Characterizations of Semicomputable Sets of Real Numbers

By

Bo Xie, Honored B.Sc, B.Eng

A Thesis

Submitted to the School of Graduate Studies

in partial fulfilment of the requirements for the degree of

Master of Science

Department of Computing and Software

McMaster University

MASTER OF SCIENCE (2004)	McMaster University
(Computing and Software)	Hamilton, Ontario

TITLE:	Characterizations of Semicomputable Sets of Real Numbers
AUTHOR:	Bo Xie, Honored B.Sc, B.Eng(McMaster University, Canada)
SUPERVISOR:	Professor Jeffery I. Zucker

Abstract

We give some characterizations of semicomputability of sets of reals by programs in certain ***While*** programming languages over a topological partial algebra of reals $\mathcal{R}$. We show that such sets are semicomputable if and only if they are any of the following:

(i) unions of effective sequences of disjoint algebraic open intervals;

(ii) unions of effective sequences of rational open intervals;

(iii) unions of effective sequences of algebraic open intervals.

For the equivalence (i), the ***While*** language must be augmented by a strong (Kleene) OR operator, and for equivalences (ii) and (iii) it must be further augmented by a strong existential quantifier over the naturals ($\boldsymbol{While}^{\exists\mathsf{N}}$).

We also show that the class of $\boldsymbol{While}^{\exists\mathsf{N}}$ semicomputable relations on reals is closed under projection. The proof makes essential use of the continuity of the operations of the algebra.

Acknowledgements

First, I am grateful to my supervisor Dr. J.I. Zucker, whose help, stimulating suggestions and encouragement helped me throughout the whole period of research and writing of this thesis. I owe him a great deal of gratitude for guiding my research.

I would also like to thank the other members of committee, Dr. Jacques Carette and Dr. Thomas Maibaum, who reviewed my work and providing me with valuable comments on this thesis.

My thanks also go to Jian Xu, Likang Zhu, Jie Liang and all the fellow students at the Computing and Software Department for their friendship, support and help.

My parents are always ready to help whichever way I need, I cannot thank them enough.

Last but not least, I would like to express my gratitude to my wife Yuan Wang, for her love, support and patience during my years of study and research.

Contents

Chapter 1

Introduction

1.1 Background

Our research in this thesis is based on computations by high level programming languages featuring the ***While*** construct over many-sorted topological partial algebras.

An algebra A is a finite family of sets

$$A_{s_1}, ..., A_{s_n}$$

called *carriers* of sort $s_1, s_2, ..., s_n$, and a finite set of operations (total or partial functions) defined over these sets[1].

An algebra is said to be standard if it contains the sort of booleans and standard boolean operators. It is N-standard if in addition, it contains the sort of naturals and the standard arithmetic operations.

Classical computability theory on naturals has been studied since the 1930's.

[1]We consider constants to be 0-ary functions.

There are many extensions of this theory to abstract structures.

One of these extensions has been the investigation of total (non-topological) algebras of reals [BCSS98]. A detailed discussion of such extensions is given in [TZ00]. We have adapted many of the definitions and proofs from [TZ00] to fit partial topological algebras.

There are two kinds of computational models for algebras: *abstract* and *concrete*. Abstract models are independent of the representations of the data type of the algebras while concrete models are dependent on such representations. The ***While*** language is an example of an abstract model. Examples of concrete models over $\mathbb{R}$ are the classical computable analysis of Pour-El and Richards [PER89], and TTE (Type-2 Theory of Effectivity) of Weihrauch [Wei00]; both these models represent reals as effective Cauchy sequences of rationals, and their equivalence follows from the results in [SHT99].

Some work in bridging the gap between abstract and concrete models is made in [TZ04a, TZ04b].We will discuss this issue again in §6.2.

In the studies of computability theories, we assume the *continuity principle* [TZ99, TZ04a]:

$$Computability \quad \Longrightarrow \quad Continuity$$

(This principle is ignored in [BCSS98].)

The ***While*** programming language is an imperative language with the basic operations of concurrent assignments, sequential composition, conditional and ‘while’ loops. The syntax and semantics of the ***While*** language are strictly defined, using “algebraic operational semantics”.

We will focus on an N-standard topological partial algebra $\mathcal{R}$, which is formed by the "N-standardization" of the ring of reals, by adding the two boolean valued partial operations:

$$\mathsf{eq_p}, \mathsf{less_p} : \mathbb{R}^2 \rightharpoonup \mathbb{B}$$

It follows from the continuity principle that these operations have to be partial. (This is because the real numbers are connected and the booleans are discrete, so the only total continuous functions from real to boolean are constants.)

Abstract models of computations such as the ***While*** language, with partial basic operations on $\mathbb{R}$, suffer from a limitation, namely the inability to implement *interleaving* or *dovetailing*. The problem is that when interleaving two processes, one process may converge and the other diverge locally (because of the partial basic operations). The resulting process will then diverge, whereas we would want it to converge. Thus we cannot even prove that the union of two semicomputable sets is semicomputable! (Concrete models do not have this limitation.)

To correct this deficiency, we establish two enhancements of the ***While*** language and construct two new languages: ***While***$^{\mathsf{OR}}$ and ***While***$^{\exists\mathsf{N}}$.

In the ***While***$^{\mathsf{OR}}$ language, we introduce a strong (Kleene) disjunction operation OR, which converges to true if either component converges to true, even if the other one diverges. By means of this, interleaving of finitely many processes can be simulated at the abstract level.

The ***While***$^{\exists\mathsf{N}}$ language includes a strong 'Exist' construct over the naturals:

$$\mathsf{x^B} := \ \mathsf{Exist\ z} : P(t, \mathsf{z})$$

where $\mathsf{z} : \mathsf{nat}$ and P is a boolean-valued procedure. By means of this, interleaving of *infinitely* many processes can be simulated at the abstract level.

We will study the structure of semicomputable sets in $\mathcal{R}$, where a set is said to be (for example) ***While*** semicomputable if it is the domain of a ***While*** computable function, or the halting set of a ***While*** procedure.

1.2 Objectives

In this thesis, we will prove certain structure theorems for semicomputable sets in $\mathcal{R}$.

For sets of reals:

(1) $\boldsymbol{While}^{\mathsf{OR}}$ semicomputable over $\mathcal{R}$ $\iff$

countable union of effective sequence of *disjoint* algebraic intervals.[2,3]

(2) $\boldsymbol{While}^{\exists\mathsf{N}}$ semicomputable over $\mathcal{R}$ $\iff$

countable union of an effective sequence of algebraic intervals.

(3) $\boldsymbol{While}^{\exists\mathsf{N}}$ semicomputable over $\mathcal{R}$ $\iff$

countable union of an effective sequence of rational intervals.[4]

We have no structure theorem for ***While*** semicomputability, only a partial result:

(4) For sets of reals:

(a) ***While*** semicomputable $\Longrightarrow$ countable union of effective sequence of rational intervals;

[2]By "interval" we will always mean *open* interval of reals.
[3]An algebraic interval is an interval between two algebraic numbers (roots of polynomials).
[4]A rational interval is an interval between two rational numbers.

(*b*) countable union of an effective sequence of *disjoint* rational intervals $\Longrightarrow$ ***While*** semicomputable

Notice that in (1) and (4), we need disjointedness because the ***While*** and $\textbf{\textit{While}}^{\mathsf{OR}}$ languages cannot implement interleaving on infinitely many of processes over partial algebras. For that we need the 'Exist' construct, as in (2) and (3).

The main steps in proving these results are:

(*a*) Engeler's Lemma, which states (roughly) that a semicomputable set can be expressed as the disjunction of an effective sequence of booleans. It is proved by constructing a computation tree for the procedure being considered.

(*b*) The Canonical Form Lemma for booleans over $\mathcal{R}$, which states that a boolean term over $\mathcal{R}$ can be expressed as a finite disjunction of finite conjunctions of polynomial inequalities.[5]

(*c*) The Characterization Lemma for booleans over $\mathcal{R}$, which states that a boolean term with only one real variable defines a union of finitely many algebraic intervals.

Note that Engeler's Lemma applies to all standard partial algebras, whereas the Canonical Form Lemma and Characterization Lemma apply only in special cases, such as ***While*** computability on $\mathcal{R}$.

The sequence of booleans given by Engeler's Lemma for $\textbf{\textit{While}}^{(\mathsf{OR})}$[6] has a *semantic disjointedness* property, which is used in the "$\Longrightarrow$" direction of the proof of (1).

[5]A polynomial inequality is a boolean term $p(x) > 0$, where $p(x)$ is a polynomial with integer coefficients.

[6] $\textbf{\textit{While}}^{(\mathsf{OR})}$ means that the case fits both the ***While*** and $\textbf{\textit{While}}^{\mathsf{OR}}$ languages.

This property does not hold for $\textbf{\textit{While}}^{\exists\mathsf{N}}$, because of the special nature of the associated "computation hypertree", which is not strictly a tree, but a directed acyclic graph (DAG).

1.3 Overview of the thesis

Chapter 2 gives the fundamental definitions of signature, algebra, standard and N-standard algebra and topological partial algebra, and a description of the topological partial algebra $\mathcal{R}$, which is used throughout the thesis.

In Chapter 3, we give the syntax and semantics of the ***While***, $\textbf{\textit{While}}^{\mathsf{OR}}$ and $\textbf{\textit{While}}^{\exists\mathsf{N}}$ languages. Our semantic definition follows the algebraic operational semantics of [TZ00].

Chapter 3 also defines the notions of computability, relative computability, semicomputability and projective semicomputability over the ***While*** language and its variants.

In Chapter 4 we prove Engeler's Lemma for the ***While***, $\textbf{\textit{While}}^{\mathsf{OR}}$ and $\textbf{\textit{While}}^{\exists\mathsf{N}}$ languages over N-standard partial algebras.

To prove this Lemma, two kinds of computation trees are constructed, one for the $\textbf{\textit{While}}^{(\mathsf{OR})}$ languages and the other, the "hypertree" for the $\textbf{\textit{While}}^{\exists\mathsf{N}}$ language.

This chapter was found to be the most mathematically challenging of the thesis.

Chapter 5 gives the main results of the thesis, namely the Structure Theorems, listed in §1.2.

Finally, this chapter also gives a proof of the theorem that $\textbf{\textit{While}}^{\exists\mathsf{N}}$ semicomputability on $\mathcal{R}$ is closed under projection, i.e., a projection of a $\textbf{\textit{While}}^{\exists\mathsf{N}}$ semicomputable set in $\mathcal{R}$ is again $\textbf{\textit{While}}^{\exists\mathsf{N}}$ semicomputable. This result is interesting because it does not hold in general over many-sorted algebras; projective semicomputability is generally more powerful (and less algorithmic) than semicomputability [TZ00]. We do not know if the result also holds for ***While*** and $\textbf{\textit{While}}^{\mathsf{OR}}$.

Chapter 2

Basic Concepts

In this chapter, we introduce the basic concepts used in this thesis, including signature, partial algebras, standard algebras, N-standard algebras, array algebras and topological algebras. We give examples of such algebras, specifically, the topological partial algebra $\mathcal{R}$ of real numbers, which will be central to our investigations.

Most of the material in this chapter is taken from [TZ99, TZ00], adapted to partial algebras.

2.1 Signatures

Definition 2.1 (Many-sorted signatures). A many-sorted *signature* Σ is a pair $\langle \boldsymbol{Sort}(\Sigma), \boldsymbol{Func}\,(\Sigma)\rangle$ where

(a) $\boldsymbol{Sort}(\Sigma)$ is a finite set of *sorts*.

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

$$\mathsf{F} :\ s_1 \times \cdots \times s_m \to s \qquad (m \geq 0)$$

Each symbol F has a *type* $s_1 \times \cdots \times s_m \to s$, 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* of F. The case $m = 0$ corresponds to *constant symbols*; we then write $\mathsf{F} : \to s$.

Definition 2.2 (Product types over Σ). A *product type* over Σ, or *Σ-product type*, is a symbol of the form $s_1 \times \cdots \times s_m$ $(m \geq 0)$, where $s_1, \ldots, s_m$ are sorts of Σ, called its *component sorts*. We write $u, v, w \ldots$ for Σ-product types.

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 (Function types). Let A be a Σ-algebra. A *function type* over Σ, or *Σ-function type*, is a symbol of the form $u \to s$, with *domain type* u and *range type* s, where u is a Σ-product type. We define $\boldsymbol{FuncType}\,(\Sigma)$ to be the set of Σ-function types, denoted $\tau,\ \tau',\ \ldots$.

Definition 2.4 (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 $\mathsf{F} : s_1 \times \cdots \times s_m \to s$, a (not necessarily total) function $\mathsf{F}^A : A_{s_1} \times \cdots \times A_{s_m} \rightharpoonup A_s$[1].

For a Σ-product type $u = s_1 \times \cdots \times s_m$, we define

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

Thus $x \in A^u$ iff $x = (x_1, \ldots, x_m)$, where $x_i \in A_{s_i}$ for $i = 1, \ldots, m$. So each Σ-function symbol $\mathsf{F} : u \to s$ has an interpretation $\mathsf{F}^A : A^u \rightharpoonup A_s$. If u is empty, *i.e.*, F is a constant symbol, then F^A is an element of A_s.

[1]We use '$\rightharpoonup$' to denote partial functions.

The algebra A is *total* if F^A is total for each Σ-function symbol F. Without such a totality assumption, A is called *partial.* In this thesis we deal mainly with partial algebras.

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

Example 2.5 (Booleans). The signature of booleans is of fundamental importance. It is defined as

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

The algebra $\mathcal{B}$ of booleans contains 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 $\Sigma(\mathcal{B})$.

Note that for convenience, we use '$\wedge$', '$\vee$' (in infix) and '$\neg$' in place of 'and' 'or' and 'not' respectively.

Example 2.6 (Naturals). The signature of naturals is defined as

signature	$\Sigma(\mathcal{N}_0)$
sorts	nat
functions	$0: \ \rightarrow$ nat,
	suc : nat $\rightarrow$ nat

The corresponding algebra of naturals $\mathcal{N}_0$ consists of the carrier $\mathbb{N}$={0,1,2,...} for sort nat and functions $0^{\mathsf{N}} : \rightarrow \mathbb{N}$, $\mathsf{suc}^{\mathsf{N}}:\mathbb{N} \rightarrow \mathbb{N}$.

Definition 2.7 (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.

- 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.
- A' is a Σ'-*expansion* of A if and only if A is the Σ-*reduct* of A.

Definition 2.8 (Σ-variables). For each Σ-sort s, there are (program) variables $\mathsf{a}^s, \mathsf{b}^s, ..., \mathsf{x}^s, \mathsf{y}^s, ...$ of sort s. Let $\boldsymbol{Var}_s(\Sigma)$ be the class of variables of sort s, and $\boldsymbol{Var}(\Sigma)$ be the class of all Σ-variables, x,y,....

Definition 2.9 (Σ-terms). Let $\boldsymbol{Term}(\Sigma)$ be the class of Σ-terms $t, \ldots$, and $\boldsymbol{Term}_s(\Sigma)$ the class of Σ-terms of sort s, defined by

$$t^s ::= \mathsf{x}^s | \mathsf{F}(t_1^{s_1}, \ldots, t_m^{s_m}) | \mathsf{if}\ b\ \mathsf{then}\ t_1^s\ \mathsf{else}\ t_2^s\ \mathsf{fi}$$

where $\mathsf{F} \in \boldsymbol{Func}\,(\Sigma)_{u \to s}$, $u = s_1 \times \cdots \times s_m$, b is a boolean term, i.e., a term of sort bool. (We are assuming here that $\mathsf{bool} \subseteq \boldsymbol{Sort}(\Sigma)$). We write $t : s$ to indicate that $t \in \boldsymbol{Term}_s$. Further,

We write $\boldsymbol{TermTup}(\Sigma)$ for the class of all tuples of distinct Σ-terms, and for $u = s_1 \times ... \times s_m$, $\boldsymbol{TermTup}_u$ for the class of u-tuple of terms, i.e.,

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

we write $t : u$ to indicate that t is a u-tuple of terms, *i.e.*, a tuple of terms of sorts $s_1, \ldots, s_m$.

We will often write $\boldsymbol{Var}$ for $\boldsymbol{Var}(\Sigma)$, $\boldsymbol{Term}$ for $\boldsymbol{Term}(\Sigma)$, etc.

Definition 2.10 (Closed terms over Σ). We define the class $\boldsymbol{T}(\Sigma)$ of *closed terms over* Σ, 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:

$$t^s ::= \mathsf{F}(t_1^{s_1}, \ldots, t_m^{s_m}) | \mathsf{if}\ b\ \mathsf{then}\ t_1^s\ \mathsf{else}\ t_2^s\ \mathsf{fi}$$

where $\mathsf{F} \in \boldsymbol{Func}\,(\Sigma)_{u \to s}$ $u = s_1 \times \cdots \times s_m$, $t_i^{s_i}$ are closed terms for $1 \le i \le m$ and b is a boolean closed term,

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

Definition 2.11 (Value of closed terms). For a Σ-algebra A and $t \in \boldsymbol{T}(\Sigma)_s$, we define the *value* $t_A \in A_s$ of t in A by structural induction on t:

$$\mathsf{F}(t_1, \ldots, t_m)_A \simeq \mathsf{F}^A((t_1)_A, \ldots, (t_m)_A)$$

$$(\mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\ \mathsf{fi})_A \simeq \begin{cases} (t_1)_A & \text{if } b_A = \mathsf{tt}, \\ (t_2)_A & \text{if } b_A = \mathsf{ff}, \\ \uparrow & \text{otherwise;} \end{cases}$$

where "$\uparrow$" means the evaluation diverges or is undefined, (and correspondingly, "$\downarrow$" means the evaluation converges) and "$\simeq$" means that either both sides diverge or converge to same value.

In particular, for m=0, *i.e.*, for a constant $\mathsf{c} : \to s$, $\mathsf{c}_A = \mathsf{c}^A$.

Definition 2.12 (Default terms; Default values).

