, Cauchy sequences, decimal expansions, etc. Thus it is natural to investigate the computability of functions on the real numbers, starting from the theory of computable functions on the naturals. Such an approach we term a *concrete computability theory.* The key idea is that of a *computable real number.* A computable real number is a number that has a computable approximation by rational numbers; the set of computable real numbers forms a real closed subfield of the reals. Computable functions on the reals are functions that can be computably approximated on computable real numbers. The study of the computability of the reals began in Turing [1936], but only later was taken up in a systematic way, in Rice [1954], Lacombe [1955] and Grzegorczyk [1955; 1957], for example.

The different representations of the reals are specified axiomatically, uniquely up to isomorphism, as a complete Archimedean ordered field. But computationally they are far from being equivalent. For instance, representing real numbers by infinite decimals leads to the problem that the trivial function $3x$ cannot be computable. If Cauchy sequences are used, however, elementary functions on the reals are computable. The problems of representation are worse when investigating computational complexity (see Ko [1991]).

It is a general problem to understand concrete representations of infinite data and, to this end, to establish a comprehensive theory of computing in topological algebras. There have been a number of approaches to computability based on concrete representations for the reals and other topological structures. Only recently have these approaches been shown to be equivalent.

The ideas about computable functions on the reals were generalised to metric spaces in Moschovakis [1964] who proved some of the special theorems of Ceitin [1959] obtained earlier with a constructive point of view.

An axiomatic approach to computability on Banach spaces is given in Pour-El and Richards [1989]. This gives general theorems about the independence of computation from representations, and provides a series of remarkable results characterising computable operators.

Computability theory on $\mathbb{N}$ includes a theory of computation for functionals on the set

$$\mathbf{B} =_{df} \ [\mathbb{N} \to \mathbb{N}]$$

which, with the product topology, is called *Baire space.* The theory of computation on $\mathbf{B}$ is called *type 2 computability theory.* Klaus Weihrauch and his collaborators, in a long series of papers, have created a fine generalisation of the theory of numberings of countable sets (recall section 1.3) to a theory of type 2 numberings of uncountable sets. In type 2 enumeration theory, numberings have the following form. Let X be a topological space. A type 2 enumeration of X is surjective partial map

$$\alpha : \ \mathbf{B} \ \to \ X$$

(cf. Definition 1.1). Computability on X is analysed using type 2 computability on $\mathbf{B}$. See, for example, Kreitz and Weihrauch [1985] and, especially, Weihrauch [1987].

A more abstract method for the systematic study of effective approximations of uncountable topological algebras has been developed by V. Stoltenberg-Hansen and J. V. Tucker. It is based on representing topological algebras with algebras built from *domains* and applying the *theory of effective domains.* This method of applying domain theory to mathematical approximation problems was first developed for topological algebras and used on completions of local rings in Stoltenberg-Hansen and Tucker [1985; 1988]. It was further developed on universal algebras in Stoltenberg-Hansen and Tucker [1991; 1993; 1995]; see also Stoltenberg-Hansen *et al.* [1994, Chapter 8]. We will sketch the basic method; an introduction can be found in Stoltenberg-Hansen and Tucker [1995]. Suppose A is a topological algebra. The idea is to build an algebra R that represents A by means of the continuous representation map $\alpha : R \to A$ and to computably approximate R.

We imagine building R from a set P of approximating data that is a computable structure (in the sense of Section 1.3). Each datum in R is approximated by some sequence of data from P. More specifically, R is a topological space obtained from P by some form of completion process in which P is dense in R. The key feature of this approach is that, since P is computable, some of the approximating sequences are computable. The subset of R consisting of the computably approximable elements forms a basis for the computable approximation of R and hence of A. We usually use a special type of approximating structure P called a *conditional upper semilattice*, and a completion process called *ideal completion.* This process yields an *algebraic domain.* The method effectively approximates a large class of examples: ultrametric algebras, locally compact Hausdorff algebras (Stoltenberg-Hansen and Tucker [1995]), and complete metric algebras (Blanck [1997]).

Similar ideas have been used in Edalat [1995a; 1995b], applying continuous domains to analytical questions, such as integration and measure.

The domain method is related to Weihrauch's generalised computability theory: a type 2 enumeration is easily shown to give a domain representation, and it is possible to construct a type 2 enumeration for a large class of domain representations (see also Weihrauch and Schreiber [1981]). Indeed, in Stoltenberg-Hansen and Tucker [1999b] there is a series of theorems that show that for a wide class of spaces the concrete models based on effective metric algebras, axiomatic computation theory, type 2 enumerability, algebraic domain representability, and continuous domain representability are all equivalent. Thus there is a stable theory of computable functions based on concrete models.

It is important to understand fully the relationship between the concrete and abstract computability theories developed here and elsewhere: in the one direction, we construct *concrete representations of abstract models*, and in the other, we *abstract from concrete models.* Let us examine this more closely.

The various concrete computability theories discussed above have a common form, which is similar to that of the theory of computable algebras (see section 1.3), one difference being that, at present, the theory of effective computation on topological algebras is not completely settled.

Let A be a topological algebra. To compute in A, a *concrete representation*

$$\alpha : R \to A \tag{7.1}$$

of A must be made where:

(i) R is a topological algebra, made from computable data types, on which we can compute; and (*ii*) α is a surjective continuous homomorphism that allows us to compute on A by computing on R.

In particular, there is a set $\boldsymbol{Comp}_\alpha(A)$ of functions on A computable in terms of the representation (7.1).

In general terms, when comparing abstract and concrete models of computation, we may expect the following situation.

Let $\boldsymbol{AbsComp}(A)$ be a set of functions on A that is computable in an abstract model of computation (e.g. the $\boldsymbol{While}$ language).

Let $\boldsymbol{ConcRep}(A)$ be a class of concrete representations of the form $\alpha : R \to A$ (e.g., a type 2 enumeration, or domain representation).

For $\alpha \in \boldsymbol{ConcRep}(A)$, let $\boldsymbol{Comp}_\alpha(A)$ be the set of functions on A computable with the representation α.

Computing with a concrete representation R of an algebra A enables more functions to be computable than with an abstract model of computation based solely on the operations. In fact, for a class of concrete models of computation, we expect the following *abstraction condition* to hold:

$$\boldsymbol{AbsComp}(A) \subseteq \bigcap\nolimits_{\alpha \in \boldsymbol{ConcRep}(A)} \boldsymbol{Comp}_\alpha(A).$$

In the case of classes of concrete models of computation that are designed to characterise the set of functions on A that can be computed, we can further postulate (using the generalised Church–Turing thesis, cf. section 8.9):

$$\boldsymbol{While}^*(A) \subseteq \bigcap\nolimits_{\alpha \in \boldsymbol{ConcRep}(A)} \boldsymbol{Comp}_\alpha(A)$$

(compare (1.1) of Section 1.3). In the known concrete models, the computable functions are continuous, therefore the continuity of the abstract computable functions is essential.

There is much to explore in the border between abstract and concrete computability. In Stewart [1999] it is shown that if A is an effective metric algebra with enumeration α, then the $\boldsymbol{While}^*$ approximable functions on A are α-effective. The converse is not true. To bridge this gap, non-deterministic choice must be added to the '$\boldsymbol{While}$' language, and many-valued functions considered (see Tucker and Zucker [2000a]).

A theory of relations (or multi-valued functions) defined by generalised Kleene schemes has been developed in Brattka [1996; 1997]. Among several important results is an equivalence between the abstract computability model based on Kleene schemes and Weihrauch's type 2 enumerability.

The distinction between abstract and concrete models made in Tucker and Zucker [1999] has practical use in classifying the many approaches to computability in concrete structures. However, this distinction needs further theoretical refinement. One is reminded of the distinction between 'internal' and 'external', applied to higher type functionals, in Normann [1982].

8 A survey of models of computability

In this section we will survey other abstract approaches to computability on abstract algebras, and discuss two generalised Church–Turing theses: one for computability of functions, and one for specification of relations, that draw support from theorems estabilishing the equivalence of different models. Earlier we have surveyed the origins of these abstract generalisations (section 1.4) and also the 'independent' development of abstract models for computation on real and complex numbers (sections 6 and 7.10).

The alternative methods for defining ***While*** computable functions are to be found in various mathematical contexts and have various objectives. Technically, they share the abstract setting of a *single-sorted abstract structure* (i.e., an algebraic or a relational structure). Here we consider their common purpose to be the characterisation of those functions effectively computable in an abstract setting: their generalisation to a class of many-sorted abstract algebras is not difficult.

The first alternative approach we look at in some detail, namely: the class of functions defined from the operations of an algebra by the application of composition, simultaneous primitive recursion and least number search, which we call the ***μPR*** *computable functions*. This model of computation was created in Tucker and Zucker [1988] with the needs of equational and logical definability in mind. A simpler generalisation using induction schemes was made early on, in Engeler [1968a]. We have found the various recursion schemes on $\mathbb{N}$ to be a primary source of technical ideas about *functions* computable on an abstract algebra A, and a useful tool for applications.

The ***While*** computable functions can also be characterised by approaches based upon

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

We will say something about each in turn.

8.1 Computability by function schemes

We will consider computability on N-standard algebras formalised by *schemes*, which apply uniformly to all algebras of some fixed N-standard signature Σ. These generalise the schemes in Kleene [1952], for constructing functions over $\mathbb{N}$ by starting with some basic functions and applying to these *composition*, *simultaneous primitive recursion* and the *constructive least number operator*. We write $\alpha, \beta, \ldots$ for schemes.

Each scheme α will have a fixed type $u \to v$, with *domain type* u and *range type* v, both product types over Σ; we will also write $\alpha : u \to v$. The semantics of such a scheme, for each $A \in \boldsymbol{NStdAlg}(\Sigma)$(the class of N-standard Σ-algebras), will then be a function

$$[\![\alpha]\!]^A : A^u \to A^v.$$

We will usually write α^A for $[\![\alpha]\!]^A$.

We will consider four notions of computability by schemes: $\boldsymbol{PR}$, $\boldsymbol{PR}^*$, $\boldsymbol{\mu PR}$ and $\boldsymbol{\mu PR}^*$, and see how they correspond with our basic notions of computability involving $\boldsymbol{While}$ and $\boldsymbol{For}$ programs.

Definition 8.1 ($\boldsymbol{PR}$ computability). Given a standard signature Σ, we will define the family

$$\boldsymbol{PR}(\Sigma) = \langle \boldsymbol{PR}(\Sigma)_{u\to v} \mid u, v \in \boldsymbol{ProdType}(\Sigma)\rangle$$

where $\boldsymbol{PR}(\Sigma)_{u\to v}$ is the set of schemes of type $u \to v$ over Σ. Then for any scheme $\alpha \in \boldsymbol{PR}(\Sigma)_{u\to v}$and any $A \in\boldsymbol{NStdAlg}(\Sigma)$, we can define a function on A:

$$\alpha^A : A^u \to A^v$$

of type $u \to v$. These schemes generalise the schemes for primitive recursive functions over $\mathbb{N}$ in Kleene [1952]. They are generated as follows.

Basic function schemes

(*i*) *Initial functions and constants.* For each Σ-product type u, Σ-sort s and function symbol $F \in \boldsymbol{Func}(\Sigma)_{u\to s}$, there is a scheme $F \in \boldsymbol{PR}(\Sigma)_{u\to s}$. On each $A \in\boldsymbol{NStdAlg}(\Sigma)$, it defines the function

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

(*ii*) *Projection.* For all $m > 0$, $u = s_1 \times \ldots \times s_m$ and i with $1 \le i \le m$, there is a scheme $\mathsf{U}_{u,i} \in \boldsymbol{PR}(\Sigma)_{u\to s_i}$. It defines the projection function $\mathsf{U}^A_{u,i} : A^u \to A_{s_i}$ on each $A \in\boldsymbol{NStdAlg}(\Sigma)$, where

$$\mathsf{U}^A_{u,i}(x_1, \ldots, x_m) = x_i$$

for all $(x_1, \ldots, x_m) \in A^u$.

(*iii*) *Definition by cases.* For every Σ-sort s there is a scheme $\mathsf{dc} \in \boldsymbol{PR}(\Sigma)_{\mathsf{B}\times s\times s\to s}$. It defines the function $\mathsf{dc}^A : \mathbb{B} \times A_s^2 \to A_s$ on each $A \in\boldsymbol{NStdAlg}(\Sigma)$, where

$$\mathsf{dc}^A(b, x, y) = \begin{cases} x & \text{if } b = \mathsf{tt} \\ y & \text{if } b = \mathsf{ff} \end{cases}$$

for all $b \in \mathbb{B}$ and $x, y \in A_s$.

Induction: Building new function schemes from old
(*iv*) *Vectorisation.* For all Σ-product types u, v, where $v = s_1 \times \ldots \times s_n$, and for all schemes $\beta_1, \ldots, \beta_n$, where $\beta_i \in \boldsymbol{PR}(\Sigma)_{u \to s_i}$ for $i = 1, \ldots, n$, there is a scheme $\alpha \equiv \mathsf{vect}_{u,v}(\beta_1, \ldots, \beta_n) \in \boldsymbol{PR}(\Sigma)_{u \to v}$. It defines the function $\alpha^A : A^u \to A^v$ on each $A \in \boldsymbol{NStdAlg}(\Sigma)$, where

$$\alpha^A(x) \;=\; (\beta_1^A(x), \ldots, \beta_n^A(x))$$

for all $x \in A^u$.
(*v*) *Composition.* For all Σ-product types u, v, w, and for all schemes $\beta \in \boldsymbol{PR}(\Sigma)_u \to v$ and $\gamma \in \boldsymbol{PR}(\Sigma)_{v \to w}$ there is a scheme $\alpha \equiv \mathsf{comp}_{u,v,w}$ $(\beta, \gamma) \in \boldsymbol{PR}(\Sigma)_{u \to w}$. It defines the function $\alpha^A : A^u \to A^w$ on each $A \in \boldsymbol{NStdAlg}(\Sigma)$, where

$$\alpha^A(x) \;=\; \gamma^A(\beta^A(x))$$

for all $x \in A^u$.
(*vi*) *Simultaneous primitive recursion.* For all Σ-product types u, v, w, and for all schemes $\beta \in \boldsymbol{PR}(\Sigma)_{u \to v}$ and $\gamma \in \boldsymbol{PR}(\Sigma)_{\mathsf{nat} \times u \times v \to v}$ there is a scheme $\alpha \equiv \mathsf{prim}_{u,v}(\beta, \gamma) \in \boldsymbol{PR}(\Sigma)_{\mathsf{nat} \times u \to v}$. It defines the function $\alpha^A : \mathbb{N} \times A^u \to A^v$ on each $A \in \boldsymbol{NStdAlg}(\Sigma)$, where

$$\begin{array}{rcl} \alpha^A(0, x) & = & \beta^A(x) \\ \alpha^A(z+1, x) & = & \gamma^A(z,\, x,\, \alpha^A(z, x)) \end{array}$$

for all $z \in \mathbb{N}$ and $x \in A^u$.

Now, for any $A \in \boldsymbol{NStdAlg}(\Sigma)$, we define

$$\boldsymbol{PR}(A) \;=\; \langle \boldsymbol{PR}(A)_{u \to v} \mid u, v \in \boldsymbol{ProdType}(\Sigma) \rangle$$

where

$$\boldsymbol{PR}(A)_{u \to v} \;=\; \{\alpha^A \mid \alpha \in \boldsymbol{PR}(\Sigma)_{u \to v}\}.$$

It turns out that a broader class of functions provides a better generalisation of the notion of primitive recursiveness, namely $\boldsymbol{PR}^*(\Sigma)$ computability.

