SEMANTICS AND UNIVERSALITY

OF NON-DETERMINISM

SEMANTICS OF NON-DETERMINISTIC PROGRAMS
AND
THE UNIVERSAL FUNCTION THEOREM
OVER ABSTRACT ALGEBRAS

BY

YUAN WANG, B.SC.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree Master of Science

McMaster University

MASTER OF SCIENCE (2001)	McMaster University
(Department of Computing and Software)	Hamilton, Ontario

TITLE: Semantics of non-deterministic programs and the Universal Function Theorem over abstract algebras

AUTHOR: Yuan Wang, B.Sc. (Beijing Institute of Technology)

SUPERVISOR: Professor Jeffery I. Zucker

NUMBER OF PAGES: vii, 99

ABSTRACT

Data types containing infinite data, such as the real numbers, functions, and bit streams, can be modeled by abstract many-sorted algebras over suitable signatures. The computability theory for deterministic programs over such algebras has been studied extensively; as a complementary investigation, we study the formal semantics and computability theory for various non-deterministic languages.

The ***ND*** programming language studied in this thesis combines the ***While*** programming language extended with *random assignment*, and the *Guarded Command Language* ***GC*** of Dijkstra. A semantic theory for ***ND*** is developed following *algebraic operational semantics*, using *semantic computation trees* labeled with states instead of the *computation sequences* used in the deterministic case. The semantics of an ***ND*** procedure is then *the set of states at all leaves of its tree, together with the* '↑' (divergence symbol) *if the tree has an infinite path.*

Since ***GC*** has (i) *finite non-determinism* (i.e. the semantic computation tree for a ***GC*** statement is finitely branching), and (ii) *localization of computation* (i.e., the output is always in the subalgebra generated by the input), the whole computation procedure can be

represented using Gödel numbering. Hence (assuming a “*term evaluation property*” for the given algebra) we can prove a *Universal Function Theorem* for ***GC***. This technique fails for the full ***ND*** language with its infinite non-determinism and failure of localization of computation.

ACKNOWLEDGEMENTS

I would like to thank Dr. Zucker, my supervisor, for his valuable guidance in the preparation of this thesis, and for his substantial help and support all the way, in my study and in my life.

Thanks to Dr. Emil Sekerinski for his useful comments, and to the other members of my Examination Committee for all their assistance. Thanks to Laurie, Sara, Chris, Derek, John and all the others for always being there whenever I needed help. Thanks to YuDong Tang and ZhiFeng Sun for their wonderful friendship and pleasant cooperation, and to all the other graduate students for their help during these two years.

Many thanks to my parents for all the love and support in so many ways. Special thanks to Lei Xu for her help, understanding and encouragement.

TABLE OF CONTENTS

CHAPTER ONE

CHAPTER TWO

CHAPTER THREE

CHAPTER FOUR

REPRESENTATIONS AND COMPUTABILITY ON A^N OF SEMANTIC FUNCTIONS

CHAPTER ONE

INTRODUCTION

The semantics and computability issues of the deterministic ***While*** programming language has been studied in the article, *Computable functions and semicomputable sets on many-sorted algebras*, J. V. Tucker and J. I. Zucker [7]. Now we want to focus on these issues of the non-deterministic programming languages, involving so-called “don’t care cases”, as a complementary study to the deterministic case.

In fact, non-deterministic programs have many practical advantages over deterministic ones. For example, let us take a look at the following deterministic program, which computes the absolute value of the input.

```
proc
    in x
    out y
begin
    if x > 0 then y := x
    elseif x = 0 then y := 0
    else x < 0 then y := -x fi
```

```
end
```

Is this program too *clumsy*?

We can use a better, non-deterministic, program to compute this function “absolute” as follows (actually, this is a ***Guarded Command language*** program):

```
proc
    in x
    out y
begin
    if x ≥ 0 → y := x | x ≤ 0 → y := -x fi
end
```

In this program, the non-deterministic case is $\mathbf{x} = 0$, and $\mathbf{y}$ can be either $\mathbf{x}$ or $-\mathbf{x}$ at this case. What is more, we leave the decision for the output $\mathbf{y}$ to the system at this case.

We can easily see that this non-deterministic program is much more *flexible, concise, convenient and powerful* than the former one. And that is also a big reason why we study the semantics and computability of non-determinism.

Note that even deterministic languages such as Pascal and C have non-deterministic aspects; for example, the *read* command in Pascal functions like a random assignment, with regard to postconditions.

In this chapter, we will introduce the non-deterministic languages, and outline our investigation of them.

1.1 *While* and *While*^{RA} programming language

Firstly, let us recall the simple imperative model, ***While***(Σ) programming language [7] for a signature Σ, whose basic computations on algebra ***A*** are performed by concurrent assignments

$$\mathbf{x}_1, \ldots, \mathbf{x}_n := t_1, \ldots, t_n$$

where $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are program variables and $t_1, \ldots, t_n$ are Σ-terms or expressions of the corresponding types ($1 \leq i \leq n$).

The control and sequencing of the basic computations are performed by the three constructs to form new statements from given statements $\boldsymbol{S}_1$, $\boldsymbol{S}_2$ and $\boldsymbol{S}$, and boolean test b:

(i) *sequential composition*: $\boldsymbol{S}_1; \boldsymbol{S}_2$,

(ii) *conditional*: **if** b **then** $\boldsymbol{S}_1$ **else** $\boldsymbol{S}_2$ **fi**,

(iii) *iteration*: **while** b **do** $\boldsymbol{S}$ **od**.

Now we extend this language with the *random assignment* $\mathbf{x}$:= **?**, which we call ***While***^{RA}, our first *non-deterministic* model, for variables $\mathbf{x}$ of every sort of Σ.

1.2 *Guarded Command Language*

Our second *non-deterministic* programming model is the so-called "*Guarded Command Language*" (***GC***) due to Edsger W. Dijkstra [3].

We give the notion of a "guarded command", whose syntax is given by:

$$b \rightarrow \boldsymbol{S}$$

where b is a boolean test and $\boldsymbol{S}$ is a statement.

The constructs of ***GC*** are derived from these guarded commands as follows, (with $k \geq 0$):

(i) the *guarded command conditional* construct,

$$\mathbf{if}\ b_1 \rightarrow \boldsymbol{S}_1 \mid \ldots \mid b_k \rightarrow \boldsymbol{S}_k\ \mathbf{fi}$$

(ii) the *guarded command iteration* construct

$$\mathbf{do}\ b_1 \rightarrow \boldsymbol{S}_1 \mid \ldots \mid b_k \rightarrow \boldsymbol{S}_k\ \mathbf{od}$$

together with *concurrent assignment* and *sequential composition* as before. (Note that we do not have random assignment in ***GC***.)