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

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

Assumption 2.13 (Instantiation). In this thesis, we will assume: For each Σ-sort s, there is a closed term $t : s$ such that for each Σ-algebra with which we deal, $t_A \downarrow$.

2.2 Standard signatures and algebras

Recall the algebra $\mathcal{B}$ of booleans defined in Example 2.5.

Definition 2.14 (Standard signatures). A signature Σ is *standard* if $\Sigma(\mathcal{B}) \subseteq \Sigma$.

Definition 2.15 (Standard algebras). Given a standard signature Σ, a Σ-algebra A is a *standard algebra* if it is an expansion of $\mathcal{B}$.

Any many-sorted signature Σ can be *standardized* to a signature Σ^{B}, by adjoining the sort bool together with the standard boolean operations; and, correspondingly, any algebra A can be *standardized* to an algebra A^{B} by adjoining the algebra $\mathcal{B}$.

Assumption 2.16 (Standardness). In this thesis, we will assume:

All signatures Σ and Σ-algebras A are standard.

2.3 N-standard signatures and algebras

We standardize and extend $\mathcal{N}_0$ to include *equality* (eq^{N}) and *order* ($\mathsf{less}^{\mathsf{N}}$) on the naturals.

Example 2.17 (Naturals with order). The signature of the standard algebra $\mathcal{N}$ of the naturals is defined as

signature	$\Sigma(\mathcal{N})$
import	$\mathcal{N}_0$, $\mathcal{B}$
functions	$\mathsf{eq}^{\mathsf{N}}, \mathsf{less}^{\mathsf{N}} : \mathsf{nat}^2 \to \mathsf{bool}$

Definition 2.18 (N-standard signature). A standard signature Σ is called *N-standard* if

$$\Sigma(\mathcal{N}) \subseteq \Sigma$$

Definition 2.19 (N-standard algebra). The corresponding Σ-algebra A is *N-standard* if it is an expansion of $\mathcal{N}$.

Any many-sorted standard signature Σ can be *N-standardized* to a signature Σ^{N}, by adjoining the sort nat together with the standard arithmatic operations; and, correspondingly, any standard algebra A can be *N-standardized* to an algebra A^{N} by adjoining the algebra $\mathcal{N}$.

2.4 Topological partial algebras

Definition 2.20 (Continuity). Given two topological spaces X and Y, a partial function $f : X \rightharpoonup Y$ is *continuous* if for every open $V \subseteq Y$,

$$f^{-1}[V] =_{df} \{x \in X | x \in \mathbf{dom}(f) \text{ and } f(x) \in V\}$$

is open in X.

Remarks 2.21.

(a) $\mathbf{dom}(f) = f^{-1}[Y]$ is open in X because Y is open.

(b) We will see that $\mathcal{R}$ (defined in Example 2.24), like all topological algebras, satisfies the *Continuity Principle* [TZ99, Chapter 6].

$$\textit{Computability} \quad \Longrightarrow \quad \textit{Continuity.}$$

In other words, we assume that all computable functions are continuous.[2]

Definition 2.22 (Topological partial algebra). A topological partial algebra is a partial Σ-algebra with topologies on the carriers such that each of the basic Σ-functions is continuous.

Definition 2.23 (N-standard topological partial algebra). An N-standard topological partial algebra is a topological partial algebra which is also an N-standard algebra, such that the carriers $\mathbb{B}$ and $\mathbb{N}$ have the discrete topologies and the carrier $\mathbb{R}$ has the usual (Euclidean) topology.

[2]We note that all the well known concrete models over $\mathbb{R}$ [PER89, Wei00] satisfies this principle.

Examples 2.24 (Real algebras). The ring of reals $\mathcal{R}_0$ has a carrier $\mathbb{R}$ of sort real:

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

Next we standardize and N-standardize $\mathcal{R}_0$ to form the N-standard real algebra $\mathcal{R}_0^{\mathsf{B,N}}$

algebra	$\mathcal{R}_o^{\mathsf{B,N}}$
import	$\mathcal{R}_o$,$\mathcal{N}$
carriers	$\mathbb{R}$

Note that $\mathcal{B} \subset \mathcal{N}$.

Then we further expand $\mathcal{R}_0^{\mathsf{B,N}}$ to a *partial algebra*:

algebra	$\mathcal{R}$
import	$\mathcal{R}_o^{\mathsf{B,N}}$
functions	less_p:$\mathbb{R}^2 \rightharpoonup \mathbb{B}$
	eq_p:$\mathbb{R}^2 \rightharpoonup \mathbb{B}$

where $\mathsf{less_p} : \mathbb{R}^2 \rightharpoonup \mathbb{B}$, and $\mathsf{eq_p} : \mathbb{R}^2 \rightharpoonup \mathbb{B}$ are partial functions defined by:[3]

$$\mathsf{less_p}(x, y) = \begin{cases} \mathtt{tt} & \text{if x} < \text{y} \\ \mathtt{ff} & \text{if x} > \text{y} \\ \uparrow & \text{if x} = \text{y} \end{cases}$$

$$\mathsf{eq_p}(x, y) = \begin{cases} \uparrow & \text{if x} = \text{y} \\ \mathtt{ff} & \text{if x} \neq \text{y} \end{cases}$$

We will sometimes use the infix '$<$' and '$=$' for '$\mathsf{less_p}$' and '$\mathsf{eq_p}$' respectively.

The motivation for the above definition of $\mathsf{less_p}$ and $\mathsf{eq_p}$ was to make these functions continuous, in accordance with the Continuity Principle, as discussed in §1.1.

[3]These definitions are motivated by the properties of theses operations in the concrete models [PER89, Wei00].

The algebra $\mathcal{R}$ becomes a *topological partial algebra* by giving $\mathbb{R}$ its usual topology, and $\mathbb{B}$ and $\mathbb{N}$ the discrete topology. This topological partial algebra will be central to our investigation in this thesis. Because of its importance, we write it out in full:

algebra	$\mathcal{R}$
carriers	$\mathbb{R}$,$\mathbb{B}$,$\mathbb{N}$
functions	$0, 1 : \rightarrow \mathbb{R}$,
	$+, \times :\ \mathbb{R}^2 \rightarrow \mathbb{R}$,
	$- :\ \mathbb{R}^2 \rightarrow \mathbb{R}$
	$\mathsf{less_p}$: $\mathbb{R}^2 \rightharpoonup \mathbb{B}$
	$\mathsf{eq_p}$: $\mathbb{R}^2 \rightharpoonup \mathbb{B}$
	true, false: $\rightarrow$ bool
	and, or: bool^2 $\rightarrow$bool
	not: bool$\rightarrow$bool
	0^{N}: $\rightarrow \mathbb{N}$
	suc$^{\mathsf{N}}$: $\mathbb{N} \rightarrow \mathbb{N}$
	less$^{\mathsf{N}}$: $\mathbb{N}^2 \rightarrow \mathbb{B}$
	eq$^{\mathsf{N}}$: $\mathbb{N}^2 \rightarrow \mathbb{B}$

Remark 2.25. The algebra $\mathcal{R}$ satisfies the Instantiation Assumption 2.12. We take the default values to be $\delta_{\mathsf{real}} = 0$, $\delta_{\mathsf{nat}} = 0^{\mathsf{N}}$, and $\delta_{\mathsf{bool}} =$ false.

Chapter 3

While computation on standard partial algebras

In this chapter we study a number of high level programming languages based on the 'while' construct, applied to a standard signature Σ. We also give semantics for these languages relative to a partial Σ-algebra A, and define the notions of *computability, semicomputability* and *projective semicomputability* for these languages on A. Most of the definitions are adapted from [TZ00], for partial algebras.

The chapter begins by defining the syntax and semantics of the imperative ***While*** programming language. Then (in Section 3.9) we extend this language with special programming constructs to form two new languages: $\boldsymbol{While}^{\mathsf{OR}}$ and $\boldsymbol{While}^{\exists \mathsf{N}}$.

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

We define the syntax of the ***While*** programming language over the signature Σ.

Definition 3.1 (Atomic statements). $\boldsymbol{AtSt}(\Sigma)$ is the class of *atomic statements* $S_{\mathsf{at}}, ...,$ defined by:

$$S_{\mathsf{at}} ::= \mathsf{skip} \mid \mathtt{x} := t$$

where $\mathtt{x} := t$ is a *concurrent assignment*, i.e., for some product type u, $\mathtt{x}$ is a tuple of distinct variables of type u and $t : u$.

Definition 3.2 (Statements). $\boldsymbol{Stmt}(\Sigma)$ is the class of *statements* $S, ...,$ generated by:

$$S ::= S_{\mathsf{at}} \mid S_1 \ ; \ S_2 \mid \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \mid \mathsf{while}\ b\ \mathsf{do}\ S_0\ \mathsf{od}$$

Definition 3.3 (Procedures). $\boldsymbol{Proc}(\Sigma)$ is the class of *procedures* $P, Q, ...,$ which have the form:

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

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

$$D \equiv \mathsf{in}\ \mathsf{a} : u\ \mathsf{out}\ \mathsf{b} : v\ \mathsf{aux}\ \mathsf{c} : w$$

where a, b and c are tuples of *input variables, output variables* and *auxiliary variables* respectively.

We stipulate:

(i) `a`, `b` and `c` each consist of distinct variables, and they are pairwise disjoint,

(ii) every variable occurring in the body S must be declared in D (among `a`, `b` or `c`). [1]

If $\mathtt{a} : u$ and $\mathtt{b} : v$, then P is said to have type $u \to v$, written $P : u \to v$. Its *input type* is u and *output type* is v. We write $\boldsymbol{Proc}(\Sigma)_{u\to v}$ for the class of Σ-procedure of type $u \to v$.

Notations 3.4.

(i) We write $\boldsymbol{Stmt}$ for $\boldsymbol{Stmt}(\Sigma)$, etc.

(ii) We will often drop the sort superscript or subscript s.

(iii) For any expression E, we define $\boldsymbol{Var}(E)$ to be the set of variables in E.

(iv) We use '$\equiv$' to denote syntactic identity between two expressions.

3.2 States

Definition 3.5 (State). For each standard Σ-algebra A, a *state* on A is a family $\langle \sigma_s \mid s \in \boldsymbol{Sort}(\Sigma) \rangle$ of functions

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

Let $\boldsymbol{State}(A)$ be the set of states on A, with elements $\sigma, \ldots$.

[1] This will not hold for the auxiliary variable in the '`Exist`' construct to be described below. (See Remarks 3.22(b).)

Notation 3.6. For $\mathtt{x} \in \boldsymbol{Var}_s$, we write $\sigma(\mathtt{x})$ for $\sigma_s(\mathtt{x})$. Also, for a tuple $\mathtt{x} \equiv (\mathtt{x}_1, \ldots, \mathtt{x}_m)$, we write $\sigma[\mathtt{x}]$ for $(\sigma(\mathtt{x}_1), \ldots, \sigma(\mathtt{x}_m))$.

Definition 3.7 (Variant of a state). Let σ be a state over A, $\mathtt{x} \equiv (\mathtt{x}_1, \ldots, \mathtt{x}_n) : u$ and $a = (a_1, \ldots, a_n) \in A^u$ (for $n \geq 1$). We define $\sigma\{\mathtt{x}/a\}$ to be the state over A formed from σ by replacing its value at $\mathtt{x}_i$ by a_i for $i = 1, \ldots, n$. That is, for all variables $\mathtt{y}$:

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

3.3 Semantics of terms

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

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

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

The definition is by structural induction on t:

$$\begin{aligned}
[\![\mathtt{x}]\!]^A\sigma &= \sigma(\mathtt{x}) \\
[\![\mathsf{F}(t_1, \ldots, t_m)]\!]^A\sigma &\simeq \begin{cases} \mathsf{F}^A([\![t_1]\!]^A\sigma, \ldots, [\![t_m]\!]^A\sigma) & \text{if } [\![t_i]\!]^A\sigma\downarrow \text{ for } 1 \leq i \leq m \\ \uparrow & \text{otherwise} \end{cases} \\
[\![\mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\ \mathsf{fi}]\!]^A\sigma &\simeq \begin{cases} [\![t_1]\!]^A\sigma & \text{if } [\![b]\!]^A\sigma \downarrow \mathtt{tt} \\ [\![t_2]\!]^A\sigma & \text{if } [\![b]\!]^A\sigma \downarrow \mathtt{ff} \\ \uparrow & \text{otherwise.} \end{cases}
\end{aligned}$$

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

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

Remarks 3.8.

(a) For a closed term t, $[\![t_1]\!]^A\sigma = t_A$ as defined in Definition 2.11, as can easily be proved by structural induction on t.

(b) Evaluation of Σ-functions is *strict*, i.e., in the evaluation of a term $\mathsf{F}(t_1, ..., t_m)$ at state σ, if any of the subterm t_i diverges at σ, the result diverges.

Definition 3.9. For any $M \subseteq \boldsymbol{Var}$, and states σ_1 and σ_2, $\sigma_1 \approx \sigma_2$ (rel M) means for all $\mathtt{x} \in M$, $\sigma_1(\mathtt{x}) = \sigma_2(\mathtt{x})$.

Lemma 3.10 (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 $[\![t]\!]^A\sigma_1 \ \simeq \ [\![t]\!]^A\sigma_2$.*

Proof. By structural induction on t. □

3.4 Algebraic operational semantics

Algebraic operational semantics is a general method for defining the meaning of a statement S, in a wide class of imperative programming languages, as a partial *state transformation.*

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

We define this via a (partial) *computation step* function

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

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

The symbol '*' is a new object which indicates that the computation is over. Thus if for any n, $\boldsymbol{Comp}^A(S, \sigma, n) = *$, then for all $m \geq n$ $\boldsymbol{Comp}^A(S, \sigma, m) = *$.

Similarly, if for some n, $\boldsymbol{Comp}^A(S, \sigma, n)\uparrow$, then for all $m \geq n$, $\boldsymbol{Comp}^A(S, \sigma, m) \uparrow$.

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

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

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

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

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.

Then, we define the "one-step computation of S at σ" function

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

by

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

Finally, define *the computation step* function $\boldsymbol{Comp}^A$ by a simple recursion on n:

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

Note that for $n = 1$, this yields

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

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

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

is called the *computation sequence generated by* S at σ. There are three possibilities for the sequence:

(a) it terminates in a final state σ_l, where $\boldsymbol{Comp}^A(S,\sigma,l+1) = *$;

(b) it is infinite (*global divergence*);

(c) it is undefined from some point on (*local divergence*).

In case (a) the computation has an output, given by the final state; in case (b) the computation is non-terminating, with infinitely many computation steps, and has no output; and in case (c) the computation is also non-terminating, and has no output, because the state at one of the computation steps is undefined, as a result of a divergent computation of a term.

Now we are ready to derive the *i/o* (*input/output*) *semantics.* First we define the *length of a computation* of a statement S, starting in state σ, as the function

$$\boldsymbol{CompLength}^A : \boldsymbol{Stmt} \times \boldsymbol{State}(A) \rightharpoonup \mathbb{N}$$

by

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

Note that $\boldsymbol{CompLength}^A(S, \sigma)\!\downarrow$ in case (a) above only. Then, we define

$$[\![S]\!]^A(\sigma) \;\simeq\; \boldsymbol{Comp}^A(S, \sigma, \boldsymbol{CompLength}^A(S, \sigma)).$$

Lemma 3.11 (Functionality lemma for computation steps). *Suppose that* $\boldsymbol{Var}(S) \subseteq M$. *If* $\sigma_1 \approx \sigma_2$ (rel M), *then for all* $n \geq 0$,

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

Proof. By induction on n, using Functionality Lemma (3.10) for terms. □

Lemma 3.12 (Functionality lemma for statements). *Suppose that* $\boldsymbol{Var}(S) \subseteq M$. *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. By the Functionality Lemma (3.11) for computation steps. □

3.5 Operational semantics of statements

We now apply the above theory to the language $\boldsymbol{While}(\Sigma)$.

There are two atomic statements: skip and *concurrent assignment.* We define $\langle\!|S|\!\rangle^A$ for these:

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

Next we define $\boldsymbol{First}$ and $\boldsymbol{Rest}^A$ by structural induction on $S \in \boldsymbol{Stmt}$.

Case 1. S is atomic.

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

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

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

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

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

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

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

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

Lemma 3.13.

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

$$\begin{aligned} [\![\mathsf{skip}]\!]^A\sigma &= \sigma \\ [\![\mathtt{x} := t]\!]^A\sigma &\simeq \sigma\{\mathtt{x}/[\![t]\!]^A\sigma\} \end{aligned}$$

(ii)

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

(iii)

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

(iv)

$$[\![S \;\equiv\; \mathsf{while}\ b\ \mathsf{do}\ S_0\ \mathsf{od}]\!]^A\sigma \;\simeq\; \begin{cases} [\![S; \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}]\!]^A\sigma & \text{if } [\![b]\!]^A\sigma \downarrow \mathtt{tt} \\ \sigma & \text{if } [\![b]\!]^A\sigma \downarrow \mathtt{ff} \\ \uparrow & \text{if } [\![b]\!]^A\sigma \uparrow. \end{cases}$$

Proof. As in [TZ99, §4.2], adapted to partial algebras.

□

3.6 *While* computability

Finally, to conclude the semantics of ***While*** programs, 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 $\sigma[\mathsf{a}] = a$ and $\sigma[\mathsf{b}]$ and $\sigma[\mathsf{c}]$ have default values by instantiation assumption 2.13. Then

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

Note that P^A is well defined, by the following Functionality Lemma.

Lemma 3.14 (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), *and* $\sigma_1[\mathsf{b}] = \sigma_2[\mathsf{b}] = \delta^v$ *and* $\sigma_1[\mathsf{c}] = \sigma_2[\mathsf{c}] = \delta^w$ *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 a), *or*

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

Proof. The result follows from the Functionality Lemma 3.12, for statements. □

Definition 3.15 (*While* computable function).

(*a*) A function f on A is said to be *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*) ***While***(A) is the class of functions ***While*** computable on A.