Definition 8.2 ($\boldsymbol{PR}^*$ computability). We define $\boldsymbol{PR}^*(\Sigma)$ to be the class of $\boldsymbol{PR}(\Sigma^*)$ schemes for which the domain and range types are in Σ, i.e.,

$$\boldsymbol{PR}^*(\Sigma) \;=_{df}\; \langle \boldsymbol{PR}(\Sigma^*)_{u \to v} \mid u, v \in \boldsymbol{ProdType}(\Sigma) \;\;\subsetneqq\;\; \boldsymbol{PR}(\Sigma^*).$$

Then any such scheme $\alpha \in \boldsymbol{PR}^*(\Sigma)_{u \to v}$ defines a function $\alpha^A : A^u \to A^v$ on each $A \in \boldsymbol{NStdAlg}(\Sigma)$.

Also $\boldsymbol{PR}^*(A)$ is the set of $\boldsymbol{PR}^*(\Sigma)$-computable functions on A.

Next we add the constructive least number operator to the $\boldsymbol{PR}$ schemes.

Definition 8.3 ($\boldsymbol{\mu PR}$ computability). The class of $\boldsymbol{\mu PR}$ *schemes* over Σ,

$$\boldsymbol{\mu PR}(\Sigma) \ = \ \langle \boldsymbol{\mu PR}(\Sigma)_u \to v \mid u, v \in \ \boldsymbol{ProdType}(\Sigma)\rangle,$$

is formed by adding to the $\boldsymbol{PR}$ schemes of Definition 8.1 the following:

(*vii*) *Least number or μ operator.* For all Σ-product types u and for all schemes $\beta \in \boldsymbol{\mu PR}_{u\times \mathsf{nat}\to\mathsf{bool}}$ there is a scheme $\alpha{\equiv}\mathsf{min}_u(\beta) \in \boldsymbol{\mu PR}(\Sigma)_{u\to\mathsf{nat}}$. It defines the function $\alpha^A : A^u \to \mathbb{N}$ on each $A \in\boldsymbol{NStdAlg}(\Sigma)$, where for all $x \in A^u$,

$$\alpha^A(x) \simeq \mu z[\beta^A(x,z) = \mathtt{tt}].$$

hat is, $\alpha^A(x) \downarrow z$ if, and only if, $\beta^A(x,y) \downarrow \mathtt{ff}$ for each $y < z$ and $\beta^A(x,z) \downarrow \mathtt{tt}$.

Also $\boldsymbol{\mu PR}(A)$ is the set of $\boldsymbol{\mu PR}(\Sigma)$computable functions on A.

Note that this scheme (as well as the scheme for simultaneous primitive recursion) uses the N-standardness of the algebra. Also, $\boldsymbol{\mu PR}$ computable functions are, in general, *partial.*

Again, however, a broader class turns out to be a better generalisation, namely:

Definition 8.4 ($\boldsymbol{\mu PR^*}$ computability). The class $\boldsymbol{\mu PR^*}(\Sigma)$ consists of those $\boldsymbol{\mu PR}(\Sigma^*)$ schemes for which the domain and range types are in Σ, i.e.,

$$\boldsymbol{\mu PR^*}(\Sigma) \ =_{df} \ \langle \boldsymbol{\mu PR}(\Sigma^*)_{u\to v} \mid u, v \in \ \boldsymbol{ProdType}(\Sigma)\rangle \ \subsetneq \ \boldsymbol{\mu PR}(\Sigma^*).$$

Also, for any $A \in\boldsymbol{NStdAlg}(\Sigma)$, $\boldsymbol{\mu PR^*}(A)$ is the set of $\boldsymbol{\mu PR^*}(\Sigma)$-computable functions on A.

We now compare the above notions of scheme computability with our notions of computability involving imperative programming languages. They correspond as follows.

Theorem 8.5. *For any N-standard Σ-algebra A,*

(*a*) $\boldsymbol{PR}(A) \ = \ \boldsymbol{For}(A)$,
(*b*) $\boldsymbol{PR^*}(A) \ = \ \boldsymbol{For^*}(A)$,
(*c*) $\boldsymbol{\mu PR}(A) \ = \ \boldsymbol{While}(A)$,
(*d*) $\boldsymbol{\mu PR^*}(A) \ = \ \boldsymbol{While^*}(A)$.

These equivalences hold uniformly over Σ.

'Uniformity over Σ' in the above theorem means (taking, for example, case (a), and writing $\boldsymbol{ForProc}(\Sigma)$ for the class of $\boldsymbol{For}(\Sigma)$ procedures) that there are effective mappings

$$\phi\colon \quad \boldsymbol{PR}(\Sigma) \to \ \boldsymbol{ForProc}(\Sigma)$$

and

$$\psi\colon \quad \boldsymbol{ForProc}(\Sigma) \to \ \boldsymbol{PR}(\Sigma)$$

(primitive recursive in the enumerated syntax) such that for all $\boldsymbol{PR}(\Sigma)$ schemes α, $\boldsymbol{For}(\Sigma)$ procedures P and N-standard Σ-algebras A,

$$[\![\phi(\alpha)]\!]^A \;=\; [\![\alpha]\!]^A \qquad \text{and} \qquad [\![\psi(P)]\!]^A \;=\; [\![P]\!]^A;$$

and similarly for parts (b), (c) and (d).

Similar uniformity results hold for the equivalences stated in the following subsections.

The above theorem can be proved by the techniques of Tucker and Zucker [1988] or Thompson [1987]. Part (a), in the classical case over $\mathbb{N}$ or $\mathbb{Z}$, was originally proved in Meyer and Ritchie [1967]. For an exposition of parts (a) and (c) in the classical case, see, for example, Brainerd and Landweber [1974], Kfoury *et al.* [1982], Davis and Weyuker [1983, Chapter 13] or Zucker and Pretorius [1993, Section 13].

Remark 8.6 (Course of values recursion). In our development above, we considered the class $\boldsymbol{PR}(\Sigma)$ of primitive recursive schemes equivalent to $\boldsymbol{For}(\Sigma)$ computability. From this we could obtain the class $\boldsymbol{PR}^*(\Sigma)$ of schemes equivalent to $\boldsymbol{For}^*(\Sigma)$ computability by operating with the same schemes $\boldsymbol{PR}$, but over the extended *array signature* Σ^*. An alternative approach for strengthening $\boldsymbol{PR}(\Sigma)$ is to maintain the signature Σ, but strengthen the *recursion scheme.* More precisely, we define the class $\boldsymbol{CR}(\Sigma)$ of *course of values recursive schemes* by *replacing* the scheme (vi) for simultaneous primitive recursion by the scheme

(vi') *Simultaneous course of values recursion.* For all Σ-product types u, v and positive integers d, and for all schemes $\beta \in \boldsymbol{CR}(\Sigma)_{u\to v}$, $\gamma \in \boldsymbol{CR}(\Sigma)_{\mathsf{nat}\times u\times v^d\to v}$ and $\delta_1,\dots,\delta_d$ where $\delta_i \in \boldsymbol{CR}(\Sigma)_{\mathsf{nat}\times u\to\mathsf{nat}}$ $(i = 1,\dots,d)$, there is a scheme

$$\alpha \equiv \mathsf{cval}_{u,v,d}(\beta,\gamma,\delta_1,\dots,\delta_d) \in \boldsymbol{CR}(\Sigma)_{\mathsf{nat}\times u\to v}.$$

It defines the function $\alpha^A : \mathbb{N}\times A^u \to A^v$ on each $A \in \boldsymbol{NStdAlg}(\Sigma)$, where

$$\alpha^A(0,x) \;=\; \beta^A(x)$$

and for $z > 0$

$$\alpha^A(z,x) \;=\; \gamma(z,\, x,\, \alpha^A(\hat{\delta}_1^A(z,x),x),\dots,\alpha^A(\hat{\delta}_d^A(z,x),x)),$$

where $\hat{\delta}_i$ are the 'reducing functions' derived from δ_i, defined by

$$\hat{\delta}_i(z,x) \;\simeq\; \min(\delta(z,x),\, z-1) \qquad \text{for } \; z > 0.$$

We also define the class $\boldsymbol{\mu CR}(\Sigma)$ of *course of values recursive schemes with the least number operator* by adjoining the scheme (vii) for the μ

operator to $\boldsymbol{CR}(\Sigma)$. We then obtain the two equivalences (cf. Theorem 8.5):

Theorem 8.7. *For any N-standard Σ-algebra A,*

(a) $\boldsymbol{CR}(A) = \boldsymbol{PR}^*(A) \quad (= \boldsymbol{For}^*(A))$
(b) $\boldsymbol{\mu CR}(A) = \boldsymbol{\mu PR}^*(A) \quad (= \boldsymbol{While}^*(A))$*.*

Part (*b*) is proved in Tucker and Zucker [1988] by showing that $\boldsymbol{\mu CR}(A) = \boldsymbol{While}^*(A)$. (Part (*a*) can be proved similarly.) This proof is more delicate than the proofs for Theorem 8.5. The direction '$\Leftarrow$' is based on local representability and term evaluation arguments.

Remark 8.8 (Some applications of the scheme models).

(1) These can be used easily in the mathematical modelling of many deterministic systems, from computers (e.g. Harman and Tucker [1993] to spatially extended non-linear dynamical systems (Holden *et al.* [1992]).
(2) The $\boldsymbol{\mu PR}$ schemes have been adapted and extended to characterise the computable relations on certain metric algebras, including the algebra of reals (Brattka [1996; 1997]).

8.2 Machine models

Perhaps the most concrete approach to generalising computability theory from $\mathbb{N}$ to an algebra A is that based upon models of machines that handle data from A. To be specific, we consider some models called *A-register machines* that generalise, to a single-sorted relational structure A, the register machine models on $\mathbb{N}$ in Shepherdson and Sturgis [1963] (see also Cutland [1980] for a development of recursive function theory using register machines); the first A-register machines appeared in Friedman [1971a].

Some of these register machine models are used in work on real number computation (Herman and Isard [1970], Shepherdson [1976] and Blum *et al.* [1989]) and have been developed further independently of the earlier literature (see our survey in section 1.4).

We will consider four types of A-register machine for an arbitrary single-sorted algebra.

A (basic) *A-register machine* has a fixed number of registers, each of which can hold a single element of A. The machine can perform the basic operations of A and decide the basic relations of A; in addition, it can relocate data and test when two registers carry the same element.

Thus, the programming language that defines the A-register machine has register names or variables $r_0, r_1, r_2, \ldots$ and labels $0, 1, 2, \ldots$, and allows instructions of the form

$$\begin{array}{l} r_\lambda \ := \ F(r_{\mu_1}, \ldots, r_{\mu_m}) \\ r_\lambda \ := \ c \\ r_\lambda \ := \ r_\mu \\ \mathsf{if}\ R(r_{\mu_1}, \ldots, r_{\mu_m})\ \mathsf{then}\ i\ \mathsf{else}\ j \end{array}$$

and, if equality is required,

$$\text{if } r_\lambda = r_\mu \text{ then } i \text{ else } j$$

wherein $\lambda, \mu, \mu_1, \mu_m \in \mathbb{N}$; $i, j \in \mathbb{N}$ are considered as labels; and F, c, R are symbols for a basic operation, constant and relation, respectively.

A program for an A-register machine is called, in Friedman [1971a], a *finite algorithmic procedure* or *fap*, and it has the form of a finite numbered or labelled list of A-register machine instructions $I_1, \ldots, I_n$. Given a formal definition of a machine state, containing the contents of registers and the label of a current instruction, is it easy to formalise an operational semantics for the finite algorithmic procedures — one in which the instructions are given their conventional meaning.

On setting conventions for input and output registers we obtain the class $\boldsymbol{FAP}(A)$ of *all partial functions on A computable by all finite algorithmic procedures on A-register machines.*

Secondly, an *A-register machine with counting* is an A-register machine enhanced with a fixed, finite number of *counting registers.* Each counting register can hold a single element of $\mathbb{N}$ and the machine is able to put 0 into a counting register, add or subtract 1 from a counting register, and test whether two counting registers contain the same number. Thus, an A-register machine with counting is an A-register machine augmented by a conventional register machine on $\mathbb{N}$. (Implicitly, this is concerned with the process of N-standardisation of the algebra A by the addition of the natural numbers $\mathbb{N}$.)

The programming language that defines the A-register machine with counting has new variables $c_0, c_1, c_2, \ldots$ for counting registers, and new instructions

$$\begin{aligned} c_\lambda &:= 0 \\ c_\lambda &:= c_\mu + 1 \\ c_\lambda &:= c_\mu - 1 \\ &\text{if } c_\lambda = c_\mu \text{ then } i \text{ else } j \end{aligned}$$

for $\lambda, \mu \in \mathbb{N}$ and $i, j \in \mathbb{N}$ considered as labels.

A program for an A-register machine with counting is called a *finite algorithmic procedure with counting* or *fapC*, and is a finite numbered list of machine instructions. Once again it is easy to give a formal semantics for the language and to rigorously define the class $\boldsymbol{FAPC}(A)$ of *all partial functions on A computable by A-register machines with counting.* The point of this model is that *it enhances computation on the abstract algebra A with computation on* $\mathbb{N}$.

The A-register machine and A-register machine with counting, and their classes of partial functions $\boldsymbol{FAP}(A)$ and $\boldsymbol{FAPC}(A)$, were introduced and studied in Friedman [1971a].

Next, an *A-register machine with stacking* is an A-register machine augmented with a stacking device into which the entire contents of the algebraic registers of the A-register machine can be copied at various points in the course of a computation.

The programming language that defines the A-register machine with stacking has a new variable s for the store or stack and the new instructions:

$$\begin{array}{l}\mathsf{stack}\ (i, r_0, \ldots, r_m)\\ \mathsf{restore}\ (r_0, \ldots, r_{j-1}, r_{j+1}, \ldots, r_m)\\ \mathsf{if}\ s = \mathsf{empty\ then}\ k\ \mathsf{else\ marker}.\end{array}$$

Here $i, j, k \in \mathbb{N}$ are considered as labels, and the machine has m registers and one stack. Intuitively, what they mean for the machine is as follows. The 'stack' instruction commands the device to copy the contents of all the registers and store at the top of a (single) stack, along with the instruction label i. The '$\mathsf{restore}$' instruction returns to the registers $r_0, \ldots, r_{j-1}, r_{j+1}, \ldots, r_m$ the values stored at the top of the stack; the value of r_j is lost (in order not to destroy the result of the subcomputation preceding the '$\mathsf{restore}$' instruction), as is the instruction label. The test instruction passes control to instruction k if the stack is empty and to the instruction indexed by the label contained in the topmost element of the stack otherwise.

A program for an A-register machine with stacking is called a *finite algorithmic procedure with stacking* or *fapS*, and is a finite numbered list of machine instructions. On formalising the semantics for the language, it is easy to define the class $\boldsymbol{FAPS}(A)$ of all partial functions on A computable by A-register machines with stacking.

Of course there are alternative designs for a stacking device of equivalent computational power. The point of this model is that first, *it enhances the bounded finite algebraic memory available in computation by an A-register machine with unbounded finite algebraic storage*, and secondly, *it does not enable us to simulate counting with natural numbers.*

Finally, an *A-register machine with counting and stacking* is an A-register machine augmented by both a counting and stacking device. A program for such a machine is called a *finite algorithmic procedure with counting and stacking* or *fapCS*, and the class of all partial functions on A computable by such machines is denoted $\boldsymbol{FAPCS}(A)$. This stack device and its associated classes of functions $\boldsymbol{FAPS}(A)$ and $\boldsymbol{FAPCS}(A)$ were introduced in Moldestad *et al.* [1980a; 1980b].

