Universality and Semicomputability for Nondeterministic Programming Languages over Abstract Algebras

Wei Jiang^{*}, Yuan Wang[†], and Jeffery Zucker[‡]

September 11, 2006

Abstract

The Universal Function Theorem (UFT) originated in 1930s with the work of Alan Turing, who proved the existence of a universal Turing machine for computations on strings over a finite alphabet. This stimulated the development of stored-program computers.

Classical computability theory, including the UFT and the theory of semicomputable sets, has been extended by Tucker and Zucker to abstract manysorted algebras, with algorithms formalized as deterministic *While* programs.

This paper investigates the extension of this work to the nondeterministic programming languages $While^{RA}$ consisting of While programs extended by random assignments, as well as sublanguages of $While^{RA}$ formed by restricting the random assignments to booleans or naturals only. It also investigates the nondeterministic language GC of guarded commands. There are two topics of investigation: (1) the extent to which the UFT holds over abstract algebras in these languages; (2) concepts of semicomputability for these languages, and the extent to which they coincide with semicomputability for the deterministic While language.

Key words and phrases: many-sorted algebras, computation on abstract data types, abstract computability, random assignments, guarded commands, nondeterminism.

^{*}School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada †IBM Canada, 820 Warden Ave., Markham, Ontario L6G 1C7, Canada

[‡]Dept of Computing and Software, McMaster University, Hamilton, Ontario L8S 4K1, Canada

1 Introduction

1.1 Nondeterministic languages

Computability theory over abstract algebras can be developed in many directions and can be applied in many areas [8]. In this paper we will emphasize computations of nondeterministic programs on abstract many-sorted algebras.

To compute on such algebras, a simple deterministic model based on the *While* language was introduced in [8], in which basic computations are performed by concurrent assignments, and control and sequencing by the three constructs: sequential composition, conditional, and iteration.

In this paper, we will study constructs for nondeterministic algorithms. We will consider two nondeterministic computation models. The main one that we will study is the $While^{RA}$ programming language, which extends the While language with random assignments

 $\mathbf{x} := ?$

The other nondeterministic computational model that we will study is the Guarded Command Language (GC) due to Dijkstra [1]. The "guarded command" has the form $b \to S$, where b is a boolean test and S is a statement. The constructs of GC includes: the guarded command conditional

if
$$b_1 \rightarrow S_1 \mid \ldots \mid b_k \rightarrow S_k$$
 fi

and the guarded command iteration

do
$$b_1 \rightarrow S_1 \mid \ldots \mid b_k \rightarrow S_k$$
 od

 $(k \ge 0)$, together with concurrent assignment and sequential composition.

1.2 Universal functions

The idea of universal computable functions originated in 1930s with the work of Turing [11], who proved the existence of a universal Turing machine for computations on strings over a finite alphabet. This concept helped to stimulate the development of stored-program computers.

The Universal Function Theorem (UFT) was extended to abstract many-sorted algebras with *deterministic* algorithms formalized as *While* programs in [8]. Let A

be an algebra with booleans and naturals. The **While** procedures P_0, P_1, P_2, \ldots of a fixed type $u \to v$ are effectively coded by natural numbers. Consider the "universal" enumerating function

$$\mathsf{Univ}^A: \mathbb{N} \times A^u \to A^v$$

defined by

$$\mathsf{Univ}^A(n,x) = P_n(x)$$

for any $x \in A^u$. It has been proved [8] that $Univ^A$ is *While* computable on A assuming a computable "term evaluation function".

In this paper we will examine universality for the $While^{RA}$ and GC languages over many-sorted algebras. GC is shown to be equivalent to $While^{RA(bool)}$, i.e., Whileprograms with random assignments restricted to the sort bool (Theorem 3.9.4), so we will concentrate on $While^{RA}$. We consider two cases: (1) $While^{RA(nat/bool)}$, where random assignments are restricted to the sorts nat and/or bool; and (2) the general case, with unrestricted random assignments.

These two cases require quite different techniques: (1) For $While^{RA(nat/bool)}$, we prove the UFT (assuming the same term evaluation property) by using *locality* of computation, which means that the output of any procedure is always in the subalgebra generated by the input (Theorem 5.4.3). (2) With unrestricted random assignments, locality of computation no longer holds, so here we use another technique, based on coding arbitrary many auxiliary variables by a fixed number of arrays. Thus we can also prove the UFT for $While^{RA}$ with unrestricted random assignments on array algebras (Theorem 5.5.3).

1.3 Nondeterministic semicomputability

The notion of *recursive enumerability* or *semicomputability* was generalized to manysorted algebras in [8]. In deterministic programming languages, a set is semicomputable if, and only if, it is the halting set of a procedure.

In this paper, we generalize this definition to nondeterministic languages, and investigate the equivalence of (a suitable notion of) semicomputability with the deterministic case. This constitutes the second part of this paper.

Again, we consider two cases: (1) $While^{RA(nat/bool)}$ semicomputability: this is found to be equivalent to $While^{N}$ semicomputability, i.e., While semicomputability with auxiliary counters (nat variables) (Theorem 6.3.3 and 6.4.1); (2) unrestricted $While^{RA}$ semicomputability: this is, in general, not equivalent to While (or $While^{N}$) semicomputability. A counterexample is again found (Theorem 6.5.5) by considering array algebras, in which $While^{RA}$ semicomputability is shown to be equivalent to projective While semicomputability, which is, in general, not equivalent to While semicomputability [8].

1.4 Overview of the sections

This paper is divided into seven sections. Section 1 is this introduction. Section 2 presents the basic algebraic notions we will need. Section 3 presents the syntax and semantics of the nondeterministic programming languages $While^{RA}$ and GC, and proves the equivalence of GC and $While^{RA(bool)}$. In Section 4, we represent or code the syntax and semantics of $While^{RA}$ computations in the algebra itself. In Section 5, we will explore the existence of a universal function for $While^{RA}$, and its sublanguages $While^{RA(nat/bool)}$. In Section 6, we investigate concepts of nondeterministic semicomputability, and see to what extent it coincides with semicomputability for the deterministic While language. Section 7 draws conclusions and lists some open problems for future work.

This paper developed from Master's theses of two of the authors [3, 12]. It is part of an ongoing research program in computation theory on many-sorted abstract algebras [7, 8, 9, 10].

1.5 Acknowledgments

This research was supported by a grant from the Natural Sciences and Engineering Research Council of Canada. We thank an anonymous referee for helpful suggestions.

2 Basic concepts

In this section, we give a brief introduction to basic algebraic concepts and notations. This section follows closely the treatment in [8].

2.1 Signatures

Definition 2.1.1 (Many-sorted signatures). A many-sorted signature Σ consists of (1) a finite set $Sort(\Sigma)$ of sorts s, \ldots , and (2) a finite set $Func(\Sigma)$ of (primitive or basic) function symbols $F: s_1 \times \ldots \times s_m \to s$ ($m \ge 0$), with $s_1, \ldots, s_m, s \in Sort(\Sigma)$. The case m = 0 corresponds to constant symbols; we then write $F: \to s$ or F: s.

Definition 2.1.2 (Product types over Σ). A $(\Sigma$ -)product type is a symbol of the form $s_1 \times \cdots \times s_m$ $(m \ge 0)$, where s_1, \ldots, s_m are Σ -sorts. We define $ProdType(\Sigma)$ to be the set of Σ -product types, denoted as u, v, w, \ldots .

Definition 2.1.3 (Σ -algebras).

A Σ -algebra A has, for each sort s of Σ , a non-empty set A_s , called the *carrier of* sort s, and for each Σ -function symbol $F : s_1 \times \cdots \times s_m \to s$, a total¹ function $F^A : A_{s_1} \times \cdots \times A_{s_m} \to A_s$. For m = 0, this gives an element $F^A \in A_s$.

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

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

So each Σ -function symbol $F: u \to s$ has an interpretation $F^A: A^u \to A_s$.

Example 2.1.4 (Signature and algebra of booleans). The signature of booleans can be defined as

```
\begin{array}{ll} \mbox{signature} & \varSigma(\mathcal{B}) \\ \mbox{sorts} & \mbox{bool} \\ \mbox{functions} & \mbox{true, false}: & \rightarrow \mbox{bool}, \\ & \mbox{and, or}: \mbox{bool}^2 \rightarrow \mbox{bool} \\ & \mbox{not}: \mbox{bool} \rightarrow \mbox{bool} \\ \mbox{end} \end{array}
```

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

Definition 2.1.5 (Function types). Let A be a Σ -algebra.

¹See Section 7, item (4).

- (a) A function type over Σ , or Σ -function type, is a symbol of the form $u \to v$, with domain type u and range type v, where u and v are Σ -product types.
- (b) For any Σ -function type $u \to v$, a function of type $u \to v$ over A is a (not necessarily total) function $f : A^u \to A^v$.

We use the following notation: if $f : A^u \to A_s$ and $x \in A^u$, then $f(x) \uparrow$ ("f(x) diverges") means that $x \notin dom(f)$; $f(x) \downarrow$ ("f(x) converges") means that $x \in dom(f)$; and $f(x) \downarrow y$ ("f(x) converges to y") means that $x \in dom(f)$ and f(x) = y.

Definition 2.1.6 (Relations; projections of relations). A relation on A of type u is a subset of A^u . We write R : u if R is a relation of type u.

Suppose R: u where $u = s_1 \times s_2 \times s_3 \times s_4 \times s_5$. Now let $v = s_1 \times s_2 \times s_3$ and $w = s_4 \times s_5$. Then the projection of R on v (or on A^v), or the projection of R off w (or off A^w), or the A^w -projection of R, is the relation S: v defined by existentially quantifying over A^w :

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

Definition 2.1.7 (Generated subalgebras). Let $X \subseteq \bigcup_{s \in Sort(\Sigma)} A_s$. Then $\langle X \rangle^A$ is the $(\Sigma$ -)subalgebra of A generated by X, *i.e.*, the smallest subalgebra of A which contains X, and $\langle X \rangle_s^A$ is the carrier of $\langle X \rangle^A$ of sort s.

Definition 2.1.8 (Closed terms over Σ). We define the class $T(\Sigma)$ of closed terms over Σ , and for each Σ -sort s, the class $T(\Sigma)_s$ of closed terms of sort s. These are generated inductively by the rule:

If $F: u \to s$ is in $Func(\Sigma)$ and $t_i \in T(\Sigma)_{s_i}$ for i = 1, ..., m, where $u = s_1 \times \cdots \times s_m$, then $F(t_1, \ldots, t_m) \in T(\Sigma)_s$. For m = 0, this corresponds to a constant F(), written F.

An important assumption we make throughout this paper is:

Assumption 2.1.9 (Instantiation). $T(\Sigma)_s$ is non-empty for each $s \in Sort(\Sigma)$.

Definition 2.1.10 (Default terms; Default values). (a) For each sort s, we pick a closed term of sort s. (There is at least one, by the Instantiation Assumption.)

We call this the *default term of sort s*, written δ^s . Further, for each product type $u = s_1 \times \cdots \times s_m$ of Σ , the *default (term) tuple of type u*, written δ^u , is the tuple of default terms $(\delta^{s_1}, \ldots, \delta^{s_m})$.

(b) Given a Σ -algebra A, for any sort s, the default (value) of sort s in A is the valuation $\delta_A^s \in A_s$ of the default term δ^s ; 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 $\delta_A^u = (\delta_A^{s_1}, \ldots, \delta_A^{s_m}) \in A^u$.

Definition 2.1.11 (Minimal carriers; minimal algebra). Let A be a Σ -algebra, and s a Σ -sort.

- (a) A is minimal at s (or the carrier A_s is minimal in A) if $A_s = \langle \emptyset \rangle_s^A$, *i.e.*, A_s is generated by the closed Σ -terms of sort s.
- (b) A is minimal if it is minimal at every Σ -sort.

2.2 Adding booleans: Standard signatures and algebras

- **Definition 2.2.1** (Standard signatures and algebras). (a) A signature Σ is standard if (i) $\Sigma(\mathcal{B}) \subseteq \Sigma$, and (ii) the function symbols of Σ include an equality operator $eq_s : s^2 \to bool$ for certain sorts s, called equality sorts.
 - (b) Given a standard signature Σ , a Σ -algebra A is standard if the carrier A_{bool} is the set of truth values $\mathbb{B} = \{\mathfrak{t}, \mathfrak{f}\}$, the standard boolean operations have their standard interpretations, and the equality operator eq_s is interpreted as *identity* on each equality sort s.

Let $StdAlg(\Sigma)$ denote the class of standard Σ -algebras.

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

Throughout this paper, we will assume:

Assumption 2.2.2 (Standardness). The signature Σ and the Σ -algebra A are standard.