Definition 3.16 (Halting set). The *halting set* of a procedure $P : u \to v$ on A is the set:

$$\boldsymbol{Halt}^A(P) =_{df} \{a \in A^u | P^A(a) \downarrow\}$$

Definition 3.17 (*While* semicomputable set). A set $R \subseteq A^u$ is ***While*** *semicomputable* on A if it is the halting set on A of some ***While*** procedure.

Definition 3.18 (Projectively *While* semicomputable set). A set $R \subseteq A^u$ is *projectively* ***While*** *semicomputable* on A iff R is the projection of a ***While*** semicomputable set on A, i.e., there exists a ***While*** semicomputable set $R' \subseteq A^{u\times v}$

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

Remark 3.19. Generally, projective semicomputability is a more powerful (and less algorithmic) concept than semicomputability. (But see Theorem 5.55!)

3.7 Nonrecursive procedure calls

In the language ***While***(Σ), we use procedures as a convenient device for defining functions. We can also use them to define a new kind of atomic statement, the *procedure call*

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

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

The semantics of ***While*** is then extended by adding the following clause to the semantics of atomic statements:

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

However, it is easy to eliminate all such procedure calls from a ***While***program statement. That is, the ***While*** language with non-recursive procedure calls is semantically equivalent to the ***While*** language without such procedure calls [TZ00, §3.9].

3.8 Relative *While* computability

Let g be a partial function

$$g \ : A^u \rightharpoonup A^v.$$

We define the programming language ***While***$(\mathtt{g})$ which extends the language ***While*** by including a special function symbol $\mathtt{g}$ of type $u \to v$. This is only used in the context of an "oracle call"

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

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

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

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

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

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

Lemma 3.20 (Transitivity of relative computability). *If f is **While** computable in $g_1, ..., g_m, h_1, ..., h_n$, and $g_1, ..., g_m$ are **While** computable in $h_1, ..., h_n$, then f is **While** computable in $h_1, ..., h_n$.*

Proof. Suppose that g_i is computable by a ***While***$(h_1, ..., h_n)$ procedure P_i for $i = 1, ..., m$. Now, given a ***While***$(g_1, ..., g_m, h_1, ..., h_n)$ procedure P for f, replace each oracle call $\mathtt{x}{:=}\mathsf{g}_i(t)$ in the body of P by the procedure call $\mathtt{x}{:=}P_i(t)$. This results in a ***While***$(h_1, ..., h_n)$ procedure which also computes f. $\square$

3.9 Expanding *While* to *While*$^{\mathsf{OR}}$ and *While*$^{\exists\mathsf{N}}$

First we extend the ***While*** language to form ***While***$^{\mathsf{OR}}$ language with the two boolean operations

$$\begin{aligned} &\mathsf{OR} &&: \mathsf{bool}^2 \rightharpoonup \mathsf{bool} \\ &\mathsf{AND} &&: \mathsf{bool}^2 \rightharpoonup \mathsf{bool} \end{aligned}$$

which we call "strong or" and "strong and" respectively [Kle71, §64], and use:

$$\triangledown \quad \text{and} \quad \vartriangle$$

as infix notations for OR and AND.

We define the semantics of these two operations:

For OR ($\triangledown$):

	tt	ff	$\uparrow$
tt	tt	tt	tt
ff	tt	ff	$\uparrow$
$\uparrow$	tt	$\uparrow$	$\uparrow$

For AND ($\triangle$):

	tt	ff	$\uparrow$
tt	tt	ff	$\uparrow$
ff	ff	ff	ff
$\uparrow$	$\uparrow$	ff	$\uparrow$

Remarks 3.21. Note that these two operations are *not* strict, unlike the boolean operations and and or (which are part of the standard signatures, see Remarks 3.8(b).)

The OR operation allows us to simulate *interleaving* or *dovetailing* at an abstract level, since it allows us to decide a disjunction of two boolean terms $b_1 \triangledown b_1$ to be true if either of these converges to tt (even if the other one diverges).

The AND operation can be defined as dual to OR:

$$b_1 \triangle b_2 \quad \Longleftrightarrow \quad \neg(\neg b_1 \triangledown \neg b_2)$$

Note that these are different from the "conditional or" and "conditional and", two other non-strict boolean operations, which are common in programming languages, but not useful for our present purpose.

We also adjoin a new boolean term

$$\mathsf{Exist}\ \mathsf{z} : P(t, \mathsf{z})$$

to the ***While*** language to form a new language ***While***$^{\exists\mathsf{N}}$, where the procedure P has the type $u \times \mathsf{nat} \to \mathsf{bool}$ and z is a "new" variable of sort nat. This will occur only in the context:

$$\mathsf{x}^{\mathsf{B}} := \ \mathsf{Exist}\ \mathsf{z} : P(t, \mathsf{z})$$

We define its semantics as:

$$[\![\ \mathsf{Exist}\ \mathsf{z} : P(t, \mathsf{z})]\!]^A \sigma \ \simeq \ \begin{cases} \mathtt{tt} & \text{if } \exists n : \ P^A([\![t]\!]^A \sigma, n) \downarrow \mathtt{tt} \\ \uparrow & \text{otherwise.} \end{cases}$$

This corresponds to the following operational semantics: interleave (or "dovetail") the computations for:

$$P^A(t, 0), \ P^A(t, 1), \ P^A(t, 2) ...$$

and return $\mathtt{tt}$ if and only if any of these procedures terminates and returns $\mathtt{tt}$, otherwise keep on going.

This operation allows us to simulate infinite interleaving or dovetailing at the abstract level. Note that this is different from "evaluating from the left", which can be simulated by a simple loop:

```
find := false;
z:=0;
repeat
  find := $P(t, z)$
  z:=z+1;
until find := true.
```

which will *diverge* e.g. in case:

$$P^A(t,1) \downarrow \mathsf{ff}, \quad P(t,2) \uparrow, \quad P(t,3) \downarrow \mathsf{tt}.$$

whereas Exist $\mathtt{z} : P(t, \mathtt{z})$ will converge to tt.

The significance of these new program constructs will be explained further in Chapter 5.

Remarks 3.22.

(a) The 'Exist' construct is "weakly" or "globally" deterministic, i.e., deterministic at abstract level, although the actual choice of $\mathtt{z}$ in a concrete implementation is nondeterministic. This is in contrast to the "choice" operator in [TZ04a], which is nondeterministic.

(b) In a procedure containing a statement $\mathtt{x}^{\mathsf{B}}$:= Exist z : $P(t, \mathsf{z})$, we do not include the variable z in the declaration, because z is bound by 'Exist' and hence invisible outside Exist z : $P(t, \mathsf{z})$.

(c) The 'Exist' construct can be implemented from the 'choose' construct [TZ04a], by

$$\mathtt{x}^{\mathsf{B}} := \textsf{Exist z} : P(t, \mathsf{z}) \quad \Longleftrightarrow \quad \mathtt{n} := \textsf{choose z} : P(t, \mathsf{z})\ ; \ \ \mathtt{x}^{\mathsf{B}} := P(t, \mathtt{n})$$

Lemma 3.23.

(1) *A* ***While*** *computable function is* ***While***^{OR} *computable.*

(2) *A* ***While***^{OR} *computable function is* ***While***$^{\exists N}$ *computable.*

Proof. (1): Obvious.

(2): A term $b_1 \triangledown b_1$ can be simulated as Exist z : $P(b_1,\ b_2, \mathsf{z})$, where P is defined as:

```
proc
in $b_1, b_2$ : bool;
        z : nat;
out   $\mathtt{x}^{\mathsf{B}}$ : bool;
begin
if z=1 $\wedge$ $b_1$ then $\mathtt{x}^{\mathsf{B}}$:= true;
else if z=2 $\wedge$ $b_2$ then $\mathtt{x}^{\mathsf{B}}$:= true;
  else $\mathtt{x}^{\mathsf{B}}$:= $b_1 \vee b_2$ ;
end.
```

Similarly the term $b_1 \vartriangle b_1$ can be simulated as Exist z : $P(b_1, b_2, \mathsf{z})$, where P is defined as:

```
proc
in $b_1, b_2$ : bool;
       z : nat;
out  $x^B$ : bool;
begin
  if z=1 $\wedge \neg b_1$ then $x^B$:= false;
  else if z=2 $\wedge \neg b_2$ then $x^B$:= false;
  else $x^B$:= $b_1 \wedge b_2$ ;
end.
```

□

Definition 3.24 (Algebraic term and program term). An *algebraic term* is (inductively) a term of the form x or $F(t_1, t_2, ..., t_m)$ where $\mathtt{x} \in$ ***Var***(Σ), $F \in$ ***Func***(Σ) and $t_1, \ldots, t_m$ are *algebraic terms*. All other terms are called *program terms*.

Thus program terms may also contain if...fi, OR, AND and Exist.

Note that all algebraic terms are evaluated strictly, which means that if any of the subterms diverge the whole term diverges. However, for program terms, this is not always true, since if...fi, OR, AND and Exist are not evaluated strictly.

3.10 *While** computation on $\mathcal{R}$

Given a standard signature Σ, and standard Σ-algebra A, we expand Σ and A in two stages: (1) N-standardise these to form Σ^{N} and A^{N}, as in §2.3; and (2) define, for each sort s of Σ, the carrier A_s^* to be the set of finite sequence or arrays a^* over A_s, of "starred sort" s^*.

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

While$^*(A)$ computation can then be defined as ***While*** computation on A^* where the input and output variables are simple, but the auxiliary variables can be arrays.

However, in the case we are interested in, i.e., $A = \mathcal{R}$, it was shown in [TZ00, §4.9] (actually for total algebras, but applicable also to $\mathcal{R}$) that computation on $\mathcal{R}$ with arrays does not increase computing power, i.e.:

$$\boldsymbol{While}^*(\mathcal{R}) \quad = \quad \boldsymbol{While}(\mathcal{R}).$$

Similarly it can be shown that:

$$\boldsymbol{While}^{\mathsf{OR}*}(\mathcal{R}) \quad = \quad \boldsymbol{While}^{\mathsf{OR}}(\mathcal{R}).$$

$$\boldsymbol{While}^{\exists\mathsf{N}*}(\mathcal{R}) \quad = \quad \boldsymbol{While}^{\exists\mathsf{N}}(\mathcal{R}).$$

So in our future investigation, we will not consider ***While****(or ***While***$^{\mathsf{OR}*}$ or ***While***$^{\exists\mathsf{N}*}$) semicomputability explicitly.

Chapter 4

Computation trees; Engeler's Lemma

Engeler's Lemma [Eng68] is an important theoretical tool for the research of this thesis. It states (roughly) that a semicomputable set can be expressed as the disjunction of an effective infinite sequence of booleans.

A proof of Engeler's Lemma for the ***While*** language on *total* algebras was given in [TZ00, §5]. In this chapter we prove Engeler's Lemma for the ***While***, ***While***$^{\mathsf{OR}}$ and ***While***$^{\exists\mathsf{N}}$ languages on a *partial* algebra A.

Our proof of Engeler's Lemma is based on computation trees (in the case of ***While***$^{(\mathsf{OR})}$) and computation hypertrees (in the case of ***While***$^{\exists\mathsf{N}}$).

For convenience, we first simplify the ***While*** programming language, and then, in the simplified language, we construct computation trees for the ***While***$^{\mathsf{OR}}$ and ***While***$^{\exists\mathsf{N}}$ languages.

4.1 ***While***$_0$ language

To simplify the formal development, we restrict the structure of ***While*** statements to a special form, and we show that all statements can be effectively transformed to this special form.

Definition 4.1 (Special form for *While* statements). A ***While*** statement S is said to be in *special form* if (inductively) it has one of the following forms:

- $S \equiv$ skip
- $S \equiv \mathtt{x}{:=}\ F(\mathtt{x}_1, ..., \mathtt{x}_n)$
- $S \equiv$ if $\mathtt{x}^{\mathsf{B}}$ then S_1 else S_2 fi
- $S \equiv$ while $\mathtt{x}^{\mathsf{B}}$ do S_0 od
- $S \equiv S_1; S_2$

where $\mathtt{x}$ and $\mathtt{x}_i$ are variables, $\mathtt{x}^{\mathsf{B}}$ is a boolean variable, and S_0, S_1 and S_2 are also in *special form.*

Note the restriction on the assignments ($\mathtt{x}^{\mathsf{B}}{:=}F(\mathtt{x}_1, ..., \mathtt{x}_n)$) and boolean tests ($\mathtt{x}^{\mathsf{B}}$). Let ***While***$_0(\Sigma)$ be the ***While***(Σ) language restricted to special form.

Lemma 4.2. *All **While** statements can be effectively transformed into **While**$_0$ statements.*

Proof. We define an effective transformation

$$S \to S^o$$

of ***While*** statements to ***While***$_0$ statements inductively, by the cases:

- For concurrent assignments $S \equiv \mathtt{x} := t$, where $\mathtt{x} \equiv \mathtt{x}_1, ..., \mathtt{x}_n$ and $t \equiv t_1, ..., t_n$, define

$$S^o \quad \equiv \quad \mathtt{z}_1 := \mathtt{x}_1; ...; \mathtt{z}_n := \mathtt{x}_n;\ (\mathtt{x}_1 := \hat{t}_1)^o; ...; (\mathtt{x}_n := \hat{t}_n)^o$$

  where $\mathtt{z}_1, ..., \mathtt{z}_n$ are new auxiliary variables and $\hat{t}_i \equiv t_i\langle(\mathtt{x}_1, ..., \mathtt{x}_n)/(\mathtt{z}_1, ..., \mathtt{z}_n)\rangle$, i.e., $\hat{t}_i$ is formed by the simultaneous substitution of $(\mathtt{z}_1, ..., \mathtt{z}_n)$ for $(\mathtt{x}_1, ..., \mathtt{x}_n)$ in t_i for $i = 1, .., n$.

- For $S \equiv \mathtt{x} := F(t_1, t_2, ..., t_m)$, define

$$S^o \quad \equiv \quad (\mathtt{z}_1 := t_1)^o;\ (\mathtt{z}_2 := t_2;)^o\ ...;\ (\mathtt{z}_m := t_m)^o;\ \mathtt{x} := F(\mathtt{z}_1, ..., \mathtt{z}_m).$$

- For $S \equiv \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}$, where b is a boolean expression, define

$$S^o \quad \equiv \quad \mathtt{x}^{\mathsf{B}} := b;\ \mathsf{if}\ \mathtt{x}^{\mathsf{B}}\ \mathsf{then}\ S_1^o\ \mathsf{else}\ S_2^o\ \mathsf{fi}$$

  where $\mathtt{x}^{\mathsf{B}}$ is a new boolean variable.

- For $S \equiv \mathsf{while}\ b\ \mathsf{do}\ S_0\ \mathsf{od}$, where b is a boolean expression, define

$$S^o \quad \equiv \quad (\mathtt{x}^{\mathsf{B}} := b)^o;\ \mathsf{while}\ \mathtt{x}^{\mathsf{B}}\ \mathsf{do}\ S_0^o;\ (\mathtt{x}^{\mathsf{B}} := b)^o\ \mathsf{od}$$

  where $\mathtt{x}^{\mathsf{B}}$ is a new boolean variable.

The transformation is clearly
- effective, and
- semantics preserving.

□

Definition 4.3 ($\boldsymbol{While}_{\mathbf{0}}^{\exists\mathsf{N}}$). We define $\boldsymbol{While}_{\mathbf{0}}^{\exists\mathsf{N}}$ to be the language formed by adding to $\boldsymbol{While}_0$ the statement

$$\mathtt{x}^{\mathsf{B}} := \ \mathsf{Exist}\ \mathsf{z} : P(t, \mathsf{z}).$$

Remarks 4.4.

(a) From now on, we will only consider $\boldsymbol{While}_0$ programs composed of statements in special form. To simplify the notation, we will write $\boldsymbol{While}$ instead of $\boldsymbol{While}_0$ and $\boldsymbol{While}^{\exists\mathsf{N}}$ instead of $\boldsymbol{While}_{\mathbf{0}}^{\exists\mathsf{N}}$.

(b) Notice that in $\boldsymbol{While}_0$ statements, the only way for a program to diverge locally is by the divergence of the right-hand side of an assignment statement.

4.2 Definability property

A definability predicate is needed in the construction of the computation tree and in the proof of Engeler's Lemma.

Definition 4.5 (Definability predicate).

(a) A *definability predicate* at sort s in a Σ-algebra A is a Σ-boolean expression $\boldsymbol{def}_s$, containing a distinguished free variable $\mathtt{x} : s$, such that (writing $\boldsymbol{def}_s(t)$ for $\boldsymbol{def}_s\langle \mathtt{x}/t\rangle$) for all $t \in \boldsymbol{Term}_s(\Sigma)$ and all $\sigma \in \boldsymbol{State}(A)$:

$$[\![\boldsymbol{def}_s(t)]\!]^A\sigma = \begin{cases} \mathtt{tt} & \text{if } [\![t]\!]\sigma\downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

(b) The Σ-algebra A has the *definability property* if it has a definability predicate at all Σ-sorts.

Lemma 4.6. *$\mathcal{R}$ has the definability property*

Proof. In $\Sigma(\mathcal{R})$, we can define $\boldsymbol{def}_s(t)$ as follows:

At sort nat,

$$\boldsymbol{def}_{\mathsf{nat}}(t) \equiv \mathsf{eq_p}(t, t)$$

At sort real,

$$\boldsymbol{def}_{\mathsf{real}}(t) \equiv \mathsf{less_p}(t, t+1)$$

For the boolean term Exist z : $P(t, \mathsf{z})$,

$$\boldsymbol{def}_{\mathsf{bool}}(\text{ Exist z : } P(t, \mathsf{z})) \equiv \text{ Exist z : } P(t, \mathsf{z})$$

For any other term t of sort bool,

$$\boldsymbol{def}_{\mathsf{bool}}(t) \equiv \mathsf{or}(t, \mathsf{not}\ t)$$

□

4.3 Gödel numbering of syntax

We assume given a family of numerical codings of the classes of syntactic expression $\ulcorner E \urcorner$ of Σ or Σ^*, i.e., a family $\boldsymbol{gn}$ of effective mappings from expressions E to natural numbers $\ulcorner E \urcorner = \boldsymbol{gn}(E)$ satisfying:

- $\ulcorner E \urcorner$ increases strictly with the complexity of (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 their respective subclasses, such as $\{ \ulcorner t \urcorner | t \in \boldsymbol{Term}\}$, $\{ \ulcorner t \urcorner | t \in \boldsymbol{Term}_s \}$ and $\{ \ulcorner S \urcorner | S \in \boldsymbol{Stmt}\}$, 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.