Of course, in the case of computability of the natural numbers $A = \mathbb{N}$ we have

$$\boldsymbol{FAP}(\mathbb{N}) \ = \ \boldsymbol{FAPC}(\mathbb{N}) \ = \ \boldsymbol{FAPS}(\mathbb{N}) \ = \ \boldsymbol{FAPCS}(\mathbb{N})$$

but in the abstract setting we have:

Theorem 8.9. *For any single-sorted algebra A, the inclusion relationship between the sets of functions is shown in Fig 17. Moreover, there exists an algebra on which the above inclusions are strict.*

$$\begin{tikzcd}
 & \boldsymbol{FAPC}(A) \arrow[dr] & \\
\boldsymbol{FAP}(A) \arrow[ur] \arrow[dr] & & \boldsymbol{FAPCS}(A) \\
 & \boldsymbol{FAPS}(A) \arrow[ur] &
\end{tikzcd}$$

Fig. 17.

This theorem is taken from Moldestad *et al.* [1980b]. It and other results about these models make clear the fact that, when computing in the abstract setting of an algebra A, adding

- computation on $\mathbb{N}$
- unbounded algebraic memory over A

both separately, and together, increases the computational power of the formalism.

The connection with the imperative models is easily described. Assuming the straightforward generalisation of the machine models to accommodate many-sorted algebra, we have:

Theorem 8.10. *For any standard* Σ*-algebra* A*,*

$$\begin{aligned} \boldsymbol{While}(A) &= \boldsymbol{FAP}(A), \\ \boldsymbol{While}^N(A) &= \boldsymbol{FAPC}(A), \\ \boldsymbol{While}^*(A) &= \boldsymbol{FAPCS}(A). \end{aligned}$$

Three other machine model formalisms of interest are the *finite algorithmic procedures with index registers* (*fapIR*) and *countable algorithmic procedures* (*cap*) in Shepherdson [1973] and the *generalised Turing algorithms* (*gTa*) in Friedman [1971a], all equivalent to $\boldsymbol{While}^*$ computability. In the obvious notation, we have:

Theorem 8.11. *For any standard* Σ*-algebra* A*,*

$$\boldsymbol{FAPCS}(A) = \boldsymbol{GTA}(A) = \boldsymbol{FAPIR}(A) = \boldsymbol{CAP}(A) = \boldsymbol{While}^*(A).$$

In addition, it is convenient at this point to mention Friedman's *effective definitional schemes* (*eds*) which are a simple and transparent technical device for defining and analysing computability on A. The effective definitional schemes have found a useful role in the logic of programs (see Tiuryn [1981b], for example).

Theorem 8.12. *For any standard* Σ*-algebra* A*,*

$$\boldsymbol{FAPCS}(A) = \boldsymbol{EDS}(A) = \boldsymbol{While}^*(A).$$

8.3 High-level programming constructs; program schemes

Practical programming languages, especially imperative languages, are a rich source of theoretical ideas about computation. However, their development, from the 1940s to the present, has not had a dominant role in shaping computability theories. The development of high-level constructs, abstract data types and non-deterministic constructs for algorithmic specification is clearly relevant.

The study of computability via machine models is akin to low-level programming, where there is a simple correspondence between instructions and machine operations. In high-level programming, abstractions away from the machine are achieved wherein a program statement or command can set off a sequence of machine operations. This break with programming a specific architecture increases the practical need for mathematical semantics. All our algebraic models are high-level since they are based on abstract data types that abstract from the data representations and their algorithms.

We have, of course, already studied some high-level constructs in the languages for ***While*** and ***While***** programs. However, in contemplating high-level constructs with regard to generalising computability theory, close attention must be paid to the ideas about algorithms that motivate their introduction. Clearly, recursion and iteration are distinct tools for *defining* algorithms in connection with procedures. Non-deterministic constructs, by contrast, are proposed as tools for *algorithm specification*, in order to abstract away from algorithmic implementation. Non-deterministic control and data commands, such as those in the *guarded command language*

$$\begin{gathered}\mathsf{if}\ b_1 \to S_1 [\!] \,, \ldots ,\, [\!]\ b_k \to S_k\ \mathsf{fi}\\ \mathsf{do}\ b_1 \to S_1 [\!] \,, \ldots ,\, [\!]\ b_k \to S_k\ \mathsf{od}\end{gathered}$$

(Dijkstra [1976]), or the *non-deterministic assignment*

$$x := y.\Phi(x,y),$$

where Φ is some condition relating y to x (Back [1983]), or the *random assignment*

$$x := ?$$

(Apt and Plotkin [1986]), are needed to express appropriately the design of an algorithm. We have examined some of these non-deterministic constructs in section 5, where we showed, for example, that the random assignment defines projectively semicomputable sets.

In building a generalisation, it is prudent to concentrate on making a comprehensive deterministic theory, having clear relations with 'classical' computability theory on $\mathbb{N}$, and its applications to other data types such as $\mathbb{R}$. Technically, to appreciate non-deterministic constructs, a deterministic theory is a necessary prerequisite. Unfortunately, there are unanswered questions as to the nature of the relationships between non-determinism, specification and non-computability, and (correspondingly) between determinism, implementation and computability. The programming of computations involving non-deterministic aspects of control, concurrency and communication is also an important topic that we leave unexplored. (We have dealt with synchronous concurrency in concurrent assignments and in the scheme of simultaneous primitive recursion in section 8.1.) We will return to the broad theme of programming languages and computability theory in section 8.9.

Here we will briefly draw attention to a body of early work on the computational power of elementary control and data structures.

The systematic classification of programming features such as iterations, recursions, 'goto's, arrays, stacks, queues and lists seems to have begun in earnest with Luckham *et al.* [1970] and Paterson and Hewitt [1970]. The central notions are that of a *program scheme* and its *interpretation* in a *model*, and that of the *equivalence* of program schemes in *all* models. These ideas may be considered as technical precursors of the corresponding syntactic and semantic concepts we use here, namely: program, state transformer semantics, abstract data type, equivalence on $\mathbb{K}$. The importance of a general syntactic notion of a program scheme that can be applied to abstract structures was discussed in Luckham and Park [1964] and Engeler [1967]. We note that in the latter paper computation over arbitrary classes of structures is treated in the course of analysing program termination by means of logical formulae from a simple fragment of $\mathcal{L}_{\omega_1,\omega}$; Engeler [1967] is the origin of algorithmic and dynamic logic.

The study of the power of programming features came to be known as *program schematology.* Like program verification, the subject was contemporary with, but independent of, research on programming language semantics. The necessity of introducing abstract structures in such a classification project is easy to understand. From the point of view of programming theory the equivalence of most algorithmic formalisms for computing on $\mathbb{N}$ with the partial recursive functions on $\mathbb{N}$ is a mixed blessing. This stability of the computational models illuminates our perception of the scope and limits of computer languages and architectures, and has many technical applications in the mathematical theory of computation. However, the restriction to $\mathbb{N}$ fails to support an analysis of the intuitive differences between programming with and without arrays, 'goto's, Boolean variables, and so forth.

The research on schematology has produced several program constructs and languages that are weaker than or equivalent to those of the four basic machine models discussed in section 8.2. We refer the reader to Greibach [1975] for a general introduction to schematology and, in particular, to Shepherdson [1985] for a detailed discussion of many important results and their relation to machine models. Other significant references are Constable and Gries [1972], Chandra [1973] and Chandra and Manna [1972].

High-level imperative programming models were slow to enter mainstream computability theory, despite attention being drawn to the value of this approach in Scott [1967]. Some early textbooks to feature such programming models were Brainerd and Landweber [1974], Manna [1974], Bird [1976] and Clark and Cowell [1976].

8.4 Axiomatic methods

In an axiomatic method one defines the concept of a *computation theory* as a set $\Theta(A)$ 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, $\Theta(A)$ can be required to contain the basic algebraic operators of A; be closed under operations such as *composition*; and, in particular, possess an enumeration for which appropriate *universality* and *s-m-n properties* (see, for example, Rogers [1967]) are true. Thus in section 4 we saw that $\boldsymbol{While}^*(A)$ is a computation theory in this sense.

It is important to note that computation theory definitions, of which there are a number of equivalent examples, require $\mathbb{N}$ to be part of the underlying structures A for the indexing of functions:

> *axiomatic methods specifically address N-standard structures and classes of N-standard structures.*

With reference to the definition sketched above, the following theorem is of importance here:

Theorem 8.13. *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 computational theory $\Theta(A)$ on A.*

The definition of a computation theory used here is from Fenstad [1975; 1980] which take up the ideas in Moschovakis [1971]. We note that

> *the $\boldsymbol{While}$ computable functions coincide with the prime computable functions of Moschovakis.*

Theorem 8.13 can be deduced using work in Moldestad *et al.* [1980b]; see also Fenstad [1980, Chapter 0].

The development of axiomatisations of computable functions includes Strong [1968] and Wagner [1969]. The axiomatisation of subrecursive functions is tackled in Heaton and Wainer [1996].

8.5 Equational definability

One of the earliest formalisations of effective computability was by means of *functions effectively reckonable in an equational calculus*, a method known as *equational* or *Herbrand–Gödel–Kleene* definability. This was the method employed to define the recursive functions in important works such as Church [1936] and Kleene [1952].

Equational definability may be generalised from $\mathbb{N}$ to an arbitrary algebra A with the natural result that, if A is an N-standard structure, equational definability is equivalent with $\boldsymbol{While}^*$ computability. The first attempt at such a generalisation is Lambert [1968]. We sketch a simpler treatment from Moldestad and Tucker [1981], adapted to many-sorted algebras.

First we choose a language $\boldsymbol{Eqn}= \boldsymbol{Eqn}(\Sigma)$ for defining equations over a signature Σ and transforming them in simple deductions. Let $\boldsymbol{Eqn}$ have constants $a, b, c, \ldots$ and variables $x, y, z, \ldots$ for data; and variables $p, q, r, \ldots$ for functions. Using the basic operations of the signature, we inductively define Σ-terms $t, \ldots$ in the usual way. An *equation* in $\boldsymbol{Eqn}$ is an expression $e \equiv (t_1 = t_2)$, where t_1 and t_2 are terms of the same sort.

A *deduction* of an equation e from a set of equations E is a list $e_1, \ldots, e_k$ of equations such that for each $i = 1, \ldots, k$ one of the following holds:

(i) $e_i \in E$;

(ii) e_i is obtained from e_j for some $j < i$ by replacing every occurrence of a variable x in e_j by a constant c;

(iii) e_i is obtained from e_j for some $j < i$ by replacing at least one occurrence of a subterm t of e_j by a constant c, where t has no free variables, and for some $j' < i$, $e_{j'} \equiv (t = c)$.

An equation e is defined to be *formally derivable* or *deducible* from E, written $E \vdash e$, if there is a deduction of e from E.

Thus, it remains to formulate equational deductions with respect to a given algebra A of signature Σ in order to formulate what it means for a function f on A to be *equationally definable* on A. This is essentially giving our system a semantics. The first semantical problem is to allow the basic operations of A to play a role in deductions from a set of equations E, and this is accomplished by permitting

$$E \vdash p(c_1, \ldots, c_n) = c \quad \text{if} \quad F^A(c_1^A, \ldots, c_n^A) = c^A.$$

This is the reason why we add the constants to $\boldsymbol{Eqn}$.

The second semantical problem is to prove a single-valuedness property of the form:

$$E \vdash p(c_1, \ldots, c_n) = a_1 \;\text{ and }\; E \vdash p(c_1, \ldots, c_n) = a_2 \implies a_1 = a_2.$$

This done, we can define $f : A^u \to A$ to be *equationally definable over* A if for some finite set of equations E and some function symbol p,

$$E \vdash p(c_1, \ldots, c_n) = c \implies f(c_1^A, \ldots, c_n^A) = c^A$$

for all constants of $\boldsymbol{Eqn}$.

Let $\boldsymbol{Eqn}(A)$ denote the set of all equationally definable functions on A.

Theorem 8.14.

(a) *For any standard Σ-algebra A,*

$$\boldsymbol{Eqn}(A) = \boldsymbol{FAPS}(A).$$

(b) *For any N-standard Σ-algebra A,*

$$\boldsymbol{Eqn}(A) = \boldsymbol{FAPCS}(A) = \boldsymbol{While}^*(A).$$

8.6 Inductive definitions and fixed-point methods

The familiar definition of the recursive functions on $\mathbb{N}$ based on the primitive recursion scheme of Dedekind and Gödel, and the least number operator of Kleene, appeared in Kleene [1936]. Kleene provided a thorough revision of the process of recursion on $\mathbb{N}$ sufficiently general to include recursion in objects of higher function type: see Kleene [1959; 1963]. In Platek [1966] there is an abstract account of higher-type recursion.

Studies of higher type inductive definitions have been taken up by D. Scott and Y. Ershov, whose work forms part of *domain theory* (see, for example, Stoltenberg-Hansen *et al.* [1994]). The central technical notion is that of fixed points of higher type operators.

In Moldestad *et al.* [1980a] Platek's methods were analysed and classified in terms of the machine models of section 8.2. Like equational definability, definability by fixed-point operators applies to an arbitrary algebra A and is there equivalent to fapS computability. Thus, this notion coincides with $\boldsymbol{While}^*$ definability in an N-standard structure. We will sketch the method (adapted to many-sorted algebras).

First we construct the language $\boldsymbol{FPD}$= $\boldsymbol{FPD}(\Sigma)$ for defining fixed-point operators. Let $\boldsymbol{FPD}$ have the data and function variables of $\boldsymbol{Eqn}$, the equation language of section 8.5. Using the basic operations of the signature Σ and the λ-abstraction notation, we create a set of *fixed-point terms* of both data and function types:

$$t \ ::= \ x \mid p \mid F \mid T(t_1, \ldots, t_n) \mid \mathsf{fp}[\lambda p \cdot y_1, \ldots, y_n \cdot t].$$

Here p is a function variable, F is a basic operation of Σ, T is a term of type function, $t_1, \ldots, t_n$ and t are terms of type data, and $y_1, \ldots, y_n$ are data variables.

Each term defines a function on each algebra A of signature Σ. The definition of the semantics of terms is by induction on their construction, the terms of the form

$$\mathsf{fp}[\lambda p \cdot y_1, \ldots, y_n \cdot t]$$

being assigned the unique least fixed point of the continuous monotonic operator defined by the notation $\lambda p \cdot y_1, \ldots, y_n \cdot t$.

A function $f : A^u \to A$ is *definable by fixed-point terms over* A if there is a term t such that for all $x \in A^u$, $f(x) \simeq t(x)$.

Let $\boldsymbol{FPD}(A)$ denote the set of all functions definable by fixed-point terms over A.

Theorem 8.15.

(a) For any standard Σ-algebra A,

$$\boldsymbol{FPD}(A) \;=\; \boldsymbol{FAPS}(A).$$

(b) For any N-standard Σ-algebra A,

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

For more details see Moldestad *et al.* [1980a].

An approach to computation on abstract data types, alternative to that presented in this chapter, is the development in Feferman [1992a; 1992b] of a theory of *abstract computation procedures*, defined by *least fixed-point schemes*, influenced by Moschovakis [1984; 1989]. The 'abstract data types' here are classes of structures similar to our standard partial many-sorted algebras, abstract in the sense that they are closed under isomorphism, and the computation procedures are abstract in the sense that they are isomorphism invariant on the data types; cf. Theorem 3.24. Types (or sorts) and operations can have an intensional or extensional interpretation.

Another treatment of inductive definitions (also influenced by Moschovakis) and a survey of their connections with machine models is given in Hinman [1999].