2.3 Adding counters: N-standard signatures and algebras

Definition 2.3.1 (N-Standard signatures and algebras). (a) A standard signature Σ is called *N*-standard if it includes (as well as **bool**) the numerical sort **nat**, and also function symbols for the standard arithmetic operations of zero, successor, equality and order on the naturals:

$$\begin{array}{rll} 0: & \to \mathsf{nat} \\ \mathsf{S}:\mathsf{nat} & \to \mathsf{nat} \\ \mathsf{eq}_\mathsf{nat}, \mathsf{less}_\mathsf{nat}:\mathsf{nat}^2 & \to \mathsf{bool}. \end{array}$$

as well as the *equality operator* eq_{nat} on nat.

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

Note that any standard Σ -algebra A can be N-standardised to a Σ^N -algebra A^N by adjoining the carrier \mathbb{N} together with the standard arithmetic operations.

Let N-StdAlg (Σ) denote the class of N-standard Σ -algebras.

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

Definition 2.4.1 (Signature Σ^* and Algebras A^*). Given a standard signature Σ , and standard Σ -algebra A, we extend Σ to Σ^* , and expand A to A^* in two stages: first, N-standardise Σ and A to form Σ^N and A^N ; then define, for each sort s of Σ , the carrier A_s^* to be the set of finite sequences or arrays a^* over A_s , of "starred sort" s^* . The resulting algebras A^* have signature Σ^* , which extends Σ^N by including, for each sort s of Σ , the new starred sorts s^* , and also the following new function symbols:

(*i*)the operator $\mathsf{Lgth}_s: s^* \to \mathsf{nat}$, where $\mathsf{Lgth}(a^*)$ is the length of the array a^* ;

(*ii*) the application operator $Ap_s : s^* \times nat \to s$, where

$$\mathsf{Ap}_{s}^{A}(a^{*},k) = \begin{cases} a^{*}[k] & \text{if } k < \mathsf{Lgth}(a^{*}), \\ \boldsymbol{\delta}^{s} & \text{otherwise,} \end{cases}$$

where $\boldsymbol{\delta}^{s}$ is the default value at sort s;

- (*iii*) the null array $\text{Null}_s : s^*$ of zero length;
- (*iv*) the operator $\mathsf{Update}_s : s^* \times \mathsf{nat} \times s \to s^*$, where $\mathsf{Update}_s^A(a^*, n, x)$ is the array $b^* \in A_s^*$ such that for all $k \in \mathbb{N}$,

$$b^*[k] = \begin{cases} a^*[k] & \text{if } k < \mathsf{Lgth}(a^*), k \neq n, \\ x & \text{if } k < \mathsf{Lgth}(a^*), k = n, \\ \delta^s & \text{otherwise}; \end{cases}$$

(v) the operator Newlength_s: $s^* \times nat \to s^*$, where Newlength_s^A(a^*, m) is the array b^* of length m such that for all k < m,

$$b^*[k] = \begin{cases} a^*[k] & \text{if } k < \mathsf{Lgth}(a^*), \\ \boldsymbol{\delta}^s & \text{otherwise;} \end{cases}$$

(vi) the equality operator on A_s^* for each equality sort s.

The significance of arrays or "starred variables" for computation is that they provide finite but unbounded memory. The reason for introducing them is the lack of effective coding of finite sequences in abstract algebras in general, in contrast to \mathbb{N} .

3 Nondeterministic languages: $While^{RA}$ and GC

In this section, we will study the two nondeterministic languages, $While^{RA}$ and GC on standard many-sorted algebras. The emphasis will be on $While^{RA}$, since (as we will see) GC is equivalent to $While^{RA(bool)}$.

3.1 Syntax of $While^{RA}$

We begin with the syntax of the language $While^{RA}(\Sigma)$, which is generated by extending $While(\Sigma)$ [8] with the random assignment ' $\mathbf{x} :=$?'.

Definition 3.1.1. $Var(\Sigma)$ is the class of Σ -variables, and $Var_s(\Sigma)$ is the class of variables of sort s.

We write $\mathbf{x} : s$ to mean that $\mathbf{x} \in Var_s(\Sigma)$, and for $u = s_1 \times \cdots \times s_m$, we write $\mathbf{x} : u$ to mean that \mathbf{x} is a *u*-tuple of *distinct variables* of sorts s_1, \ldots, s_m , respectively. We write $VarTup(\Sigma)$ for the class of all tuples of distinct Σ -variables, and $VarTup_u(\Sigma)$ for the class of all *u*-tuples of distinct Σ -variables.

Definition 3.1.2. $Term(\Sigma)$ is the class of Σ -terms t, \ldots , and for each Σ -sort s, $Term_s(\Sigma)$ is the class of terms of sort s. These are defined by:

$$t^s ::= \mathbf{x}^s \mid F(t_1^{s_1}, ..., t_m^{s_m}) \mid \text{if } b \text{ then } t_1^s \text{ else } t_2^s \text{ fi},$$

where $\mathbf{x}^s : s, F : u \to s$ is in $Func(\Sigma)$, and $u = s_1 \times \cdots \times s_m (m \ge 0)$, and $b \in Term_{bool}(\Sigma)$.

We write $TermTup(\Sigma)$ for the class of all tuples of Σ -terms, and, for $u = s_1 \times \cdots \times s_m$, $TermTup_u(\Sigma)$ for the class of *u*-tuples of terms.

We write t:s or t^s to indicate that $t \in Term_s(\Sigma)$. 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 .

Definition 3.1.3. $AtSt(\Sigma)$ is the class of atomic statements S_{at}, \ldots , defined by:

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

where $\mathbf{x} := t$ is a *concurrent assignment* with $\mathbf{x} : u$ and t : u for some product type u, and $\mathbf{x} :=$? is a *random assignment* to a variable \mathbf{x} of some Σ -sort s.

Definition 3.1.4. Stmt(Σ) is the class of statements S, \ldots , generated by:

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

Definition 3.1.5. $Proc(\Sigma)$ is the class of procedures P, Q, \ldots , which have the form

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

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

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

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

If $\mathbf{a} : u$ and $\mathbf{b} : v$, then P has type $u \to v$, written $P : u \to v$. Its input type is u, and its output type is v. We write $\operatorname{Proc}(\Sigma)_{u \to v}$ for the class of Σ -procedures of type $u \to v$.

We often write **Term** for **Term**(Σ), **Proc** for **Proc**(Σ), etc. when the signature Σ is known or not important.

We write ' \equiv ' for syntactic identity.

3.2 States

A state on a Σ -algebra A is a family $\langle \sigma_s | s \in Sort(\Sigma) \rangle$ of functions $\sigma_s : Var_s \to A_s$. Let State(A) be the set of states on A, with elements σ, \ldots . For $\mathbf{x} \in Var_s$, we often write $\sigma(\mathbf{x})$ for $\sigma_s(\mathbf{x})$. Also, for a tuple $\mathbf{x} \equiv (\mathbf{x}_1, \ldots, \mathbf{x}_m)$, we write $\sigma[\mathbf{x}]$ for $(\sigma(\mathbf{x}_1), \ldots, \sigma(\mathbf{x}_m))$.

Definition 3.2.1 (Variant of a state). Let σ be a state over A, $\mathbf{x} \equiv (\mathbf{x}_1, \dots, \mathbf{x}_m) : u$ and $a = (a_1, \dots, a_m) \in A^u$ (for $m \ge 1$). Then $\sigma\{\mathbf{x}/a\}$ is the variant of σ at \mathbf{x} by a, i.e., the state defined by:

$$\sigma\{\mathbf{x}/a\}(\mathbf{y}) = \begin{cases} \sigma(\mathbf{y}) & \text{if } \mathbf{y} \neq \mathbf{x}_i \text{ for } i = 1, \dots, m \\ a_i & \text{if } \mathbf{y} \equiv \mathbf{x}_i. \end{cases}$$

3.3 Semantics of terms

For $t \in Term_s$, we define the function $\llbracket t \rrbracket^A$: $State(A) \to A_s$ where $\llbracket t \rrbracket^A \sigma$ is the value of t in A at state σ . The definition is by structural induction on t:

$$\begin{split} \llbracket \mathbf{x} \rrbracket^A \sigma &= \sigma(\mathbf{x}) \\ \llbracket F(t_1, \dots, t_m) \rrbracket^A \sigma &= F^A(\llbracket t_1 \rrbracket^A \sigma, \dots, \llbracket t_m \rrbracket^A \sigma) \\ \llbracket \mathbf{i} \mathbf{f} \ b \ \mathbf{then} \ t_1 \ \mathbf{else} \ t_2 \ \mathbf{fi} \rrbracket^A \sigma &= \begin{cases} \llbracket t_1 \rrbracket^A \sigma & \mathrm{if} \ \llbracket b \rrbracket^A \sigma = \mathbf{t} \\ \llbracket t_2 \rrbracket^A \sigma & \mathrm{if} \ \llbracket b \rrbracket^A \sigma = \mathbf{f}. \end{split}$$

Note that for a constant $F :\to s$, this gives $\llbracket F() \rrbracket^A \sigma = F^A \in A_s$.

For a *tuple* of terms $t = (t_1, \ldots, t_m)$, we write $\llbracket t \rrbracket^A \sigma =_{df} (\llbracket t_1 \rrbracket^A \sigma, \ldots, \llbracket t_m \rrbracket^A \sigma)$.

Definition 3.3.1. For any $M \subseteq Var$, and states σ_1 and σ_2 , $\sigma_1 \approx \sigma_2$ (rel M) means $\sigma_1 \upharpoonright M = \sigma_2 \upharpoonright M$.

Lemma 3.3.2 (Functionality lemma for terms). For any term t and states σ_1 and σ_2 , if $\sigma_1 \approx \sigma_2$ (rel var(t)), then $[t]^A \sigma_1 = [t]^A \sigma_2$.

Proof: By structural induction on t.

3.4 Algebraic operational semantics

We will interpret $While^{RA}$ programs as many-valued state transformations, and their meaning functions as many-valued functions on A. Our approach follows and extends the algebraic operational semantics of [8].

Notation 3.4.1 (Many-valued functions). (a) We write $F : X \rightrightarrows Y$ and $F : X \rightrightarrows^+$ Y for $F : X \rightarrow \mathcal{P}(Y)$ and $F : X \rightarrow \mathcal{P}^+(Y)$ respectively, where $\mathcal{P}(X)$ and $\mathcal{P}^+(X)$ are (respectively) the sets of all subsets of Y, and all non-empty subsets of Y. (b) We write Y^{\uparrow} for $Y \cup \{\uparrow\}$, where ' \uparrow ' denotes divergence.

Definition 3.4.2 (Many-valued function composition). Given many-valued functions $g: A \rightrightarrows B$ and $h: B \rightrightarrows C$, we define the composed function $h \circ g: A \rightrightarrows C$ as follows:

$$h \circ g(x) = \{h(y) \mid y \in g(x)\}$$

for all $x \in A$. We write h(g(x)) for $h \circ g(x)$.

For the $While^{RA}$ language, we will define the meaning of a statement S to be a state transformation:

$$\llbracket S \rrbracket^A : State(A) \Rightarrow^+ State(A)^{\uparrow}$$

The definition is by structural induction on S. First we define

$$\langle S_{\mathsf{at}} \rangle^A \sigma : State(A) \implies^+ State(A).$$

for atomic statements S_{at} :

$$\begin{aligned} &\langle \mathsf{skip} \rangle^A \sigma &= \{ \sigma \} \\ &\langle \mathsf{x} := t \rangle^A \sigma &= \{ \sigma \{ \mathsf{x} / \llbracket t \rrbracket^A \sigma \} \} \\ &\langle \mathsf{x} := ? \rangle^A \sigma &= \{ \sigma' \mid \sigma'(\mathsf{y}) = \sigma(\mathsf{y}) \text{ for all } \mathsf{y} \not\equiv \mathsf{x} \} \end{aligned}$$

Next we define the functions

$$First : Stmt \rightarrow AtSt$$

 $Rest^A : Stmt \times State(A) \rightarrow Stmt,$

where, for a statement S and state σ , First(S) is an atomic statement which gives the *first* step in the execution of S (in any state), and $Rest^A(S,\sigma)$ is a statement which gives the *rest* of the execution in state σ . The definitions of First(S) and $Rest^A(S,\sigma)$ proceed by structural induction on S: (i)

$$First(S) = \begin{cases} S & \text{if } S \text{ is atomic} \\ First(S_1) & \text{if } S \equiv S_1; S_2 \\ skip & \text{otherwise.} \end{cases}$$

(*ii*) **Rest**^A(S, σ) is defined as follows.