4.4 Computation tree for $\boldsymbol{While}^{\mathsf{OR}}(\Sigma)$

We define a computation tree $\mathcal{T}[S, \mathtt{x}]$ for a ***While*** statement S on $\mathcal{R}$, where $\boldsymbol{Var}(S) \subseteq \mathtt{x} : u = u_1 \times u_2 \times ... \times u_n$. The computation tree $\mathcal{T}[S, \mathtt{x}]$ is like an "unfolded flow chart" for S.

This is a simplified version of the computation tree defined in [TZ00, §5.10], adapted for the $\boldsymbol{While}_0$ language (which we call ***While*** now) and for *partial* algebras.

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 assignments and boolean tests.

Furthermore, each edge of $\mathcal{T}[S, \mathtt{x}]$ is labelled with a *syntactic state*, i.e., a tuple of terms $t \equiv t_1, ..., t_n$ with $t_i \in \boldsymbol{Term}_{\mathtt{x},s}$. 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* $\mathcal{T}^{-}[S_1, \mathtt{x}]$ which is just $\mathcal{T}[S, \mathtt{x}]$ without the 's' node.

We will also use the notation $\mathcal{T}[S, t]$ for the tree formed from $\mathcal{T}[S, \mathtt{x}]$ by replacing all edges labelled t' by $t'\langle \mathtt{x}/t \rangle$.

The definition is by structural induction on S.

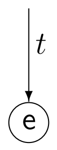
(i) $S \equiv \mathsf{skip}$. Then $\mathcal{T}[S, \mathtt{x}]$ is:

$$\begin{tikzcd}
\mathsf{s} \arrow[d, "\mathtt{x}"] \\
\mathsf{e}
\end{tikzcd}$$

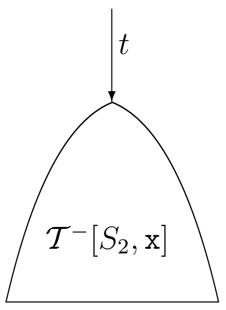
(ii) $S \equiv \mathtt{x}_j := F(\mathtt{x}_{j_1}, ..., \mathtt{x}_{j_m})$. Then $\mathcal{T}[S, \mathtt{x}]$ is:

$$\begin{tikzcd}
\mathsf{s} \arrow[d, "\mathtt{x}"] \\
\boxed{\mathtt{x}_j := F(\mathtt{x}_{j_1}, ..., \mathtt{x}_{j_m})} \arrow[d, "t \equiv (\mathtt{x}_1, ..., \mathtt{x}_{j-1}, F(\mathtt{x}_{j_1}, ..., \mathtt{x}_{j_m}), \mathtt{x}_{j+1}, ...\mathtt{x}_n)"] \\
\mathsf{e}
\end{tikzcd}$$

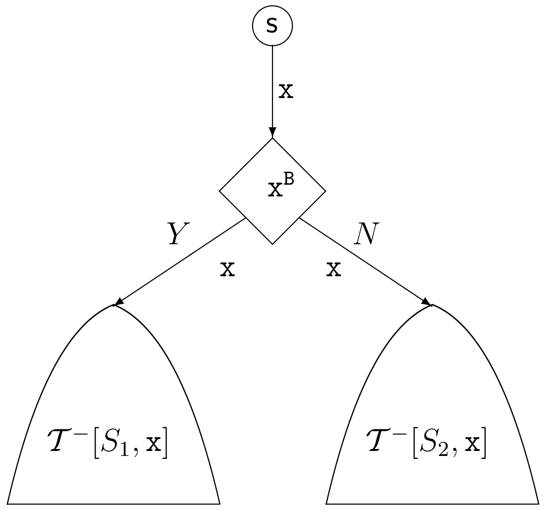
(*iii*) $S \equiv S_1; S_2$. Then $\mathcal{T}[S, \mathtt{x}]$ is formed from $\mathcal{T}[S_1, \mathtt{x}]$ by replacing each 'e' leaf



by the tree



(*iv*) $S \equiv$ if $\mathtt{x}^\mathtt{B}$ then S_1 else S_2 fi. Then $\mathcal{T}[S, \mathtt{x}]$ is:



s
x
$\mathtt{x}^\mathtt{B}$
Y
N
x
x
$\mathcal{T}^-[S_1, \mathtt{x}]$
$\mathcal{T}^-[S_2, \mathtt{x}]$

(v) $S \equiv \mathsf{while}\ \mathsf{x}^{\mathsf{B}}\ \mathsf{do}\ S_1\ \mathsf{od}$. For the sake of this case, we temporarily adjoin another kind of leaf to our tree formalism, labelled 'i' (for incomplete computation). Then $\mathcal{T}[S, \mathsf{x}]$ is defined as the "limit" of the sequence of trees $\mathcal{T}_n$, where $\mathcal{T}_0$ is defined as in Figure 1 and $\mathcal{T}_{n+1}$ is formed from $\mathcal{T}_n$ by replacing each i-leaf as in Figure 2 by the tree in Figure 3, where $\mathcal{T}_i^-[S_1, t]$ is formed from $\mathcal{T}^-[S_1, t]$ by replacing all e-leaves in the latter by i-leaves.

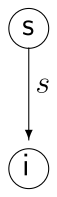


Figure 1

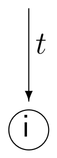


Figure 2

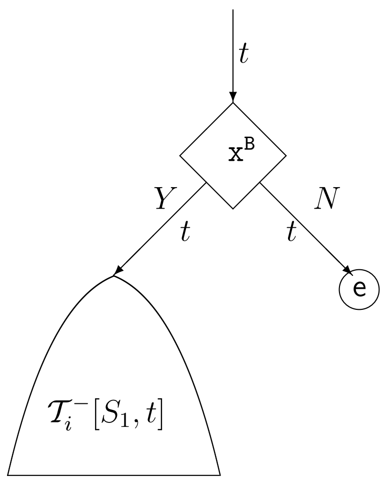


Figure 3

Remark 4.7. 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 an r.e. (recursively enumerable) set of numbers, with index primitive recursive in $\ulcorner S \urcorner$ and $\ulcorner \mathtt{x} \urcorner$.

4.5 Engeler's Lemma for *While*

We will show that the halting set of a ***While***, $\boldsymbol{While}^{\mathsf{OR}}$ and $\boldsymbol{While}^{\exists\mathsf{N}}$ procedure can be expressed as the countable disjunction of an effective infinite sequence of booleans. We must therefore first consider carefully the different possible semantics of infinite disjunctions in 3-valued logics[1]:

[1]Semantics of finite disjunctions and conjunctions were given by the definitions of the strict and strong boolean operators (and, or, AND, OR) in Chapter 3 (§3.9)

Discussion 4.8 (Four semantics of infinite disjunctions). Let (b_k) be a sequence of booleans. There are (at least) 4 different "reasonable" semantic definitions of the infinite disjunction of the b_k for 3 valued logics:

$$\bigvee_{k=0}^{\infty} b_k$$

(1) *Sequential evaluation* (i.e., "*evaluation from the left*") with two possible outputs ($\mathtt{tt}$ and $\uparrow$):

$$[\![\bigvee_{k=0}^{\infty} b_k]\!]^A\sigma = \begin{cases} \mathtt{tt} & \text{if } \exists k, [\![b_k]\!]^A\sigma \downarrow \mathtt{tt} \text{ and } \forall i < k, [\![b_i]\!]^A\sigma \downarrow \mathtt{ff} \\ \uparrow & \text{otherwise} \end{cases}$$

This definition is ***While*** computable, since we can evaluate b_k $(k = 0, 1, ...)$ one by one until:

• for some k, b_k converges to $\mathtt{tt}$, and all earlier b_j converges to $\mathtt{ff}$, or

• for some k, evaluation of b_k *diverges* and all earlier b_j converges to $\mathtt{ff}$ (local divergence), or

• all the b_k converge to $\mathtt{ff}$ (global divergence).

In the two latter cases evaluation of the infinite disjunction diverges.

(2) *Interleaving*, or "strong Kleene evaluation" with two possible outputs:

$$[\![\bigvee_{k=0}^{\infty} b_k]\!]^A\sigma = \begin{cases} \mathtt{tt} & \text{if } \exists k, [\![b_k]\!]^A\sigma \downarrow \mathtt{tt} \\ \uparrow & \text{otherwise} \end{cases}$$

This definition is not (in general) ***While*** computable, but it is ***While***$^{\exists \mathsf{N}}$ computable, by the semantic definition of $\mathsf{Exist\ z} : P(t, \mathsf{z})$ (§3.9). This is the definition we will use in this thesis, e.g. in the formulation of Engeler's Lemma 4.13.

(3) *Sequential evaluation*, with three possible outputs (tt, ff and $\uparrow$):

$$[\![\bigvee_{k=0}^{\infty} b_k]\!]^A\sigma = \begin{cases} \mathsf{tt} & \text{if } \exists k, [\![b_k]\!]^A\sigma \downarrow \mathsf{tt} \text{ and } \forall i < k, [\![b_i]\!]^A\sigma \downarrow \mathsf{ff} \\ \mathsf{ff} & \text{if } \forall k, [\![b_k]\!]^A\sigma \downarrow \mathsf{ff} \\ \uparrow & \text{otherwise} \end{cases}$$

This can be viewed as a generalization of Kleene's weak 3-valued disjunction [Kle71, §64].

(4) "*Strong Kleene evaluation*" with three possible outputs:

$$[\![\bigvee_{k=0}^{\infty} b_k]\!]^A\sigma = \begin{cases} \mathsf{tt} & \text{if } \exists k, [\![b_k]\!]^A\sigma \downarrow \mathsf{tt} \\ \mathsf{ff} & \text{if } \forall k, [\![b_k]\!]^A\sigma \downarrow \mathsf{ff} \\ \uparrow & \text{otherwise} \end{cases}$$

This can be viewed as generalization of Kleene's strong 3-valued disjunction [Kle71, §64].

Intuitively, definitions (1) and (2) are "concretely computable" [2] but definitions (3) and (4) are not.

[2]Refer to the §1.1 of this thesis.

Definition 4.9 (Strong equivalence of booleans). Two Σ-boolean b_1 and b_2 are *strongly equivalent over* A iff $\forall \sigma \in \boldsymbol{State}(A)$,

$$\begin{array}{lcl} [\![b_1]\!]^A\sigma \downarrow \mathtt{tt} & \Longleftrightarrow & [\![b_2]\!]^A\sigma \downarrow \mathtt{tt} \\ [\![b_1]\!]^A\sigma \downarrow \mathtt{ff} & \Longleftrightarrow & [\![b_2]\!]^A\sigma \downarrow \mathtt{ff} \\ [\![b_1]\!]^A\sigma \uparrow & \Longleftrightarrow & [\![b_2]\!]^A\sigma \uparrow \end{array}$$

Definition 4.10 (Weak equivalence of booleans). Two Σ-boolean b_1 and b_2 are *weakly equivalent over* A iff $\forall \sigma \in \boldsymbol{State}(A)$,

$$[\![b_1]\!]^A\sigma \downarrow \mathtt{tt} \quad \Longleftrightarrow \quad [\![b_2]\!]^A\sigma \downarrow \mathtt{tt}$$

Note that weak equivalence of booleans is closed under '$\wedge$' and '$\vee$', but not under '$\neg$'.

Definition 4.11. For any boolean term b with $\boldsymbol{Var}(b) \subseteq \mathtt{x} : u$, and $a \in A^u$, we write $b[a]$ for $[\![b]\!]^A\sigma \downarrow \mathtt{tt}$ for some $\sigma \in \boldsymbol{State}(A)$ where $\sigma[\mathtt{x}] = a$.

Note that this is well-defined, by the Functionality Lemma (3.10) for terms.

Definition 4.12 (Relation defined by boolean). A Σ-boolean term b with $\boldsymbol{Var}(b) \subseteq \mathtt{x} : u$, is said to define a relation $R \subseteq A^u$ (w.r.t $\mathtt{x}$) iff $\forall a \in A^u$

$$a \in R \quad \Longleftrightarrow \quad b[a].$$

Lemma 4.13 (Engeler's Lemma for *While*). *If a relation $R \subseteq A^u$ is **While** semicomputable over a standard partial Σ-algebra A, then R can be defined by the disjunction of an effective sequence of Σ-booleans over A.*

Note 4.14. Here "effective" means that the sequence of Gödel numbers of the booleans is recursive.

Proof. Suppose R is the halting set in A of the ***While*** procedure:

$$P \;\equiv\; \mathsf{proc\ in\ a\ out\ b\ aux\ c\ begin}\ S\ \mathsf{end}. \tag{4.1}$$

For each leaf λ of the computation tree $\mathcal{T}[S, \mathtt{x}]$ there is a boolean $b_{S,\lambda}$ with variables among $\mathtt{x} \equiv (\mathsf{a}, \mathsf{b}, \mathsf{c})$ which expresses the conjunction of results of the tests and definability predicates along the path from the root to λ.

An assignment node $\mathtt{x} := t^s$ in the path contributes to $b_{S,\lambda}$ the conjunct of definability

$$\ldots \wedge \boldsymbol{def}_{\boldsymbol{s}}(t) \wedge \ldots\ ,$$

which guarantees that the term $b_{S,\lambda}$ converges only if the evaluation of the term t converges at this point. If the path goes to the left through a test node $\mathtt{x}^{\mathsf{B}}$, we add the conjunct

$$\ldots \wedge \mathtt{x}^{\mathsf{B}} \wedge \ldots$$

to $b_{S,\lambda}$, and if the path goes to the right through $\mathtt{x}^{\mathsf{B}}$, we add the conjunct

$$\ldots \wedge \neg\mathtt{x}^{\mathsf{B}} \wedge \ldots\ .$$

(Since the boolean test only has the form of a boolean variable $\mathtt{x}^{\mathsf{B}}$, we do not need to add the $\boldsymbol{def}_{\mathsf{bool}}$ predicate here.)

We can *effectively enumerate* the leaves of the computation tree (using Remarks 4.7) to obtain a sequence of leaves $\lambda_0, \lambda_1, ...$, by increasing the depth, and, at a given depth, going from left to right. (If we have not reached an n^{th} leaf after searching through all the nodes of depth less than or equal to n we can return the default value false, to ensure an infinite output sequence.)

Define $\boldsymbol{Halt_S}$ as the countable disjunction of $b_{S,\lambda}$:

$$\boldsymbol{Halt}_S \equiv_{df} \bigvee_{k=0}^{\infty} b_{S,k}$$

where $b_{S,k} \equiv_{df} b_{S,\lambda_k}$, which expresses the condition under which the computation of S will eventually halt.

Note we are using the "interleaving 2-output" semantics defined in the case (2) of Discussion 4.8 for $\boldsymbol{Halt}_S$. Hence for the procedure P of equation 4.1, we can see that :

$$P^A(a) \downarrow \quad \Longleftrightarrow \quad \boldsymbol{Halt}_S[a].$$

Therefore $\forall a \in A^u$

$$a \in R \quad \Longleftrightarrow \quad P^A(a) \downarrow \;\Longleftrightarrow \quad \boldsymbol{Halt}_S[a] \quad \Longleftrightarrow \quad \bigvee_{k=0}^{\infty} b_{S,k}[a]$$

(Note that here "$\Longleftrightarrow$" stands for weak semantic equivalence.)

Hence if a relation R is semicomputable over a standard partial Σ-algebra A, then it can be expressed as the disjunction of an effective countable sequence of booleans over A.

$\square$

Note that the same proof holds for the $\boldsymbol{While}^{\mathsf{OR}}$ language (with the booleans $b_{S,k}$ containing 'OR' and 'AND'). Hence we also have:

Corollary 4.15 (Engeler's Lemma for $\boldsymbol{While}^{\mathsf{OR}}$). *If a relation R is $\boldsymbol{While}^{\mathsf{OR}}$ semicomputable over a standard partial Σ-algebra A, then R can be defined by the disjunction an effective countable sequence of Σ-booleans over A.*

4.6 Computation tree for $\boldsymbol{While}^{\exists\mathsf{N}}$

In order to prove Engeler's Lemma for the $\boldsymbol{While}^{\exists\mathsf{N}}$ language, we define (inductively) the computation tree for $\boldsymbol{While}^{\exists\mathsf{N}}$ statements, following the cases in the definition of the computation tree in §4.4. We add the case:

(vi) $S \equiv x_j := \mathsf{Exist\ z} : P(t, \mathsf{z});\ S_2$, where P is defined as:

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

Note that if $S \equiv x_j := \mathsf{Exist\ z} : P(t, \mathsf{z})$, (with S_2 missing), we let $S_2 \equiv \mathsf{skip}$.