8.7 Set recursion

Given a structure on A one can construct a set-theoretic hierarchy $\boldsymbol{H}(\mathrm{A})$ over A, taking A as so-called urelements, and, depending upon the construction, develop a recursion theory on $\boldsymbol{H}(\mathrm{A})$. This is the methodology in Normann [1978] where combinatorial operations on sets are employed to make a generalisation of computability. In Moldestad and Tucker [1981], Normann's set recursion schemes are applied to the domain $\boldsymbol{HF}(A)$, the set of hereditarily finite subsets, so as to invest the general construction with computational content. $\boldsymbol{HF}(A)$ is inductively defined as follows:

(*i*) $A \subseteq \boldsymbol{HF}(A)$;

(*ii*) if $a_1, \ldots, a_n \in \boldsymbol{HF}(A)$ then $\{a_1, \ldots, a_n\} \in \boldsymbol{HF}(A)$, $n \geq 0$.

Thus, $\emptyset \in \boldsymbol{HF}(A)$, a copy of $\mathbb{N}$ is imbedded in $\boldsymbol{HF}(A)$, and copies of A^n ($n = 2, 3, \ldots$) are embedded in $\boldsymbol{HF}(A)$. From computability on $\boldsymbol{HF}(A)$ a notion of computability on A, *set recursiveness*, is easily obtained. Then, writing $\boldsymbol{SR}(A)$ for the class of set-recursive functions on A, we have:

Theorem 8.16. *For any standard Σ-algebra A,*

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

8.8 A generalised Church–Turing thesis for computability

The $\boldsymbol{While}^*$ computable functions are a mathematically interesting and useful generalisation 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 generalisation 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 Tucker and Zucker [1988].

Consider the following naive attempt at a generalisation of the Church–Turing thesis.

Thesis 8.17 (A naive generalised Church–Turing thesis for computability).

(*a*) The functions 'effectively computable' on a many-sorted algebra A are precisely the functions $\boldsymbol{While}^*$ computable on A.

(*b*) The families of functions 'effectively computable' uniformly over a class $\mathbb{K}$ of such algebras are precisely the families of functions uniformly $\boldsymbol{While}^*$ computable over $\mathbb{K}$.

Consider now: what can be meant by 'effective computability' on an abstract algebra or class of algebras?

In the standard situation of calculation with $\mathbb{N}$, the idea of effective computability is complicated, as it is made up from many philosophical and mathematical ideas about the nature of finite computation with finite or concrete elements. For example, its analysis raises questions about the mechanical representation and manipulation of finite symbols; about the equivalence of data representations; and about the formalisation of *constituent concepts* such as *algorithm*; *deterministic procedure*; *mechanical procedure*; *computer program*; *programming language*; *formal system*; *machine*; and the functions definable by these entities.

The idea of effective computability is particularly deep and valuable because of the close relationships that can be shown to exist between its distinct constituent concepts. However, only *some* of these constituent concepts can be reinterpreted or generalised to work in an abstract setting; and hence the general concept, and term, of 'effective computability' does not belong in a generalisation of the Church–Turing thesis. In addition, since finite computation on finite data is truly a fundamental phenomenon, it is approriate to preserve the term with its established special meaning.

In seeking a generalisation of the Church–Turing thesis we are trying to make explicit certain primary informal concepts that are formalised 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 structures to abstract structures, and hence three points of view from which to consider the $\boldsymbol{While}^*$ computable functions.

First, 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 is 'a concrete algebra considered unique only up to isomorphism'.

Secondly, there is *formal logic*, which is a theory about the scope and limits of axiomatisations and formal reasonings. Here structures and classes of structures are used to discuss formal systems and axiomatic theories in terms of consistency, soundness, completeness, and so on.

Thirdly, in programming language theory, there is *data type theory*, which is about data types that users 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 a computation. Each of these theories, because of its special concerns and technical emphasis, leads to its own theory of computability on abstract structures.

Suppose, for example, the $\boldsymbol{While}^*$ 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 and fields, etc.) leads to a formalisation of a generalised Church–Turing thesis tailored to the language and use of an algebraist:

Thesis 8.18 (Generalised Church–Turing thesis for algebraic computability).

(*a*) The functions computable by finite deterministic algebraic algorithms on a many-sorted algebra A are precisely the functions $\boldsymbol{While}^*$ computable on A.

(*b*) The families of functions uniformly so computable over a class $\mathbb{K}$ of such algebras are precisely the families of functions uniformly $\boldsymbol{While}^*$ computable over $\mathbb{K}$.

An account of computability on abstract structures from the point of view of algebra is given in Tucker [1980].

Now suppose that the $\boldsymbol{While}^*$ computable functions are considered with the needs of computer science in mind. The context of studies of data types, programming and specification constructs, etc., leads to a formulation tailored to the language and use of a computer scientist:

Thesis 8.19 (Generalised Church–Turing thesis for programming

languages). Consider a deterministic programming language over an abstract data type $\boldsymbol{dt}$.

(a) The functions that can be programmed in the language on an algebra A which represents an implementation of $\boldsymbol{dt}$, are the same as the functions $\boldsymbol{While}^*$ programmable on A.

(b) The families of functions that can be programmed in the language uniformy over a class $\mathbb{K}$ of implementations of $\boldsymbol{dt}$, are the same as the families of functions $\boldsymbol{While}^*$ programmable over $\mathbb{K}$.

The thesis has been discussed in Tucker and Zucker [1988].

The logical view of computable functions and sets, with its focus on axiomatic theories and reasoning, is a more abstract view of computation than the view from algebra and data type theory, with their focus on algorithms and programs. The logical view is directed at the specification of computations.

8.9 A Church–Turing thesis for specification

In the course of our study, we have met logical and non-deterministic languages that define in a natural way the projectively computable sets (and, equivalently, the projectively semicomputable sets). These languages are motivated by the wish to specify problems and computations, and to leave open all or some of the details of the programs that will solve the problems and perform the computations.

To better understand the role of the projective computable sets, we introduce the idea of an *algorithmic specification language* which includes some ideas about *non-deterministic programming languages*. The properties that characterise an algorithmic specification language are forms of algorithmically validating a specification. An algorithmic specification language is an informal concept that is intended to complement that of a deterministic programming language. The problem we consider is that of formalising the informal notion of an algorithmic specification language by means of a generalised Church–Turing thesis for specification, based on projectively computable sets.

There are four basic components to a computation:

(0) a *data type*;

(1) a *specification of a task* to be performed or problem to be solved;

(2) *specifications for algorithms* whose input/output behaviour accomplishes the task or solves the problem; and

(3) *algorithms* with appropriate i/o behaviour.

We model mathematically these components of a computation, by assuming that:

(0°) a data type is a *many-sorted algebra*, or class of algebras;

(1°) a specification of the task or problem is defined by a *relation* on the algebra;

(2°) specifications of algorithms for the task or problem are defined by *functions* on the algebra; and
(3°) algorithms are defined by *programs* that compute functions on the algebra.

Usually, the relations, functions and programs are defined uniformly over a class of algebras.

Given a specification

$$S \subseteq A^u \times A^v$$

on an algebra A, the *task* is: for all $x \in A^u$, to calculate all or some $y \in A^v$ such that $R(x,y)$ holds, if any such y exist. The set

$$D =_{df} \{x \in A^u \mid \exists y R(x,y)\}$$

may be called the *domain of the task*.

Thus the task of computing the relation can be expressed in the following *functional form*:

$$\hat{R} : A^u \to \mathcal{P}(A^v)$$

(where $\mathcal{P}(A^v)$ is the power set of A^v), defined for $x \in A^u$ by

$$\hat{R}(x) =_{df} \{y \in A^v \mid R(x,y)\}.$$

Quite commonly, the task is 'simplified' to computing one or more so-called *selection functions* for the relation.

Definition 8.20 (Selection functions). Let $R \subseteq A^u \times A^v$ be a relation. A function

$$f : A^u \to A^v$$

is a *selection function* for R if

(*i*) $\forall x[\exists y R(x,y) \Rightarrow f(x)\downarrow$ and $R(x, f(x))]$; and
(*ii*) $\forall x[f(x)\downarrow \Rightarrow R(x, f(x))]$.

Notice that the domain and range of a selection function f are projections:

$$\begin{array}{lcl} \boldsymbol{dom}(f) & = & \{x \in A^u \mid \exists y R(x,y)\}, \\ \boldsymbol{ran}(f) & = & \{y \in A^v \mid \exists x R(x,y)\}. \end{array}$$

Note also that

> *any partial function f is definable as the unique selection function for its graph $G(f) = \{(x,y) \mid f(x)\downarrow y\}$.*

Other sets of use in specification theory can be derived from these sets (e.g. weakest preconditions and strongest postconditions — see Tucker and Zucker [1988]).

To define and reason about computations on a data type, we must define a class of relations, functions and programs on an algebra A. The key ideas are those of formal languages that define functions, called *programming languages*, and those that define relations, called *specification languages*.

The relation between a programming language $\boldsymbol{P}$ and a specification language $\boldsymbol{S}$ is that of *satisfaction*

$$\models \ \subseteq \ \boldsymbol{P} \times \boldsymbol{S}$$

defined for $p \in \boldsymbol{P}$ and $s \in \boldsymbol{S}$ by

$$p \models s \iff \textit{the function defined by } p \textit{ is a selection function for the relation defined by } s.$$

What properties of relations are needed for a specification language?

We propose two properties. The first is that it should be possible to '*validate*' ('test', 'check', ...) data against each specification. A basic question is, therefore:

> *For any given data x and y, can we validate whether or not the given y is a valid output for the given input x?*

We define the following informal concept:

Definition 8.21 (Algorithmic specification language). An *algorithmic specification language* is a language in which any data for any task can be validated.

The process of validation depends on the relations defined by the specification. Our theory of computability on algebras presents three cases:

Definition 8.22 (Algorithmic validation of specifications). Let S be a specification language.

(a) S has *decidable validation* if each relation it defines is computable.
(b) S has *semidecidable validation* if each relation it defines is semicomputable.
(c) S has *projectively decidable validation* if each relation it defines is projectively computable.

The second property is '*adequacy*'. A specification language may be quite expressive, containing specifications for tasks for which there does not exist an algorithmic solution. It should, however, be capable of expressing at least all those tasks which are algorithmic. We therefore define the following informal concept:

Definition 8.23 (Adequate specification language). A specification language is *adequate* if all computations can be specified in it.

To specify a function is to define a relation for which it is a selection function. Recall that any function is definable as the unique selection function for its graph. Consider the adequacy of an algorithmic specification language with each of the three types of algorithmic validation above.

(a) If a specification language has *decidable validation* then not every partial computable function can be specified uniquely, since the graph of a partial computable function need not be computable (by a standard result of classical computation theory).

(b) If a specification language has *semidecidable validation* then every partial computable function can be specified uniquely, since the graph of a partial computable function is semicomputable.

(c) Thus a specification language with *projectively decidable validation* is also adequate for the definition of all possible computations.

Furthermore, there are many occasions when the adequacy of a specification formalism demands greater expressiveness. The problem is to allow a class of specifications that extends that of the semicomputable relations, and yet *retains some chance of an effective test or check.*

For example, let $E \subseteq A$ be a computable subset of an algebra A and consider the membership relation for the subalgebra $\langle E \rangle$ of A generated by E; using established notations, this is defined by:

$$a \in \langle E \rangle \Leftrightarrow \exists k \geq 0 \exists e_1, \ldots, e_k \in E \exists t \in \boldsymbol{Term}(\Sigma)[\boldsymbol{TE}(t, e_1, \ldots, e_k) = a].$$

This relation is not semicomputable, nor even projectively computable over A, but projectively computable over A^*.

In examples of the above kind, the computation and specification of y from x involves a finite sequence of auxiliary data z^* that is 'hidden' from R, but can be recovered from the specification and algorithm. This type of specification R has the form

$$R(x, y) \Leftrightarrow \exists z^* R_0(x, y, z^*),$$

where R_0 is computable. That is, R is projectively computable (or semicomputable) over A^*.

This is a weak form of the concept of a specification that can be validated algorithmically.

We have seen a number of methods, involving logical and non-deterministic languages, all of which define the projections of computable sets (or, equivalently, of semicomputable sets); we recall them briefly:

(i) *Projections in first-order languages.* Consider the first-order languages

$$\boldsymbol{Lang}(\Sigma) \qquad \text{and} \qquad \boldsymbol{Lang}(\Sigma^*)$$

over the signatures Σ and Σ^* with their usual semantics. The relations that are $\boldsymbol{\Sigma}_1$ definable in these languages are the projectively $\boldsymbol{While}$ and $\boldsymbol{While}^*$ computable sets.

(*ii*) *Horn clause languages.* In Tucker and Zucker [1989; 1992a] we studied a generalisation of logic programming languages based on Horn clauses, and a semantics based on resolution. The relations definable in this specification-cum-programming language were the projectively $\boldsymbol{While}^*$ computable sets. The logic programming model was shown to be equivalent to certain classes of logically definable functions (Fitting [1981]).

(*iii*) *Other definabilities.* In Fitting [1981] the relations are shown to be equivalent to those definable in Montague [1968]. Hence, by work in Gordon [1970], these all coincide with the search computable functions of Moschovakis [1969a]. A summary of these results is contained in Tucker and Zucker [1988, section 7].

(*iv*) *Non-deterministic programming languages.* Finally, recall from section 5 that we have seen that constructs allowing non-deterministic choices of data, state, or control in programming languages also lead to the projectively computable sets. In particular, the models

$\boldsymbol{While}^*$ *computability with initialisation* and
$\boldsymbol{While}^*$ *computability with random assignments*

were analysed.

The equivalence results suggest that the concepts of projective computablity and semicomputability are *stable* in the analysis of models of specification. The concept of an algorithmic specification language in its weak form, together with all the above equivalence results, leads us to formulate the following generalised Church–Turing thesis for specification, to complement that for computation:

Thesis 8.24 (Generalised Church–Turing thesis for specification on abstract data types). Consider an adequate algorithmic specification language $\boldsymbol{S}$ over an abstract data type $\boldsymbol{dt}$.

(*a*) The relations on a many-sorted algebra A implementing $\boldsymbol{dt}$ that can be specified in $\boldsymbol{S}$ are precisely the projectively $\boldsymbol{While}^*$ computable relations on A.

(*b*) The families of relations over a class $\mathbb{K}$ of such algebras implementing $\boldsymbol{dt}$, that can be specified in $\boldsymbol{S}$, uniformly over $\mathbb{K}$, are precisely the families of uniformly projectively $\boldsymbol{While}^*$ computable relations over $\mathbb{K}$.

This thesis has been discussed in Tucker and Zucker [1988].

8.10 Some other applications

Computations on many-sorted algebras lead to many investigations and applications. We conclude by mentioning two.

(*i*) *Provably computable selection functions.* In this chapter we have not dealt with proof systems, or the connections between *provability* and *computability*. In Tucker and Zucker [1988] we developed one such connection, namely the use of proof systems for *verifying program correctness*.

Another connection is based on classical proof theory, and its application to computability on the naturals. In Tucker *et al.* [1990] and Tucker and Zucker [1993] we investigated the *generalisation of a particular problem in classical proof theory* to the context of N-standard many-sorted signatures and algebras. Specifically, we developed classical and intuitionistic formal systems for theories over N-standard signatures Σ. We showed, in the case of universal theories (i.e., theories with axioms containing only universal quantifiers) that, in either of these systems:

if an existential assertion is provable, then it has a $\boldsymbol{PR}^*(\Sigma)$ *selection function.*

(Recall the discussion of selection functions in section 8.9.) It follows that

if a $\boldsymbol{\mu PR}^*(\Sigma)$ *function scheme is provably total, then it is extensionally equivalent over* Σ *to a* $\boldsymbol{PR}^*(\Sigma)$ *scheme.*

The methods are proof-theoretical, involving cut elimination. These results generalise to an abstract setting previous results of Parsons [1971; 1972] and Mints [1973] over the natural numbers.