Case 1. S is atomic.

$$Rest^A(S,\sigma) = skip.$$

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

$$\operatorname{\operatorname{\textit{Rest}}}^{A}(S,\sigma) = \begin{cases} S_{2} & \text{if } S_{1} \text{ is atomic} \\ \operatorname{\operatorname{\textit{Rest}}}^{A}(S_{1},\sigma); S_{2} & \text{otherwise.} \end{cases}$$

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

$$\mathbf{Rest}^{A}(S,\sigma) = \begin{cases} S_{1} & \text{if } \llbracket b \rrbracket^{A}\sigma = \mathbf{t} \\ S_{2} & \text{if } \llbracket b \rrbracket^{A}\sigma = \mathbf{f} \end{cases}$$

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

$$\operatorname{\operatorname{\operatorname{Rest}}}^{A}(S,\sigma) = \begin{cases} S_{0}; S & \text{if } \llbracket b \rrbracket^{A} \sigma = \mathfrak{t} \\ \operatorname{skip} & \text{if } \llbracket b \rrbracket^{A} \sigma = \mathfrak{f}. \end{cases}$$

Now we define the one-step computation function

$$CompStep^{A} : Stmt \times State(A) \Rightarrow^{+} State(A)$$

as

$$CompStep^{A}(S,\sigma) = \langle First(S) \rangle^{A} \sigma$$

Note that $CompStep^A$ is a many-valued function.

Finally, we construct a semantic computation tree $CompTree^{A}(S,\sigma)$ for a $While^{RA}$ statement S at a state σ . This tree branches according to all possible outcomes of the one-step computation function $CompStep^{A}(S,\sigma)$. Each node is labelled by a state σ' , with the initial state σ as the root. Each edge is labelled with an atomic statement.

First, we need to define a bounded computation tree, $CompTree_bdd^A(S, \sigma, n)$, which is the computation tree $CompTree^A(S, \sigma)$ up to stage n, by recursion on n: Base case: $CompTree_bdd^A(S, \sigma, 0)$ consists of only the root node:

\bigcap	\sum	σ
\mathcal{L}	Γ	

Induction step: (i) For S atomic, there are 3 cases:

1. $S \equiv \text{skip.}$ Then $CompTree_bdd^A(S, \sigma, n+1)$ is:



2. $S \equiv \mathbf{x} := t$. Then **CompTree_bdd**^A $(S, \sigma, n + 1)$ is:



3. $S \equiv \mathbf{x} := ?$. Then **CompTree_bdd**^A $(S, \sigma, n+1)$ is:



The leaves are the outcomes of the execution of S, i.e. the set $\{\sigma', \sigma'', \sigma''', \dots\}$ of all states which are variants of σ at **x**. Each edge is labelled by S.

(*ii*) For S not atomic, **CompTree_bdd**^A(S, σ , n + 1) is formed by attaching the subtree **CompTree_bdd**^A(**Rest**^A(S, σ), σ' , n) to each $\sigma' \in \langle First(S) \rangle^{A} \sigma$:



From the construction, any actual computation of statement S at state σ corresponds to a path from the root. There are two possibilities for any such path: (1) *finite*, ending in a leaf labelled with a final state of the computation; and (2) *infinite*, indicating divergence. Now we can define

$$CompTree^{A}(S,\sigma) = \bigcup_{n=0}^{\infty} CompTree_{-}bdd^{A}(S,\sigma,n)$$

where $\bigcup_{n=0}^{\infty}$ is a suitable "limiting" operation on increasing sequence of trees. The following lemma is needed for the semantics of *While*^{*RA*} statements (Theorem 3.5.1).

Lemma 3.4.3. Let n > 0.

- (a) If $S \in AtSt$, then CompTree_bdd^A (S, σ, n) is formed by attaching to the root $\{\sigma\}$, the leaf set $\langle\!\langle S \rangle\!\rangle^A \sigma$ and edges between them labelled with S;
- (b) If $S \equiv S_1; S_2$, then **CompTree_bdd**^A (S, σ, n) is formed by attaching subtrees **CompTree_bdd**^A $(S_2, \sigma', n - m)$ to each leaf σ' of **CompTree_bdd**^A (S_1, σ, n) , of depth $m \leq n$;

- (c) If $S \equiv \text{if } b$ then S_1 else S_2 fi, then $CompTree_bdd^A(S, \sigma, n)$ is formed by attaching to the root $\{\sigma\}$, the subtree $CompTree_bdd^A(S_i, \sigma, n-1)$, where if $[\![b]\!]^A \sigma = \mathfrak{t}$ then i = 1, else i = 2;
- (d) If $S \equiv$ while b do S_1 od, then $CompTree_bdd^A(S, \sigma, n)$ is formed by attaching to the root $\{\sigma\}$, the subtree $CompTree_bdd^A(S_1; S, \sigma, n-1)$ if $[\![b]\!]^A \sigma = \mathfrak{t}$; nothing, otherwise.

Proof:

See Appendix 1 for details.

3.5 Semantics of $While^{RA}$ statements

We define the i/o semantics of $While^{RA}$ statements

$$\llbracket S \rrbracket^A : State(A) \Rightarrow^+ State(A)^{\uparrow}$$

as follows: $\llbracket S \rrbracket^A \sigma$ consists of:

- (1) the set of states of all leaves of $CompTree^{A}(S, \sigma)$, and also
- (2) \uparrow , provided that **CompTree**^A(S, σ) has an infinite path.

This definition satisfies the usual desirable properties, as shown by:

Theorem 3.5.1. (a) For S atomic, $[S]^A = \langle S \rangle^A$, *i.e.*,

$$\begin{bmatrix} \mathsf{skip} \end{bmatrix}^A \sigma &= \{ \sigma \} \\ \llbracket \mathbf{x} := t \rrbracket^A \sigma &= \{ \sigma \{ \mathbf{x} / \llbracket t \rrbracket^A \sigma \} \} \\ \llbracket \mathbf{x} := ? \rrbracket^A \sigma &= \{ \sigma' \mid \sigma'(\mathbf{y}) = \sigma(\mathbf{y}) \text{ for all } \mathbf{y} \neq \mathbf{x} \} \end{cases}$$

(b)

$$\llbracket S_1; S_2 \rrbracket^A \sigma \simeq \llbracket S_2 \rrbracket^A (\llbracket S_1 \rrbracket^A \sigma).$$

Note the use of *multi-valued function composition* (Definition 3.4.2) here. (c)

$$\llbracket \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi} \rrbracket^A \sigma \simeq \begin{cases} \llbracket S_1 \rrbracket^A \sigma & \text{if } \llbracket b \rrbracket^A \sigma = \texttt{tt} \\ \llbracket S_2 \rrbracket^A \sigma & \text{if } \llbracket b \rrbracket^A \sigma = \texttt{ff} \end{cases}$$

(d)

$$\llbracket \text{while } b \text{ do } S \text{ od} \rrbracket^A \sigma \simeq \begin{cases} \llbracket S; \text{while } b \text{ do } S \text{ od} \rrbracket^A \sigma & \text{if } \llbracket b \rrbracket^A \sigma = \texttt{tt} \\ \{\sigma\} & \text{if } \llbracket b \rrbracket^A \sigma = \texttt{ff} \end{cases}$$

Proof: The proof uses Lemma 3.4.3. See Appendix 2 for details.

Definition 3.5.2. For $M \subseteq Var$ and $U_1, U_2 \subseteq State(A)^{\uparrow}$:

(a)
$$U_1 \subseteq U_2$$
 (rel M) means:
(i) $\forall \sigma_1 \in U_1 \exists \sigma_2 \in U_2, \sigma_1 \approx \sigma_2$ (rel M), and
(ii) $\uparrow \in U_1 \Rightarrow \uparrow \in U_2$.

(b) $U_1 \approx U_2$ (rel M) means: $U_1 \subseteq U_2$ (rel M) and $U_2 \subseteq U_1$ (rel M).

Definition 3.5.3. For $M \subseteq Var$ and trees T_1, T_2 with nodes labelled by states: $T_1 \approx T_2$ (rel M) means that there is an isomorphism between T_1 and T_2 , such that for every pair of corresponding nodes labelled σ_1 and $\sigma_2, \sigma_1 \approx \sigma_2$ (rel M).

Lemma 3.5.4. Suppose $Var(S) \subseteq M$. If $\sigma_1 \approx \sigma_2$ (rel M) then

$$CompTree_bdd^{A}(S, \sigma_{1}, n) \approx CompTree_bdd^{A}(S, \sigma_{2}, n) (rel M).$$

Proof: Induction on *n*. For the case $S \equiv \mathbf{x} := t$, use the functionality lemma (3.3.2) for terms.

Lemma 3.5.5 (Functionality lemma for semantic computation trees). Suppose $Var(S) \subseteq M$. If $\sigma_1 \approx \sigma_2$ (rel M) then

$$CompTree^{A}(S, \sigma_1) \approx CompTree^{A}(S, \sigma_2) (rel M).$$

Proof: From Lemma 3.5.4.

Lemma 3.5.6 (Functionality lemma for $While^{RA}$ statements). Suppose $Var(S) \subseteq M$. If $\sigma_1 \approx \sigma_2$ (rel M) then $[\![S]\!]^A \sigma_1 \approx [\![S]\!]^A \sigma_2$ (rel M).

Proof: From the functionality lemma (3.5.5) for semantic computation trees.

3.6 Semantics of $While^{RA}$ procedures

Now let a: u, b: v, c: w. If

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

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

$$\llbracket P \rrbracket^A : A^u \implies A^{v\uparrow}.$$

We write P^A for $\llbracket P \rrbracket^A$. For $a \in A^u$, let σ be any state on A such that $\sigma[\mathbf{a}] = a$, $\sigma[\mathbf{b}] = \delta^v$, and $\sigma[\mathbf{c}] = \delta^w$, where δ^v and δ^w are the default tuples of type v and w. Then

 $P^A(a) \ = \{ \sigma'[\mathbf{b}] \ | \ \sigma' \in [\![S]\!]^A \sigma \} \ \cup \{ \uparrow \ | \ \uparrow \in [\![S]\!]^A \sigma \}.$

The following lemma shows that P^A is well defined.

Lemma 3.6.1 (Functionality lemma for procedures). Suppose

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

If $\sigma_1 \approx \sigma_2$ (rel a), then $\llbracket S \rrbracket^A \sigma_1 \approx \llbracket S \rrbracket^A \sigma_2$ (rel b).

Proof: From the functionality lemma (3.5.2) for statements.

3.7 The language GC

,

Another nondeterministic computational model is the *Guarded Command Language* $GC(\Sigma)$ introduced by Edsger W.Dijkstra [1]. The class of statements S is generated by the rules

$$S ::= \mathsf{skip} \parallel \mathtt{x} := t \parallel S_1; S_2 \parallel \mathsf{if} \dots \mathsf{fi} \parallel \mathsf{do} \dots \mathsf{od}_2$$

The nondeterminism here arises not from random assignments, but from the two constructs which replace the *conditional* and *iteration* in $While^{RA}$:

(i) the guarded command conditional

$$\text{if } b_1 \to S_1 | \dots | b_k \to S_k \text{ fi} \tag{3.1}$$

(ii) the guarded command iteration

do
$$b_1 \to S_1 | \dots | b_k \to S_k$$
 od (3.2)

where $k \ge 0$, and b_i are Σ -terms of sort **bool**.

We give only informal semantics for GC, since, as we will see, it can be replaced by $While^{RA(bool)}$.

- (i) For the guarded command conditional (3.1), if any of the boolean tests is true, then one of the corresponding statements is executed; otherwise, the procedure halts. (Hence, for k = 0, if fi corresponds to halt.)
- (*ii*) For the guarded command iteration (3.2), repeatedly execute any one of the statements for which the corresponding boolean test is true, until none of these boolean tests is true. (Hence, for k = 0, do od corresponds to skip.)

3.8 $While^{RA*}$ computability

Recall that $While(\Sigma)$ is the $While^{RA}(\Sigma)$ language without random assignments. Then a $While^{N}(\Sigma)$ procedure is a $While(\Sigma^{N})$ procedure in which the input and output variables have sorts in Σ (but the auxiliary variables may have sort nat). Similarly a $While^{*}(\Sigma)$ procedure is a $While(\Sigma^{*})$ procedure in which the input and output variables have sorts in Σ (but the auxiliary variables may have sort nat or "starred").

Thus $While^{N}(\Sigma)$ and $While^{*}(\Sigma)$ procedures P define functions P^{A} on Σ -algebras A.

In the same way, we define $While^{RA*}(\Sigma)$ procedures as $While^{RA}(\Sigma)$ procedures in which the auxiliary variables (only) may be of sort **nat** or starred.