In particular, if $k = 0$, we define the two guarded command constructs as

$$\mathbf{if\ fi} \equiv \mathbf{halt}$$

$$\mathbf{do\ od} \equiv \mathbf{skip}$$

1.3 *ND* programming language and Semantics of *ND*

For the purpose of finding a uniform method to develop the semantics for both *non-deterministic* programming languages, we combine them into one so-called programming language ***ND*** (for ***Non-Determinism***), which also combines their constructs as follows,

(i) *concurrent assignment*,

(ii) *random assignment*,

(iii) *sequential composition*,

(iv) the *guarded command conditional*,

(v) the *guarded command iteration*.

To compute functions on ***A***, we formulate a simple class of function procedures based on ***ND*** statements of the form

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

where **a**, **b**, **c** are lists of input, output and auxiliary variables, respectively, and ***S*** is an ***ND*** statement.

The following diagram shows their relationship:

$$\begin{tikzcd}
& \boldsymbol{GC} \arrow[dr] & \\
\boldsymbol{While} \arrow[ur] \arrow[dr] & & \boldsymbol{ND} \\
& \boldsymbol{While}^{\boldsymbol{RA}} \arrow[ur] &
\end{tikzcd}$$

The *operational semantics* of an ***ND*** statement is a function that, given an initial state, constructs a *semantic computation tree* labeled with states. Then, the input/output (i/o) semantics of an ***ND*** statement is the set of states at all leaves of the semantic computation tree, together with ‘↑’ (divergence) if there exists an infinite path in this tree.

Thus, we interpret statements as *many-valued state transformations*, and function procedures as *many-valued functions* on any standard algebra ***A***. Our approach follows the *algebraic operational semantics* (first developed systematically in [5] and used in [7, section 3.4]). The main difference is that we use the *semantic computation tree* ***CompTree***, defined via a function ***CompTreeStage***(***S***, *σ*, *n*) representing the first *n* steps of ***CompTree***, instead of the *computation sequence* ***Comp*** in [7].

As a result, we give a uniform semantics for ***ND*** statements and procedures, by defining the *operational semantics* and the *semantic computation tree* in Chapter 3.

1.4 Universal Function Theorem for *GC*

We are very interested in whether or not a given programming language ***L*** over a signature Σ satisfies a *Universal Function Theorem* (***UFT***). This means answering the following questions:

Let ***A*** *be a Σ-algebra. Does there exist a universal* **L**(Σ) *program* $\mathbf{U}_{prog}$ *that can simulate and perform the computations of all programs in* **L**(Σ) *on all inputs from* ***A****? Is there a universal* **L**(Σ) *procedure* $\mathbf{U}_{proc} \in$ ***Proc***(Σ) *that can compute all the* **L** *computable functions on* ***A****?*

We have not been able to answer this question for the full non-deterministic language ***ND***, but only for the sub-language ***GC***.

This question involves representing faithfully the syntax and semantics of ***GC*** computations using functions on ***A***, and we need the techniques of Gödel numbering, state (and state set) representation, symbolic computations on terms, and localization of computation (explained below).

Because of the structure of the guarded command statements in ***GC***, its semantic computation tree is only *finitely branching*. Then, we also have the following two important properties for the *semantic computation tree* of ***GC*** statements:

(i) at each step, we only have finitely many leaves, which can all be coded by a single Gödel number,

(ii) localization of computation: the output is always in the subalgebra generated from the input.

Moreover, since the *term evaluation function* is ***While*** computable in most commonly used algebras such as semi-groups, groups, rings, boolean algebras and subalgebras [7, Examples 4.5], it is reasonable to assume the *term evaluation property* ([7, Definition 4.4]). Then, we can show that

*for any given Σ-algebra **A**, there is a universal **GC** procedure over **A**.*

Unfortunately, the same technique does not work for the ***While***^{RA} programming language because of (i) the infinite branching of its computation trees, and (ii) the fact that the output is *not* necessarily in the subalgebra generated by the input. In fact, we do not even know whether the ***UFT*** hold for ***While***^{RA}.

1.5 Background: significance of the Universal Function Theorem

The origin of the ***UFT*** lies in the work of Turing [9] who (in the context of his Turing machine formalism for classical computation theory on strings over a finite alphabet) proved the existence of a universal Turing machine.

The ***UFT*** in [7] can be viewed as an extension of this result to *abstract data types*, with algorithms formalized as *deterministic* ***While*** *programs*.

The ***UFT*** presented here can be viewed as a further extension of this result, to *non-deterministic programming languages*.

1.6 Overview of the chapters

Here is the structure of this thesis.

We begin, in Chapter 1, by introducing the non-deterministic languages (***While*** and ***While***^{RA} in section 1.1, ***GC*** in section 1.2, and ***ND*** in section 1.3) and outline our investigation of them in section 1.4.

In Chapter 2, we define some basic algebraic concepts, such as signatures (in section 2.1) and algebras, and establish notations. The study in this thesis is based on *standard* and ***N***-*standard algebras*, studied in sections 2.3 and 2.4.

In Chapter 3, we will study the syntax and semantics of ***ND*** on standard algebras by means of imperative programming models. We start by defining the non-deterministic programming language ***ND*** = ***ND***(Σ), which combines the programming language ***While*** extended with ‘*random assignment*’ and ‘***Guarded Command Language***’, and may be interpreted on any many-sorted Σ-algebra.

We will define in detail the abstract syntax (in section 3.1) and semantics of this language (in section 3.2 – 3.6). Our approach follows the algebraic operational semantics defined in [7, section 3.4]; however, we introduce a semantic computation tree for the semantics of ***ND*** statements, instead of the computation sequence used in the deterministic case [7]. Then, the semantics of an ***ND*** statement is, the set of states at all leaves of the semantic computation tree, together with ‘↑’ (divergence) if there exists an infinite path in this tree.

Then, we give a definition for ***ND*** computable functions in two cases, one for multi-valued functions and the other for single-valued functions (see Definition 3.14).

In Chapter 4, we prove the Universal Function Theorem for ***GC***, assuming a “*term evaluation property*” for the given algebra.