(*ii*) *Computation on stream algebras.* A *stream over a set* A is a sequence of data from A

$$\ldots,\ a(t),\ \ldots$$

indexed by time $t \in \boldsymbol{T}$. Discrete time $\boldsymbol{T}$ is modelled by the naturals $\mathbb{N}$, and the space of all streams over A is the set $[\mathbb{N} \to A]$ of functions from $\mathbb{N}$ to A.

Streams are ubiquitous in computing. In hardware, where clocking and timing are important, most systems process streams (see McEvoy and Tucker [1990] and Möller and Tucker [1998]). Models of stream computation are needed for any wide spectrum specification method such as FOCUS (see Broy *et al.* [1993]).

A general theory of stream processing is given in Stephens [1997].

There is a strong need to incorporate stream computation in a general theory of computation on many-sorted algebras. Some first steps in this direction, partly motivated by technical questions arising in an algebraic study of stream processing by *synchronous concurrent algorithms* (see Thompson and Tucker [1991]), were taken in Tucker and Zucker [1994; 1998].

Another approach to this problem has been developed in Feferman [1996], within (an extensional version of) the framework of computation theory on abstract data types presented in Feferman [1992a; 1992b], as summarised in section 8.6.

The relationship between these two theories of stream computations remains to be investigated.

We conclude with a brief survey of the former approach (Tucker and Zucker [1994]). Here we consider the following problem: Given a algebra A (which we suppose for notational simplicity is single-sorted), consider stream transformations of the form

$$f: [\mathbb{N} \to A]^m \to [\mathbb{N} \to A]$$

as well as their *cartesian* or *uncurried* forms

$$\boldsymbol{cart}(f): [\mathbb{N} \to A]^m \times \mathbb{N} \to A$$

defined by

$$\boldsymbol{cart}(f)(\xi, n) \;=\; f(\xi)(n).$$

We ask the following questions.

> *For any algebra A, what are the computable stream transformations over A?*
>
> *What is their relation to the computable functions on A?*

To answer these, we extend A to the stream algebra $\bar{A}$ (section 2.8), and consider various models of computation $\mathrm{MC}(A)$ over A, as well as the corresponding models of computation $\mathrm{MC}(\bar{A})$ over $\bar{A}$. These models of computation MC include the schemes

$$\boldsymbol{PR}, \quad \boldsymbol{PR}^*, \quad \boldsymbol{\mu PR}, \quad \boldsymbol{\mu PR}^*.$$

We also consider the operation of *stream abstraction* or *currying* inverse to $\boldsymbol{cart}$: for any function

$$g: D \times \mathbb{N} \to A,$$

where D is any cartesian product of carriers of $\bar{A}$, construct the function

$$\boldsymbol{\lambda abs}(g): D \to [\mathbb{N} \to A]$$

defined by

$$(\boldsymbol{\lambda abs}(g))(d)(n) \;=\; g(d, n).$$

The addition of this construct to models of computation MC leads to models of computation $\lambda\mathrm{MC}(\bar{A})$:

$$\lambda\boldsymbol{PR}, \quad \lambda\boldsymbol{PR}^*, \quad \lambda\mu\boldsymbol{PR}, \quad \lambda\mu\boldsymbol{PR}^*.$$

We investigate the relationships between these various models; for example, we prove some *computational conservativity* results: for any function f on A,

$$f \in \boldsymbol{PR}(\bar{A}) \iff f \in \boldsymbol{PR}(A)$$

and similarly for $\lambda\boldsymbol{PR}^*$, $\lambda\mu\boldsymbol{PR}$ and $\lambda\mu\boldsymbol{PR}^*$. We also show that computability is *not invariant under Cartesian forms*, i.e., there are functions f such that

$$f \notin \boldsymbol{PR}(\bar{A}) \qquad \text{but} \qquad \boldsymbol{cart}(f) \in \boldsymbol{PR}(\bar{A})$$

and similarly for $\lambda\boldsymbol{PR}^*$, $\lambda\mu\boldsymbol{PR}$ and $\lambda\mu\boldsymbol{PR}^*$. Further, '$\lambda$-elimination' does not hold, i.e., there are functions f such that

$$f \in \lambda\boldsymbol{PR}(\bar{A}) \qquad \text{but} \qquad f \notin \boldsymbol{PR}(\bar{A});$$

for example, the function $\mathsf{const}^A\colon A \to [\mathbb{N} \to A]$, which maps data $a \in A$ to the stream $\mathsf{const}^A(a) \in [\mathbb{N} \to A]$ with constant value a, is in $\lambda\boldsymbol{PR}(\bar{A})$ but not in $\boldsymbol{PR}(\bar{A})$, or even in $\boldsymbol{\mu PR}^*(\bar{A})$. However, we do have *$\lambda$-elimination + cartesian form*, in the sense that

$$f \in \lambda\boldsymbol{PR}(\bar{A}) \iff \boldsymbol{cart}(f) \in \boldsymbol{PR}(\bar{A}),$$

and similarly for $\lambda\boldsymbol{PR}^*$, $\lambda\mu\boldsymbol{PR}$ and $\lambda\mu\boldsymbol{PR}^*$.

There are advantages to working with stream transformers via their cartesian forms. It is then true, but difficult to show, that the class of computable functions so defined is closed under composition (Stephens and Thompson [1996]).

Suppose now we ask for a model of computability to satisfy a *generalised Church–Turing thesis* for stream computations. The model $\boldsymbol{\mu PR}^*(\bar{A})$ obtained from our previous generalised Church–Turing thesis on arbitrary standard algebras (section 8.8, substituting $\bar{A}$ for A) would be too weak, since (as we have seen) even the constant stream function const^A is not computable in it. However, we can show, as a corollary of the *computational conservativity* results, that the following models of computation are equivalent:

$$\lambda\mu\boldsymbol{PR}(\bar{A}), \quad \lambda\mu\boldsymbol{PR}(\bar{\bar{A}}), \quad \lambda\mu\boldsymbol{PR}^*(\bar{A}), \quad \lambda\mu\boldsymbol{PR}(\bar{A}^*).$$

This shows that the model

$$\lambda\mu\boldsymbol{PR}(\bar{A})$$

is robust, and suggests it as a good candidate for a *generalised Church–Turing thesis for stream computations.*

(*iii*) *Equational specification of computable functions.* Many functions are defined as solutions of systems of equations from, for example, datatype theory or real analysis. Sometimes considerable effort is expended in devising algorithms to implement or compute these functions; this is the *raison d'être* of numerical methods for differential and integral equations.

It is possible to develop a theory of equational specifications for functions on algebras, including topological algebras. In Tucker and Zucker [2000b] it is shown that any $\boldsymbol{While}^*$ approximable function on a total metric algebra is the unique solution of a finite system of conditional equations, which can be chosen uniformly over all algebras of the signature, and over all $\boldsymbol{While}^*$ computations. The converse however is not true; specifiability by conditional equations is a more powerful device than $\boldsymbol{While}^*$ approximation – how much more powerful, remains to be investigated.

References

The great majority of the publications listed here are referenced in the text. Some papers, however, marked with a star next to the date, are not so referenced. They are included here as a guide to further reading, either because they shed some light on the historical development of the subject, or because they provide useful further information in certain areas, such as program verification and computation on the reals.

[American Standards Association, 1963] American Standards Association. Proposed American standard flowchart symbols for information processing, *Communications of the Association for Computing Machinery* 6:601–604, 1963.

[Apt, *1981] K. R. Apt. Ten years of Hoare's logic: A survey — Part 1, *ACM Transactions on Programming Languages and Systems* 3:431–483, 1981.

[Apt and Plotkin, 1986] K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment, *Journal of the Association for Computing Machinery*, 33:724–767, 1986.

[Arbib and Give'on, *1968] M. A. Arbib and Y. Give'on. Algebra automata I: Parallel programming as a prolegomena to the categorical approach, *Information and Control* 12:331–345, 1968.

[Ashcroft and Manna, *1971] E. Ashcroft and Z. Manna. The translation of 'go to' programs to 'while' programs, *Information Processing*, 71:147–152, 1971.

[Ashcroft and Manna, *1974] E. Ashcroft and Z. Manna. Translation program schemas to while-schemas, *SIAM Journal of Computing* 4:125–146, 1974.

[Asser, *1960] G. Asser. Rekursive Wortfunktionen, *Zeitschrift für mathematische Logik und Grundlagen der Mathematik* 6:258–278, 1960.

[Asser, 1961] G. Asser. Funktionen-Algorithmen und Graphschemata, *Zeitschrift für mathematische Logik und Grundlangen der Mathematik* 7:20–27, 1961.

[Back, 1983] R. J. R. Back. A continuous semantics for unbounded nondeterminism, *Theoretical Computer Science* 23:187–210, 1983.

[de Bakker, 1980] J. W. de Bakker. *Mathematical Theory of Program Correctness*, Prentice Hall, 1980.

[Banachowski *et al.*, *1977] L. Banachowski, A. Kreczmar, G. Mirkowska, H. Rasiowa and A. Salwicki. An introduction to algorithmic logic, mathematical investigations in the theory of programs. In *Mathematical Foundations of Computer Science*, A. Mazurkiewicz and Z. Pawlak eds, pp. 7–99, Banach Center Publications, 1977.

[Barwise, *1975] J. Barwise. *Admissible Sets and Structures*, Springer-Verlag, 1975.

[Becker, 1986] E. Becker. On the real spectrum of a ring and its application to semialgebraic geometry, *Bulletin of the American Mathematical*

Society (N.S.) 15:19–60, 1986.

[Bergstra and Tucker, *1980a] J. A. Bergstra and J. V. Tucker. A natural data type with a finite equational final semantics specification but no effective equational initial semantics specification. *Bulletin of the European Association for Theoretical Computer Science* 11:23–33, 1980.

[Bergstra and Tucker, *1980b] J. A. Bergstra and J. V. Tucker. A characterisation of computable data types by means of a finite equational specification method. In *7th International Colloquium on Automata, Languages and Programming, Noordwijkerhout, The Netherlands, July 1980.* J. W. de Bakker and J. van Leeuwen eds, Lecture Notes in Computer Science 85, pp. 76–90, Springer-Verlag, 1980.

[Bergstra and Tucker, *1982a] J. A. Bergstra and J. V. Tucker. The completeness of the algebraic specification methods for data types, *Information & Control* 54:186–200, 1982.

[Bergstra and Tucker, *1982b] J. A. Bergstra and J. V. Tucker. Some natural structures which fail to possess a sound and decidable Hoare-like logic for their while-programs, *Theoretical Computer Science* 17:303–315, 1982.

[Bergstra and Tucker, *1982c] J. A. Bergstra and J. V. Tucker. Expressiveness and the completeness of Hoare's logic, *Journal of Computer & Systems Science* 25:267–284, 1982.

[Bergstra and Tucker, *1982d] J. A. Bergstra and J. V. Tucker. Two theorems about the completeness of Hoare's logic, *Information Processing Letters*, 15:143–149, 1982.

[Bergstra and Tucker, *1983a] J. A. Bergstra and J. V. Tucker. Hoare's logic and Peano's arithmetic, *Theoretical Computer Science* 22:265–284, 1983.

[Bergstra and Tucker, *1983b] J. A. Bergstra and J. V. Tucker. Initial and final algebra semantics for data type specifications: two characterization theorems, *SIAM Journal of Computing* 12:366–387, 1983.

[Bergstra and Tucker, *1984a] J. A. Bergstra and J. V. Tucker. Hoare's logic for programming languages with two data types, *Theoretical Computer Science* 28:215–221, 1984.

[Bergstra and Tucker, *1984b] J. A. Bergstra and J. V. Tucker. The axiomatic semantics of programs based on Hoare's logic, *Acta Informatica* 21:293–320, 1984.

[Bergstra and Tucker, *1987] J. A. Bergstra and J. V. Tucker. Algebraic specifications of computable and semicomputable data types, *Theoretical Computer Science* 50:137–181, 1987.

[Bergstra *et al.*, *1982] J. A. Bergstra, J. Tiuryn and J. V. Tucker. Floyd's principle, correctness theories and program equivalence, *Theoretical Computer Science* 17:113–149, 1982.

[Bird, 1976] R. Bird. *Programs and Machines: An Introduction to the Theory of Computation*, John Wiley and Sons, 1976.

[Bishop, 1967] E. Bishop. *Foundations of Constructive Analysis*, McGraw-Hill, 1967.

[Bishop and Bridges, 1985] E. Bishop and D. Bridges. *Constructive Analysis*, Springer-Verlag, 1985.

[Blanck, 1997] J. Blanck. Domain representability of metric spaces, *Annals of Pure and Applied Logic* 83:225–247, 1997.

[Blum and Smale, 1993] L. Blum and S. Smale. The Gödel incompleteness theorem and decidability over a ring. In *From Topology to Computation: Proceedings of the Smalefest*, M. W. Hirsch, J. E. Marsden and M. Schub eds, pp. 321–339, Springer-Verlag, 1993.

[Blum *et al.*, 1989] 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.

[Blum *et al.*, 1996] L. Blum, F. Cucker, M. Shub and S. Smale. Complexity and real computation: A manifesto, *International Journal of Bifurcation and Chaos* 6(1):3–26, 1996.

[Böhm and Jacopini, 1966] C. Böhm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules, *Communications of the Association for Computing Machinery*, 9:366–371, 1966.

[Brainerd and Landweber, 1974] W. S. Brainerd and L. H. Landweber. *Theory of Computation*, John Wiley & Sons, 1974.

[Brattka, 1996] V. Brattka. Recursive characterisation of computable real-valued functions and relations, *Theoretical Computer Science* 162:45–77, 1996.

[Brattka, 1997] V. Brattka. Order-free recursion on the real numbers, *Mathematical Logic Quarterly* 43:216–234, 1997.

[Bröcker and Lander, 1975] T. Bröcker and L. C. Lander. *Differentiable Germs and Catastrophes*, London Mathematical Society, Lecture Notes Series 17, Cambridge University Press, 1975.

[Brown *et al.*, *1972] S. Brown, D. Gries and T. Szymanski. Program schemes with pushdown stores, *SIAM Journal of Computing* 1:242–268, 1972.

[Broy *et al.*, 1993] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner and R. Weber. The design of distributed sytems: An introduction to FOCUS, Technical Report TUM-19202-2, Institut für Informatik, Technical University of Munich, January 1993.

[Burris and Sankappavanar, 1981] S. Burris and H. P. Sankappavanar. *A Course in Universal Algebra*, Springer-Verlag, 1981.

[Byerly, *1993] R. E. Byerly. Ordered subrings of the reals in which output sets are recusively enumerable, *Proceedings of the American Mathematical Society* 118:597–601, 1993.

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

[Chandra, 1973] A. K. Chandra. On the properties and applications of program schemas, PhD. thesis, Department of Computer Science, Stanford University, 1973.

[Chandra, 1974] A. K. Chandra. The power of parallelism and nondeterminism in programming. In *Information Processing '74*, North-Holland, pp. 461–465, 1974.

[Chandra and Manna, 1972] A. K. Chandra and Z. Manna. Program schemes with equality. In *Proceedings of the 4th Annual ACM Symposium on the Theory of Computing*, Denver, Col. pp. 52–64, Association for Computing Machinery, 1972.