3.9 Equivalence of $While^{RA(bool)}$ and GC

 $While^{RA(bool)}$ is the restriction of the $While^{RA}$ language to random assignments on booleans only. Its semantic computation tree shares the property with GC that at each level, there are only finitely many leaves. This leads us to the question: Is $While^{RA(bool)}$ equivalent to GC? In order to answer these questions precisely, we must first give some definitions.

Definition 3.9.1. Let $\mathcal{L}_1(\Sigma)$ and $\mathcal{L}_2(\Sigma)$ be two programming languages over Σ .

- (i) $\mathcal{L}_1(\Sigma) \preceq \mathcal{L}_2(\Sigma)$ means that $\mathcal{L}_1(\Sigma)$ can be *compiled* in $\mathcal{L}_2(\Sigma)$, *i.e.*, there exists an effective transformation of \mathcal{L}_1 -procedures to \mathcal{L}_2 -procedures, which preserves semantics.
- (*ii*) $\mathcal{L}_1(\Sigma) \approx \mathcal{L}_2(\Sigma)$ means: $\mathcal{L}_1(\Sigma) \preceq \mathcal{L}_2(\Sigma)$ and $\mathcal{L}_2(\Sigma) \preceq \mathcal{L}_1(\Sigma)$.

Definition 3.9.2. For any Σ -language \mathcal{L} and Σ -structure A, $\mathcal{L}(A)$ is the set of all \mathcal{L} -computable functions over A.

Remark 3.9.3. $(i)\mathcal{L}_1(\Sigma) \preceq \mathcal{L}_2(\Sigma) \implies \mathcal{L}_1(A) \subseteq \mathcal{L}_2(A).$ $(ii)\mathcal{L}_1(\Sigma) \approx \mathcal{L}_2(\Sigma) \implies \mathcal{L}_1(A) = \mathcal{L}_2(A)$

Theorem 3.9.4 (Equivalence Theorem).

$$While^{RA(bool)}(\Sigma) \approx GC(\Sigma)$$

Proof: (1) First we will prove $While^{RA(bool)}(\Sigma) \preceq GC(\Sigma)$, by defining an effective transformation of $While^{RA(bool)}$ procedures to GC procedures. Clearly, the statements skip, $\mathbf{x} := t$, and $S_1; S_2$ can be translated to themselves. Next, the conditional

```
if b then S_1 else S_2 fi
```

is translated into the GC statement

if
$$b \to S_1 \mid \neg b \to S_2$$
 fi.

Also the iteration

while
$$b \text{ do } S \text{ od}$$

is translated into the GC statement

do
$$b \rightarrow S \mid \neg b \rightarrow$$
 skip od.

Finally, we translate the random boolean assignment b :=? into the following **GC** procedure:

```
proc out b : bool
begin
if true \rightarrow b := true
true \rightarrow b := false
fi
```

end

This completes the proof in the one direction.

(2) We must prove that $GC(\Sigma) \preceq While^{RA(bool)}(\Sigma)$. We do this in two stages. Stage 1: Show that

$$GC(\Sigma) \preceq While^{RA(bool)}(\Sigma) + halt$$

We must transform the two kinds of guarded commands (3.1) and (3.2) into $While^{RA(bool)}$ procedures. We illustrate (3.1) with the simple case k = 2:

if
$$b_1 \rightarrow S_1 \mid b_2 \rightarrow S_2$$
 fi

which is translated into the following $While^{RA(bool)}$ procedure

```
proc
         aux b : bool
begin
         if b_1 \wedge \neg b_2
           then S_1
           else if \neg b_1 \land b_2
                  then S_2
                   else if b_1 \wedge b_2
                          then b :=?;
                                  if b then {\cal S}_1
                                         else S_2
                                  fi
                          else halt
                        fi
                  fi
         fi
end
```

Likewise, we illustrate (3.2) with the simple case

do $b_1 \rightarrow S_1 \mid b_2 \rightarrow S_2 \operatorname{od}$

which is translated as:

```
proc aux b : bool
begin
while b_1 \lor b_2 do
if b_1 \land \neg b_2 then S_1
else if \neg b_1 \land b_2 then S_2
else b :=?;
if b then S_1
else S_2
fi
fi
fi
od
```

end

Similar procedures can be used to simulate the more general case where k > 2. Stage 2: Show that

$$While^{RA(bool)}(\Sigma) + halt \preceq While^{RA(bool)}(\Sigma).$$

By Mirkowska's theorem [6, 2, 4], every while program can be efficiently transformed into one with a single while loop (with additional boolean variables), *i.e.*,

$$While(\Sigma) \preceq While^{1}(\Sigma),$$
 (3.3)

where $While^1$ is the set of While procedures with only one 'while' loop. This transformation can be easily modified so as to show

$$While(\Sigma) + halt \preceq While^{1}(\Sigma)$$
 (3.4)

(some details are given in Lemma 3.9.5 below), and similarly:

$$While^{RA(bool)}(\Sigma) + halt \preceq While^{RA(bool)}(\Sigma).$$
 (3.5)

Combining stages 1 and 2, we get:

$$GC(\Sigma) \preceq While^{RA(bool)}(\Sigma),$$

proving the theorem.

Remark 3.9.5. (Proof of Mirkowska's Theorem for $While^{RA} + halt$). We give an outline of the proof for (3.4), modifying Mirkowska's proof for (3.3). More details of the latter can be found in [6, 2, 4]. Consider a *While* program S.

First, rewrite S as a 'goto' program $S' \equiv$

$$1: S_1;$$

 $2: S_2;$
 \vdots
 $L: S_L$

with L elementary statements labelled by the integers $1, \ldots, L$. Each S_i is either (1) an assignment, or (2) a conditional jump 'if b then goto j else goto k', where b is some boolean term, and $1 \le j, k \le L$, or (3) halt.

Now represent the labels $1, \ldots, L$ by K-tuples of truth values, where $K = \lceil \log(L+1) \rceil$:

$$\bar{1} \equiv (\text{true}, \text{true}, \dots, \text{true})$$
$$\bar{2} \equiv (\text{true}, \text{true}, \dots, \text{false})$$
$$\vdots$$
$$\bar{L} \equiv (\text{true}, \text{false}, \dots, \text{false})$$
$$\overline{L+1} \equiv (\text{false}, \text{false}, \dots, \text{false}).$$

Here $\overline{L+1}$ will represent the 'halt' condition. Also introduce a variable (actually a K-tuple of boolean variables) for the labels:

$$label = (b_1, b_2, \cdots, b_K) : bool^K$$

and a boolean variable over.

Finally, rewrite S' as a $While(\Sigma)$ statement \tilde{S} , which first initializes the variable **over** to false, and then has a single loop containing (using some obvious pseudo-code) a huge case statement:

```
while not over do

case label of

\overline{1}: \tilde{S}_1;

\overline{2}: \tilde{S}_2;

\vdots

\overline{L}: \tilde{S}_L;

\overline{L+1}: over := true

esac

od
```

where for $i = 1, \ldots, L$:

- (1) if S_i is an assignment, then $\tilde{S}_i \equiv S_i$; label := $\overline{i+1}$,
- (2) if S_i is if b then goto j else k fi, then $\tilde{S}_i \equiv$ label := if b then \bar{j} else \bar{k} fi,
- (3) if $S_i \equiv$ halt, then $\tilde{S}_i \equiv$ label $:= \overline{L+1}$,

This is clear that S is semantically equivalent to S', which in turn is semantically equivalent to S.

This proves (3.4). The assertion (3.5) is proved similarly.

Remark 3.9.6. This theorem fails for $While^{RA(nat)}$, which has random assignments on naturals, i.e.,

$$While^{RA(nat)}(\Sigma) \not\preceq GC(\Sigma).$$

Consider, for example, the simple $While^{RA(nat)}$ procedure

 $P \equiv \text{proc out } n : \text{nat begin } n := ? \text{ end}$

Suppose we try to simulate this by a GC procedure $P' \equiv$

```
proc out n : nat
aux b : bool
begin
n := 0;
b := true
do b \rightarrow n := n + 1 | b \rightarrow b := false od
end
```

Then the semantic computation tree for P' contains an infinite path (indicating the possibility of divergence), which does not occur in the semantic computation tree for P. So, P' is not semantically equivalent to P.

In fact there is no GC procedure which simulates P. For suppose Q is such a procedure, then the semantic computation tree for Q must (1) be finitely branching, like all GC trees, (2) have infinitely many leaves, like the tree for P, and (3) not have an infinite path, again like the tree for P. But these 3 conditions together contradict König's Lemma.

Because of the above equivalence theorem, from now on we focus our attention on the $While^{RA}$ programming language rather than GC.

4 Representations of *While*^{*RA*} semantic functions

To examine to what extent the $While^{RA}$ language and its various sublanguages satisfy the Universal Function Theorem, we need to represent faithfully the syntax and semantics of $While^{RA}$ computations using functions on A.

In this section, we apply the techniques of Gödel numbering and state representations. More accurately, for Gödel numbering to be possible, we need the sort nat, and so we will investigate the possibility of representing the syntax of a standard Σ -algebra A (not in A itself, but) in its N-standardisation A^N , or (failing that) in the array algebra A^* .

4.1 Gödel numbering of syntax

We assume given a family of numerical codings, or Gödel numberings, of the classes of syntactic expressions of Σ and Σ^* , *i.e.*, a family gn of effective mappings from expressions E to natural numbers $\lceil E \rceil = gn(E)$, which satisfy certain basic properties: $(i) \lceil E \rceil$ increases strictly with the complexity of E, and in particular, the code of an expression is larger than those of its subexpressions; (ii) sets of codes of the various syntactic classes, and of their respective subclasses, such as $\{\lceil t \rceil \mid t \in Term\}$, $\{\lceil t \rceil \mid t \in Term_s\}$, etc., are primitive recursive; (iii) we can go primitive recursively from codes of expressions to codes of their immediate subexpressions, and vice versa.

This means that we can primitive recursively simulate all operations involved in processing the syntax of the programming language.

We will use the notation $\lceil Term \rceil =_{df} \{ \lceil t \rceil \mid t \in Term \}$, etc., for sets of Gödel numbers of syntactic expressions.

4.2 Representation of states

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

The state representing function, $\operatorname{Rep}_{\mathbf{x}}^{A}$: $\operatorname{State}(A)^{\uparrow} \to A^{u\uparrow}$, is defined by

$$Rep_{x}^{A}(\sigma) = \sigma[x]$$

and

$$Rep_{x}^{A}(\uparrow) = \uparrow$$

4.3 Representation of term evaluation

Let **x** be a *u*-tuple of variables. Let $Term_{\mathbf{x}}(\Sigma)$ be the class of all Σ -terms with variables among **x** only, and for all sorts *s* of Σ , let $Term_{\mathbf{x},s}(\Sigma)$ be the class of such terms of sort *s*. Similarly we write $TermTup_{\mathbf{x}}(\Sigma)$ for the class of all term tuples with variables among **x** only, and $TermTup_{\mathbf{x},v}(\Sigma)$ for the class of all *v*-tuples of such terms.