In section 4.1 – 4.9, we will represent the semantic functions defined in Chapter 3, using the techniques of Gödel numbering, state (and state set) representations, symbolic computations on terms. In section 4.10, we study the computability of all the semantic representing functions by assuming the *term evaluation property*. In section 4.11, we prove the Universal Function Theorem for ***GC*** on ***A***. This makes use of (i) finite branching of the semantic computation tree for ***GC***, allowing its representation by Gödel numbering, and (ii) localization of computation. However, this theorem fails for the full

ND language with its infinite non-determinism (from ***While***^{RA}), where neither (i) nor (ii) holds.

Finally, in the Appendix, we give some details of the proofs of the important theorems and lemmas in Chapter 1 – 4. Most of them are proved by structural induction, and some of them involve interesting techniques.

CHAPTER TWO

SIGNATURES AND ALGEBRAS

In this section, we will define some basic algebraic concepts, such as signatures and algebras, and establish notations. We will use many-sorted algebras equipped with booleans, which we call *standard algebras*. Sometimes we use algebras with the natural numbers as well, which we call ***N****-standard algebras*. This section is essentially taken from [7, section 2].

2.1 Signatures

Definition 2.1 (Many-sorted signatures).

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

$$\boldsymbol{F} : s_1 \times \cdots \times s_m \to s.$$

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

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

Definition 2.2 (Product types over Σ).

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

A Σ-algebra $\boldsymbol{A}$ has, for each sort s of Σ, a non-empty set $\boldsymbol{A}_s$, called the *carrier of sort* s, and for each Σ-function symbol $\boldsymbol{F} : s_1 \times \cdots \times s_m \to s$, a function

$$\boldsymbol{F}^{\boldsymbol{A}} : \boldsymbol{A}_{s_1} \times \ldots \times \boldsymbol{A}_{s_m} \to \boldsymbol{A}_s .$$

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

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

Thus $x \in \boldsymbol{A}^u$ if, and only if, $x = (x_1, \ldots, x_m)$, where $x_i \in \boldsymbol{A}_{s_i}$ for $i = 1, \ldots, m$. So each Σ-function symbol $\boldsymbol{F} : u \to s$ has an interpretation $\boldsymbol{F}^{\boldsymbol{A}} : \boldsymbol{A}^u \to \boldsymbol{A}_s$. If u is empty, i.e., $\boldsymbol{F}$ is a constant symbol, then $\boldsymbol{F}^{\boldsymbol{A}}$ is an element of $\boldsymbol{A}_s$.

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

Assumption 2.4

The algebras $\boldsymbol{A}$ are total, i.e., $\boldsymbol{F}^{\boldsymbol{A}}$ is total for each Σ-function symbol $\boldsymbol{F}$.

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

We will use the following perspicuous notation for signatures Σ:

signature	Σ	
sorts		
	$\ldots$	
	$s,$	$(s \in \boldsymbol{Sort}(\Sigma))$
	$\ldots$	
functions	$\ldots$	
	$\boldsymbol{F} : s_1 \times \cdots \times s_m \to s,$	$(\boldsymbol{F} \in \boldsymbol{Func}(\Sigma))$
	$\ldots$	
end		

and for Σ-structures $\boldsymbol{A}$:

algebra $\boldsymbol{A}$
Carriers
…
A_s, $(s \in \boldsymbol{Sort}(\Sigma))$
…
functions …
$F^A : A_{s_1} \times \ldots \times A_{s_m} \to A_s$, $(F \in \boldsymbol{Func}(\Sigma))$
…
end

Examples 2.5[1]

(a) The algebra of natural N_0 = (**N**; 0, **succ**) has a signature containing the sort **nat** and the function symbols 0 : → **nat** and **succ** : **nat** → **nat**. We can display this signature thus:

signature $\Sigma(N_0)$
sorts **nat**
functions 0 : → **nat**,
succ : **nat** → **nat**
end

and the algebra thus:

algebra N_0
carriers **N**
functions 0 : → **N**,
succ : **N** → **N**
end

---

[1] Refer to [7, section 2.1] for more examples.

from which the signature can be inferred. Below, we will often display the algebra instead of the signature.

(b) The ring of reals $\boldsymbol{R}_0 = (\mathbf{R};\ 0,\ 1,\ +,\ -,\ \times)$ has a carrier $\mathbf{R}$ of sort **real**, and can be displayed as follows:

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

2.2 Terms

For details, we refer to [4, section 1 and 2]. Here we give the definition for default terms, which will be used in the following sections.

Definition 2.6 (Default terms; default values).[2]

(a) For each sort s, we pick a closed term of sort s, and 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 tuple of type u*, written $\boldsymbol{\delta}^u$, is the tuple of default terms $(\boldsymbol{\delta}^{s_1}, \ldots, \boldsymbol{\delta}^{s_m})$.

---

[2] The assumption that this is always possible is called the *Instantiation Assumption* in [7, Assumption 2.13].

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

2.3 Adding booleans: Standard signatures and algebras

A very important signature for our purposes is the signature of *booleans*:

signature Σ(***B***)
sorts **bool**
functions **true, false :** → **bool,**
and, or : $\mathbf{bool}^2$ → **bool,**
not: bool → **bool**
end

The algebra ***B*** of booleans, with signature Σ(***B***), has the carrier **B** = {**tt, ff**} of sort **bool**, and, as constants and functions, the standard interpretations of the function and constant symbols of Σ(***B***). Thus, for example, $\mathbf{true}^{\boldsymbol{B}}$ = **tt** and $\mathbf{false}^{\boldsymbol{B}}$ = **ff**.

We are interested in those signatures and algebras which contain Σ(***B***) and ***B***.

Definition 2.7 (Standard signatures and algebras).

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

(i) Σ(***B***) ⊆ Σ, and

(ii) the function symbols of Σ include a conditional

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

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

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

for certain sorts s of Σ, called equality sorts.

(b) Given a standard signature Σ, a Σ-algebra $\boldsymbol{A}$ is a *standard algebra* if

(i) It is an expansion of $\boldsymbol{B}$, and

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

$$\mathbf{if}_s(b,x,y) = \begin{cases} x & \text{if } b = \mathbf{tt} \\ y & \text{if } b = \mathbf{ff} \end{cases}$$

and $\mathbf{eq}_s$ is interpreted as the *identity* on each equality sort s.