[Chandra and Manna, 1975] A. K. Chandra and Z. Manna. On the power of programming features, *Journal of Computer Languages* 1:219–232, 1975.

[Chaplin, 1970] N. Chaplin. Flowcharting with the ANSI Standard, *Computer Surveys* 2:119–30, 1970.

[Church, 1936] A. Church. An unsolvable problem of elementary number theory, *American Journal of Mathematics* 58: pp. 345–363, 1936.

[Clark and Cowell, 1976] K. L. Clark and D. F. Cowell. *Programs, Machines and Computation: An Introduction to the Theory of Computing*, McGraw-Hill, 1976.

[Clarke, *1979] E. M. Clarke, Jr. Programming language constructs for which it is impossible to obtain good Hoare-like axioms, *Journal of the ACM* 26:126–147, 1979.

[Clarke, *1984] E. M. Clarke, Jr. The characterization problem for Hoare logic, *Philisophical Transactions of the Royal Society of London A*:312;423–440, 1984.

[Clarke *et al.*, *1983] E. M. Clarke, Jr., S. German and J. Y. Halpern. Effective axiomatization of Hoare logics, *Journal of the Association of Computing Machinery*, 30:612–636, 1983.

[Constable and Gries, 1972] R. L. Constable and D. Gries. On classes of program schemata, *SIAM Journal of Computing* 1:66–118, 1972.

[Cook, *1978] S. A. Cook. Soundness and completeness of an axiom system for program verification, *SIAM Journal of Computing* 7:70–90, 1978; corrigendum [1981], *ibid.* 10, 612.

[Cucker *et al.*, *1994] F. Cucker, M. Shub and S. Smale. Separation of complexity classes in Koiran's weak model, *Theoretical Computer Science* 13:3–14, 1994.

[Cutland, 1980] N. J. Cutland. *Computability: An Introduction to Recursive Function Theory*, Cambridge University Press, 1980.

[Dańko, *1983] W. Dańko. Algebraic properties of finitely generated structures. In *Proceedings in Logics of Programs and their Applications, Poznań 1980*, A. Salwicki ed., Lecture Notes in Computer Science 148, pp. 118–131, Springer-Verlag, 1983.

[Davis, 1958] M. Davis *Computability and Unsolvability.* McGraw-Hill, 1958. (Reprinted [1983], Dover.)

[Davis and Weyuker, 1983] M. D. Davis and E. P. Weyuker. *Computability, Complexity, and Languages* Academic Press, 1983.

[Davis *et al.*, 1994] M. D. Davis, R. Sigal and E. P. Weyuker. *Computability, Complexity, and Languages* (2nd edition), Academic Press, 1994.

[Devaney, 1989] R. Devaney. *An Introduction to Chaotic Dynamical Systems*, Addison Wesley, 1989.

[Dijkstra, 1976] E. W. Dijkstra. *A Discipline of Programming*, Prentice Hall, 1976.

[Dugundji, 1966] J. Dugundji. *Topology*, Allyn and Bacon, 1966.

[Edalat, 1995a] A. Edalat. Domain theory and integration, *Theoretical Computer Science* 151:163–193, 1995.

[Edalat, 1995b] A. Edalat. Dynamical systems, measures, and fractals via domain theory, *Information & Computation* 120:32–48, 1995.

[Ehrig and Mahr, 1985] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, EATCS Monographs 6:, Springer-Verlag, 1985.

[Eilenberg and Elgot, *1968] S. Eilenberg and C. C. Elgot. Iteration and recursion, IBM Research Report RC-2148, 1968.

[Eilenberg and Elgot, *1970] S. Eilenberg and C. C. Elgot. *Recursiveness*, Academic Press, 1970.

[Eilenberg and Wright, *1967] S. Eilenberg and J. B. Wright. Automata in general algebras, *Information & Control* 11:452–470, 1976.

[Elgot, *1966a] C. C. Elgot. Abstract algorithms and diagram closure. In *Programming Languages*, Genuys ed., pp. 1–42, Academic Press, 1966, also: Preprint, IBM Laboratory Vienna (1966) (presented at NATO Summer School on Programming Languages, Villard-de-Lans (Isère), France, September 1966).

[Elgot, *1966b] C. C. Elgot. A notion of interpretability of algorithms in algorithms, Preprint, IBM Laboratory Vienna (presented at NATO Summer School on Programming Languages, Villard-de-Lans (Isère), France, September 1966).

[Elgot, *1966c] C. C. Elgot. Machine species and their computation languages. In *Formal Language Description Languages for Computer Programming, Proceedings of the IFIP Working Conference on Formal Language Description Languages, Amsterdam*, T. B. Steel, Jr., ed., North-Holland, pp. 160–178, 1966.

[Elgot, *1970] C. C. Elgot. The common algebraic structure of exit-automata and machines, IBM Research Report RC-2744, 1970.

[Elgot, *1971] C. C. Elgot. Algebraic theories and program schemes. In *Symposium on Algorithmic Semantics of Languages*, E. Engeler ed., Lecture Notes in Mathematics 188, Springer-Verlag, 1971.

[Elgot and Robinson, *1964] C. C. Elgot and A. Robinson. Random-access, stored-program machines. An approach to programming

Langueages, *Journal of the Association for Computing Machinery*, 11:365–399, 1964.

[Elgot *et al.*, *1966] C. C. Elgot, A. Robinson and J. D. Rutledge. Multiple control computer models, IBM Research Report RC-1622, 1966.

[Enderton, 1977] H. B. Enderton. Elements of recursion theory. In *Handbook of Mathematical Logic*, J. Barwise, ed., pp. 527-566, North-Holland, 1977.

[Engeler, 1967] E. Engeler. Algebraic properties of structures, *Mathematical Systems Theory* 1:183–195, 1967.

[Engeler, 1968a] E. Engeler. *Formal Languages: Automata and Structures*, Markham Publishing Co., 1968.

[Engeler, 1968b] E. Engeler. Remarks on the theory of geometrical constructions. In *The Syntax and Semantics of Infinitary Languages*, Lecture Notes in Mathematics 72, pp. 64–76, Springer-Verlag, 1968.

[Engeler, 1971] E. Engeler. Structure and meaning of elementary programs. In *Symposium on Semantics of Algorithmic Languages*, Lecture Notes in Mathematics 188, pp. 89–101, Springer-Verlag, 1971.

[Engeler, 1975a] E. Engeler. On the solvability of algorithmic problems, *in Logic Colloquium '73*, H. E. Rose and J. C. Shepherdson eds, pp. 231–251, North-Holland, 1975.

[Engeler, 1975b] E. Engeler. Algebraic logic. In *Foundations of Computer Science*, Mathematical Centre Tracts No. 63, J. W. de Bakker, ed., Amsterdam, pp. 57–85, 1975.

[Engeler, 1993] E. Engeler. *Algebraic Properties of Structures*, World Scientific, 1993.

[Ershov, 1958] A. P. Ershov. On operator algorithms, *Doklady Akademii Nauk SSSR* 122:967–970, 1958 (in Russian), translated in *Automation Express* 1:20–23, 1959.

[Ershov, 1960] A. P. Ershov. Operator algorithms I, *Problemi Kibernetiki* 3, 1960. (in Russian), translated in *Problems of Cybernetics* 3, 1962.

[Ershov, 1962] A. P. Ershov. Operator algorithms II, *Problemi Kibernetiki* 8:211–233 (in Russian), 1962.

[Ershov, *1981] A. P. Ershov. Abstract computability on algebraic structures. In *Algorithms in Modern Mathematics and Computer Science*, A. P. Ershov and D. E. Knuth, eds, Lecture Notes in Computer Science 122, Springer-Verlag, 1981.

[Ershov and Shura-Bura, 1980] A. P. Ershov and M. R. Shura-Bura. The early development of programming in the USSR. In *A History of Computing in the Twentieth Century*, E. N. Metropolis, J. Howlett and G.-C. Rota eds, pp. 137–196, Academic Press, 1980.

[Feferman, 1992a] S. Feferman. A new approach to abstract data types, I: Informal development, *Mathematical Structures in Computer Science* 2: pp. 193–229, 1992.

[Feferman, 1992b] S. Feferman. A new approach to abstract data types, II: Computability on ADTs as ordinary computation. In *Computer Science Logic*, E. Börger, g. Jäger, H. Kleine Büning and M. M. Richter eds, Lecture Notes in Computer Science 626, pp. 79–95, Springer-Verlag, 1992.

[Feferman, 1996] S. Feferman. Computation on abstract data types: The extensional approach, with an application to streams, *Annals of Pure and Applied Logic* 81:75–113, 1996.

[Fenstad, 1975] J. E. Fenstad. Computation theories: an axiomatic approach to recursion on general structures. In *Logic Conference, Kiel 1974*, G. Müller, A. Oberschelp and K. Potthoff eds, Lecture Notesin Mathematics 499, pp. 143–168, Springer-Verlag, 1975.

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

[Fitting, 1981] M. Fitting. *Fundamentals of Generalized Recursion Theory*, North-Holland, 1981.

[Floyd, *1967] R. W. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, J. T. Schärtz ed, pp. 19–32, American Mathematical Society, 1967.

[Friedman, 1971a] H. Friedman. Algebraic procedures, generalized Turing algorithms, and elementary recursion theory. In *Logic Colloquium '69*, R. O. Gandy and C. M. E. Yates eds, pp. 361–389, North-Holland, 1971.

[Friedman, *1971b] H. Friedman. Axiomatic recursive function theory. In *Logic Colloquium '69*, R. O. Gandy and C. M. E. Yates eds, pp. 113–137, North-Holland, 1971.

[Friedman and Mansfield, *1992] H. Friedman and R. Mansfield. Algebraic procedures, *Transactions of the American Mathematical Society* 332:297–312, 1992.

[Gabrovsky, *1976] P. Gabrovsky. Models for an axiomatic theory of computability, PhD thesis, Syracuse University, 1976.

[Gandy, *1980] R. O. Gandy. Church's thesis and principles for mechanisms. In *The Kleene Symposium*, J. Barwise, H. J. Keisler and K. Kunen eds, pp. 123–148, North-Holland, 1980.

[Garland and Luckham, *1973] S. J. Garland and D. C. Luckham. Program schemes, recursion schemes, and formal languages, *Journal of Computer and Systems Science* 7:119–160, 1973.

[Goguen *et al.*, *1978] J. A. Goguen, J. W. Thatcher and E. G. Wagner. An initial approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology, 4: Data Structuring*, R.T. Yeh ed, pp. 80–149, Prentice Hall, 1978.

[Goldstine and von Neumann, 1947] H. H. Goldstine and J. von Neumann. *Planning and Coding Problems for an Electronic Computing Instrument*, Institute for Advanced Study, Princeton, 1947. Reprinted in A. H. Traub ed, *John von Neumann's Collected Works, 5*, pp. 80–235, Pergamon, 1963.

[Goodstein, 1964] R. L. Goodstein. *Recursive Number Theory*, North-Holland, 1964.

[Gordon, 1970] C. E. Gordon. Comparisons between some generalizations of recursion theory, *Compositio Mathematica* 22:333–346, 1970.

[Gordon, *1971] C. E. Gordon. Finitistically computable functions and relations on an abstract structure (abstract), *Journal of Symbolic Logic* 36:704, 1971.

[Gorn, *1961] S. Gorn. Specification languages for mechanical languages and their processors, *Communications of the Association for Computing Machinery* 4:532–542, 1961.

[Grabowski, *1981] M. Grabowski. Full weak second-order logic versus algorithmic logic, Colloquia In *Proceedings in Mathematical Logic in Computer Science*, Mathematica Societatis János Bolyai 26, pp. 471–483, North-Holland, 1981.

[Grabowski and Kreczmar, *1978] M. Grabowski and A. Kreczmar. Dynamic theories of real and complex numbers. In *Mathematical Foundations of Computer Science '78*, J. Winkowski ed., Lecture Notes in Computer Science 64, pp. 239–249, Springer-Verlag, 1978.

[Greibach, 1975] S. A. Greibach. *Theory of Program Structures: Schemes, Semantics, Verification*, Lecture Notes in Computer Science 36, Springer-Verlag, 1975.

[Grzegorczyk, 1955] A. Grzegorczyk. Computable functions, *Fundamenta Mathematicae* 42:168–202, 1955.

[Grzegorczyk, 1957] A. Grzegorczyk. On the defintions of computable real continuous functions, *Fundamenta Mathematicae* 44:61–71, 1957.

[Guttag, *1977] J. V. Guttag. Abstract data types and the development of data structures, *Communications of the Association for Computing Machinery* 20:396–404, 1977.

[Harel, *1980] D. Harel. On folk theorems, *Communications of the Association for Computing Machinery* 23, 1980.

[Harel, *1984] D. Harel. Dynamic logic. In *Handbook of Philisophical Logic, II*, D. Gabbay and F. Guenthner eds, pp. 497–604, Reidel, 1984.

[Harel *et al.*, *1977] D. Harel, A. R. Meyer and V. R. Pratt. Computability and completeness in logics of programs. In *Proceedings of the 9th Annual ACM Symposium on the Theory of Computing*, Boulder, Colorado, 1977.

[Harman and Tucker, 1993] N. A. Harman and J. V. Tucker. Algebraic methods and the correctness of microprocessors. In *Correct Hardware Design and Verification Methods*, G. J. Milne and L. Pierre eds, Lecture Notes in Computer Science 683, pp. 92–108, Springer-Verlag, 1993.

[Harnik, *1975] V. Harnik. Effective proper procedures and universal classes of program schemata, *Journal of Computer & Systems Science* 10:44-61, 1975.

[Heaton and Wainer, 1996] A. Heaton and S. Wainer. Subrecursion theories. In *Computability, Enumerability, Unsolvability*, S. B. Cooper, S.

S. Wainer and T. A. Slaman eds, London Mathematical Society Lecture Note Series, Cambridge University Press, 1996.

[Hemmerling, *1994] A. Hemmerling. On genuine complexity and kinds of nondeterminism, *Journal of Information Processing and Cybernetics* 30:77–96, 1994.

[Hemmerling, *1995] A. Hemmerling. Computability and complexity over structures of finite type, Preprint Nr. 2-1995, Preprint-Reihe Mathematik, Ernst-Moritz-Arndt-Universität, Greifswald, 1995.

[Hemmerling, *1998] A. Hemmerling. Computability of string functions over algebraic structures, *Mathematical Logic Quarterly* 44:1–44, 1998.

[Herman and Isard, 1970] G. T. Herman and S. D. Isard. Computability over arbitrary fields, *Journal of the London Mathematical Society* 2:73–79, 1970.

[Hinman, 1999] P. G. Hinman. Recursion on abstract structures. In *Handbook of Computability Theory*, E. Griffor ed., pp. 315–359. Elsevier, 1999.

[Hoare, *1969] C. A. R. Hoare. An axiomatic basis for computer programming, *Communications of the Association of Computing Machinery* 12:576–583, 1969.

[Hobley *et al.*, *1988] K. M. Hobley, B. C. Thompson and J. V. Tucker. Specification and verification of synchronous concurrent algorithms: a case study of a convoluted algorithm. In *The Fusion of Hardware Design and Verification (Proceedings of IFIP Working Group 10.2 Working Conference)*, G. Milne ed, pp. 347–374, North-Holland, 1988.

[Hocking and Young, 1961] J. G. Hocking and G. S. Young. *Topology*, Addison Wesley, 1961.

[Holden *et al.*, 1992] A. V. Holden, M. Poole, J. V. Tucker and H. Zhang. Coupled map lattices as computational systems. *American Institute of Physics – Chaos*, 2: 367–376, 1992.