The tree for S is formed from the tree:

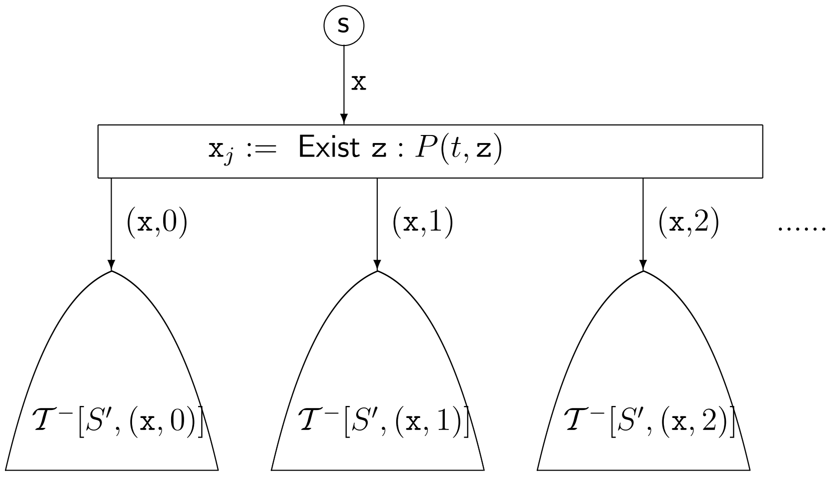
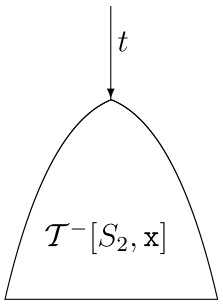

s
x
$\mathtt{x}_j :=$ Exist z : $P(t, \mathsf{z})$
(x,0)
(x,1)
(x,2)
......
$\mathcal{T}^-[S', (\mathtt{x}, 0)]$
$\mathcal{T}^-[S', (\mathtt{x}, 1)]$
$\mathcal{T}^-[S', (\mathtt{x}, 2)]$

Figure 4

by replacing each '$\mathtt{e}$' leaf of the trees $\mathcal{T}^-[S', (\mathtt{x}, i)]$ $(i = 0, 1, 2, ...)$ by the tree



and then collapsing these multiple occurrences of the subtree $\mathcal{T}^-[S_2, \mathtt{x}]$ to form the tree shown in Figure 5.

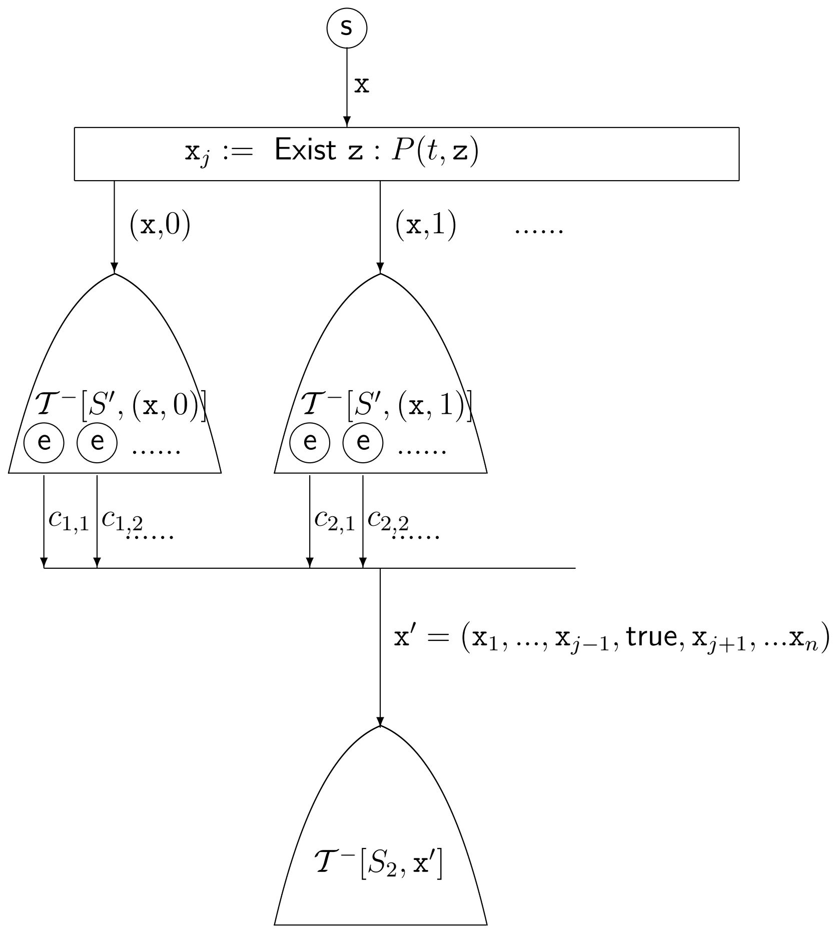


Figure 5

We call the subtrees $\mathcal{T}^-[S', (\mathtt{x}, i)]$ $(i = 0, 1, 2, ...)$ appearing in Figure 4 *proc-subtrees* of the whole computation tree.

Define a *channel* in the tree of Figure 5 to be a path through one of the "former leaves" of a proc-subtree $\mathcal{T}^-[S', (\mathtt{x}, \bar{i})]$, namely $c_{i,j}$ in Figure 5, where "i" refers to the i^{th} proc-subtree, and "j" refers to the j^{th} "former leaf" of the proc-subtree. Note that in this tree, there are (countably) infinitely many channels from the root to the subtree $\mathcal{T}^-[S_2, \mathsf{x}']$. We can effectively enumerate these channels by renaming channel $c_{i,j}$ as c_k where $k = \ulcorner(i,j)\urcorner$ [3].

Let $\mathcal{T}[S, \mathsf{x}]$ be a computation tree defined as above. Strictly speaking, $\mathcal{T}[S, \mathsf{x}]$ is not a tree, but a DAG (directed acyclic graph). However, if we consider the node $\mathsf{x}^{\mathsf{B}}{:=}$ $\mathsf{Exist\ z} : P(t, \mathsf{z})$ as an *atomic node* and ignore the internal details of the node, we get a computation tree just like those constructed in Section 4.4. We call the expanded tree a *hypertree* and the expanded node for $\mathsf{x}^{\mathsf{B}}{:=}$ $\mathsf{Exist\ z} : P(t, \mathsf{z})$ a *hypernode.* We can reduce the hypertree to a *reduced tree*, and a hypernode to a *reduced node*, when we ignore the details of the $\mathsf{Exist\ z} : P(t, \mathsf{z})$ node.

Notice that there are no leaves in the proc-subtrees because we have replaced all the leaves with the subtree $\mathcal{T}^-[S_2, \mathsf{x}]$. So we can effectively enumerate the leaves of the hypertree $\mathcal{T}[S, \mathsf{x}]$ as we did in the proof of Engeler's Lemma 4.13.

We define a *hyperpath* to be the route in a hypertree from the root of $\mathcal{T}[S, \mathsf{x}]$ to a leaf. At a hypernode of the hypertree, the hyperpath goes through a specific channel. Similarly, we define a *reduced path* as a path in the reduced tree, ignoring the details of the hypernodes.

We exhibit a hyperpath as in Figure 6. The picture shows part of the hypertree. (Note that the e_1, $e_2, \ldots$ are used to denote edges of the hypertree, not syntactic states.) To simplify the drawing, we ignore the details of the proc-subtree, leaving only the enumerated channels of each hypernode.

From the root 's' to the leaf 'e' of the hypertree, there is one reduced path corresponding to infinitely many hyperpaths, e.g. hyperpath$(1,4)$ consists of the edge e_1, the channel c_1 of the first hypernode, the edges e_2, e_3, e_4, e_5, e_6, the channel c_4 of the second hypernode, and the edges e_7, e_8.

[3]Gödel numbers of pairs will be defined in §5.1.

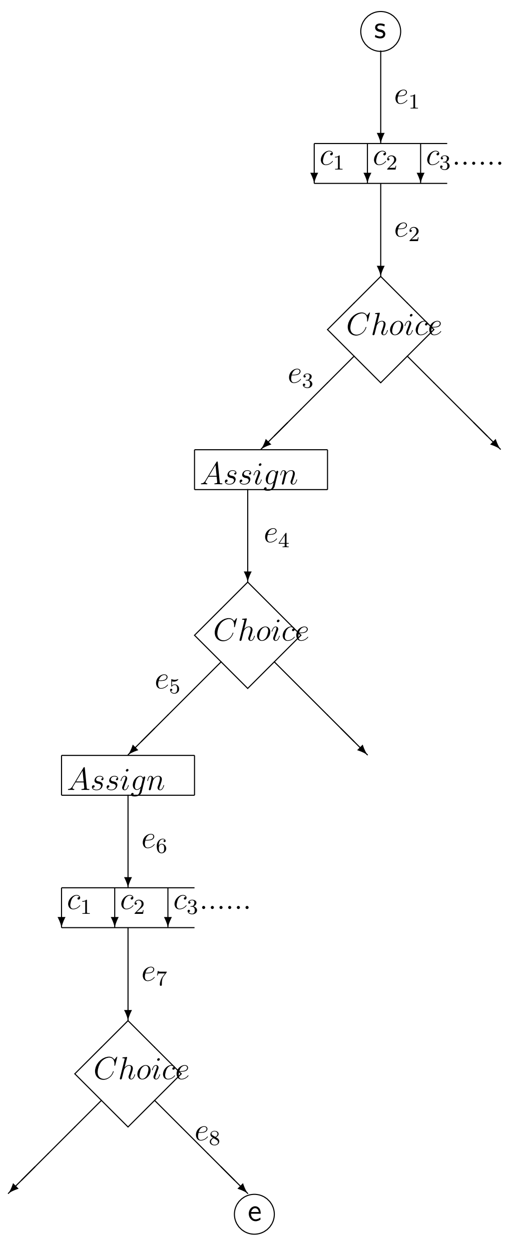

s
e_1
c_1 c_2 c_3……
e_2
$Choice$
e_3
$Assign$
e_4
$Choice$
e_5
$Assign$
e_6
c_1 c_2 c_3……
e_7
$Choice$
e_8
e

Figure 6

We can then enumerate all the hyperpaths of each leaf as follows:

Let hyperpath$(i_1, ..., i_m)$ be the hyperpath through the i_k^{th} channel at the k^{th} hypernodes on the route (where $k = 1, 2, ..., m$, and there are m hypernodes on the hyperpath). Then we rename the hyperpath$(i_1, ..., i_m)$ as hyperpath(i) where $i = \ulcorner(i_1, ..., i_m)\urcorner$.[4]

Finally, using the enumeration of the leaves of the hypertree as above and the enumeration of the hyperpaths of each leaf, we can enumerate all hyperpaths of a computation tree in a straightforward way.

Remarks 4.16.

(a) Notice that in an 'Exist' term of $\mathsf{Exist}\ \mathsf{z} : P(t, \mathsf{z})$ in the tree, there are (in general) other $\mathsf{Exist}\ \mathsf{z} : P'(t', \mathsf{z})$ terms in the procedure P. Then (recursively) we expand all such proc-subtrees in P so as to form a hypertree without any 'Exist' nodes.

(b) On each hyperpath, there are only two kinds of node:

- assignment nodes $x_j := F(t_1, ..., t_m)$ without the 'Exist' construct, or
- branching nodes at a boolean variable x^{B}.

(c) To each leaf corresponds (infinitely) many hyperpaths because of the multiple channels through the hypernodes which lead to that leaf.

(d) Note that hyperpaths which end in leaves are finite. Since any countably branching tree has only countably many finite paths, it is clear that the set of all hyperpaths ending in leaves and hence all booleans, for these paths, is countable.

[4]Gödel numbers of tuples will be defined in §5.1.

The important thing is to show that they can be effectively listed. This can be done by an extension of the method used for listing leaves in the computation tree for $\boldsymbol{While}^{(\mathsf{OR})}$. (See proof of Lemma 4.13.)

We proceed in stages. At stage n, create partial hypertree down to depth n (only), and at 'Exist' node only the first n branches. This has only finitely many paths ending in leaves. The whole procedure is effective.

Lemma 4.17 (Engeler's Lemma for $\boldsymbol{While}^{\exists\mathsf{N}}$). *If a relation R is $\boldsymbol{While}^{\exists N}$ semicomputable over a standard partial Σ-algebra A, then R can be defined by the disjunction of an effectively countable sequence of Σ-booleans over A.*

Proof. Suppose R is the halting set in A of the $\boldsymbol{While}^{\exists\mathsf{N}}$ procedure:

$$P \;\equiv\; \mathsf{proc\ in\ a\ out\ b\ aux\ c\ begin}\ S\ \mathsf{end}. \tag{4.2}$$

Consider the enumeration of the hyperpaths:

$$\rho_0,\ \rho_1,\ \rho_2,\ \cdots$$

described above. For each hyperpath ρ (not leaf, see Remark 4.16(c)) of the computation tree $\mathcal{T}[S, \mathtt{x}]$ there is a boolean $b_{S,\lambda}$ with variables among $\mathtt{x} \equiv (\mathsf{a}, \mathsf{b}, \mathsf{c})$ which expresses the conjunction of results of the tests and definability predicates from the root to a leaf of the $\mathcal{T}[S, \mathtt{x}]$ along ρ.

An assignment node $\mathtt{x} := t$ in the hyperpath, where t is *not* of the form of $\mathsf{Exist}\ \mathsf{z} : P(t', \mathsf{z})$, contributes to $b_{S,\rho}$ the conjunction

$$\ldots \wedge \boldsymbol{def}_{\boldsymbol{s}}(t) \wedge \ldots$$

which guarantees that the $b_{S,\rho}$ converges only if the evaluation of the term t converges at this point.

Suppose the hyperpath goes through a test node $\mathtt{x}^{\mathsf{B}}$, where the most recent assignment to $\mathtt{x}^{\mathsf{B}}$ was *not* of the form

$$\mathtt{x}^{\mathsf{B}} := \ \mathsf{Exist\ z} : P(t, \mathsf{z}). \tag{4.3}$$

Then if the hyperpath goes to the left branch we add the conjunctions $... \wedge \mathtt{x}^{\mathsf{B}} \wedge ...$ to $b_{S,\rho}$, and if the hyperpath goes to the right through $\mathtt{x}^{\mathsf{B}}$, we add the conjunctions $... \wedge \neg \mathtt{x}^{\mathsf{B}} \wedge ....$

If the hyperpath goes through a test node $\mathtt{x}^{\mathsf{B}}$, and the most recent assignment to $\mathtt{x}^{\mathsf{B}}$ *was* of the form like equation (4.3), then the hyperpath goes only to the left and $b_{S,\rho}$ is unchanged, (since in this case, $\mathtt{x}^{\mathsf{B}}$ cannot be false, and adding a conjunct true is redundant).

Define $b_{S,k} \equiv_{df} b_{S,\rho_k}$.

Then the halting set of a procedure

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

is defined by

$$\bigvee_{k=0}^{\infty} b_{S,k}$$

exactly as in the proof of Engeler's Lemma 4.13 for the ***While*** language.

□

Chapter 5

Structure theorems for semicomputable sets over $\mathcal{R}$

In this chapter, we present our structure theorems characterizing the ***While***, ***While***$^{\mathsf{OR}}$ and ***While***$^{\exists\mathsf{N}}$ semicomputable sets over $\mathcal{R}$. We will discuss the limitations of the ***While*** language in this regard and show how the ***While***$^{\mathsf{OR}}$ and the ***While***$^{\exists\mathsf{N}}$ languages correct these deficiencies.

In this Chapter we write $\Sigma = \Sigma(\mathcal{R})$ and Σ^{OR} for Σ augmented by OR, AND.

Let $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ be the set of natural numbers, integers, rational numbers, and real numbers respectively.

5.1 Definition and Gödel numbering of syntax:

Definition 5.1 (Gödel numberings of some mathematical objects).

(a) **Integers**. For integers n we define:

$$\ulcorner n \urcorner = \begin{cases} 2 * n & \text{if } n \geq 0 \\ -2 * n - 1 & \text{otherwise} \end{cases}$$

(b) **Pairs**. Define:

$$\mathsf{pair} : \mathbb{N}^2 \to \mathbb{N}$$

by

$$\mathsf{pair}(x, y) = \langle x, y \rangle = 2^x(2y + 1) - 1$$

Note that in this case the pair function is one-one and onto.

(c) **Tuples**. Define:

$$\mathsf{tuple} : \mathbb{N}^* \to \mathbb{N}$$

by

$$\begin{aligned} \mathsf{tuple}() &= 0 \\ \mathsf{tuple}(x_1, ..., x_n) &= \langle n, \langle x_1, ..., x_n \rangle \rangle = \langle n, \langle x_1, \langle x_2, \langle ... \langle x_{n-1}, x_n \rangle ... \rangle \rangle \rangle \rangle \end{aligned}$$

(d) **Rational numbers**: We can express each rational number r uniquely as $\frac{m}{n+1}$, where $m \in \mathbb{Z}$, $n \in \mathbb{N}$ and $\mathsf{gcd}(m, n + 1) = 1$. This induces, via the Gödel numberings of integers and of pairs, an effective *enumeration* or *listing* of $\mathbb{Q}$. We then define $\ulcorner r \urcorner = n$, where r is the n^{th} element in the list.

Notes:

(1) The above Gödel numberings of $\mathbb{N}^*$, $\mathbb{Z}$ and $\mathbb{Q}$ are surjective and therefore give us *effective enumerations* of pairs, tuples, integers and rationals.

(2) From now on, by "effective(ly)", we mean *effective* in the Gödel numberings of the mathematical objects referred to.

Definition 5.2. (Inverse functions) The left inverse $\mathsf{lt} : \mathbb{N} \to \mathbb{N}$ and right inverse $\mathsf{rt} : \mathbb{N} \to \mathbb{N}$ are defined as:

$$\begin{aligned} \mathsf{lt}(\langle x, y \rangle) &= x, \\ \mathsf{rt}(\langle x, y \rangle) &= y. \end{aligned}$$

Lemma 5.3. *The functions* pair*,* lt *and* rt *are primitive recursive.*

Proof. See [ZP93, §7.1] □

Definition 5.4 (Computable sequence of rationals). A sequence $(r_0, r_1, r_2, ...)$ of rationals is *computable* if there is a total recursive function $f : \mathbb{N} \to \mathbb{N}$ such that $f(n) = \ulcorner r_n \urcorner$.

In such a case, a Gödel number of the sequence can be defined as a Gödel number or index of the function f.

Definition 5.5 (Computable real number). A real number x is *computable* if there exists a computable sequence (r_n) of rational numbers which converges to x and a computable modulus of convergence, i.e., a total recursive strictly increasing function $M : \mathbb{N} \to \mathbb{N}$ such that:

$$\forall n, |r_n - x| < 2^{-M(n)}.$$

A Gödel number of the computable real number is then defined as:

$$\langle e, m \rangle$$

where e is an index of the sequence (r_n), and m is an index of the modulus of convergence M.