Remark 2.8

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

Examples 2.9

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

(b) The standard algebra of naturals $\boldsymbol{N}$ is formed by standardizing the algebra $\boldsymbol{N}_0$ of Example 2.5 (a), with **nat** as an equality sort, and, further, adjoining the order relation $\mathbf{less_{nat}}$ on $\mathbf{N}$:

algebra $\boldsymbol{N}$
import $\boldsymbol{N}_0$, $\boldsymbol{B}$
functions $\mathbf{if_{nat}} : \mathbf{B} \times \mathbf{N}^2 \to \mathbf{N}$,
$\mathbf{eq_{nat}}, \mathbf{less_{nat}} : \mathbf{N}^2 \to \mathbf{B}$
end

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

algebra $\boldsymbol{R}$
import $\boldsymbol{R}_0$, $\boldsymbol{B}$
functions $\mathbf{if_{real}} : \mathbf{B} \times \mathbf{R}^2 \to \mathbf{R}$,
$\mathbf{eq_{real}}: \mathbf{R}^2 \to \mathbf{B}$
end

(d) Refer to [7, section 2.4] for more examples of *standard algebras*.

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

Assumption 2.10 (Standardness).

The signature Σ and the Σ-algebra ***A*** are standard.

We let ***StdAlg***(Σ) denote the class of all standard Σ-algebras.

2.4 Adding counters: *N*-standard signatures and algebras

Definition 2.11

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

$$\begin{array}{rll} \mathbf{0}: & & \to \mathbf{nat} \\ \mathbf{S}: & \mathbf{nat} & \to \mathbf{nat} \\ \mathbf{less_{nat}}: & \mathbf{nat}^2 & \to \mathbf{bool} \end{array}$$

as well as the *conditional* $\mathbf{if_{nat}}$ and the *equality operator* $\mathbf{eq_{nat}}$ on **nat**.

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

Definition 2.12

(a) The ***N**-standardization* Σ^N of a standard signature Σ is formed by adjoining the sort **nat** and the operations **0**, **S**, $\mathbf{eq_{nat}}$, $\mathbf{less_{nat}}$ and $\mathbf{if_{nat}}$.

(b) The ***N**-standardization* A^N of a standard Σ-algebra A is the Σ^N-algebra formed by adjoining the carrier **N** together with the corresponding standard operations to A, thus

algebra	A^N
import	A
carriers	$\mathbf{N}$
functions	$\mathbf{0} : \to \mathbf{N}$
	$\mathbf{S} : \mathbf{N} \to \mathbf{N}$
	$\mathbf{if_{nat}} : \mathbf{B} \times \mathbf{N}^2 \to \mathbf{N}$
	$\mathbf{eq_{nat}}, \mathbf{less_{nat}} : \mathbf{N}^2 \to \mathbf{B}$
end	

Examples 2.13

(a) The simplest ***N***-standard algebra is the algebra ***N*** of Example 2.9 (b).

(b) We can ***N***-standardize the real ring ***R*** of Example 2.9 (c) to form the algebra R^N.

Remark 2.14

For any standard A, both A and N are Σ-*reducts* of the the ***N**-standardization* A^N.

2.5 Other important algebras

In this subsection, we briefly mention some other important algebras,

(i) add the unspecified value **u**: algebras $\boldsymbol{A}^{\mathbf{u}}$ of signature $\Sigma^{\mathbf{u}}$,

(ii) add arrays: algebras $\boldsymbol{A}^*$ of signature Σ^*,

(iii) add streams: algebras $\overline{\boldsymbol{A}}$ of signature $\overline{\Sigma}$.

Since we mainly focus on the *standard algebras* and ***N****-standard algebras*, we will not give any details for these three algebras here (see [7, section 2.6 – 2.8] for details).

Remark 2.15

The array algebra $\boldsymbol{A}^*$ will be used in Chapter 4 for the *Universal Function Theorem*.

CHAPTER THREE

SYNTAX AND SEMANTICS OF *ND* ON STANDARD ALGEBRAS[1]

In this section, we will study the syntax and semantics of ***ND*** on standard algebras by means of imperative programming models. We start by defining the non-deterministic programming language ***ND*** = ***ND***(Σ), which combines the programming language ***While*** extended with '*random assignment*' (studied in [7]) and '***Guarded Command Language***' (studied in [3] by Djikstra), and may be interpreted on any many-sorted Σ-algebra.

We will define in detail the abstract syntax (in section 3.1) and semantics of this language (in section 3.2 – 3.6). Our approach follows the *algebraic operational semantics* developed in [5] and used in [7]; however, we introduce a *semantic computation tree* for the semantics of ***ND*** statements, instead of the *computation sequence* used in the deterministic case [7]. Then the semantics of an ***ND*** statement is the *set of states* at all leaves of the semantic computation tree, together with '↑' (divergence) if there exists an infinite path in this tree.

---

[1] Cf. [7, section 3].

3.1 Syntax of *ND*(Σ)

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

(a) ***Var*** = ***Var***(Σ) is the class of Σ-variables, and $\boldsymbol{Var}_s$ is the class of variables of sort s. 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*, i.e., a tuple of variables of sorts $s_1, \ldots, s_m$, respectively.

Further, we write ***VarTup*** = ***VarTup***(Σ) for the class of all tuples of Σ-variables, and $\boldsymbol{VarTup}_u$ for the class of all u-tuples of Σ-variables.

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

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

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

Note again that Σ-constants are constructed as 0-ary functions, and so enter the definition of ***Term***(Σ) via clause (ii), with $m = 0$.

We write $\boldsymbol{type}(t) = s$ or $t : s$ to indicate that $t \in \boldsymbol{Term}_s$.

Further, we write ***TermTup*** = ***TermTup***(Σ) for the class of all tuples of Σ-terms, and, for $u = s_1 \times \cdots \times s_m$, $\boldsymbol{TermTup}_u$ for the class of u-tuples of terms, i.e.,

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

We write ***type***$(t) = u$ or $t : u$ to indicate that t is a *u-tuple* of terms, i.e., a tuple of terms of sorts $s_1, \ldots, s_m$.

For the sort **bool**, we have the class of *boolean terms* or *booleans* ***Bool***$(\Sigma) =_{df}$ ***Term***$_{\text{bool}}$, denoted either t^{bool} … (as above) or b, …

This class is given (according to the above definition of ***Term***$_s$) by:

$$b ::= \mathbf{x}^{\text{bool}} \mid \boldsymbol{F}(t) \mid \mathbf{eq}_s(t_1^s, t_2^s) \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{not}(b) \mid \mathbf{and}(b_1, b_2) \mid \mathbf{or}(b_1, b_2) \mid \mathbf{if}(b, b_1, b_2),$$

where $\boldsymbol{F}$ is a Σ-function symbol of type $u \to \mathbf{bool}$ and s is an equality sort.