[Ianov, 1960] I. Ianov. The logical schemes of algorithms, *Problemi Kibernetiki* 1:75–127, 1958 (in Russian). Translated in *Problems of Cybernetics* 1:82–140, 1960.

[Igerashi, *1968] S. Igerashi. An axiomatic approach to the equivalence problems of algorithms with applications, Report from the Computing Centre of the University of Tokyo 1, 1968.

[Ivanov, *1986] L. L. Ivanov. *Algebraic Recursion Theory*, Ellis Horwood, 1986.

[Jacopini, *1966] G. Jacopini. Macchina Universale di von Neumann ad unico comando incondizionato, *Calcolo* 3:23–29, 1966.

[Jervis, 1988] C. A. Jervis. On the specification, implementation and verification of data types, PhD thesis, Department of Computer Studies, University of Leeds, 1988.

[Kaluzhnin, 1961] A. Kaluzhnin. Algorithmization of mathematical problems, *Problemi Kibernetiki* 2, 1959 (in Russian). Translated *in Problems of Cybernetics* 2, 1961.

[Kaphengst, *1959] M. Kaphengst. Eine abstrakte programmgesteuerte Rechenmaschine, *Zeitschrift für mathematische Logik und Grundlagen der Mathematik* 5: 366–379, 1959.

[Kaplan, *1969] D. M. Kaplan. Regular expressions and the equivalence of programs, *Journal of Computer & System Sciences*, 3:361–386, 1969.

[Karp and Miller, *1969] R. M. Karp and R. E. Miller. Parallel program schemata, *Journal of Computer & Systems Science* 3:147–195, 1969.

[Kelley, 1955] J. L. Kelley. *General Topology*, Van Nostrand, 1955. Reprinted Springer-Verlag, 1975.

[Kfoury, *1972] A. J. Kfoury. Comparing algebraic structures up to algorithmic equivalence. In *International Colloquium on Automata, Languages and Programming, Paris, 1972*, M. Nivat, ed., pp. 235–264, North-Holland, 1972.

[Kfoury, *1983] A. J. Kfoury. Definability by programs in first-order structures, *Theoretical Computer Science* 25:1–66, 1983.

[Kfoury, *1985] A. J. Kfoury. Definability by deterministic and nondeterministic programs (with applications to first-order dynamic logic), *Information & Control* 65:98–121, 1985.

[Kfoury and Park, *1975] A. J. Kfoury and D. M. Park. On the termination of program schemas, *Information & Control* 29:243–251, 1975.

[Kfoury and Urzyczyn, *1985] A. J. Kfoury and P. Urzyczyn. Necessary and sufficient conditions for the universality of programming formalisms, *Acta Informatica* 22:347–377, 1985.

[Kfoury *et al.*, 1982] A. J. Kfoury, R. N. Moll and M. A. Arbib. *A Programming Approach to Computability*, Springer-Verlag, 1982.

[Kleene, 1936] S. C. Kleene. General recursive functions of natural numbers, *Mathematische Annalen* 112:727–742, 1936.

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

[Kleene, 1959] S. C. Kleene. Recursive functionals and quantifiers of finite types I, *Transactions of the American Mathematical Society* 91:1–52, 1959.

[Kleene, 1963] S. C. Kleene. Recursive functionals and quantifiers of finite types II, *Transactions of the American Mathematical Society* 108:106–142, 1963.

[Klop, 1992] J. W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science*, Vol. 1, S. Abramsky, D. Gabbay and T. Maibaum eds, pp. 2–116, Clarendon Press, 1992.

[Knuth, *1974] D. E. Knuth. Structured programming with 'go to' statements, *Computing Surveys* 6:261–301, 1974.

[Knuth and Prado, *1980] D. Knuth and L. T. Prado. The early development of programming languages. In *A History of Computing in the Twentieth Century*, N. Metropolis, J. Howlett and G.-C. Rota eds, pp. 197–273, Academic Press, 1980.

[Ko, 1991] K.-I. Ko. *Complexity Theory of Real Functions*, Birkhäuser, 1991.

[Kolmogorov, *1953] A. N. Kolmogorov. O ponyatii algoritma, *Uspekhi Matematicheskikh Nauk* 814:175–176, 1953.

[Kreczmar, *1977] A. Kreczmar. Programmability in fields, *Fundamenta Informaticae* 1:195–230, 1977.

[Kreisel, 1971] G. Kreisel. Some reasons for generalizing recursion theory. In *Logic Colloquium '69*, R. O. Gandy and C. M. E. Yates eds, pp. 139–198, North-Holland, 1971.

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

[Kreitz and Weihrauch, 1985] C. Kreitz and K. Weihrauch. Theory of representations, *Theoretical Computer Science* 38:35–53, 1985.

[Lacombe, 1955] D. Lacombe. Extension de la notion de fonction récursive aux fonctions d'une ou plusieurs variables réelles, I, II, III, *Comptes Rendus de ;'Académie des Sciences Paris* 240:2470–2480, 241:13–14, 151–153, 1955.

[Lacombe, *1971] D. Lacombe. Recursion theoretic structure for relational systems. In *Logic Colloquium '69*, R. O. Gandy and C. M. E. Yates eds, pp. 3–18, North-Holland, 1971.

[Lambert, 1968] W. M. Lambert, Jr. A notion of effectiveness in arbitrary structures, *Journal of Symbolic Logic*, 33:577–602, 1968.

[Lauer, 1967] P. E. Lauer. The formal explicates of the notion of algorithm, Technical Report TR 25.072, IBM Laboratory, Vienna, 1971.

[Lauer, 1968] P. E. Lauer. An introduction to H. Thiele's notions of algorithm, algorithmic process and graph-schemata calculus, Technical Report TR 25.079, IBM Laboratory, Vienna, 1968.

[Levien, *1962] R. E. Levien. Set-theoretic formalizations of computational algorithms, computable functions and general purpose computers. In *Proceedings of the Symposium on the Mathematical Theory of Automation, New York*, American Mathematical Society, pp. 101–123, 1962.

[Lucas *et al.*, 1968] P. Lucas, P. E. Lauer and H. Stigleitner. Method and notation for the formal definition of programming languages, Technical Report TR 25.087, IBM Laboratory, Vienna, 1968.

[Luckham and Park, 1964] D. Luckham and D. M. Park. The undecidability of the equivalence problem for program schemata, Report 1141, Bolt, Beranek and Newman Inc., 1964.

[Luckham *et al.*, 1970] D. Luckham, D. M. Park and M. S. Paterson. On formalized computer programs, *Journal of Computer & Systems Science* 4:220–249, 1970.

[Machtey and Young, 1978] M. Machtey and P. Young. *An Introduction to the General Theory of Algorithms*, North-Holland, 1978.

[Mahr and Makowsky, *1984] B. Mahr and J. A. Makowsky. Characterizing specification languages which admit initial semantics, *Theoretical*

Computer Science 31:49–59, 1984.

[Makowsky and Sain, *1989] J. A. Makowsky and I. Sain. Weak second order characterization of various program verification systems, *Theoretical Computer Science* 66:239–321, 1989.

[Mal'cev, 1973] A. I. Mal'cev. *Algebraic Systems*, Grundlehren der mathematischen Wissenschaften 192, Springer-Verlag, 1973.

[Manna, 1974] Z. Manna. *Mathematical Theory of Computation*, McGraw-Hill, 1974.

[Martin and Tucker, *1988] A. R. Martin and J. V. Tucker. The concurrent assignment representation of synchronous systems, *Parallel Computing* 9:227–256, 1988.

[McCarthy, *1960] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I, *Communications of the Association for Computing Machinery* 3:184–195, 1960.

[McCarthy, *1962] J. McCarthy. Towards a mathematical science of computation, *in Proceedings of the International Congress of Information Processing* pp. 21–34, 1962.

[McCarthy, 1963] J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hershberg, eds., North Holland, Amsterdam, pp. 33–70, 1963.

[McEvoy and Tucker, 1990] K. McEvoy and J. V. Tucker, eds. *Theoretical Foundations of VLSI Design*, Cambridge University Press, 1990.

[McKenzie *et al.*, 1987] R. N. McKenzie, G. F. McNulty and W. F. Taylor. *Algebras, Lattices, Varieties* 1, Wadsworth and Brookes Cole, 1987.

[McNaughton, 1982] R. McNaughton. *Elementary Computability, Formal Languages and Automata*, Prentice Hall, 1982.

[Meer, *1993] K. Meer. *Komplexitätsbetrachtungen für reelle Maschinenmodelle*, Shaker Verlag, 1993.

[Meer and Michaux, *1996] K. Meer and C. Michaux. A survey on real structural complexity theory, *Bulletin of the Belgian Mathematical Socitety* 3:113–148, 1996.

[Megiddo, *1993] N. Megiddo. A general NP-completeness theorem. In *From Topology to Computation: Proceedings of the Smalefest*, M. W. Hirsch, J. E. Marsden and M. Schub eds, pp. 432–442, Springer-Verlag, 1993.

[Meinke and Tucker, 1992] K. Meinke and J.V. Tucker. Universal algebra. In *Handbook of Logic in Computer Science*, Vol. 1, S. Abramsky, D. Gabbay and T. Maibaum eds, pp. 189–411, Clarendon Press, 1992.

[Melzak, *1961] Z. A. Melzak. An informal arithmetical approach to computability and computation, *Canadian Mathematical Bulletin*, 4:279–293, 1961.

[Meseguer and Goguen, 1985] J. Meseguer and J. A. Goguen. Initiality, induction and computability. In *Algebraic Methods in Semantics*, M. Nivat and J. Reynolds eds, pp. 459–541, Cambridge University Press, 1985.

[Meyer and Ritchie, 1967] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. In *Proceedings of the 22nd National Conference, Association for Computing Machinery*, pp. 465–469, Thompson Book Company, 1967.

[Meyer and Tiuryn, *1982] A. R. Meyer and J. Tiuryn. A note on equivalences among logics of programs. In *Proceedings in Logics of Programs, Yorktown Heights 1981*, D. Kozen ed, Lecture Notes in Computer Science 92, Springer-Verlag, 1982.

[Michaux, *1989] C. Michaux. Une remarque à propos des machines sur **R**. In *Comptes Rendus de l'Académie des Sciences Paris, Série I* 309: pp. 435–437, 1989.

[Michaux, *1990] C. Michaux. Machines sur les réels et problèmes NP-complets. In *Séminaire de structures algébriques ordonnées*, Prépublication de l'equipe de logique mathématique de Paris 7, 1990.

[Michaux, *1991] C. Michaux. Ordered rings over which output sets are recursively enumerable, *Proceedings of the American Mathematical Society*, 112:569–575, 1991.

[Milner, *1969] Equivalences on program schemes. *Journal of Computer & System Sciences* 4: 205–219, 1969.

[Mints, 1973] G. Mints. Quantifier-free and one-quantifier systems, *Jounal of Soviet Mathematics* 1:71–84, 1973.

[Moldestad and Tucker, 1981] J. Moldestad, and J. V. Tucker. On the classification of computable functions in an abstract setting. Unpublished manuscript, 1981.

[Moldestad *et al.*, 1980a] J. Moldestad, V. Stoltenberg-Hansen and J. V. Tucker. Finite algorithmic procedures and inductive definability, *Mathematica Scandinavica* 46:62–76, 1980.

[Moldestad *et al.*, 1980b] J. Moldestad, V. Stoltenberg-Hansen and J. V. Tucker. Finite algorithmic procedures and inductive definability, *Mathematica Scandinavica* 46:77–94, 1980.

[Montague, 1968] R. Montague. *Recursion theory as a branch of model theory, Logic*, Methodology & Philosophy of Science III, B. van Rootselaar and J. F. Staal eds, North-Holland, pp. 63–86, 1968.

[Moschovakis, 1964] Y. N. Moschovakis. Recursive metric spaces, *Fundamenta Mathematicae* 55:215–238, 1964.

[Moschovakis, 1969a] Y. N. Moschovakis. Abstract first-order computability I, *Transactions of the American Mathematical Society* 138:427–464, 1969.

[Moschovakis, *1969b] Y. N. Moschovakis. Abstract first-order computability II, *Transactions of the American Mathematical Society* 138:465–504, 1969.

[Moschovakis, *1969c] Y. N. Moschovakis. Abstract computability and invariant definability, *Journal of Symbolic Logic*, 34:605–633, 1969.

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

[Moschovakis, *1974] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*, North-Holland, 1974.

[Moschovakis, 1984] Y. N. Moschovakis. Abstract recursion as a foundation for the theory of recursive algorithms. In *Computation and Proof Theory*, Lecture Notes in Mathematics 1104, pp. 289–364, Springer-Verlag, 1984.

[Moschovakis, 1989] Y. N. Moschovakis. The formal language of recursion, *Journal of Symbolic Logic* 54:1216–1252, 1989.

[Müller and Tucker, 1998] B. Müller and J. V. Tucker, eds. *Prospects for Hardware Foundations.* Lecture Notes in Computer Science 1546, Springer-Verlag, 1998.

[Normann, 1978] D. Normann. Set-recursion. In *Generalized Recursion Theory II: Proceedings of the 1977 Oslo Symposium*, J. E. Fenstad, R. O. Gandy and G. E. Sacks eds, pp. 303–320, North-Holland, 1978.

[Normann, 1982] D. Normann. External and internal algorithms on the continuous functionals. In *Patas Logic Symposium*, G. Metakides, ed, Studies in Logic, vol. 109, pp. 137–144. North-Holland, 1982.

[Parsons, 1971] C. Parsons. On a number theoretic choice scheme II (Abstract), *Journal of Symbolic Logic* 36:587, 1971.

[Parsons, 1972] C. Parsons. On n-quantifier induction, *Journal of Symbolic Logic* 37:466–482, 1972.

[Paterson, *1967] M. S. Paterson. Equivalence problems in a model of computation, PhD thesis, Cambridge University, 1967.

[Paterson, *1968] M. S. Paterson. Program schemata, *Machine Intelligence* 3:19–31, 1968.

[Paterson and Hewitt, 1970] M. S. Paterson and C. E. Hewitt. Comparative schematology. In *Record of Project MAC Conference on Concurrent Systems and Parallel Computation* pp. 119–128, ACM; also MIT, AI Technical Memo 201, 1979.

[Péter, 1958] R. Péter. Graphschemata und rekursive Funktionen, *Dialectica* 2:373–393, 1958.

[Péter, 1967] R. Péter. *Recursive Functions*, Academic Press, 1967.

[Phillips, 1992] I. C. C. Phillips. Recursion theory. In *Handbook of Logic in Computer Science*, Vol. 1, S. Abramsky, D. Gabbay and T. Maibaum eds, pp. 79–187, Clarendon Press, 1992.

[Plaisted, *1972] D. Plaisted. Program schemas with counters. In *Proceedings of the 4th Annual ACM Simposium on the Theory of Computing, Denver, Col.* pp. 44-51, Association for Computing Machinery, 1972.

[Platek, 1966] R. A. Platek. Foundations of recursion theory, PhD thesis, Department of Mathematics, Stanford University, 1966.

[Pour-El and Caldwell, 1975] M. B. Pour-El and J. C. Caldwell. On a simple definition of computable function of a real variable with applications to functions of a complex variable, *Zeitschrift für mathematische Logik und Grundlagen der Mathematik* 21:1–19, 1975.

[Pour-El and Richards, 1989] M. B. Pour-El and J. I. Richards. *Computability in Analysis and Physics*, Springer-Verlag, 1989.

[Rice, 1954] H. G. Rice. Recursive real numbers, *Proceedings of the American Mathematical Society* 5:784–791, 1954.

[Rogers, 1967] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, 1967.