The term evaluation function on A relative to \mathbf{x} , $\mathbf{TE}_{\mathbf{x},s}^{A}$: $\mathbf{Term}_{\mathbf{x},s} \times \mathbf{State}(A) \rightarrow A_{s}$, defined by

$$TE_{\mathbf{x},s}^{A}(t,\sigma) = \llbracket t \rrbracket^{A} \sigma,$$

is *represented* by the function, $te_{\mathbf{x},s}^A$: $\lceil Term_{\mathbf{x},s} \rceil \times A^u \rightarrow A_s$, defined by

$$te_{\mathbf{x},s}^{A}(\ulcorner t\urcorner, a) = \llbracket t \rrbracket^{A} \sigma,$$

where σ is any state on A such that $\sigma[\mathbf{x}] = a$. This is well defined, by the functionality lemma (3.3.2) for terms. In other words, the following diagram commutes:



Similarly, for a product type v, we will define an evaluating function for *tuples* of terms, $te_{\mathbf{x},v}^A$: $\lceil TermTup_{\mathbf{x},v} \rceil \times A^u \rightarrow A^v$, by

$$te_{\mathbf{x},v}^{A}(\ulcorner t\urcorner, a) = \llbracket t \rrbracket^{A} \sigma.$$

4.4 Representation of the *First* and *Rest* functions

For $\mathbf{x} : u$, let $Stmt_{\mathbf{x}}$ be the class of statements with variables among \mathbf{x} only, and define

$$Rest_{x}^{A} =_{df} Rest^{A} \upharpoonright (Stmt_{x} \times State(A)).$$

(see § 3.4). Then *First* and *Rest*^A_x are *represented* by the functions

$$\begin{array}{ll} \textit{first} & : \ulcorner \textit{Stmt} \urcorner \to \ulcorner \textit{AtSt} \urcorner \\ \textit{rest}_{\mathtt{x}}^{A} & : \ulcorner \textit{Stmt}_{\mathtt{x}} \urcorner \times A^{u} \to \ulcorner \textit{Stmt}_{\mathtt{x}} \urcorner \end{array}$$



which are defined so as to make the following diagrams commute:

4.5 Representation of statement evaluation

For $\mathbf{x} : u$, let $AtSt_{\mathbf{x}}$ be the class of atomic statements with variables among \mathbf{x} only. The atomic statement evaluation function on A relative to \mathbf{x} , $AE_{\mathbf{x}}^{A} : AtSt_{\mathbf{x}} \times State(A) \Rightarrow^{+} State(A)$, defined by

$$\boldsymbol{AE}_{\mathbf{x}}^{A}(S,\sigma) = \llbracket S \rrbracket^{A} \sigma,$$

is *represented* by the function, $ae_{x}^{A}: \ \ \Box AtSt_{x} \ \supseteq \times A^{u} \ \Longrightarrow^{+} \ A^{u}$, defined by

$$ae_{\mathbf{x}}^{A}(\ulcorner S\urcorner, a) = \{\sigma'[\mathbf{x}] \mid \sigma' \in \langle\!\langle S \rangle\!\rangle^{A} \sigma\},\$$

where σ is any state such that $\sigma[\mathbf{x}] = a$. This commutes the following diagram:



Now let $Stmt_x$ be the class of statements with variables among x only. The statement evaluation function on A relative to x, SE_x^A : $Stmt_x \times State(A) \Rightarrow^+$ $State(A)^{\uparrow}$, defined by

$$\boldsymbol{SE}_{\mathbf{x}}^{A}(S,\sigma) = [\![S]\!]^{A}\sigma$$

is represented by the function, se_{x}^{A} : $\ \ \, Stmt_{x}^{\neg} \times A^{u} \ \Rightarrow^{+} \ A^{u\uparrow}$, defined by

 $se_{\mathbf{x}}^{A}(\ulcornerS\urcorner, a) = \{\sigma'[\mathbf{x}] \mid \sigma' \in \llbracketS\rrbracket^{A}\sigma\} \cup \{\uparrow \mid CompTree^{A}(S, \sigma) \text{ has an infinite path}\}$

where σ is any state on A such that $\sigma[\mathbf{x}] = a$. This makes the following diagram commute:



4.6 Representation of procedure evaluation

It is a rather subtle matter to represent the class $Proc_{u \to v}$ of all $While^{RA}$ procedures of type $u \to v$, since it requires a coding for *arbitrary tuples of auxiliary variables*. For now we consider a restricted version, for the subclass of $Proc_{u \to v}$ with auxiliary variables of a given fixed type.

So let $\mathbf{a}, \mathbf{b}, \mathbf{c}$ be pairwise disjoint lists of variables, with types $\mathbf{a} : u, \mathbf{b} : v$ and $\mathbf{c} : w$. Let $\operatorname{Proc}_{\mathbf{a},\mathbf{b},\mathbf{c}}$ be the class of $While^{RA}$ procedures of type $u \to v$, with declaration in $\mathbf{a} : u$ out $\mathbf{b} : v$ aux $\mathbf{c} : w$. The procedure evaluation function on Arelative to $\mathbf{a}, \mathbf{b}, \mathbf{c}, \operatorname{PE}_{\mathbf{a},\mathbf{b},\mathbf{c}}^{A} : \operatorname{Proc}_{\mathbf{a},\mathbf{b},\mathbf{c}} \times A^{u} \rightrightarrows A^{v^{\uparrow}}$, defined by

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

is represented by the function, $pe_{a,b,c}^A$: $\lceil Proc_{a,b,c} \rceil \times A^u \implies A^{v\uparrow}$, defined by

$$pe_{a,b,c}^{A}(\ulcorner P\urcorner, a) = P^{A}(a).$$

This makes the following diagram commute:



In the next section, we will investigate the computability of these sematic representing functions.

5 While^{RA} computability and universality

In this section, we will explore the problem of the existence of a Universal Function Theorem (UFT) for $While^{RA}$: Is there a universal $While^{RA}$ procedure of a given type that can compute all the $While^{RA}$ computable functions on A? We will investigate this problem in two cases: (1) for $While^{RA(nat/bool)}$, i.e., $While^{RA}$ with random assignments restricted to sorts nat and/or bool; (2) for the unrestricted $While^{RA}$ language.

5.1 Term evaluation property

In order to study further the computability of the representing functions given in Section 4, we must make an assumption on the algebra.

Definition 5.1.1 (Term evaluation property). The algebra A has the *term evaluation property* (*TEP*) if and only if for all \mathbf{x} and s, the term evaluation representing function $te_{\mathbf{x},s}^{A}$ is *While* computable on A^{N} .

Remark 5.1.2. (a) Many well-known varieties (i.e., equationally axiomatisable classes of algebras) have the TEP; for example, semigroups, groups and rings with or without unity. This follows from the effective normalizability of the terms of these varieties. It is therefore a very reasonable assumption to make on algebras, as in the UFT for $While^{RA(nat/bool)}$ (Theorem 5.4.3 below), where we actually prove the equivalence of the TEP with the UFT. For more on the UFT, see [8, Examples 4.5]. (b) Also, by [8, Corollary 4.7], $te_{x,s}^A$ is always $While^*$ computable on A^N . Therefore, for any algebra A, the algebra A^* always has the TEP.

5.2 Locality of computation

A programming language \mathcal{L} over a signature Σ is said to satisfy *locality of compu*tation if for any \mathcal{L} -procedure $P: u \to v$, and any (N-)standard Σ -algebra A, the output of P^A applied to any input in A is contained in the subalgebra of A generated by that input.

In [8, §3.8], it is shown that the *While* language satisfies locality of computation.

The issue of locality of computation is important in investigating the UFT for various languages, as we will see below ($\S5.3$ and $\S5.4$). For now, we point out that

- Full While^{RA} clearly does not satisfy locality, since random assignments for arbitrary sorts take us, in general, out of the subalgebra generated by the input.
- (2) However $While^{RA(nat/bool)}$ does satisfy locality. The proof extends that in [8] for *While*, by noting that for (N-)standard algebras, random assignments for sorts bool and nat do not take us out of the subalgebra generated by the input, since all of \mathbb{B} and \mathbb{N} are contained in any such subalgebra.

5.3 Computability of semantic representing functions

By examining the definitions of the various semantic functions in Section 3, we can infer the computability of the corresponding representing functions, as shown below (Theorem 5.3.4).

Remark 5.3.1 (Procedure calls; Many-valued composition). (a) In the fragments of program code below, we use extensions of the $While^{RA}$ (etc.) languages by (non-recursive) procedure calls

$$\mathbf{x} := P(t)$$

where t is a term tuple of the same type as the input variables. This can be eliminated by the well-known method of replacing the procedure's name P by its body [TZ00, Sec. 3.9] which also works for many-valued procedures.

(b) Also, many-valued composition of procedures (see Definition 3.4.2) can be handled similarly, e.g.

$$\mathbf{x} := P_1(P_2(t))$$

can be rewritten as

$$\mathbf{z} := P_2(t);$$
$$\mathbf{x} := P_1(\mathbf{z})$$

and the procedure calls can then be eliminated as in part (a).

Lemma 5.3.2. The function $first : \mathbb{N} \to \mathbb{N}$ is primitive recursive, and hence *While* computable on A^N , for any standard Σ -algebra A [8, §4.7].

Lemma 5.3.3. Let \mathbf{x} be a tuple of program variables and A a standard Σ -algebra.

- (a) $rest_{\mathbf{x}}^{A}$ is While computable in $\langle te_{\mathbf{a},s}^{A} | s \in Sort(\Sigma) \rangle$ on A^{N} .
- (b) $ae_{\mathbf{x}}^{A}$ is $While^{RA}$ computable in $\langle te_{\mathbf{a},s}^{A} | s \in Sort(\Sigma) \rangle$ on A^{N} .
- (c) se_{x}^{A} is $While^{RA}$ computable in ae_{x}^{A} and $rest_{x}^{A}$ on A^{N} .
- (d) $pe_{a,b,c}^{A}$ is **While** computable in se_{x}^{A} on A^{N} , where $x \equiv a, b, c$.

Proof:

We prove parts (a)-(d) by examining the definitions of the semantic functions and giving informal algorithms.

- (a) The semantic definition of Rest^A is a structural recursion on statements with one inductive case $S \equiv S_1; S_2$, and the other three as basic cases. Therefore, the representing function $\operatorname{rest}^A_{\mathfrak{x}}$ is definable by course of values recursion on \mathbb{N} . Here we need $\langle te^A_{\mathfrak{a},s} | s \in \operatorname{Sort}(\Sigma) \rangle$ to evaluate the boolean tests.
- (b) With the input $\lceil S \rceil$ and $a = (a_1, \dots, a_n) \in A^u$ (where $u = s_1 \times \dots \times s_n$), to evaluate $ae_x^A(\lceil S \rceil, a)$, there are 3 cases, for the 3 kinds of atomic statements, which can be distinguished primitive recursively in $\lceil S \rceil$:
 - (i) $S \equiv \text{skip}$, then output = $\{a\}$;

(*ii*)
$$S \equiv \mathbf{y} := t$$
, then output = { (b_1, \dots, b_m) }, where for $i = 1, \dots, m$:

$$b_i = \begin{cases} te_{\mathbf{x},s_i}^A(\ulcorner t_i \urcorner, a) & \text{if } x_i \text{ is in the tuple } \mathbf{y}, \\ a_i & \text{otherwise.} \end{cases}$$

(*iii*)
$$S \equiv \mathbf{x}_i :=$$
?, then output = { $(a_1, \cdots, a_{i-1}, b, a_{i+1}, \cdots, a_n) \mid b \in A_{s_i}$ }.

It follows that $ae_{\mathbf{x}}^{A}$ is $While^{RA}$ computable in $\langle te_{\mathbf{a},s}^{A} | s \in Sort(\Sigma) \rangle$ on A^{N} . (c) The following procedure computes $se_{\mathbf{x}}^{A}$.

```
proc in s : nat, a : u
out b : u
begin
while s \neq \lceil skip \rceil do
s, a := rest_x^A(s, a), ae_x^A(first(s), a);
od
b := a
end
```

Note that the while loop iterates single steps in the (nondeterministic) updating of the "snapshot" (s,a), where s and a represent the current statement and current state respectively. The computation terminates if and when $s = \lceil skip \rceil$.

(d) Finally, $pe_{a,b,c}^{A}(\ulcorner P \urcorner, a)$ is easily computable from $se_{x}^{A}(\ulcorner S \urcorner, x)$, where x = (a, b, c).

Theorem 5.3.4 (Computability theorem for the semantic representing functions). Under the TEP assumption:

- (a) ae_x^A is **While**^{RA} computable, and $rest_x^A$ is **While** computable, on A^N ;
- (b) se_x^A is **While**^{RA} computable on A^N ;
- (c) $pe_{a,b,c}^{A}$ is **While**^{RA} computable on A^{N} .

Proof: From Lemma 5.3.3, and transitivity of relative computability [8, Lemma 3.32].

5.4 Universal procedure for $While^{RA(nat/bool)}$

We use the notation $While^{RA(nat/bool)}$ for any of the following: $While^{RA(nat)}$, $While^{RA(bool)}$, or $While^{RA(nat,bool)}$, i.e., the $While^{RA}$ language with random assignments restricted (respectively) to variables of sorts nat, bool, or both.

First we must define uniform (in **x**) versions of ae_x^A , $rest_x^A$ and se_x^A . The new definitions differ from the old in that their outputs are not sets of tuples of values, but sets of Gödel numbers of tuples of terms in the input variables, which is made possible by *locality of computation* (as discussed in §5.2).

The problem, in general, is that we cannot deal with procedures with unbounded sequences of auxiliary variables using the procedure evaluation $pe_{a,b,c}^{A}$ (see Theorem 5.3.4). However, by locality of computation again, we can represent all the procedure variables (including auxiliary variables) by tuples of terms in the input variables, which can be coded by single Gödel numbers.

We will prove the UFT for $While^{RA(nat/bool)}$ (assuming the TEP). We give the details for $While^{RA(nat)}$, but note that exactly the same reasoning holds for $While^{RA(bool)}$ and $While^{RA(nat,bool)}$.