(c) ***AtSt*** = ***AtSt***(Σ) is the class of atomic statements $\boldsymbol{S}_{\text{at}}$, …, defined by:

$$\boldsymbol{S}_{\text{at}} ::= \mathbf{skip} \mid \mathbf{x} := t \mid \mathbf{x} := \mathbf{?},$$

where $\mathbf{x} := t$ is the *concurrent assignment*, where for some product type u, $\mathbf{x} : u$ and $t : u$, and $\mathbf{x} := \mathbf{?}$ is the random assignment, for $\mathbf{x} : s$.

(d) ***Stmt*** = ***Stmt***(Σ) is the class of statements $\boldsymbol{S}$, ….generated by the following rules:

$$\boldsymbol{S} ::= \boldsymbol{S}_{\text{at}} \mid \boldsymbol{S}_1 ; \boldsymbol{S}_2 \mid \mathbf{if}\ b_1 \to \boldsymbol{S}_1 \mid \ldots \mid b_k \to \boldsymbol{S}_k\ \mathbf{fi} \mid \mathbf{do}\ b_1 \to \boldsymbol{S}_1 \mid \ldots \mid b_k \to \boldsymbol{S}_k\ \mathbf{od} \quad (k \geq 0).$$

(e) $\boldsymbol{Proc} = \boldsymbol{Proc}(\Sigma)$ is the class of procedures $\boldsymbol{P}$, $\boldsymbol{Q}$, … in the form

$$\boldsymbol{P} \equiv \textbf{proc } \boldsymbol{D} \textbf{ begin } \boldsymbol{S} \textbf{ end},$$

where $\boldsymbol{D}$ is the *variable declaration* and $\boldsymbol{S}$ is the *body*. Here $\boldsymbol{D}$ has the form

$$\boldsymbol{D} \equiv \textbf{in a out b aux c},$$

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

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

(ii) every variable occurring in the body $\boldsymbol{S}$ must be declared in $\boldsymbol{D}$ (among **a**, **b**, or **c**),

(iii) the *input variables* **a** must not occur on the lhs (left-hand side) of assignments in $\boldsymbol{S}$,

(iv) (*Initialization conditions*) $\boldsymbol{S}$ has the form $\boldsymbol{S}_{init};\boldsymbol{S}'$, where $\boldsymbol{S}_{init}$ is a *concurrent assignment* which *initializes* all the *output* and *auxiliary variables*, i.e., assigns to each of them the default term (see Definition 2.6) of the same sort.

Each variable occurring in the declaration of a procedure *binds* all free occurrences of that variable in that body.

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

Note 3.1

(a) We get ***GC*** as a sub-language of ***ND*** by removing *random assignment* from ***ND***.

(b) We get $\boldsymbol{While}^{\boldsymbol{RA}}$ as a sub-language of ***ND*** by using (only) the following special forms for the guarded command constructs:

$$\textbf{if } b \rightarrow \boldsymbol{S}_1 \mid \neg b \rightarrow \boldsymbol{S}_2 \textbf{ fi}$$

$$\textbf{do } b \rightarrow \boldsymbol{S} \textbf{ od}$$

Notation 3.2

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

(b) We will use $\boldsymbol{E}$, $\boldsymbol{E}'$, $\boldsymbol{E}_1$, … to denote syntactic expressions of any of the three classes ***Term***, ***Stmt*** and ***Proc***.

(c) For any such expression ***E***, we define ***var***(***E***) to be the set of variables occurring in ***E***.

(d) We use '≡' to denote syntactic identity between two expressions.

Remark 3.3 (Structural induction).

We will often prove assertions about, or define constructs on, expressions ***E*** of a particular syntactic class (such as ***Term***, ***Stmt***, or ***Proc***) by *structural induction* (*or recursion*) on ***E***, following the inductive definition of that class.

Section 3.2 – 3.6 will focus on the semantics of ***ND*** (cf. [7, sections 3.2 – 3.6]).

3.2 States

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

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

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

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

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

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

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

3.3 Semantics of terms

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

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

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

The definition is by structural induction on t:

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

$$[\![\boldsymbol{F}(t_1, \ldots, t_m)]\!]^{\boldsymbol{A}}\sigma = \boldsymbol{F}^{\boldsymbol{A}}([\![t_1]\!]^{\boldsymbol{A}}\sigma, \ldots, [\![t_m]\!]^{\boldsymbol{A}}\sigma).$$

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

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

Definition 3.4

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

Lemma 3.5 (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 $[\![t]\!]^{\boldsymbol{A}}\sigma_1 = [\![t]\!]^{\boldsymbol{A}}\sigma_2$.

Proof. By structural induction on t (see Appendix 1 for details). ■

3.4 Algebraic operational semantics

We will interpret programs as many-valued state transformations, and function procedures as many-valued functions on $\boldsymbol{A}$. Our approach follow the *algebraic operational semantics*, first developed in [5], and used in [7, section 3.4]. It is a general method for any programming language: we can define these three functions $\lhd\ \rhd$ (semantics of atomic statements), ***first*** and $\boldsymbol{Rest}^{\boldsymbol{A}}$ to develop the semantics of this language.

3.4.1 Semantics of atomic statements.

Firstly, we define the meaning of an atomic statement $\boldsymbol{S}_{\mathrm{at}} \in \boldsymbol{AtSt}$, to be a function

$$\lhd \boldsymbol{S}_{\mathrm{at}} \rhd^{\boldsymbol{A}} \sigma : \boldsymbol{State}(\boldsymbol{A}) \rightarrow \boldsymbol{P}(\boldsymbol{State}(\boldsymbol{A}))^{+},$$

where $\boldsymbol{P}(\boldsymbol{X})^+$ means the set of all *non-empty* subsets of a set $\boldsymbol{X}$ (see [8, Notation 3.2.1]).