Lemma 5.6. *For each computable real number x, we can effectively construct sequences of rational numbers (r_n) and (s_n) such that (s_n) is increasing and (r_n) is decreasing and:*

$$\forall n : \ 0 < (x - r_n) < 2^{-n} \quad \textit{and}$$
$$\forall n : \ 0 < (s_n - x) < 2^{-n}.$$

Proof. The index of the computable real number x gives us an effective sequence (r'_n) such that:[1]

$$\forall n, \ |r'_n - x| < 2^{-n}$$

Define a sequence (r''_n) by:

$$r''_n = r'_n - 2^{-(n-1)}.$$

Since $|r'_n - x| < 2^{-n}$, we have

$$x - 2^{-n} < r'_n < x + 2^n,$$

[1]Since the modular function M is increasing, and $M(0) \geq 0$, we have $M(n) \geq n$ by induction on n.

and therefore:

$$\begin{aligned} x - 2^{-n} - 2^{-n+1} &< r''_n &< x + 2^{-n} - 2^{-n+1} \\ x - 2^{-n-2} - 2^{-n-1} &< r''_{n+2} &< x + 2^{-n-2} - 2^{-n-1}. \end{aligned}$$

Furthermore:

$$r''_n < x - 2^{-n} < x - 2^{-n-2} - 2^{-n-1} < r''_{n+2} < x. \tag{5.1}$$

Then we construct an increasing rational sequence (r_n) by

$$r_n = r''_{2n}.$$

by (5.1) (r_n) is increasing, and we can easily verify that

$$0 < (x - r_n) < 2^{-n}.$$

Symmetrically, we can effectively construct a decreasing rational sequence (s_n) such that:

$$0 < (s_n - x) < 2^{-n}.$$

□

Definition 5.7 (Integer polynomials). We write $\mathbb{Z}[\mathtt{x}_1, ..., \mathtt{x}_m]$ for the set of polynomials in m indeterminates $\mathtt{x}_1, ..., \mathtt{x}_m$ with coefficients in $\mathbb{Z}$.

Note that a polynomial $p(\mathtt{x}_1, ..., \mathtt{x}_m) \in \mathbb{Z}[\mathtt{x}_1, ..., \mathtt{x}_m]$ defines a polynomial function $p : \mathbb{R}^m \to \mathbb{R}$ in an obvious way. We give a Gödel numbering of $\mathbb{Z}[\mathtt{x}_1, ..., \mathtt{x}_m]$ in two steps:

(a) Code $a_i * x_1^{e_1} * x_2^{e_2} * x_3^{e_3} ... x_m^{e_m}$ as:

$$g_i = \langle \ulcorner a_i \urcorner, e_1, ..., e_m \rangle$$

(b) Then define

$$\ulcorner \sum_{i=0}^{n} (a_i \prod_{j=0}^{m} x_j^{e_j^i}) \urcorner = \langle g_1, ..., g_n \rangle$$

Definition 5.8 (Degree of a polynomial). For a unary integer polynomial

$$p(\mathtt{x}) \;=\; \sum_{i=0}^{n} (a_i \mathtt{x}^{e_i})$$

define the *degree* of p:

$$\boldsymbol{deg}(p) =_{df} \mathsf{max}\{e_i \mid 0 \leq i \leq n \text{ and } a_i \neq 0\}$$

Definition 5.9 (Polynomial inequality). A *polynomial inequality* is an expression of the form $p(\mathtt{x}) > 0$, where p is an integer polynomial.

Definition 5.10 (Algebraic numbers).

(a) An *algebraic number* is a root of an integer polynomial in one indeterminate.

(b) *Gödel numbering*: For an algebraic number a, define

$$\ulcorner a \urcorner = \langle \ulcorner p \urcorner, k \rangle$$

where a is the k^{th} smallest real root of the integer polynomial p.[2]

(c) Note that we can make this Gödel numbering surjective by letting $\langle \ulcorner p \urcorner, k\rangle$ be the biggest root of p if the polynomial p has fewer than k distinct roots.

Lemma 5.11. *Let $\mathbb{A}$, $\mathbb{R}_c$ be the set of algebraic real number and computable real numbers respectively. Then:*

$$\mathbb{Q} \subseteq \mathbb{A} \subseteq \mathbb{R}_c \subseteq \mathbb{R}$$

Note that the above embeddings are effective in the respective Gödel numberings. In other words, there is a computable function $f : \mathbb{N} \to \mathbb{N}$ such that if k is the Gödel number of a rational, the $f(k)$ is the Gödel number of the same number viewed as an algebraic number, etc.

Definition 5.12 (Algebraic, rational and computable real intervals).

(a) On $\mathbb{R}$, the intervals:

$$(a, b), \quad (-\infty, a), \quad (b, \infty)$$

are called *algebraic, rational* and *computable* real intervals if $a < b$ and a and b are algebraic, rational and computable reals respectively.

[2]We ignore multiplicity of roots. Also for uniqueness, we can assume that p is the minimal polynomial for a.

(*b*) *Gödel numberings.* We define:

$$\begin{aligned}
\ulcorner(a,b)\urcorner &= \langle\langle\ulcorner a\urcorner,\ulcorner b\urcorner\rangle,0\rangle\\
\ulcorner(b,\infty)\urcorner &= \langle\ulcorner b\urcorner,1\rangle\\
\ulcorner(-\infty,a)\urcorner &= \langle\ulcorner a\urcorner,2\rangle
\end{aligned}$$

Lemma 5.13. *Let (α,β) be an algebraic interval. We can effectively find a sequence of expanding rational intervals (r_i,s_i) such that*

$$(\alpha,\beta)=\bigcup_{i=0}^{\infty}(r_i,s_i)$$

Proof. By Lemmas 5.6 and 5.11. □

5.2 Basic algebraic results:

The following results can be found with proofs in standard texts in algebra [Lan90, Wae64], real analysis [Roy66, Rud76], and constructive analysis [PER89, Wei00].

Theorem 5.14. *An integer polynomial of degree n has no more than n real roots.*

Theorem 5.15 (Intermediate value theorem). *Let f be a function which is continuous on the closed interval $[a,b]$. Suppose $f(a)$ and $f(b)$ have different signs. Then there exists $c\in(a,b)$ such that $f(c)=0$.*

Corollary 5.16. *Let p be a unary polynomial and α_1 and α_2 two consecutive (distinct) roots of p. Then either*

$$\begin{aligned}
&\forall x\in(\alpha_1,\alpha_2),\ p(x)>0\\
\textit{or}\quad &\forall x\in(\alpha_1,\alpha_2),\ p(x)<0.
\end{aligned}$$

Corollary 5.17. *A unary polynomial of degree n with $m(\leq n)$ distinct real roots*

$$\alpha_1 < ... < \alpha_m$$

defines $m+1$ algebraic intervals:

$$(-\infty,\ \alpha),\ (\alpha_1,\ \alpha_2), ...(\alpha_{m-1},\ \alpha_m),\ (\alpha_m,\ \infty),$$

in each of which p is either only positive or only negative.

Lemma 5.18. *For two distinct computable reals c_1 and c_2, we can effectively decide whether $c_1 < c_2$ or $c_2 < c_1$.*

Lemma 5.19. *Given any unary polynomial p of degree n, we can find, effectively in $\ulcorner p \urcorner$:*

(1) *The number of distinct roots $m(\leq n)$ of p, and, writing these roots as*

$$\alpha_1\ <\ \alpha_2\ <\ ...\ <\ \alpha_m :$$

(2) (*a*) *a rational less than α_1,*

(*b*) *a rational between α_k and α_{k+1}, for $1 \leq k < m$, and*

(*c*) *a rational bigger than α_m.*

Proof. From Sturm's theorem [Wae64].

□

5.3 Canonical form for $\Sigma(\mathcal{R})$ booleans

Note: Unless otherwise stated, the following definitions and lemmas refer to the Σ^{OR}-language with the Σ-language as a special case.

Definition 5.20. A *positive boolean combination* of a set of booleans is a (finite) boolean expression build up from the atoms by '$\wedge$', '$\vee$', '$\triangle$', '$\triangledown$', *i.e.*, without '$\neg$'.

Lemma 5.21 (Strong canonical form for booleans over $\mathcal{R}$). *A Σ-boolean over $\mathcal{R}$ with variables among* $\mathtt{x} \equiv \mathtt{x_1}, ..., \mathtt{x_n}$ *of sort* real *only, not containing* AND, OR, *or* Exist, *is effectively strongly semantically equivalent over $\mathcal{R}$ to positive boolean combination of equations and inequalities of the form:*

$$p(\mathtt{x}) = 0 \quad \textit{and} \quad q(\mathtt{x}) > 0$$

where p and q are integer polynomials in $\mathtt{x}$.

Proof. First, we can show that:

(a) A term t : real over $\mathcal{R}$ with variables among $\mathtt{x}$ is effectively semantically equivalent to a "conditional polynomial" in $\mathtt{x}$ over $\mathbb{Z}$ in the form:

$$\mathsf{if} \quad b_1 \;\rightarrow\; p_1 \mid ... \mid b_k \;\rightarrow\; p_k \;\; \mathsf{fi}$$

where b_i are booleans and p_i are integer polynomials for $0 < i \leq k$.

(b) A term t : bool over $\mathcal{R}$ with variables among $\mathtt{x}$ is effectively strongly equivalent over $\mathcal{R}$ to a boolean combination of equations and inequalities:

$$p(\mathtt{x}) = 0 \quad \text{and} \quad q(\mathtt{x}) > 0$$

where p and q are integer polynomials.

Assertions (a) and (b) are proved simultaneously, by structural induction on t.

Next, we convert the boolean combination constructed in (b) to a *positive* boolean combination of polynomial equalities and inequalities by eliminating all negations, by means of the following strong equivalences:

$$\begin{array}{rcl}
\neg(b_1 \ \triangledown \ b_2) & \Longleftrightarrow & \neg b_1 \ \ \triangle \neg \ b_2 \\
\neg(b_1 \ \triangle \ b_2) & \Longleftrightarrow & \neg b_1 \ \triangledown \neg \ b_2 \\
\neg(b_1 \ \vee \ b_2) & \Longleftrightarrow & \neg b_1 \ \wedge \ \neg b_2 \\
\neg(b_1 \ \wedge \ b_2) & \Longleftrightarrow & \neg b_1 \ \vee \ \neg b_2
\end{array}$$

$$\begin{array}{rclcl}
\neg\neg b & \Longleftrightarrow & b & & \\
\neg(p(x) > 0) & \Longleftrightarrow & (p(x) < 0) & \Longleftrightarrow & (-p(x) > 0) \\
\neg(p(x) = 0) & \Longleftrightarrow & (p(x) > 0) \vee (-p(x) > 0) & &
\end{array}$$

$\square$

Corollary 5.22 (Weak canonical form for booleans over $\mathcal{R}$). *A Σ-boolean with variables among* $\mathtt{x} \equiv \mathtt{x_1}, ..., \mathtt{x_n}$ *of sort* real *only, not containing* AND*,* OR*, or* Exist*, is effectively weakly semantically equivalent over $\mathcal{R}$ to a positive boolean combination of inequalities of the form:*

$$p(\mathtt{x}) > 0$$

where p is an integer polynomial in $\mathtt{x}$.

Proof. By the semantics of $\mathsf{eq_p}$, any conjunction containing a term $p(\mathtt{x}) = 0$ will either diverge, or converge to ff, and can therefore be deleted from the disjunction given by Lemma 5.21.

□

Lemma 5.23 (Characterization Lemma for boolean terms over $\mathcal{R}$). *A Σ^{OR}-boolean with one variable $x \in \mathbb{R}$ uniquely defines a union of finitely many disjoint algebraic intervals.*

Proof. First convert the boolean to its weak canonical form given by Lemma 5.22. We then proceed by structural induction in b.

Base case: $b \equiv p(\mathtt{x}) > 0$, where $p(x)$ integer polynomial and x : real. Use Corollary 5.17.

Induction step: This follows from the fact that the class of finite unions of algebraic intervals is closed under (binary) union and (binary) intersection.

The proof is straightforward except for a subtle point due to the different semantics of '$\vee$' and '$\triangledown$'. To clarify this, we consider the special case that two booleans each define a single algebraic interval, say b_1 defines (α_1, β_1), and b_2 defines $(\alpha_2,\ \beta_2)$, where:

$$\alpha_1 < \alpha_2 < \beta_1 < \beta_2.$$

Then:

(1) $b_1 \wedge b_2$ defines the algebraic interval:

$$(\alpha_2, \beta_1);$$

(2) $b_1 \bigtriangleup b_2$ also defines the algebraic interval:

$$(\alpha_2, \beta_1);$$

(3) $b_1 \vee b_2$ defines the disjoint union of the three algebraic intervals:

$$(\alpha_1, \alpha_2)\ \cup\ (\alpha_2, \beta_1)\ \cup\ (\beta_1, \beta_2);$$

(4) $b_1 \bigtriangledown b_2$ defines the single algebraic interval:

$$(\alpha_1, \beta_2).$$

$\square$

This lemma will be used in the proofs of Lemmas 5.43 and 5.45.

Remark 5.24. In the example given in the above proof, the fact that $b_1 \wedge b_2$ and $b_1 \vartriangle b_2$ defines the same interval, whereas $b_1 \vee b_2$ and $b_1 \triangledown b_2$ do not, is related to the fact that the former pair are weakly semantically equivalent whereas the latter pair are not. (See definitions of 'AND' and 'OR' in §3.9). In this case when $x = \alpha_2$ or $x = \beta_1$,

$$(b_1 \vee b_2[x]) \uparrow \quad \text{but} \quad (b_1 \triangledown b_2)[x] \downarrow \mathtt{tt}$$

Note that the following two lemmas (5.26 and 5.28) apply to all standard partial algebras A.

Definition 5.25 (Semantic disjointedness). A sequence $b_0, b_1, b_2, ...$ of boolean terms is *semantically disjoint* over A if *for any state* σ on A and any n,

$$[\![b_n]\!]^A\sigma \downarrow \mathtt{tt} \quad \Longrightarrow \quad (\forall i \neq n, [\![b_i]\!]^A\sigma \downarrow \mathtt{ff} \text{ or } [\![b_i]\!]^A\sigma \uparrow)$$

Lemma 5.26 (Disjointedness Lemma). *The sequence of computable boolean terms generated from a* ***While***$^{\mathsf{OR}}$ *computation tree as in Lemma* 4.13 *is semantically disjoint.*

Proof. Let i, j be distinct natural numbers and $b_{S,i} \equiv b_{S,i_1} \wedge ... \wedge b_{S,i_m}$ and $b_{S,j} \equiv b_{S,j_1} \wedge ... \wedge b_{S,j_n}$. From the definition of $b_{S,k}$ in Lemma 4.13, $b_{S,k}$ defines the path from the root to the k^{th} leaf of the computation tree of S. Therefore for the path from the root to the i^{th} leaf and the path from the root to the j^{th} leaf, there must be a branching node x^{B} where the two paths depart from each other. i.e. $\exists l < \mathsf{min}(m, n)$ such that

$$b_{S,i_1} \equiv b_{S,j_1}, ...b_{S,i_{(l-1)}} \equiv b_{S,j_{(l-1)}},$$

but

$$b_{S,i_l} = \neg(b_{S,j_l}).$$

So if for some σ, $[\![b_{S,i}]\!]^A\sigma \downarrow \mathtt{tt}$, then $[\![b_{S,i_l}]\!]^A\sigma \downarrow \mathtt{tt}$ for all l and so for any $j \neq i$ $b_{S,j} \equiv ... \wedge \neg\, b_{S,i_l} \wedge ...$ cannot converge to $\mathtt{tt}$.

□

This Lemma will be used in in the proof of Lemmas 5.36 and 5.37, and hence in the proof of Structure Theorems 5.52 and 5.49

Remark 5.27. Note that the Disjointedness Lemma does not hold for the boolean sequence generated from the $\boldsymbol{While}^{\exists \mathsf{N}}$ computation trees, since (unlike the case with ordinary computation trees) many hyperpaths in hypertree may end in the same leaf. (See the definition of "hypertree" in Chapter 4, and Remark 4.16(c).)

Lemma 5.28. *If an effective sequence of booleans* (b_k) *is semantically disjoint over* $\mathcal{R}$*, then, for all* $x \in \mathbb{R}^m$,

$$\bigvee_{k=0}^{\infty} b_k[x]$$

can be evaluated from the left.[3]

Proof. Since the b_k's are semantically disjoint, and in $\mathcal{R}$, divergence can only appear at the boundary points of the algebraic intervals, if for some σ there exists a k such that $[\![b_k]\!]^A\sigma \downarrow \mathtt{tt}$, then for all $i \neq k$, $[\![b_i]\!]^A\sigma \downarrow \mathtt{ff}$. Further if for some σ there exists k such that $[\![b_k]\!]^A\sigma \uparrow$, then there cannot exist b_i, such that $[\![b_i]\!]^A\sigma \downarrow \mathtt{tt}$. □

[3] *See the definition of "evaluation from the left" in case* (1) *of Discussion* 4.8.

This lemma is crucial in the proof of Lemmas 5.43 and 5.45, where "evaluation from the left" is implemented as evaluation by a 'while' loop. (See Discussion 5.48.)

Next we prepare several lemmas for the structural theorems in the next section.

Lemma 5.29. *There are* ***While**(*$\mathcal{R}$*)* *computable embeddings:*

(*a*) *of* $\mathbb{N}$ *into* $\mathbb{R}$

$$\boldsymbol{\iota}_{\mathsf{N}} : \mathbb{N} \to \mathbb{R}$$

(*b*) *of* $\mathbb{Z}$ *into* $\mathbb{R}$

$$\boldsymbol{\iota}_{\mathsf{Z}} : \mathbb{N} \to \mathbb{R}$$

where for any $a \in \mathbb{Z}$

$$\boldsymbol{\iota}_{\mathsf{Z}}(\ulcorner a \urcorner) = a$$

Proof. (*a*): By a simple while loop.

(*b*): Clear.