[Rutledge, *1964] J. D. Rutledge. On Ianov's program schemata, *Journal of the Association for Computing Machinery* 11:1–9, 1964.

[Rutledge, *1970] J. D. Rutledge. Parallel processes—schemata and transformations, IBM Research Report RC-2912 1970. Also *in Architecture and Design of Digital Computers*, Boulaye ed., pp. 91–129, Dunod, 1971.

[Rutledge, *1973] J. D. Rutledge. Program Schemes as Automata 1, *Journal of Computer & System Sciences* 7:543–578, 1973.

[Saint John, *1994] R. Saint John. *Output sets, halting sets and an arithmetical hierarchy for ordered substrings of the real numbers under Blum/Shub/Smale computation*, Technical Report TR-94-035, ICSI, Berkeley, CA, 1994.

[Saint John, *1995] R. Saint John. *Theory of computation for the real numbers and subrings of the real numbers following Blum/Shub/Smale*, Dissertation, University of California at Berkeley, 1995.

[Sanchis, *1988] L. E. Sanchis. *Reflexive Structures*, Springer-Verlag, 1988.

[Schreiber, *1975] P. Schreiber. *Theorie der geometrischen Konstruktionen*, Deutscher Verlag der Wissenschaften, Berlin, 1975.

[Scott, 1967] D. Scott. Some definitional suggestions for automata theory, *Journal of Computer & System Sciences* 1:187–212, 1967.

[Scott, *1970a] D. S. Scott. The lattice of flow diagrams, Programming Research Group, Oxford, 1970.

[Scott, *1970b] D. S. Scott. Outline of a mathematical theory of computation. In *Proceedings of the 4th Annual Princeton Conference on Information Sciences & Systems, Princeton University*, pp. 169–176, 1970. Also Technical Monograph PRG-2, Porgramming Research Group, Oxford University, 1970.

[Scott and Strachey, *1971] D. S. Scott and C. Strachey. Towards a mathematical semantics for computer languages, *in Proceedings of the Symposium on Computers & Automata*, J. Fox ed., Polytechnic Institute of Brooklyn. Also in Technical Monograph 6, Programming Research Group, Oxford, 1971.

[Shafarevich, 1977] I. R. Shafarevich. *Basic Algebraic Geometry*, Springer-Verlag, 1977.

[Shepherdson, 1973] J. C. Shepherdson. Computations over abstract structures: serial and parallel procedures and Friedman's effective definitional schemes, *in Logic Colloquium '73*, H. E. Rose and J. C. Shepherdson eds, pp. 445–513, North-Holland, 1973.

[Shepherdson, 1976] J. C. Shepherdson. On the definition of computable function of a real variable, *Zeitschrift für mathematische Logik und Grundlagen der Mathematik* 22:391–402, 1976.

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

[Shepherdson, *1994] J. C. Shepherdson. Mechanisms for computing over arbitrary structures. In *The Universal Turing Machine, A Half-Century Survey*, R. Herken ed, pp. 581–601, Oxford University Press, 1994.

[Shepherdson and Sturgis, 1963] J. C. Shepherdson and H. E. Sturgis. Computability of recursive functions, *Journal of the Association for Computing Machinery* 10:217–255, 1963.

[Simmons, 1963] G. F. Simmons. *Introduction to Topology and Modern Analysis*, McGraw-Hill, 1963.

[Smyth, 1992] M. B. Smyth. Topology. In *Handbook of Logic in Computer Science*, Vol. 1, S. Abramsky, D. Gabbay and T. Maibaum eds, pp. 641–761, Clarendon Press, 1992.

[Soskov, *1987] I. N. Soskov. Prime computability on partial structures. In *Mathematical Logic and its Applications*, D. G. Skordev ed, pp. 341–350, Plenum Press, 1987.

[Soskov, *1989a] I. N. Soskov. Definability via enumerations, *Journal of Symbolic Logic*, 54:428–440, 1989.

[Soskov, *1989b] I. N. Soskov. An external characterization of the prime computability, *Annuaire de l'Université Sofia* 83:89–110, 1089.

[Soskov, *1990] I. N. Soskov. Computability by means of effectively definable schemes and definability via enumerations, *Archive for Mathematical Logic* 29:187–200, 1990.

[Soskova, *1994] A. A. Soskova An external approach to abstract data types I, *Annuaire de l'Université Sofia* 87, 1994.

[Soskova and Soskov, *1990] A. A. Soskova and I. N. Soskov. Effective enumerations of abstract structures. In *Heyting '88*, P. Petkov ed, pp. 361–372, Plenum Press, 1990.

[Sperschneider and Antoniou, 1991] V. Sperschneider and G. Antoniou. *Logic: A Foundation for Computer Science*, Addison Wesley, 1991.

[Stephens, 1997] R. Stephens. A survey of stream processing, *Acta Informatica* 34:491–541, 1997.

[Stephens and Thompson, 1996] R. Stephens and B. C. Thompson. Cartesian stream transformer composition, *Fundamenta Informaticae* 25:123–174, 1996.

[Stephenson, 1996] K. Stephenson. An algebraic approach to syntax, semantics and computation, PhD thesis, Department of Computer Science, University of Wales, Swansea, 1996.

[Stewart, 1999] K. Stewart. Abstract and concrete models of computation over metric algebras. PhD thesis, Computer Science Department, University of Wales, Swansea, 1999.

[Stoltenberg-Hansen, 1979] V. Stoltenberg-Hansen. Finite injury arguments in infinite computation theories, *Annals of Mathematical Logic* 16:57–80, 1979.

[Stoltenberg-Hansen and Tucker, 1985] V. Stoltenberg-Hansen and J. V. Tucker. Complete local rings as domains, Technical Report 1.85, Centre for Theoretical Computer Science, University of Leeds, 1985.

[Stoltenberg-Hansen and Tucker, 1988] V. Stoltenberg-Hansen and J. V. Tucker. Complete local rings as domains, *Journal of Symbolic Logic* 53:603–624, 1988.

[Stoltenberg-Hansen and Tucker, 1991] V. Stoltenberg-Hansen and J. V. Tucker. Algebraic and fixed point equations over inverse limits of algebras, *Theoretical Computer Science* 87:1–24, 1991.

[Stoltenberg-Hansen and Tucker, 1993] V. Stoltenberg-Hansen and J. V. Tucker. Infinite systems of equations over inverse limits and infinite synchronous concurrent algorithms. In *Semantics: Foundations and Applications* J. W. de Bakker, W.-P. de Roever and G. Rozenberg, eds. Lecture Notes in Computer Science 666, pp. 531–562, Springer-Verlag, 1993.

[Stoltenberg-Hansen and Tucker, 1995] V. Stoltenberg-Hansen and J. V. Tucker. Effective algebras. In *Handbook of Logic in Computer Science*, Vol. 4, S. Abramsky, D. Gabbay and T. Maibaum eds, pp. 357–526, Oxford University Press, 1995.

[Stoltenberg-Hansen and Tucker, 1999a] V. Stoltenberg-Hansen and J. V. Tucker. Computable rings and fields. In *Handbook of Computability Theory*, E. Griffor ed, North-Holland, 1999.

[Stoltenberg-Hansen and Tucker, 1999b] V. Stoltenberg-Hansen and J. V. Tucker. Concrete models of computation for topological algebras. *Theoretical Computer Science*, 219:347–378, 1999.

[Stoltenberg-Hansen *et al.*, 1994] V. Stoltenberg-Hansen, I. Lindström and E. Griffor. *Mathematical Theory of Domains*, Cambridge University Press, 1994.

[Strong, 1968] H. R. Strong, Jr. Algebraically generalised recursive function theory, *IBM Journal of Research and Development*, 12:465–475, 1968.

[Strong, *1971] H. R. Strong, Jr. Translating recursion equations into flowcharts, *Journal of Computer & Systems Science* 5:254–285, 1971.

[Thiele, 1966] H. Thiele. *Wissenschaftstheoretische Untersuchungen in algorithmischen Sprachen*, VEB Deutcher Verlag der Wissenschaften, 1966.

[Thompson, 1987] B. C. Thompson. A mathematical theory of synchronous concurrent algorithms, PhD thesis, School of Computer Studies, University of Leeds, 1987.

[Thompson and Tucker, 1991] B. C. Thompson and J. V. Tucker. Algebraic specification of synchronous concurrent algorithms and architectures, Research Report CSR 9-91, University College of Swansea, 1991.

[Tiuryn, *1981a] J. Tiuryn. Unbounded program memory adds to expressive power of first-order dynamic logic. In *Proceedings of the 22nd Annual Symposium on the Foundations of Computer Science, Nashville*, pp. 335–339, IEEE, 1981.

[Tiuryn, 1981b] J. Tiuryn. Logic of effective definitions, *Fundamenta Informaticae* 4:629–660, 1981.

[Tiuryn, *1981c] J. Tiuryn. A survey of the logic of effective definitions. In *Logics of Programs: Workshop, ETH Zürich, May–July 1979*, E. Engeler ed., Lecture Notes in Computer Science 125, pp. 198–245, Springer-Verlag, 1981.

[Troelstra and van Dalen, 1988] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics: An Introduction, Vols. I and II*, North-Holland, 1988.

[Tucker, 1980] J. V. Tucker. Computing in algebraic systems, *Recursion Theory, Its Generalisations and Applications*, F. R. Drake and S. S. Wainer eds., London Mathematical Society Lecture Note Series 45, pp. 215–235, Cambridge University Press, 1980.

[Tucker, *1991] J. V. Tucker. Theory of computation and specification over abstract data types and its applications. In *NATO Advanced Study Institute, International Summer School at Marktoberdorf, 1989, on Logic, Algebra and Computation*, F. L. Bauer ed, pp. 1–39, Springer-Verlag, 1991.

[Tucker and Zucker, 1988] J. V. Tucker and J. I. Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics*, CWI Monographs 6:, North-Holland, 1988.

[Tucker and Zucker, 1989] J. V. Tucker and J. I. Zucker. Horn programs and semicomputable relations on abstract structures, *in 16th International Colloquium on Automata, Languages and Programming, Stresa, Italy, July 1989*, G. Ausiello, M. Dezani-Ciancaglini and S. Ronchi della Rocca eds., Lecture Notes in Computer Science 372:, Springer-Verlag, pp. 745–760, 1989.

[Tucker and Zucker, *1991] J. V. Tucker and J. I. Zucker. Projections of semicomputable relations on abstract data types, *International Journal of Foundations of Computer Science* 2:267–296, 1991.

[Tucker and Zucker, 1992a] J. V. Tucker and J. I. Zucker. Deterministic and nondeterministic computation, and Horn programs, on abstract data types, *Journal of Logic Programming* 13:23–55, 1992.

[Tucker and Zucker, *1992b] J. V. Tucker and J. I. Zucker. Examples of semicomputable sets of real and complex numbers. In *Constructivity in Computer Science: Summer Symposium, San Antonio, Texas, June 1991*, J. P. Myers, Jr. and M. J. O'Donnell eds, Lecture Notes in Computer Science 613, pp. 179–198, Springer-Verlag, 1992.

[Tucker and Zucker, *1992c] J. V. Tucker and J. I. Zucker. Theory of computation over stream algebras, and its applications. In *Mathematical Foundations of Computer Science 1992: 17th International Symposium, Prague*, I.M. Havel and V. Koubek eds, Lecture Notes in Computer Science 629, pp. 62–80, Springer-Verlag,1992.

[Tucker and Zucker, 1993] J. V. Tucker and J. I. Zucker. Provable computable selection functions on abstract structures. In *Proof Theory*, P. Aczel, H. Simmons and S. S. Wainer eds, Cambridge University Press, pp. 277–306, 1993.

[Tucker and Zucker, 1994] J. V. Tucker and J. I. Zucker. Computable functions on stream algebras. In *NATO Advanced Study Institute, International Summer School at Marktoberdorf, 1993, on Proof and Computation*, H. Schwichtenberg ed, pp. 341–382, Springer-Verlag, 1994.

[Tucker and Zucker, 1998] J. V. Tucker and J. I. Zucker. Infinitary algebraic specifications for stream algebras. Report CAS 98-02, Department of Computing and Software, McMaster University, 1998.

[Tucker and Zucker, 1999] J. V. Tucker and J. I. Zucker. Computation by 'While' programs on topological partial algebras. *Theoretical Computer Science*, 219:379–420, 1999.

[Tucker and Zucker, 2000a] J. V. Tucker and J. I. Zucker. Abstract versus concrete comptability over partial metric algebras (in preparation).

[Tucker and Zucker, 2000b] J. V. Tucker and J. I. Zucker. Abstract computability, algebraic specificationa nd initiality. Technical Report No. CAS 2000-01-J2, Department of Computing & Software, McMaster University, Hamilton, Canada.

[Tucker *et al.*, 1990] J. V. Tucker, S. S. Wainer and J. I. Zucker. Provable computable functions on abstract data types. In *17th International Colloquium on Automata, Languages and Programming*, Warwick University, England, July 1990, M. S. Paterson ed, Lecture Notes in Computer Science 443, pp. 660–673, Springer-Verlag, 1990.

[Turing, 1936] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society* 42: pp. 230–265; correction [1937], *ibid.* 43, pp. 544–546. Reprinted [1965], *The Undecidable*, M. Davis, ed., Raven Press, 1936.

[Urzyczyn, *1981a] P. Urzyczyn. Algorithmic triviality of abstract structures, *Fundamenta Informaticae* 4:819–849, 1981.

[Urzyczyn, *1981b] P. Urzyczyn. The unwind property in certain algebras, *Information & Control* 50:91–109, 1981.

[Urzyczyn, *1982] P. Urzyczyn. On the unwinding of flow-charts with stacks, *Fundamenta Informaticae* 4:119–126, 1982.

[Urzyczyn, *1983] P. Urzyczyn. Nontrivial definability by flow-chart programs, *Information & Control*, 58:101–112, 1983.

[Voorhes, 1958] E. A. Voorhes. Algebric formulation of the notion of flow-diagrams, *Communications of the Association for Computing Machinery* 1:4–8, 1958.

[Wagner, *1965] E. A. Wagner. Uniformly reflexive structures: An axiomatic approach to computability. In *Logic, Computability & Automation: Joint PADC–AIAC Symposium, Trinkaus Manor, Oriskany, New York*, 1965.

[Wagner, 1969] E. A. Wagner. Uniformly reflexive structures: On the nature of Gödelizations and relative computability, *Transactions of the American Mathematical Society* 144:1–41, 1969.

[Walker and Strong, *1973] S. Q. Walker and H. R. Strong. Characterizations of flowchartable recursions, *Journal of Computer & Systems Science* 7:404–447, 1973.

[Warner, *1993] S. Warner. *Topological Rings*, Mathematics Studies 178, North-Holland, 1989.

[Wechler, 1992] W. Wechler. *Universal Algebra for Computer Scientists*, EATCS Monographs 25, Springer-Verlag, 1992.

[Weihrauch, 1987] K. Weihrauch. *Computability*, EATCS Monographs 9, Springer-Verlag, 1987.

[Weihrauch and Schreiber, 1981] K. Weihrauch. Embedding metric spaces into complete partial orders. *Theoretical Computer Science* 16:5–24, 1981.

[van Wijngaarden, *1966] A. van Wijngaarden. Numerical analysis as an independent science, *BIT* 6:68–81, 1966.

[Wirsing, 1991] M. Wirsing. Algebraic specification. In *Handbook of Theoretical Computer Science Vol.B: Formal Methods and Semantics*, J. van Leeuwen ed, pp. 675–788, North-Holland, 1991.

[Zucker and Pretorius, 1993] J. I. Zucker and L. Pretorius. Introduction to computability theory, *South African Computer Journal* 9:3–30, 1993.