This is defined by

$$\lhd \mathbf{skip} \rhd^{\boldsymbol{A}} \sigma = \{\, \sigma \,\},$$

$$\lhd \mathbf{x} := t \rhd^{\boldsymbol{A}} \sigma = \{\, \sigma\{\mathbf{x} \,/\, [\![t]\!]^{\boldsymbol{A}}\sigma\} \,\},$$

$$\lhd \mathbf{x} := \mathbf{?} \rhd^{\boldsymbol{A}} \sigma = \{\, \sigma' \mid \sigma' \text{ agrees with } \sigma \text{ on all variables, except } \mathbf{x} \}.$$

3.4.2 The *First* and *Rest* operations.

Secondly, we have two functions

$$\boldsymbol{First} : \boldsymbol{Stmt} \rightarrow \boldsymbol{AtSt},$$

$$\boldsymbol{Rest}^{\boldsymbol{A}} : \boldsymbol{Stmt} \times \boldsymbol{State}(\boldsymbol{A}) \rightarrow \boldsymbol{P}(\boldsymbol{Stmt}).$$

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

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

(i) $$\boldsymbol{First}(\boldsymbol{S}) = \begin{cases} \boldsymbol{S} & \text{if } \boldsymbol{S} \text{ is atomic} \\ \boldsymbol{First}(\boldsymbol{S}_1) & \text{if } \boldsymbol{S} \equiv \boldsymbol{S}_1 ; \boldsymbol{S}_2 \\ \mathbf{skip} & \text{otherwise} \end{cases}$$

(ii) $\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$ is defined as follows,

Case 1. If $\boldsymbol{S}_{\text{at}}$ is atomic, then $\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}_{\text{at}}, \sigma) = \{\ \mathbf{skip}\ \}$,

Case 2. If $\boldsymbol{S} \equiv \boldsymbol{S}_1; \boldsymbol{S}_2$, where $\boldsymbol{S}_1, \boldsymbol{S}_2 \in \boldsymbol{Stmt}$. Then

$$\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S},\sigma) = \begin{cases} \{\ \boldsymbol{S}_2\ \} & \text{if } \boldsymbol{S}_1 \text{ is atomic} \\ \bigcup \{\boldsymbol{S}_1'; \boldsymbol{S}_2 \mid \boldsymbol{S}_1' \in \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}_1,\sigma)\} & \text{otherwise} \end{cases}$$

Case 3. If $\boldsymbol{S} \equiv \mathbf{if}\ b_1 \to \boldsymbol{S}_1 \mid \ldots \mid b_k \to \boldsymbol{S}_k\ \mathbf{fi}$. Then

$$\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S},\sigma) = \bigcup_{i=1}^{k} \{\ \boldsymbol{S}_i \mid [\![b_i]\!]^{\boldsymbol{A}}\sigma = \mathbf{tt}\ \}.$$

Case 4. If $\boldsymbol{S} \equiv \mathbf{do}\ b_1 \to \boldsymbol{S}_1 \mid \ldots \mid b_k \to \boldsymbol{S}_k\ \mathbf{od}$. Then

$$\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S},\sigma) = \begin{cases} \bigcup_{i=1}^{k} \{\ \boldsymbol{S}_i; \boldsymbol{S} \mid [b_i]^{\boldsymbol{A}}\sigma = \mathbf{tt}\} & \text{if for some } i, [b_i]^{\boldsymbol{A}}\sigma = \mathbf{tt} \\ \{\mathbf{skip}\} & \text{otherwise} \end{cases}$$

This completes the definitions of ***First*** and $\boldsymbol{Rest}^{\boldsymbol{A}}$.

Note 3.6

(a) $\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$ is finite (can be easily proved by structural induction on $\boldsymbol{S}$).

(b) If for all $i = 1, \ldots, k$, $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \mathbf{ff}$, then

$$\boldsymbol{Rest}^{A}(\boldsymbol{S},\sigma)=\begin{cases} 0 & \text{in case 3 (corresponding to } \textbf{halt} \text{ command),} \\ \{\,\textbf{skip}\,\} & \text{in case 4.} \end{cases}$$

3.4.3 One-step computation function

From the ***First*** function we can define the one-step computation function

$$\boldsymbol{CompStep}^{\boldsymbol{A}} : \boldsymbol{Stmt} \times \boldsymbol{State}(\boldsymbol{A}) \rightarrow \boldsymbol{P}(\boldsymbol{State}(\boldsymbol{A}))^{+},$$

as $\boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma) = \lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}} \sigma$.

3.4.4 The semantic computation tree

Now, we will define a very important concept in our approach: the *semantic computation tree* $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$ of an ***ND***-statement ***S*** at a state σ is an ω-branching tree

$$\boldsymbol{CompTree}^{\boldsymbol{A}} : \boldsymbol{Stmt} \times \boldsymbol{State}(\boldsymbol{A}) \rightarrow \boldsymbol{P}(\ (\boldsymbol{State}(\boldsymbol{A}))^{\leq\omega}\ )^{+},$$

branching according to all possible outcomes (i.e., “output states”) of the one-step computation function $\boldsymbol{CompStep}^{\boldsymbol{A}}$. Each node of this tree is labeled by a state.

Here $(\boldsymbol{State}(\boldsymbol{A}))^{\leq\omega}$ denotes the set of all finite and infinite sequences from $\boldsymbol{State}(\boldsymbol{A})$, interpreted as the paths through the semantic computation tree.

In the definition, we have '↑' as the symbol for divergence, which indicates that the computation tree has an infinite path.

Any actual computation of statement $\boldsymbol{S}$ at state σ corresponds to one of the paths through this tree. The possibilities for any such path are:

(i) it is finite, ending in a leaf containing a state: the final state of the computation,

(ii) it is infinite (global divergence or ↑).

We define the *semantic computation tree* via a function

$$\boldsymbol{CompTreeStage^A} : \boldsymbol{Stmt} \times \boldsymbol{State}(\boldsymbol{A}) \times \mathbf{N} \rightarrow \boldsymbol{P}(\,(\boldsymbol{State}(\boldsymbol{A}))^{<\omega}\,)^{+},$$

where $\boldsymbol{CompTreeStage^A}(\boldsymbol{S}, \sigma, n)$ represents the first n steps of $\boldsymbol{CompTree^A}(\boldsymbol{S}, \sigma)$. Here $(\boldsymbol{State}(\boldsymbol{A}))^{<\omega}$ denotes the set of finite sequences from $\boldsymbol{State}(\boldsymbol{A})$, interpreted as finite initial segments of the paths through the semantic computation tree.