$\square$

Lemma 5.30. *There is a* ***While**(*$\mathcal{R}$*)* *computable function:*

$$\boldsymbol{eval} : \mathbb{N} \times \mathbb{R} \to \mathbb{R}$$

such that for any integer polynomial $p \in \mathbb{Z}[\mathtt{x}]$*, and* $a \in \mathbb{R}$*:*

$$\boldsymbol{eval}(\ulcorner p \urcorner, a) = p(a)$$

Proof. The function ***eval*** is defined by induction on $\boldsymbol{deg}(p)$.

□

Note that this is a special case of the *term evaluation property (TEP)* over $\mathcal{R}$ [TZ00, §4.7]. In fact, this is the main step in proving the TEP for $\mathcal{R}$.

Lemma 5.31. *There is a* ***While*($\mathcal{R}$)** *computable function:*

$$\boldsymbol{less}_{\mathsf{Q}} : \mathbb{N} \times \mathbb{R} \rightharpoonup \mathbb{B}$$

such that for $r \in \mathbb{Q}$ and $x \in \mathbb{R}$:

$$\boldsymbol{less}_{\mathsf{Q}}(\ulcorner r \urcorner, x) \quad \Longleftrightarrow \quad r < x$$

where "$\Longleftrightarrow$" is strong semantic equivalence.

Proof. Let $r = \frac{m}{n+1}$, then by the assumptions on Gödel numbering in §4.3, we can primitively recursively retrieve $\ulcorner m \urcorner$ and $\ulcorner n \urcorner$ from $\ulcorner r \urcorner$. Then

$$r < x \quad \Longleftrightarrow \quad \boldsymbol{\iota}_{\mathsf{Z}}(\ulcorner m \urcorner) < (\boldsymbol{\iota}_{\mathsf{N}}(n) + 1) \times x$$

where "$\Longleftrightarrow$" is strong semantic equivalence.

□

Lemma 5.32. *There is a* $\boldsymbol{While}^{OR}(\mathcal{R})$ *computable function:*

$$\boldsymbol{less}_{\mathsf{A}} : \mathbb{N} \times \mathbb{R} \rightharpoonup \mathbb{B}$$

such that for $\alpha \in \mathbb{A}$[4] *and* $x \in \mathbb{R}$*:*

$$\boldsymbol{less}_{\mathsf{A}}(\ulcorner\alpha\urcorner, x) \Longleftrightarrow \alpha < x$$

where "$\Longleftrightarrow$*" is strong semantic equivalence.*

Proof. By Lemma 5.3, we can effectively retrieve from $\ulcorner\alpha\urcorner$ the Gödel numbers $\ulcorner p\urcorner$ and $\ulcorner k\urcorner$, where α is the k^{th} root of p. Then by Lemma 5.30, we have a ***While*** computable function ***eval*** such that

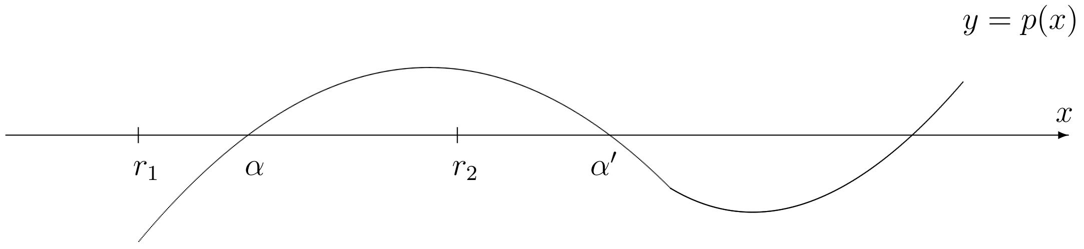
$$\boldsymbol{eval}(\ulcorner p\urcorner, a) = p(a)$$

Also by Lemma 5.19, we can effectively find two rationals r_1 and r_2, such that $r_1 < \alpha < r_2$, and α is the only root of p between these two rationals.

There are four cases, which can be effectively distinguished by Sturm's Theorem:

[4] $\mathbb{A}$ *is the set of algebraic numbers, as in Lemma* 5.11.

Case 1 :

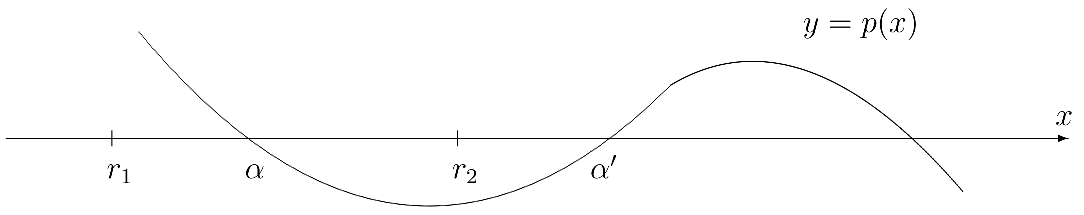


We can see that $\alpha < x$ if and only if:

$$[(r_1 < x) \wedge (p(x) > 0)] \triangledown [r_2 < x]$$

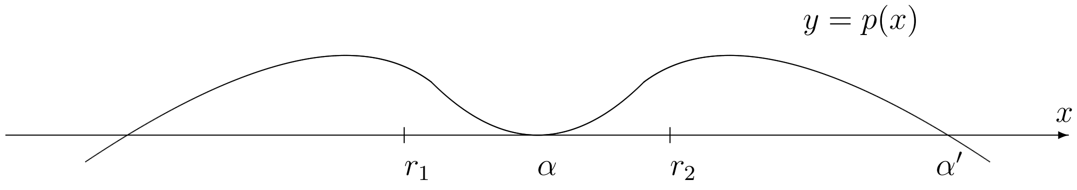
Note the use of the strong disjunction here! (See Remark 5.33 below.)

Case 2 :



This reduces to Case 1 by replacing $p(x)$ by $-p(x)$.

Case 3 :

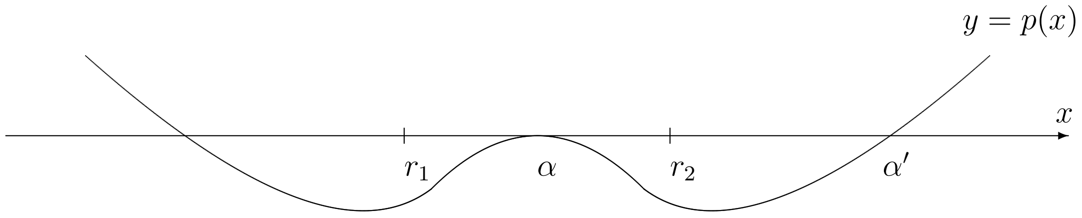


In this case α is a repeated root of p. Then by choosing r_1 and r_2 sufficiently close to α (so that $p'(r_2) > 0$, $p'(r_1) < 0$ and there is no root of $p'(x)$ between r_1 and α, or between α and r_2) we have $\alpha < x$ iff:

$$[(r_1 < x) \wedge (p'(x) > 0)] \triangledown [r_2 < x]$$

where p' is the derivative of p.

Case 4 :



This reduced to Case 3 by replacing $p(x)$ by $-p(x)$. □

Note that all the above operations on polynomials are effective; for example, $\ulcorner -p \urcorner$ and $\ulcorner p' \urcorner$ are primitive recursive in $\ulcorner p \urcorner$.

Remark 5.33 (Need for strong disjunction). In Case 1, if $x = r_2$, then the disjunct $(r_2 < x)$ will diverge, and so we need '$\triangledown$' to make the whole expression converge. Similarly for the other cases. (See also Remark 5.24.)

Lemma 5.34. *There is a **While** computable function*

$$\mathsf{In}_\mathsf{Q} : \mathbb{N} \times \mathsf{real} \rightharpoonup \mathit{bool}$$

such that:

$$\mathsf{In}_\mathsf{Q}(\ulcorner (r_1,\ r_2) \urcorner, x) \simeq \begin{cases} \mathtt{tt} & \textit{if} \ \ x \in (r_1,\ r_2) \\ \mathtt{ff} & \textit{if} \ \ x < r_1 \ \ \textit{or} \ \ \ r_2 < x \\ \uparrow & \ \textit{otherwise} \end{cases}$$

where r_1 and r_2 are rational numbers.

Proof. By assumptions of Gödel numberings in §4.3, we can primitive recursively retrieve $\ulcorner r_1 \urcorner$ and $\ulcorner r_2 \urcorner$ from $\ulcorner (r_1, r_2) \urcorner$ and therefore define:

$$\mathsf{In}_\mathsf{Q}(\ulcorner (r_1,\ r_2) \urcorner, x) \simeq_{df} \boldsymbol{less}_\mathbf{Q}(\ulcorner r_1 \urcorner, x) \wedge (\neg\ \boldsymbol{less}_\mathbf{Q}(\ulcorner r_2 \urcorner, x))$$

which is ***While*** computable by Lemma 5.31.

□

Lemma 5.35. *There is a* ***While***^{OR} *computable function*

$$\mathsf{In}_\mathsf{A} : \mathbb{N} \times \mathsf{real} \rightharpoonup \mathit{bool}$$

such that:

$$\mathsf{In}_\mathsf{A}(\ulcorner(\alpha,\ \beta)\urcorner, x) \simeq \begin{cases} \mathtt{tt} & \text{if } x \in (\alpha,\ \beta) \\ \mathtt{ff} & \text{if } x < \alpha \text{ or } \beta < x \\ \uparrow & \text{otherwise} \end{cases}$$

where α and β are algebraic numbers.

Proof. Like the proof of Lemma 5.34, except that instead of ***less***$_\mathbf{Q}$, we use ***less***$_\mathbf{A}$, which is ***While***$^{\mathsf{OR}}$ computable by Lemma 5.32. □

5.4 Characterizations of semicomputable real sets

In this section, we prove the "$\Longrightarrow$" direction of the structure theorems.

Lemma 5.36. *If a set $R \subseteq \mathbb{R}$ is* ***While*** *semicomputable over $\mathcal{R}$, then R can be expressed as the countable union of an effective sequence of disjoint algebraic intervals.*

Proof. If $R \subseteq \mathbb{R}$ is ***While*** semicomputable, then by Engeler's lemma (4.13) for the ***While*** language,

$$a \in R \iff \bigvee_{k=0}^{\infty} b_k[a]$$

for an effective sequence (b_k) of booleans. By the Characterization Lemma (5.23) each b_k defines a finite union of effective disjoint algebraic intervals.

By the Disjointedness Lemma (5.26), the sequence (b_k) is semantically disjoint over $\mathcal{R}$, and hence the unions of algebraic intervals defined by different b_k's are disjoint.

□

Lemma 5.37. *If a set $R \subseteq \mathbb{R}$ is **While**^{OR} semicomputable over $\mathcal{R}$, then R can be expressed as the countable union of an effective sequence of disjoint algebraic intervals.*

Proof. Like the proof of Lemma 5.36, except that we use Corollary 4.15 (Engeler's Lemma for ***While***$^{\mathsf{OR}}$) in place of Lemma 4.13 (Engeler's Lemma for ***While***), and notice that the Characterization Lemma (5.23) also applies to Σ^{OR}-booleans.

□

Lemma 5.38. *If a set $R \subseteq \mathbb{R}$ is **While**^{OR} semicomputable over $\mathcal{R}$, then R can be expressed as the countable union of an effective sequence of rational intervals.*

Proof. From Lemmas 5.37 and 5.13.

□

Remarks 5.39.

(*a*) We lose disjointedness here for rational intervals (compared to Lemma 5.37), because the the sequence of rational intervals generated by Lemma 5.13 are not disjoint.

(*b*) We could also have proved this lemma as an immediate consequences of Lemma 5.41 below, in which case we lose the disjointedness of the rationals because of the failure of semantic disjointedness of the boolean sequence for the hypertrees for ***While***$^{\exists\mathsf{N}}$ computation. (See Remark 5.27.)

Lemma 5.40. *If a set $R \subseteq \mathbb{R}$ is* ***While***$^{\exists N}$ *semicomputable over $\mathcal{R}$, then R can be expressed as the countable union of an effective sequence of algebraic intervals.*

Proof. By Engeler's Lemma for ***While***$^{\exists N}$ (4.17), a ***While***$^{\exists N}$ semicomputable set over $\mathcal{R}$ can be expressed as a countable disjunction of Σ^{OR}-booleans, to which the Characterization Lemma 5.23 still applies.

□

Lemma 5.41. *If a set $R \subseteq \mathbb{R}$ is* ***While***$^{\exists N}$ *semicomputable over $\mathcal{R}$, then R can be expressed as the countable union of an effective sequence of rational intervals.*

Proof. By Lemmas 5.40 and 5.13.

□

Remark 5.42. Note again the lack of disjointedness of the algebraic and rational sequences obtained by Lemma 5.40 and 5.41 for the ***While***$^{\exists N}$ computation hypertree.

5.5 Unions of effective sequences of intervals are semicomputable

In this section, we will prove the reverse direction of the structure theorems.

Lemma 5.43. *The countable union of an effective sequence of disjoint rational intervals is* ***While*** *semicomputable over $\mathcal{R}$.*

Proof. An effective sequence of rational intervals gives us a total recursive function $f : \mathbb{N} \to \mathbb{N}$ such that $f(n)$ is the Gödel number of the n^{th} rational interval. So the countable union of an effective sequence of disjoint rational intervals is equivalent to the halting set of the procedure

```
proc
in  r : real;
aux i : nat;
begin
  i := 0;
  while not(In_Q(P_f(i), r))
  do i := i + 1 od
end
```

where P_f is the ***While***$(\mathcal{N})$ (and hence ***While***$(\mathcal{R})$) procedure which computes f.

By Lemma 5.34, In_Q is ***While*** computable, and so the above procedure is ***While*** computable.

□

Remark 5.44. This result is related to Lemma 5.28, which states that a disjunction of an effective sequence of semantically disjoint booleans can be evaluated "from the left", i.e., by a 'while' loop.

Lemma 5.45. *The countable union of an effective sequence of disjoint algebraic intervals is* ***While***$^{\mathsf{OR}}$ *semicomputable over* $\mathcal{R}$.

Proof. Exactly the same as previous Lemma, but instead of $\mathsf{In}_\mathsf{Q}(f(\mathtt{i}), r)$, we must use $\mathsf{In}_\mathsf{A}(P_f(\mathtt{i}), \mathtt{r})$ which is ***While***$^{\mathsf{OR}}$ computable, by Lemma 5.35.

□

Lemma 5.46. *The countable union of an effective sequence of algebraic intervals is* ***While***$^{\exists N}$ *semicomputable over* $\mathcal{R}$.

Proof. An effective sequence of algebraic intervals gives us a total ***While*** computable function $f : \mathbb{N} \to \mathbb{N}$ such that $f(n)$ returns the Gödel number of the n^{th} algebraic interval.

Further by Lemma 5.35, there is a ***While***$^{\mathsf{OR}}$ computable function $\mathsf{In_A}$ such that $\forall x \in \mathbb{R}$:

$$\mathsf{In_A}(\ulcorner(\alpha,\ \beta)\urcorner, x) \iff x \in (\alpha,\ \beta)$$

By Lemma 3.23, $\mathsf{In_A}$ is ***While***$^{\exists \mathsf{N}}$ computable.

So the countable union of an effective sequence of algebraic intervals is equivalent to the halting set of the following ***While***$^{\exists \mathsf{N}}$ procedure:

```
proc
in   a : real;
out b : bool;
begin
  b:=  Exist z : $P$(a, z)
end
```

where P is the procedure defined as:

```
proc
in    a : real;
      z : nat;
out b : bool;
begin
   b:= In_A(P_f(z),a);
end
```

where P_f : nat $\rightharpoonup$ nat is the ***While***$(\mathcal{N})$ (and hence ***While***$(\mathcal{R})$) procedure which computes f.

$\square$

Lemma 5.47. *The countable union of an effective sequence of rational intervals is* ***While***$^{\exists N}$ *semicomputable over* $\mathcal{R}$.

Proof. From Lemma 5.46 (since by Lemma 5.11, a rational interval is an algebraic interval).

$\square$

Discussion 5.48. When a sequence of rational or algebraic intervals is disjoint, we can represent their union as the halting set of a ***While***$^{(\mathsf{OR})}$ procedure (as in Lemmas 5.36 and 5.37), since it can be evaluated from the left, i.e., by a 'while' loop.

However when the intervals are not disjoint, their union must be evaluated by a ***While***$^{\exists \mathsf{N}}$ procedure, using the 'Exist' construct (as in Lemma 5.46).

5.6 Structure theorems for semicomputable sets over $\mathcal{R}$

We conclude with our three structure theorems for $\boldsymbol{While}^{\mathsf{OR}}$ and $\boldsymbol{While}^{\exists\mathsf{N}}$ semicomputable sets over $\mathcal{R}$.

Theorem 5.49. *A subset of $\mathbb{R}$ is $\boldsymbol{While}^{\mathsf{OR}}$ semicomputable over $\mathcal{R}$ if and only if it can be expressed as a countable union of an effective sequence of disjoint algebraic intervals.*

Proof. By Lemmas 5.45 and 5.37.

□

Theorem 5.50. *A subset of $\mathbb{R}$ is $\boldsymbol{While}^{\exists\mathsf{N}}$ semicomputable over $\mathcal{R}$ if and only if it can be expressed as a countable union of an effective sequence of algebraic intervals.*

Proof. By Lemmas 5.46 and 5.40.

□

Theorem 5.51. *A subset of $\mathbb{R}$ is $\boldsymbol{While}^{\exists\mathsf{N}}$ semicomputable over $\mathcal{R}$ if and only if it can be expressed as a countable union of an effective sequence of rational intervals.*

Proof. By Lemmas 5.47 and 5.41.

□

We do not have a structure theorem for ***While*** semicomputable sets. We do however, have a partial result:

Theorem 5.52. *For subsets of* $\mathbb{R}$*,*

(a) ***While*** *semicomputable over* $\mathcal{R}$ $\Longrightarrow$

union of effective sequence of rational intervals.

(b) *union of effective sequence of disjoint rational intervals* $\Longrightarrow$

While *semicomputable over* $\mathcal{R}$*.*

Proof. (a) By Lemma 5.38.

(b) By Lemma 5.43.

Remark 5.53. See Remarks 5.39 and 5.44, for the reasons that disjointedness is lost in part (a), but needed in part (b).