Definition 5.4.1 (Uniform versions of the semantic representing functions). (a) The function

$$aeu^A$$
: $\ulcorner VarTup \urcorner × \ulcorner AtSt \urcorner \Rightarrow^+ \ulcorner TermTup \urcorner$

is defined by: for any $\mathbf{x} : w$ and $S \in \mathbf{AtSt}_{\mathbf{x}}$, we have $\mathbf{aeu}^{A}(\lceil \mathbf{x} \rceil, \lceil S \rceil) \subseteq \lceil \mathbf{TermTup}_{\mathbf{x},w} \rceil$, such that for any $x \in A^{w}$,

$$te^{A}_{\mathbf{x},w}(aeu^{A}(\lceil \mathbf{x} \rceil, \lceil S \rceil), x) = ae^{A}_{\mathbf{x}}(\lceil S \rceil, x).$$

(b) The function

is defined by: for any $\mathbf{x} : w$ extending $\mathbf{a} : u, S \in Stmt$ and $a \in A^u$ (putting $w = u \times v$):

$$\operatorname{restu}_{a}^{A}(\lceil x \rceil, \lceil S \rceil, a) = \operatorname{rest}_{x}^{A}(\lceil S \rceil, (a, \delta_{A}^{v})).$$

(c) The function

is defined by: for any $\mathbf{x} : w$ extending $\mathbf{a} : u, S \in Stmt$ and $a \in A^u$ (putting $w = u \times v$):

$$seu_{a}^{A}(\lceil x \rceil, \lceil S \rceil, a) = se_{x}^{A}(\lceil S \rceil, (a, \delta_{A}^{v})).$$

Lemma 5.4.2. The function seu_{a}^{A} is $While^{RA(nat/bool)}$ computable on A^{N} , for any standard Σ -algebra A with the TEP.

Proof:

We must show that

- (i) aeu^A is $While^{RA(nat/bool)}$ computable on A^N . Similar to the proof of Lemma 5.3.3 (a).
- (ii) $restu_{a}^{A}$ is While computable on A^{N} . Similar to the proof of Lemma 5.3.3 (b). Note that to evaluate terms in the course of computing $restu_{a}^{A}$ (e.g. boolean tests) $t \in Term_{x,s}$, which may contain variables in x other than a, we use (assuming $x \equiv (a, y)$ where a : u, y : v) $te_{a,s}^{A}(t', a)$, where t' is formed from t by replacing the variables y by δ^{v} .
- (*iii*) seu_{a}^{A} is $While^{RA(nat)}$ computable in aeu^{A} and $restu_{a}^{A}$ on A. Similar to the proof of Lemma 5.3.3 (c).

These three results together give the desired conclusion.

Theorem 5.4.3 (Universal Function Theorem for $While^{RA(nat/bool)}$). For any N-standard Σ -algebra A, the following are equivalent:

(i) A has the TEP,

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

$$\mathsf{Univ}_{u \to v}: \ \ulcorner \mathbf{Proc}_{u \to v}^{\mathbf{RA}(\mathsf{nat/bool}) \, \urcorner} \times A^u \ \rightrightarrows^+ \ A^{v \, \uparrow}$$

which is universal for $While^{RA(nat/bool)}$ procedures of type $u \to v$ on A, in the sense that for all $While^{RA(nat/bool)}$ procedures $P: u \to v$ and $a \in A^u$,

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

Proof:

 $(i) \Rightarrow (ii)$ Assume A has the TEP. We give an informal description of the algorithm represented by the procedure $\mathsf{Univ}_{u \to v}$. With *input* ($\ulcorner P \urcorner$, a), where $P \in \mathbf{Proc}_{u \to v}$ and $a \in A^u$, suppose

$$P~\equiv~$$
 proc in a out b aux c begin S end

where $\mathbf{a}: u, \mathbf{b}: v$ and $\mathbf{c}: w$. The *output* is then

$$te_{\mathbf{a},v}^{A}(seu_{\mathbf{a}}^{A}(\ulcorner\mathbf{a}, \mathbf{b}, \mathbf{c}^{\neg}, \ulcornerS^{\neg}, (a, \delta_{A}^{v}, \delta_{A}^{w})), a),$$
(5.1)

which is $While^{RA(nat/bool)}$ computable by Lemma 5.4.2.

 $(ii) \Rightarrow (i)$ This is the "easy" direction, as in the proof of [8, Theorem 4.14]. \Box Note the use of many-valued composition in (5.1) (*cf.* Remark 5.3.1 (*b*)). As an immediate consequence, we have:

Corollary 5.4.4 (Universal Function Theorem for $While^{RA(nat)}$).

If A has the TEP, then for $A \in StdAlg(\Sigma)$ and all Σ -product types u, v, there is a $While^{RA(nat)}(\Sigma^N)$ procedure

$$\mathsf{Univ}_{u \to v}^A : \ \ulcorner \mathbf{Proc}_{u \to v}^{\mathbf{RA}(\mathsf{nat})} \urcorner \times A^u \ \rightrightarrows \ A^{v\uparrow}$$

which is universal for $While^{RA(nat)}$ procedures of type $u \to v$ on A, in the sense that for all $While^{RA(nat)}$ procedures $P: u \to v$ and $a \in A^u$,

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

Corollary 5.4.5 (Universal Function Theorem for $While^{RA(bool)}$).

For $A \in StdAlg(\Sigma)$ and all Σ -product types u, v, there is a $While^{RA(bool)}(\Sigma^N)$ procedure

$$\mathsf{Univ}_{u \to v}^{A} : \ \ulcorner \mathbf{Proc}_{u \to v}^{\mathbf{RA}(\mathsf{bool})} \underset{\mathsf{nat} \times u \to v}{\overset{\mathsf{nat}}{\to}} \lor A^{u} \ \rightrightarrows \ A^{v^{\uparrow}}$$

which is universal for $While^{RA(bool)}$ procedures of type $u \to v$ on A, in the sense that for all $While^{RA(bool)}$ procedures $P: u \to v$ and $a \in A^u$,

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

Corollary 5.4.6 (Universal Function Theorem for GC).

For $A \in StdAlg(\Sigma)$ and all Σ -product types u, v, there is a $GC(\Sigma)$ procedure

$$\mathsf{Univ}_{u \to v}^A: \ \ulcorner \mathbf{Proc}_{u \to v} \urcorner \times A^u \ \rightrightarrows \ A^{v^{+}}$$

which is universal for GC procedures of type $u \to v$ on A, in the sense that for all $GC(\Sigma)$ procedures $P: u \to v$ and $a \in A^u$,

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

Proof: From the Equivalence Theorem (3.9.4).

5.5 Universal procedure for $While^{RA}$

We now consider the $While^{RA}$ language over Σ , extending While by the random assignment $\mathbf{x} :=$? for variables \mathbf{x} of every sort of Σ .

Since the variable \mathbf{x} in a random assignment can have any sort s, for which the carrier A_s might not be minimal (for example, A_s could be \mathbb{R}), we cannot code the possible new values of \mathbf{x} with closed terms of sort s, and represent the output as a term in the input variables. In other words, locality of computation fails (as discussed in §5.2). Therefore we cannot use the method for $While^{RA(nat/bool)}$ to prove the UFT over $While^{RA}$.

To solve this problem, we use arrays or starred variables (cf. Section 2.4).

Definition 5.5.1. Suppose $Sort(\Sigma) = \{s_1, \ldots, s_k\}$.

(i) A sequence of variables is in Σ -standard form if it has the form

$$\mathsf{c}_1^*, \mathsf{t}_1, \mathsf{c}_2^*, \mathsf{t}_2, \ldots, \mathsf{c}_k^*, \mathsf{t}_k,$$

where $c_i^* : s_i^*, t_i : s_i$.

(ii) A procedure is in Σ -standard auxiliary form if its auxiliary variables are in Σ -standard form.

Let $Proc_{u \to v}^{*}(\Sigma)$ be the class of $While^{RA*}$ procedures of type $u \to v$.

Lemma 5.5.2. Any $While^{RA*}$ procedure P on A can be effectively transformed into a $While^{RA*}$ procedure \hat{P} of the same type in Σ -standard auxiliary form.

Proof: Suppose

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

where $\mathbf{a} : u$, and $\mathbf{b} : v$. Note that \mathbf{c} could include starred variables.

It is clear how to code a finite sequence of (starred and unstarred) variables of each sort by a single starred variable of that sort [4, Chapter 6]. Therefore, for each sort $s_i \in Sort(\Sigma)$, we have a starred variable \hat{c}_i^* , and an unstarred variable t_i (for "temporary", to help with random assignments, as we will see).

Assignments to variables of sort s or s^* can be simulated by Update_s in an obvious way. For random assignment, consider (for simplicity) the case of a random assignment to a simple variable of sort s_i : $\mathbf{x} :=$?. Suppose \mathbf{x} is coded as $\hat{\mathbf{c}}_i^*[j]$, then we simulate this random assignment with the pair of statements:

$$\begin{aligned} \mathbf{t}_i &:= ?; \\ \hat{\mathbf{c}}_i^* &:= & \mathsf{Update}_{s_i}^A(\hat{\mathbf{c}}_i^*, j, \mathbf{t}_i) \end{aligned}$$

Similarly, a random assignment to a starred variable can be simulated with Update_s using a loop.

In this way, we effectively transform P into a procedure in \varSigma -standard auxiliary form

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

where $\hat{c} \equiv \hat{c}_1^*, t_1, \hat{c}_2^*, t_2, \dots, \hat{c}_k^*, t_k$.

Theorem 5.5.3 (Universal Function Theorem for $While^{RA*}$). For $A \in StdAlg(\Sigma)$ and all Σ -product types u, v, there is a $While^{RA*}$ procedure

$$\mathsf{Univ}_{u \to v}: \ \ulcorner \mathbf{Proc}^*_{u \to v} \urcorner \times A^u \ \rightrightarrows \ A^{v\uparrow}$$

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

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

Proof:

Let P be any $While^{RA*}$ procedure of type $u \to v$. By Lemma 5.5.2, we effectively transform P to a procedure $\hat{P} \in Proc^*_{u \to v}$ in Σ -standard auxiliary form

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

where $\hat{c} : \hat{w}$ (say).

By Theorem 5.3.4, there is a $While^{RA*}$ procedure $pe_{a,b,\hat{c}}^A$ which is universal for $While^{*RA}$ procedures of type $u \to v$ with auxiliary variables of type \hat{w} . The required universal $While^{RA*}$ procedure for type $u \to v$,

$$\mathsf{Univ}_{u \to v}: \ \lceil \mathbf{Proc}_{u \to v} \rceil \times A^u \ \rightrightarrows \ A^{v\uparrow}$$

cam then be defined by

$$Univ_{u \rightarrow v} \equiv pe_{a, b, \hat{c}}.$$

Note that we have proved the Universal Function Theorem for $While^{RA*}$, not $While^{RA}$. We return to this point in Section 7.

6 While^{RA} semicomputability

The notion of recursive enumerability and *While* semicomputability was generalised to many-sorted algebras in [8]. We now consider the question: Is *While*^{*RA*} semicomputability equivalent to *While* semicomputability? In other words:

Given a $While^{RA}(\Sigma)$ procedure P and standard Σ -algebra A, is there a $While(\Sigma)$ procedure with the same halting set as P in A? (The concept of "halting set" will be explained below.) We will consider this question separately for $While^{RA(bool)}$, $While^{RA(nat)}$, and "full" $While^{RA}$.

6.1 $While^{RA}$ computability

Before exploring $While^{RA}$ semicomputability, we review $While^{RA}$ computability. We distinguish two types of $While^{RA}$ functions on A: (i) multi-valued functions,

$$F: A^u \rightrightarrows A^{v\uparrow},$$

(ii) single-valued functions, i.e., partial functions

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

Actually, a single-valued function is a special case of a multi-valued function.

Definition 6.1.1 (*While*^{*RA*} computability). Let $P : u \to v$ be a *While*^{*RA*} procedure.

- (i) A multi-valued function $F : A^u \rightrightarrows^+ A^{v\uparrow}$ is **While**^{RA} computable on A by P if $F = P^A$.
- (ii) A single-valued function $f : A^u \rightharpoonup A^v$ is **While**^{RA} computable on A by P if for any $a \in A^u$,

$$P^{A}(a) = \begin{cases} \{f(a)\} & \text{if } f(a) \downarrow \\ \{\uparrow\} & \text{if } f(a) \uparrow. \end{cases}$$

Before tackling the question about $While^{RA}$ semicomputability, we first ask whether random assignments enhance the computing power of While(A) for singlevalued functions on any standard algebra A, *i.e.*, for any single-valued $While^{RA}$ computable (partial) function $f : A^u \rightharpoonup A^v$, is f While computable?