This function is defined by a simple tail recursion on n:

Base case: $\boldsymbol{CompTreeStage^A}(\boldsymbol{S}, \sigma, 0) = \{\, \sigma \,\}$, i.e., just the root containing σ,

Inductive step: $\boldsymbol{CompTreeStage^A}(\boldsymbol{S}, \sigma, n+1)$ is formed by attaching to the root $\{\sigma\}$ the following:

(i) for $\boldsymbol{S}$ atomic: the leaf $\{\sigma'\}$, for each $\sigma' \in \lhd \boldsymbol{S} \rhd^{\boldsymbol{A}}\sigma$

(ii) for $\boldsymbol{S}$ not atomic: the subtree $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}', \sigma', n)$, for each $\sigma' \in \boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$ and $\boldsymbol{S}' \in \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$

Then, $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$ is defined as the 'limit' over n of $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n)$, i.e., $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma) = \bigcup_{n=0}^{\infty} \boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n)$

Remark 3.7 (Tail recursion).

Consider the recursive definition of $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}$. In the 'recursive call' (ii) of the inductive step, notice that (1) $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}$ is on the 'outside', and (2) the parameter *changes* (from $\boldsymbol{S}$ to $\boldsymbol{S}'$, and σ to σ', for each $\boldsymbol{S}' \in \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$ and $\sigma' \in \boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$). Such a definitional scheme is said to be *tail recursive*.

3.5 Semantics of *ND* statements

From the semantic computation tree, we define the i /o semantics of statements

$$[\![\boldsymbol{S}]\!]^{\boldsymbol{A}} : \boldsymbol{State}(\boldsymbol{A}) \to \boldsymbol{P}(\boldsymbol{State}(\boldsymbol{A}) \cup \{\uparrow\}),$$

as follows: $[\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma$ is the set of states at all leaves in $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$, together with '↑' if $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$ has an infinite path.

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

Theorem 3.8

(a) For $\boldsymbol{S}_{\text{at}}$ atomic, $[\![\boldsymbol{S}_{\text{at}}]\!]^{\boldsymbol{A}} = \lhd \boldsymbol{S}_{\text{at}} \rhd^{\boldsymbol{A}}$, i.e.,

$$\lhd \textbf{skip} \rhd^{\boldsymbol{A}} \sigma = \{ \sigma \},$$

$$\lhd \mathbf{x} := t \rhd^{\boldsymbol{A}} \sigma = \{ \sigma\{\mathbf{x} / [\![t]\!]^{\boldsymbol{A}} \sigma \} \},$$

$$\lhd \mathbf{x} := \textbf{?} \rhd^{\boldsymbol{A}} \sigma = \{ \sigma' \mid \sigma' \text{ agrees with } \sigma \text{ on all variables, except } \mathbf{x} \},$$

(b) If $\boldsymbol{S} \equiv \boldsymbol{S}_1 ; \boldsymbol{S}_2$, then $[\![\boldsymbol{S}]\!]^{\boldsymbol{A}} \sigma = \bigcup \{ [\![\boldsymbol{S}_2]\!]^{\boldsymbol{A}} \tau \mid \tau \in [\![\boldsymbol{S}_1]\!]^{\boldsymbol{A}} \sigma \}$,

(c) If $\boldsymbol{S} \equiv \textbf{if } b_1 \rightarrow \boldsymbol{S}_1 \mid \dots \mid b_k \rightarrow \boldsymbol{S}_k \textbf{ fi}$, then, $[\![\boldsymbol{S}]\!]^{\boldsymbol{A}} \sigma = \bigcup_{i=1}^{k} \{ [\![\boldsymbol{S}_i]\!]^{\boldsymbol{A}} \sigma \mid [\![b_i]\!]^{\boldsymbol{A}} \sigma = \textbf{tt} \}$,

(d) If $\boldsymbol{S} \equiv \textbf{do } b_1 \rightarrow \boldsymbol{S}_1 \mid \dots \mid b_k \rightarrow \boldsymbol{S}_k \textbf{ od}$, then,

$$[\![\boldsymbol{S}]\!]^{\boldsymbol{A}} \sigma = \begin{cases} \bigcup_{i=1}^{k} \{ [\boldsymbol{S}_i ; \boldsymbol{S}]^{\boldsymbol{A}} \sigma \mid [b_i]^{\boldsymbol{A}} \sigma = \textbf{tt} \} & \text{if for some } i, [b_i]^{\boldsymbol{A}} \sigma = \textbf{tt} \\ \{ \sigma \} & \text{otherwise} \end{cases}$$

In particular, for ***While***$^{\boldsymbol{RA}}$, case (c) and (d) turn into simple forms (see Note 3.1 (b)):

(c)' $\boldsymbol{S} \equiv \textbf{if } b_1 \rightarrow \boldsymbol{S}_1 \mid \neg b_1 \rightarrow \boldsymbol{S}_2 \textbf{ fi}$. Then,

$$[\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma = \begin{cases} [\boldsymbol{S}_1]^{\boldsymbol{A}}\sigma & \text{if } [b_1]^{\boldsymbol{A}}\sigma = \textbf{tt} \\ [\boldsymbol{S}_2]^{\boldsymbol{A}}\sigma & \text{otherwise} \end{cases}$$

(d)' $\boldsymbol{S} \equiv \textbf{do}\, b \to \boldsymbol{S}_0 \textbf{od}$. Then,

$$[\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma = \begin{cases} [\boldsymbol{S}_0; \boldsymbol{S}]^{\boldsymbol{A}}\sigma & \text{if } [b]^{\boldsymbol{A}}\sigma = \textbf{tt} \\ \{\sigma\} & \text{otherwise} \end{cases}$$

We prove Theorem 3.8 via the following *lemmas*.

Lemma 3.9

Assume $n > 0$.

(a) If $\boldsymbol{S}_{\text{at}} \in \boldsymbol{AtSt}$, then $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_{\text{at}}, \sigma, n)$ is formed by *attaching to* the root $\{\sigma\}$, the leaf $\{\tau\}$, for each $\tau \in \lhd \boldsymbol{S}_{\text{at}} \rhd^{\boldsymbol{A}}\sigma$.

(b) (*Interesting case*) If $\boldsymbol{S} \equiv \boldsymbol{S}_1;\boldsymbol{S}_2$, then $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n)$ is formed by *attaching subtree(s)* $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_2, \tau, n\text{-}d')$ to each leaf $\{\tau\}$ of $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma, n)$, where d' is the depth of $\{\tau\}$ in $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma, n)$.