$\square$

5.7 Projectively *While*$^{\exists \mathsf{N}}$ semicomputable sets

We now prove that for the ***While***$^{\exists \mathsf{N}}$ language, projectively semicomputability is equivalent to semicomputability, i.e., semicomputability is closed under projection over $\mathbb{R}$.

Lemma 5.54. *For a continuous partial function* $b : \mathbb{R}^n \rightharpoonup \mathbb{B}$*, if there exists an* n*-tuple of reals* $x = (x_1, ..., x_n)$ *such that* $b(x) \downarrow \mathtt{tt}$*, then there exists an* n*-tuple of rationals* $r = (r_1, ..., r_n)$ *such that* $b(r) \downarrow \mathtt{tt}$.

Proof. For a continuous function $b : \mathbb{R}^n \rightharpoonup \mathbb{B}$, suppose that there exists a real tuple $x = (x_1, ..., x_m) \in \mathbb{R}^n$ such that $b(x) \downarrow \mathtt{tt}$. Then there exists $\delta > 0$ such that for all real tuples $y = (y_1, ..., y_n)$ in the set:

$$\mathsf{N}(x, \delta) =_{df} \{(y_1, ..., y_n) | \sqrt{(x_1 - y_1)^2 + ... + (x_n - y_n)^2} < \delta\}$$

$b(y) \downarrow \mathtt{tt}$. Because of the density of $\mathbb{Q}$ in $\mathbb{R}$, there exists a rational tuple $r = (r_1, ..., r_n) \in \mathsf{N}(x, \delta)$.

$\square$

Theorem 5.55. *A set $R \subseteq \mathbb{R}^n$ is **While**$^{\exists N}$ projectively semicomputable over $\mathcal{R}$ if and only if R is **While**$^{\exists N}$ semicomputable over $\mathcal{R}$.*

Proof. "$\Leftarrow$": Trivial

"$\Rightarrow$": Suppose R is projectively semicomputable. Then there exists another relation $R' \subset \mathbb{R}^{m+n}$, such that for all $x \in \mathbb{R}$:

$$\begin{aligned} x \in R \quad &\Longleftrightarrow \quad \exists y \in \mathbb{R}^m (x, y) \in R' \\ &\Longleftrightarrow \quad \exists y \in \mathbb{R}^m \bigvee_k b_k[x, y] \\ &\qquad\quad \text{for some effective sequence of } \Sigma\text{-booleans } (b_k) \\ &\qquad\quad \text{by Engeler's Lemma 4.13 for } \textbf{\textit{While}}, \text{ applied to } R', \\ &\Longleftrightarrow \quad \bigvee_k \exists y \in \mathbb{R}^m b_k[x, y] \\ &\Longleftrightarrow \quad \bigvee_k \exists r \in \mathbb{Q}^m b_k[x, r], \text{ by Lemma 5.54} \end{aligned}$$

It is not hard to see that we can construct an effective double sequence $(b_{k,l})$ of booleans, such that for all k, l, if $l = \ulcorner k \urcorner$ then

$$b_{k,l}[x] \quad \Longleftrightarrow \quad b_k[x, r]$$

and:

$$x \in R \quad \Longleftrightarrow \quad \bigvee_k \bigvee_l b_{k,l}[x]. \tag{5.2}$$

Finally, by a method similar to the proof of Lemma 5.46, we can show that the right hand side of (5.2) is the halting set of a ***While***$^{\exists \mathsf{N}}$ procedure.

□

Essentially, the proof involves replacing existential quantification over $\mathbb{R}$ by existential quantification over $\mathbb{Q}$ (using continuity and density of $\mathbb{Q}$ in $\mathbb{R}$), and then replacing the latter by countable disjunctions.

Remarks 5.56.

(a) We do not know if this result holds for ***While*** or ***While***$^{\mathsf{OR}}$.

(b) In total (non-topological) algebras $\mathcal{R}_t$ over the reals, the continuity argument in the above proof would not work, and in fact, the theorem fails! A counterexample is given in [TZ00, §6.2].

Chapter 6

Conclusion and future work

6.1 Conclusions

In this thesis we investigated computability, or rather semicomputability, for the ***While*** language and certain extensions (***While***$^{\mathsf{OR}}$ and ***While***$^{\exists\mathsf{N}}$) over a topological partial algebra $\mathcal{R}$ on the reals. We proved Structure Theorems for semicomputable sets in $\mathcal{R}$:

(1) A subset of $\mathbb{R}$ is ***While***$^{\mathsf{OR}}$ semicomputable over $\mathcal{R}$ $\iff$

> it can be expressed as a countable union of an effective sequence of *pairwise disjoint* algebraic intervals.

(2) A subset of $\mathbb{R}$ is ***While***$^{\exists\mathsf{N}}$ semicomputable over $\mathcal{R}$ $\iff$

> it can be expressed as a countable union of an effective sequence of algebraic intervals.

(3) A subset of $\mathbb{R}$ is $\boldsymbol{While}^{\exists\mathsf{N}}$ semicomputable over $\mathcal{R}$ $\Longleftrightarrow$

it can be expressed as a countable union of an effective sequence of rational intervals.

We have no structure theorem for the ***While*** language, only the partial result.

(4) For subsets of $\mathbb{R}$,

(a) ***While*** semicomputable over $\mathcal{R}$

$\Longrightarrow$ union of effective sequence of rational intervals.

(b) union of effective sequence of *pairwise disjoint* rational intervals

$\Longrightarrow$ ***While*** semicomputable over $\mathcal{R}$.

And finally, we showed:

(5) $\boldsymbol{While}^{\exists\mathsf{N}}$ semicomputability over $\mathcal{R}$ is closed under projection.

6.2 Future work and conjectures

We list some future work in this area, and conjectures:

(1) Investigating the structure of semicomputable subsets of $\mathbb{R}^n$ for $n > 1$.

Although the Canonical Form Lemma (5.21) still holds, the Characterization Lemma (5.23) does not, and so this is a harder problem. Note (again) that Engeler's Lemma holds for all standard partial algebras.

In searching for a suitable characterization lemma for the multi-dimensional case, Strum's Theorem, is no longer helpful. A more useful approach may be the method of cylindrical algebraic decomposition, applied to semialgebraic sets. [1]

(2) To expand $\mathcal{R}$ by including division by naturals:

$$\mathsf{div}^{\mathsf{N}}(x,n) \equiv_{df} \frac{x}{n+1}$$

where $x \in \mathbb{R}$ and $n \in \mathbb{N}$. This is clearly a total function.

This seems fairly straightforward. With this expansion, we can directly embed $\mathbb{Q}$ in $\mathbb{R}$. This makes the proofs of several theorems, such as Theorem 5.55, easier. The Canonical Form and Characterization Lemmas still hold, as do the Structure Theorems.

Thus it seems clear that the five structure theorems listed in §6.1 still hold for the algebra $\mathcal{R}+\mathsf{div}^{\mathsf{N}}$.

(3) To expand $\mathcal{R}$ to an algebra $\mathcal{R}^{\mathsf{div}}$, which includes the partial division operation:

$$\mathsf{div}(x,y) \equiv_{df} \frac{x}{y}$$

where $x, y \in \mathbb{R}$.

This expansion is a major step compared to (2). The Canonical Form Lemma now becomes: a boolean over $\mathcal{R}^{\mathsf{div}}$ can be expressed as a finite disjunction of finite conjunctions of *rational function inequalities* $r(x) > 0$ where:

$$r(x) = \frac{p(x)}{q(x)} \tag{6.1}$$

[1] We are grateful to Dr. Carette for this insight.

with $p(x), q(x) \in \mathbb{Z}[x]$.

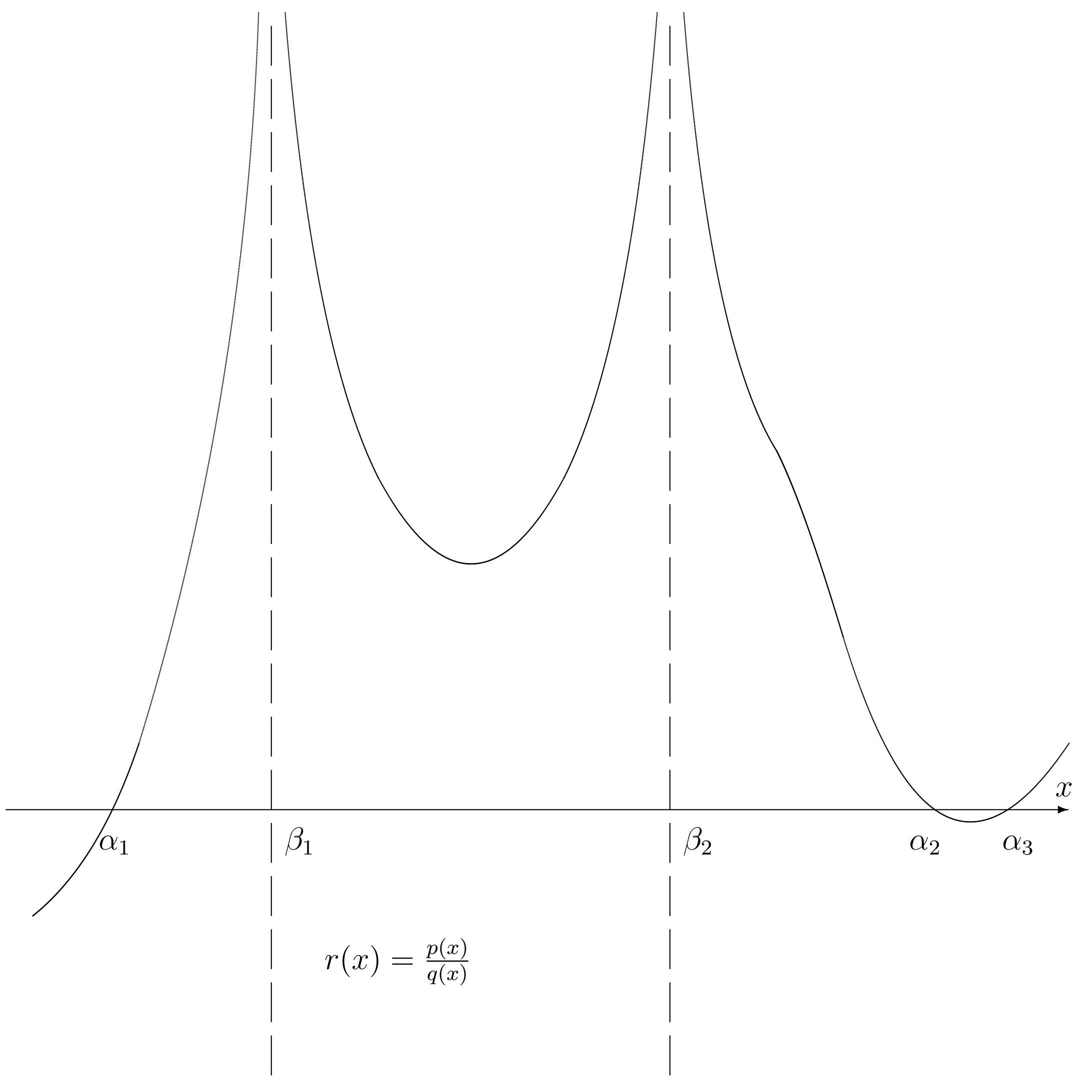


Figure 7

The important thing to notice is that the zeros and poles of rational functions are algebraic numbers, since for equation 6.1, the zeros and poles of $r(x)$ are respectively the roots of $p(x)$ and $q(x)$.

Thus the graph of $y = r(x)$ has a form like in Figure 7.

with zeros at $\alpha_1,\ \alpha_2,\ \alpha_3$ and poles at $\beta_1,\ \beta_2$.

From this it can be seen that the Characterization Lemma still holds for $\mathcal{R}^{\mathsf{div}}$. In fact:

$$\frac{p(x)}{q(x)} > 0 \quad \Longleftrightarrow \quad (p(x) \times q(x)) > 0$$

where "$\Longleftrightarrow$" is weak semantic equivalence.

We conjecture that the three Structure Theorems in §6.1 still hold for $\mathcal{R}^{\mathsf{div}}$.

(4) To bridge the gap between an abstract model (e.g. $\boldsymbol{While}^{\exists\mathsf{N}}$) and a concrete model of computation over $\mathbb{R}$ (e.g. Weihrauch's TTE [Wei00])

We have proved (Structure Theorem (3) in §6.1) that for a relation R on $\mathbb{R}$:

$$R \text{ is } \boldsymbol{While}^{\exists\mathsf{N}} \text{ semicomputable in } \mathcal{R} \quad \Longleftrightarrow \quad R = \bigcup_k I_k \tag{6.2}$$

where I_k is an effective sequence of rational intervals. On the other hand, Weihrauch has shown [Wei00] that for his concrete model:

$$R \text{ is TTE-semicomputable} \quad \Longleftrightarrow \quad R = \bigcap_j \bigcup_k I_{j,k} \tag{6.3}$$

where $(I_{j,k})$ is an effective double sequence of rational intervals.

We can try to bridge the gap between (6.2) and (6.3) by generalizing the notion of semicomputability in $\mathcal{R}$ to that of *approximable* $\boldsymbol{While}^{\exists\mathsf{N}}$ *semicomputability*, where a set $R \subseteq \mathbb{R}^n$ is said to be approximably $\boldsymbol{While}^{\exists\mathsf{N}}$ semicomputable if for some $\boldsymbol{While}^{\exists\mathsf{N}}$

procedure $P : \mathsf{nat} \times \mathsf{real} \rightharpoonup \mathsf{bool}$, we have, writing $P_n^{\mathcal{R}}(x) \ =_{df} \ P^{\mathcal{R}}(n,x)$:

$$R = \bigcap_n \boldsymbol{Halt}^{\mathcal{R}}(P_n^{\mathcal{R}}).$$

We then conjecture that for a set $R \subseteq \mathbb{R}^n$:

$$R \text{ is approximably } \boldsymbol{While}^{\exists\mathsf{N}} \text{ semicomputable} \quad \Longleftrightarrow \quad R \text{ is TTE semicomputable.}$$

The motivation for this conjecture, and the reason for the terminology "approximably semicomputable", is by analogy with the "completeness theorem" in [TZ04a], where for partial topological algebras A (including $\mathcal{R}$) satisfying certain general conditions, it was proved that

$$\boldsymbol{WhileCC} \text{ approximable computability} \quad \Longleftrightarrow \quad \text{concrete computability on } A.$$

Here ***WhileCC*** is the ***While*** language extended by a nondeterministic "countable choice" operator, and a function $f : A^u \rightharpoonup A^v$ is said to be approximably ***WhileCC*** computable if for some ***WhileCC*** procedure

$$P : \mathsf{nat} \times u \to v$$

the sequence of (many-valued) functions

$$P_n^A : A^u \rightharpoonup A^v$$

converges or *approximates* to f (in a suitable sense).

In other words, for abstract computability to correspond to concrete computability, it must be augmented by

(a) a non-deterministic choice operator 'choose' on $\mathbb{N}$,

(b) approximability of computations.

Similarly, in the present case, we conjecture that for abstract semicomputability to correspond to concrete semicomputability, it must be augmented by

(a) the 'Exist' operator on $\mathbb{N}$,

(b) approximability, which here means countable intersection.

Note that our 'Exist' operator can be viewed as a (weakly deterministic) special case of the 'choose' operator. In fact it can be clearly defined by 'choose':

$$\mathtt{x}^{\mathtt{B}} := \ \mathsf{Exist}\ \mathtt{z} : P(t,\mathtt{z}) \quad \Longleftrightarrow \quad \mathtt{n} := \mathsf{choose}\ \mathtt{z} : P(t,\mathtt{z})\ ;\ \mathtt{x}^{\mathtt{B}} := P(t,\mathtt{n}).$$

(See Remarks 3.22 (b, c))

Bibliography

[BCSS98] Lenore Blum, Felipe Cucker, Michael Shub, and Steve Smale. *Complexity and Real Computation.* Springer-Verlag, 1998.

[Eng68] E. Engeler. *Formal Languages: Automata and Structures.* Markham Publishing Co, 1968.

[Kle71] Stephen Cole Kleene. *Introduction to metamathematics.* Wolters-Noordhoff, 1971.

[Lan90] Serge Lang. *Undergraduate Algebra.* Elementary Mathematics. Springer-Verlag, 1990.

[PER89] B. Pour-El and Jonathan I. Richards. *Computability in Analysis and Physics.* Springer-Verlag, 1989.

[Roy66] H. L. Royden. *Real Analysis.* The Macmillan Company, sixth edition, 1966.

[Rud76] Water Rudin. *Principle of Mathematical Analysis.* McGraw-Hill Inc., New York, third edition, 1976.

[SHT99] V. Stoltenberg-Hansen and J.V. Tucker. Concrete models of computation for topological algebras. *Theoretical Computer Science*, 219:347–378, 1999.

[TZ99] J.V. Tucker and J.I. Zucker. Computation by ‘while’ programs on topological partial algebras. *Theoretical Computer Science, 219*, pages 379–420, 1999.

[TZ00] J.V. Tucker and J.I. Zucker. *Computable functions and semicomputable sets on many-sorted algebras*, volume 5 of *Handbook of Logic in Computer Science*, section 1.2, pages 317–523. Oxford University Press, 2000.

[TZ04a] J.V. Tucker and J.I. Zucker. Abstract versus concrete computation on metric partial algebras. *ACM Transactions on Computational Logic*, 2004. To appear.

[TZ04b] J.V. Tucker and J.I. Zucker. Computable total functions on metric algebras, universal algebraic specifications and dynamical systems. *Journal of Logical and Algebraic Programming*, 2004. To appear.

[Wae64] B.L. Van Der Waerden. *Modern Algebra*, volume 1. Frederick Ungar Publishing Co., New York, second edition, 1964.

[Wei00] Klaus Weihrauch. *Computable Analysis, An Introduction.* Springer-Verlag, 2000.

[ZP93] J.I. Zucker and L. Pretorius. Introduction to computability theory. *South African Computer Journal*, 9:3–30, 1993.