The answer is "Yes". We will prove it as follows.

Theorem 6.1.2. For any $A \in StdAlg(\Sigma)$ and any single-valued function $f : A^u \rightharpoonup A^v$, if f is *While*^{*RA*} computable, then f is *While* computable.

Proof: By replacing all random assignments in the *While*^{*RA*} procedure with assignments of default values of the same sort using the Instantiation Assumption (2.1.10), we effectively construct a *While* procedure which must also compute f.

Remark 6.1.3. There are two notions of deterministic computation [9, Remark 3.2.6]: (i) strong deterministic computation, the common concept, in which each step of the computation is determinate; and (ii) weak deterministic computation, in which the output (or divergence) is uniquely determined by the input, but the steps in the computation are *not* necessarily determinate.

In this sense, a single-valued $While^{RA}$ function results from a weak deterministic computation. Hence Theorem 6.1.2 indicates that (weak) $While^{RA}$ determinism is equivalent to (strong) While determinism.

6.2 Definition of $While^{RA}$ Semicomputability

We first clarify the definition of $While^{RA}$ semicomputability for nondeterministic languages. Since there exist many computation sequences for a given input, we have two possible definitions of $While^{RA}$ semicomputability. We say that a relation R is $While^{RA}$ semicomputable if it is the *halting set* of a $While^{RA}$ procedure, which can be defined as *either* (1) the set of inputs for which *all* computation sequences halt, *or* (2) the set of inputs for which *some* computation sequence halts.

The second definition turns out to be more tractable mathematically, so we choose to work with it.

Definition 6.2.1. Let P be a *While*^{*RA*} procedure, with input variables a : u. The *halting set* of P on A is the set of tuples $a \in A^u$ such that when a is initialised to a, then execution of P halts for *some sequence of values* for the random assignments.

Definition 6.2.2. A relation on A is $While^{RA}$ semicomputable on A if it is the halting set of a $While^{RA}$ procedure on A.

Since we are only concerned with the *domains* or *halting sets* of procedures, we will ignore their output variables (or assume they have been re-labelled as auxiliary variables).

6.3 $While^{RA(bool)}$ Semicomputability

In this section, we will show that, given a $While^{RA(bool)}$ procedure

$$P \equiv \text{proc in a aux c begin } S \text{ end},$$
 (6.1)

(with a: u and c: w), there is a **While**^N procedure with the same halting set as P.

For $\mathbf{x} \equiv \mathbf{a}, \mathbf{c}$ in (6.1), we define a function *isleaf* $_{\mathbf{x}}^{A}(\ulcorner S \urcorner, a, n)$, which tests whether any leaves, indicating terminating computations, occur in the semantic computation tree for S with input a, by a depth of n.

Definition 6.3.1. The function

$$isleaf_{x}^{A}: \lceil Stmt_{x} \rceil \times A^{u} \times \mathbb{N} \implies \mathbb{B}$$

is defined by tail recursion:

Base case: $isleaf_{x}^{A}(\lceil S \rceil, a, 0) = ff.$ Inductive step: (i) for S atomic, $\textit{isleaf}_{\mathtt{x}}^{A}(\ulcorner S \urcorner, a, n+1) = \mathtt{t}$

(ii) for S not atomic, if First(S) is not a random assignment, then

$$\textit{isleaf}_{\mathtt{x}}^{A}(\ulcorner S\urcorner, a, n+1) = \textit{isleaf}_{\mathtt{x}}^{A}(\textit{rest}_{\mathtt{x}}^{A}(\ulcorner S\urcorner, a), \textit{ae}_{\mathtt{x}}^{A}(\textit{First}(\ulcorner S\urcorner, a)), n),$$

otherwise, if $First(S) \equiv b := ?$ then

$$\begin{aligned} \textit{isleaf}_{\mathtt{x}}^{A}(\lceil S \rceil, a, n+1) = & \textit{isleaf}_{\mathtt{x}}^{A}(\textit{rest}_{\mathtt{x}}^{A}(\lceil S \rceil, a), a\{\mathtt{b}/\mathtt{t}\}, n) \\ & \text{or } \textit{isleaf}_{\mathtt{x}}^{A}(\textit{rest}_{\mathtt{x}}^{A}(\lceil S \rceil, a), a\{\mathtt{b}/\mathtt{f}\}, n). \end{aligned}$$

where if $\mathbf{x} \equiv (\mathbf{x}_1, \cdots, \mathbf{x}_n)$, $a \equiv (a_1, \cdots, a_n)$ and $\mathbf{b} \equiv \mathbf{x}_i$ $(1 \le i \le n)$, then $a\{\mathbf{b}/\mathbf{t}\} = (a_1, \cdots, a_{i-1}, \mathbf{t}, a_{i+1}, \cdots, a_n)$, and similarly for $a\{\mathbf{b}/\mathbf{f}\}$.

Lemma 6.3.2. The function *isleaf*^A_x is *While* computable on A^N .

Proof: This follows from the *While* computability of *first*, *rest*^A_x and *ae*^A_x (Lemmas 5.3.1 and 5.3.2, and Theorem 5.3.3). \Box

Theorem 6.3.3 ($While^{RA(bool)}$ Semicomputability Theorem).

Let A be a standard Σ -algebra with the TEP. For any $While^{RA(bool)}$ procedure P, there is a $While^N$ procedure P' with the same halting set on A as P.

Proof: Here is an informal algorithm for P'. With P and S as in (6.1), and input a, test *isleaf*^A_x($\ulcorner S \urcorner$, $(a, \delta^w), n$) for $n = 0, 1, 2, \cdots$. Halt if and when this gives a result of **t**. More formally, P' can be defined as follows.

```
proc in a : u

aux n : nat, c : w, continue : bool

begin

n := 0;

continue := true;

while continue do

if isleaf_x^A(\ulcorner S\urcorner, (a, \delta^w), n)

then continue := false

else n := n + 1

fi

od

end
```

Since the function $isleaf_{x}^{A}$ is *While* computable on A^{N} by Lemma 6.3.2, its call in the above procedure can be eliminated, resulting in a *While*^N procedure (see Remark 5.3.1 (a)).

Remark 6.3.4. The proof actually gives an algorithm for a transformation

$$P \mapsto P'$$

of a $While^{RA(bool)}(\Sigma)$ procedure P to a $While^{N}(\Sigma)$ procedure P' with the same halting set as P, on any standard Σ -algebra with TEP. This transformation is uniform relative to the term evaluation subroutine.

Note the use of $While^N$, rather than While, semicomputability. We return to this point in Section 7. In any case, we have:

Corollary 6.3.5. For any N-standard Σ -algebra A with the TEP, $While^{RA(bool)}$ semicomputability is equivalent to While semicomputability on A.

6.4 $While^{RA(nat)}$ Semicomputability

In the this section, we turn to $While^{RA(nat)}$ semicomputability. The method in the proof of Theorem 6.3.3 cannot be applied directly to $While^{RA(nat)}$, since with random assignments to nat, the semantic computation tree has (computably) infinite branching, causing a problem with the definition of $isleaf_x^A$.

We solve this problem by *dovetailing* the traversal of nodes. So at stage n, we only consider the finite set of nodes that are not only of depth $\leq n$, but also (hereditarily, up to the root) among the first n children of the parent nodes. The predicate $isleaf_{\mathbf{x}}^{A}(\lceil S \rceil, n)$ is re-defined so as to search only these nodes at stage n. (We omit details.)

From this we can prove, in exactly the same way as for Theorem 6.3.3:

Theorem 6.4.1 ($While^{RA(nat)}$ Semicomputability Theorem).

Let A be an N-standard Σ -algebra with the TEP. For any $While^{RA(nat)}$ procedure P, there is a *While* procedure P' with the same halting set on A as P.

Corollary 6.4.2. For any N-standard Σ -algebra A with the TEP, $While^{RA(nat)}$ semicomputability is equivalent to While semicomputability on A.

Corollary 6.4.3. For any N-standard Σ -algebra A with the TEP, $While^{RA(nat,bool)}$ semicomputability is equivalent to While semicomputability on A.

Remark 6.4.4. By similar reasoning, the same result holds for languages $While^{RA(s)}$ over any standard A with the TEP, with random assignments on sort s only, where the carrier A_s is minimal, since the elements of A_s can then be (*While*) effectively enumerated.

$6.5 \quad While^{RA} \text{ Semicomputability}$

In this sections, we turn to (unrestricted) $While^{RA}$, and ask: for any standard (or N-standard) Σ -algebra A with the TEP, and $While^{RA}$ procedure P, is there a While procedure with the same halting set on A? We will show that the answer is negative in general. For our counterexample, we actually work with computations over A^* .

First we must introduce another notion of semicomputability: *projective While semicomputability*. The definition is taken from [8].

Definition 6.5.1. *R* is projectively $While(\Sigma)$ semicomputable on *A* if, and only if, *R* is a projection of a *While*(Σ) semicomputable relation on *A*.

Definition 6.5.2. *R* is projectively **While**^{*}(Σ) semicomputable on *A* if, and only if, *R* is a projection on *A* of a **While**(Σ^*) semicomputable relation on *A*^{*}.

Lemma 6.5.3. On any standard Σ -algebra with the TEP,

 $While^{RA*}$ semicomputability \iff projective $While^*$ semicomputability.

Proof: A proof is given in [8, Theorem 5.75], using Engeler's Lemma. \Box

Lemma 6.5.4. On any standard Σ -algebra with the TEP,

projective $While^*$ semicomputability $\stackrel{\longleftarrow}{\Rightarrow} While^*$ semicomputability.

Proof: A counterexample for the direction " \Longrightarrow " is given in [8, § 6.2].

Theorem 6.5.5 (*While^{RA}* Semicomputability Theorem). There is a standard signature Σ , and a standard Σ -algebra A, and a relation on A^* which is *While^{RA}* semicomputable, but not *While* semicomputable.

Proof: By Lemmas 6.5.3 and 6.5.4 applied to A^* .

Remark 6.5.6. (a) The same counterexample for Lemma 6.5.4 also works for Theorem 6.5.5.

(b) This counterexample was only obtained by considering array algebras. We return to this point in the next section.

7 Conclusion

In this paper, we have studied *nondeterministic* languages over abstract many-sorted algebras, specifically (1) the $While^{RA}$ language, which extends the While language with random assignments, as well as its sublanguage $While^{RA(nat/bool)}$ formed by random assignments to nat and/or bool; (2) the guarded command language GC, which is equivalent to $While^{RA(bool)}$. The investigation was made in two directions: generalizing the Universal Function Theorem for deterministic languages in [8] to the nondeterministic case, and examining semicomputability for such nondeterministic languages.

An interesting aspect of this investigation is the extension of algebraic operational semantics [8] by the use of many-valued semantic functions. Instead of the computation sequences used in the deterministic languages, we use semantic computation trees to indicate the computations of $While^{RA}$ statements, and multi-valued functions to represent the semantics of $While^{RA}$ procedures.

Many questions concerning these issues remain open. To name some of them: (1) For the nondeterministic $While^{RA}$ language, does the UFT hold for procedures without arrays variables? (See Theorem 5.5.3 and the comment following it.) (2) Does the result

$$While^{RA(bool)}$$
 semicomputability \iff $While$ semicomputability

also hold for non-N-standard algebras? (See Corollary 6.3.5.)(3) Our counterexample to the equivalence

$While^{RA}$ semicomputability \iff While semicomputability

(Theorem 6.5.5) uses random assignments to array variables (see Remark 6.5.6 (b)). Can one find a counterexample with random assignments to simple variable only? (4) In order to focus on the problems at hand, we have avoided issues of partiality by assuming that the algebras are total, i.e., all the function symbols of the signature are interpreted as total functions. With partial algebras (as studied, for example, in [9] and [5]), subtle problems arise in connection with the semantics. We predict that our main results hold also in that case; however, this should be investigated.

Appendix. Proofs omitted from previous sections

A1. Proof of Lemma 3.4.3.

Proof:

(a) It is trivial by the definition of $CompTree_bdd^A$.

(b) We prove it by two cases.

Case 1: $S_1 \in AtSt$.

By the definition, $CompTree_bdd^A(S, \sigma, n)$ is formed by attaching to the root $\{\sigma\}$ the subtree $CompTree_bdd^A(Rest^A(S, \sigma), \sigma', n-1)$, for each $\sigma' \in CompTree^A(S, \sigma)$. Since in the case that S_1 is atomic,

$$CompTree^{A}(S,\sigma) = \langle First(S) \rangle^{A} \sigma = \langle First(S_{1}) \rangle^{A} \sigma = \langle S_{1} \rangle^{A} \sigma,$$

and $\operatorname{Rest}^{A}(S, \sigma) = S_{2}$, what we want turns to be that $\operatorname{CompTree_bdd}^{A}(S, \sigma, n)$ is formed by attaching to the root $\{\sigma\}$ the subtree $\operatorname{CompTree_bdd}^{A}(S_{2}, \sigma', n-1)$, for each $\sigma' \in \langle S_{1} \rangle^{A} \sigma$.

By (a), **CompTree_bdd**^A(S_1, σ, n) is a one-step tree with each leaf $\sigma' \in \langle S_1 \rangle^A \sigma$, with the depth of 1. Therefore (b) is proved for this case.

Case 2: S_1 is not atomic. We use induction on n to prove this.

Base case: n = 1.

By the definition, $CompTree_bdd^A(S, \sigma, 1)$ is formed by attaching to the root $\{\sigma\}$ the subtree $CompTree_bdd^A(Rest^A(S, \sigma), \sigma', 0)$, for each $\sigma' \in CompStep^A(S, \sigma)$, i.e., attaching to the root $\{\sigma\}$ the node $\{\sigma'\}$ for each σ' in $CompStep^A(S, \sigma)$.

Since S_1 is not atomic, $CompTree_bdd^A(S_1, \sigma, 1)$ has no leaf. Therefore, (b) amounts to saying that $CompTree_bdd^A(S, \sigma, 1)$ is formed by

CompTree_bdd^A($S_1, \sigma, 1$), i.e., attaching to the root { σ } the node { σ' } for each σ' in **CompStep**^A(S_1, σ).

Since $S \equiv S_1; S_2,$

$$CompStep^{A}(S, \sigma) = \langle First(S) \rangle^{A} \sigma = \langle First(S_{1}) \rangle^{A} \sigma = CompStep^{A}(S_{1}, \sigma).$$

So (b) is proved for the base case.

Inductive step:

Induction hypothesis: Assume that $CompTree_bdd^A(S, \sigma, n)$ is formed by attaching subtrees $CompTree_bdd^A(S_2, \sigma', n-m)$ to each leaf σ' of $CompTree_bdd^A(S_1, \sigma, n)$, where m is the depth of σ' in $CompTree_bdd^A(S_1, \sigma, n)$.

We want to prove that $CompTree_bdd^A(S, \sigma, n + 1)$ is formed by attaching subtrees

 $CompTree_bdd^{A}(S_{2}, \sigma', n+1-m)$ to each leaf σ' of $CompTree_bdd^{A}(S_{1}, \sigma, n+1)$, where m is the depth of σ' in $CompTree_bdd^{A}(S_{1}, \sigma, n+1)$.

By the definition, $CompTree_bdd^A(S, \sigma, n + 1)$ is formed by attaching to the root $\{\sigma\}$ the subtree $CompTree_bdd^A(Rest^A(S, \sigma), \sigma', n)$, for each $\sigma' \in CompStep^A(S, \sigma)$. And in this case we have

$$CompStep^{A}(S,\sigma) = \langle First(S) \rangle^{A} \sigma = \langle First(S_{1}) \rangle^{A} \sigma = CompStep^{A}(S_{1},\sigma).$$

and $\operatorname{Rest}^{A}(S, \sigma) = \operatorname{Rest}^{A}(S_{1}, \sigma); S_{2}.$

Then, $CompTree_bdd^A(S, \sigma, n+1)$ is formed by attaching to the root $\{\sigma\}$ the subtree

 $CompTree_bdd^A(Rest^A(S_1, \sigma); S_2, \sigma', n)$, for each $\sigma' \in CompStep^A(S_1, \sigma)$.

By induction hypothesis, $CompTree_bdd^A(Rest^A(S_1, \sigma); S_2, \sigma', n)$ is formed by attaching the subtree $CompTree_bdd^A(S_2, \sigma'', n - m)$ to each leaf σ'' of

 $CompTree_bdd^A(Rest^A(S_1, \sigma), \sigma', n)$, for each $\sigma' \in CompStep^A(S_1, \sigma)$, where m is the depth of $\{\sigma''\}$ in

 $CompTree_bdd^A(Rest^A(S_1, \sigma), \sigma', n).$

Now, **CompTree_bdd**^A(S, σ , n + 1) is formed by,

(i) attaching to the root $\{\sigma\}$, $CompTree_bdd^A(Rest^A(S_1, \sigma), \sigma', n)$, for each $\sigma' \in CompStep^A(S_1, \sigma)$.

(*ii*) attaching the bounded computation tree $CompTree_bdd^A(S_2, \sigma'', n - m)$ to each leaf σ'' of $CompTree_bdd^A(Rest^A(S_1, \sigma), \sigma', n)$, where *m* is the depth of σ'' in

 $CompTree_bdd^A(Rest^A(S_1, \sigma), \sigma', n).$

Obviously, step (i) is the definition of $CompTree_bdd^A(S_1, \sigma, n+1)$, and the depth of leaf σ'' in $CompTree_bdd^A(S_1, \sigma, n+1)$, we say, m' = m+1, where m is the depth of σ'' in $CompTree_bdd^A(Rest^A(S_1, \sigma), \sigma', n)$.

Therefore we reach the result that $CompTree_bdd^A(S, \sigma, n + 1)$ is formed by attaching subtrees $CompTree_bdd^A(S_2, \sigma'', n + 1 - m)$ to each leaf σ'' of the subtree $CompTree_bdd^A(S_1, \sigma, n + 1)$, where m is the depth of σ'' in the subtree $CompTree_bdd^A(S_1, \sigma, n + 1)$.

(c) Since $S \equiv \text{if } b$ then S_1 else S_2 fi,

$$CompStep^{A}(S, \sigma) = \langle First(S) \rangle^{A} \sigma = \langle skip \rangle^{A} \sigma = \{\sigma\}$$

and

$$Rest^{A}(S,\sigma) = S_{i}$$

where if $\llbracket b \rrbracket^A \sigma = \mathsf{t}$ then i = 1, else i = 2.

By the definition, (c) is true.

(d) Since $S \equiv$ while b do S_1 od.

$$CompStep^{A}(S,\sigma) = \langle\!\langle First(S) \rangle\!\rangle^{A} \sigma = \langle\!\langle skip \rangle\!\rangle^{A} \sigma = \{\sigma\}$$

and

$$\operatorname{\operatorname{\operatorname{Rest}}}^{A}(S,\sigma) = \begin{cases} S_{1}; S & \text{if } \llbracket b \rrbracket^{A} \sigma = \mathfrak{t} \\ \operatorname{skip} & \text{if } \llbracket b \rrbracket^{A} \sigma = \mathfrak{f}. \end{cases}$$

By the definition, it is easy to prove (d) holds.

A2. Proof of Theorem 3.5.1.

Proof:

Each part of this proof uses the corresponding part of Lemma 3.4.3.

(a) This is trivial.

(b) From Lemma 3.4.3 (b), take the "union" over *n* for all **CompTree_bdd**^A(S, σ, n), then we get that the set of nodes of **CompTree**^A(S, σ) is formed by attaching the set of nodes of **CompTree**^A(S_2, σ') to each leaf { σ' } of **CompTree**^A(S_1, σ).

Hence, the leaves of $CompTree^{A}(S, \sigma)$ are formed from the leaves of computation tree

 $CompTree^{A}(S_{2}, \sigma')$ for each leaf $\{\sigma'\}$ of $CompTree^{A}(S_{1}, \sigma)$, *i.e.*, $[S_{2}]^{A}([S_{1}]^{A}\sigma)$.

Particularly, if there is an infinite path in $CompTree^{A}(S_{1}, \sigma)$ or any subtree $CompTree^{A}(S_{2}, \sigma')$ for each leaf $\{\sigma'\}$ of $CompTree^{A}(S_{1}, \sigma)$, the extension of this path in $CompTree^{A}(S, \sigma)$ is also an infinite path.

(c) From Lemma 3.4.3 (c), take the "union" over *n* for all **CompTree_bdd**^A(S, σ, n), then we get that the nodes of **CompTree**^A(S, σ) is formed by attaching to the root $\{\sigma\}$, the nodes of the subtree **CompTree**^A(S_i, σ), where if $[\![b]\!]^A \sigma = \mathfrak{t}$ then i = 1, else i = 2.

So the leaves of $CompTree^{A}(S, \sigma)$ are formed from the leaves of $CompTree^{A}(S_{i}, \sigma)$, where if $[\![b]\!]^{A}\sigma = t$ then i = 1, else i = 2.

Also, if there exists an infinite path in $CompTree^{A}(S_{i}, \sigma)$, where if $[\![b]\!]^{A}\sigma = t$ then i = 1, else i = 2, there must be an infinite path in $CompTree^{A}(S, \sigma)$, by extending the infinite path in $CompTree^{A}(S_{i}, \sigma)$ one step up to the root $\{\sigma\}$.

Therefore, we proved (c) is true.

(d) From Lemma 3.4.3 (d), take the "union" over n for all $CompTree_bdd^A(S, \sigma, n)$, then we get that the nodes of $CompTree^A(S, \sigma)$ are formed by attaching to the root $\{\sigma\}$:

- (*i*) the nodes of the subtree $Comp Tree^{A}(S_1; S, \sigma)$, if $\llbracket b \rrbracket^{A} \sigma = tt$;
- (*ii*) otherwise, the leaf $\{\sigma\}$.

So the leaves of **CompTree**^A(S, σ) are formed from,

- (i) the leaves of the subtree $Comp Tree^{A}(S_{1}; S, \sigma)$, if $\llbracket b \rrbracket^{A} \sigma = \texttt{tt}$;
- (*ii*) otherwise, the leaf $\{\sigma\}$.

If there exists an infinite path in $Comp Tree^{A}(S_{1}, \sigma)$, when $\llbracket b \rrbracket^{A} \sigma = \mathfrak{t}$, there must be an infinite path in $Comp Tree^{A}(S, \sigma)$, by extending the infinite path in the subtree $Comp Tree^{A}(S_{1}, \sigma)$ one step up to the root $\{\sigma\}$.

Therefore, (d) has been proved.

References

- DIJKSTRA, E. W. A Discipline of Programming. Prentice Hall, Englewood Cliffs, New Jersey, 1976.
- [2] HAREL, D. On folk theorems. In Communications of the ACM 23, vol. 7. 1980, pp. 379–389.
- [3] JIANG, W. Universality and semicomputability for non-deterministic programming languages over abstract algebras. Master's thesis, Department of Computing and Software, McMaster University, 2002. Technical Report CAS 03-01-JZ, Department of Computing and Software, McMaster University, 2003.
- [4] KOSTER, J. Relative strengths of while-programs with and without counters and stacks. Master's thesis, Department of Computing and Software, McMaster University, 2002. Technical Report CAS 02-05-JZ, Department of Computing and Software, McMaster University, 2002.
- [5] Luo, L. Specifiability and computability of functions by equations on partial algebras. Master's thesis, Department of Computing and Software, McMaster University, 2003. Technical Report CAS 03-07-JZ, Department of Computing and Software, McMaster University, 2003.
- [6] MIRKOWSKA, G. Algorithmic logic and its applications. PhD thesis, University of Warsaw, 1972.
- [7] TUCKER, J. V., AND ZUCKER, J. I. Computability by 'while' programs on topological partial algebras. *Theoretical Computer Science 219* (1999), 379–420.
- [8] TUCKER, J. V., AND ZUCKER, J. I. Computable functions and semicomputable sets on many-sorted algebras. In *Handbook of Logic in Computer Science*, D. M. G. S. Abramsky and T. S. E. Maibaum, Eds., vol. 5. Oxford University Press, 2000, pp. 317–523.
- [9] TUCKER, J. V., AND ZUCKER, J. I. Abstract versus concrete computation on metric partial algebras. ACM Transactions on Computational Logic 5 (2004), 611–668.

- [10] TUCKER, J. V., AND ZUCKER, J. I. Computable total functions on metric algebras, algebraic specifications and dynamical systems. *Journal of Logic and Algebraic Programming 62* (2005), 71–108.
- [11] TURING, A. M. On computable numbers, with an application to the entscheidungs problem. Proceedings of the London Mathematical Society 42 (1936), 230–265. With correction, *ibid.*, 43, 544–546, 1937. Reprinted in *The Undecidable*, M. Davis, ed., Raven Press, 1965.
- [12] WANG, Y. Semantics of non-deterministic programs and the universal function theorem over abstract algebras. Master's thesis, Department of Computing and Software, McMaster University, 2001. Technical Report CAS 01-03-JZ, Department of Computing and Software, McMaster University, 2001.