(c) If $\boldsymbol{S} \equiv \textbf{if}\ b_1 \rightarrow \boldsymbol{S}_1\ |\ldots|\ b_k \rightarrow \boldsymbol{S}_k\ \textbf{fi}$, then ***CompTreeStage***$^{\boldsymbol{A}}$(***S***, σ, n) is formed by *attaching to* the root $\{\sigma\}$, the subtree(s) ***CompTreeStage***$^{\boldsymbol{A}}$($\boldsymbol{S}_i$, σ, n-1), for those i ($1 \leq i \leq$ k) where $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \mathbf{tt}$.

(d) If $\boldsymbol{S} \equiv \textbf{do}\ b_1 \rightarrow \boldsymbol{S}_1\ |\ldots|\ b_k \rightarrow \boldsymbol{S}_k\ \textbf{od}$, then ***CompTreeStage***$^{\boldsymbol{A}}$(***S***, σ, n) is formed by *attaching to* the root $\{\sigma\}$,

(i) the subtree(s) ***CompTreeStage***$^{\boldsymbol{A}}$($\boldsymbol{S}_i$;***S***, σ, n-1), for those i, where $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \mathbf{tt}$, if for some i, $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \mathbf{tt}$,

(ii) the leaf $\{\sigma\}$ otherwise.

Proof. By structural induction on ***S*** (see Appendix 2 for details). ■

Now, we can prove Theorem 3.8 via the above Lemmas. As an example, we give the proof for case (b). Please see to Appendix 3, for the proof in the other cases.

Proof for Theorem 3.8 (b).

From Lemma 3.9 (b), take the 'limit' over n for all ***CompTreeStage***$^{\boldsymbol{A}}$(***S***, σ, n), Then we have, ***CompTree***$^{\boldsymbol{A}}$(***S***,σ) is formed by attaching ***CompTree***$^{\boldsymbol{A}}$($\boldsymbol{S}_2$, τ) to each leaf $\{\tau\}$ of ***CompTree***$^{\boldsymbol{A}}$($\boldsymbol{S}_1$, σ).

So, the leaves of ***CompTree***$^{\boldsymbol{A}}$(***S***, σ) are formed from the leaves of ***CompTree***$^{\boldsymbol{A}}$($\boldsymbol{S}_2$, τ), for each leaf $\{\tau\}$ of ***CompTree***$^{\boldsymbol{A}}$($\boldsymbol{S}_1$, σ). Also (trivially), if there is an infinite path in

$\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma)$ or any $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}_2, \tau)$, for each leaf $\{\tau\}$ of $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma)$, this path or its extension (in $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}_2, \tau)$) to the root $\{\sigma\}$, is just an infinite path in $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$.

By the definition (of semantics of ***ND*** statements), $[\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma$ is the set of states at all leaves in $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$, together with '↑' if $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$ has an infinite path.

$\bigcup \{ [\![\boldsymbol{S}_2]\!]^{\boldsymbol{A}}\tau \mid \tau \in [\![\boldsymbol{S}_1]\!]^{\boldsymbol{A}}\sigma \}$ is the set of states at all leaves in $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}_2, \tau)$, for each leaf $\{\tau\}$ of $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma)$, together with '↑' if there is an infinite path in either $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma)$ or any $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}_2, \tau)$, for each leaf $\{\tau\}$ of $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma)$.

From this, it follows that $[\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma = \bigcup \{ [\![\boldsymbol{S}_2]\!]^{\boldsymbol{A}}\tau \mid \tau \in [\![\boldsymbol{S}_1]\!]^{\boldsymbol{A}}\sigma \}$. ■

For the semantics of procedures, we need the following. Let $\boldsymbol{M} \subseteq \boldsymbol{Var}_s$, and $\sigma_1, \sigma_2 \in \boldsymbol{State}(\boldsymbol{A})$.

Definition 3.10

Let $\boldsymbol{C}_1, \boldsymbol{C}_2 \subseteq \boldsymbol{State}(\boldsymbol{A}) \cup \{\uparrow\}$. We say

$$\boldsymbol{C}_1 \approx \boldsymbol{C}_2 \text{ (rel } \boldsymbol{M}\text{)},$$

($\boldsymbol{C}_1$ agrees with $\boldsymbol{C}_2$ on $\boldsymbol{M}$) if only and if,

$$\forall \sigma_1 \in \boldsymbol{C}_1, \exists \sigma_2 \in \boldsymbol{C}_2, \sigma_1 \approx \sigma_2 \text{ (rel } \boldsymbol{M}\text{)},$$

and

$$\forall \sigma_2 \in \boldsymbol{C}_2, \exists \sigma_1 \in \boldsymbol{C}_1, \sigma_1 \approx \sigma_2 \text{ (rel } \boldsymbol{M}\text{)},$$

and

$$\uparrow \in \boldsymbol{C}_1 \Leftrightarrow \uparrow \in \boldsymbol{C}_2.$$

Lemma 3.11

Suppose $\boldsymbol{var}(\boldsymbol{S}) \subseteq \boldsymbol{M}$. If $\sigma_1 \approx \sigma_2$ (rel $\boldsymbol{M}$), then

$$\lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}} \sigma_1 \approx \lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}} \sigma_2 \text{ (rel } \boldsymbol{M}\text{)},$$

$$\text{and } \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_1) = \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_2).$$

Proof. By structural induction on $\boldsymbol{S}$ and Definition 3.10 (see Appendix 4 for details). ■

Definition 3.12 (Set of Leaf States).

$$\boldsymbol{LS}^{\boldsymbol{A}} : \boldsymbol{Stmt} \times \boldsymbol{State}(\boldsymbol{A}) \times \mathbf{N} \rightarrow \boldsymbol{P}(\ (\boldsymbol{State}(\boldsymbol{A})\ )$$

means the *set of states at the leaves* of $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$ in $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n)$. We define function $\boldsymbol{LS}^{\boldsymbol{A}}$ by tail recursion on n as follows, *Base case*: $\boldsymbol{LS}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, 0) = 0$, i.e. no leaf state.

Inductive step:

(i) for $\boldsymbol{S}$ atomic: $\boldsymbol{LS}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n{+}1) = \lhd \boldsymbol{S} \rhd^{\boldsymbol{A}}\sigma$,

(ii) for $\boldsymbol{S}$ not atomic:

$$\boldsymbol{LS}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n{+}1) = \bigcup \{\boldsymbol{LS}^{\boldsymbol{A}}(\boldsymbol{S}', \sigma', n) \mid \sigma' \in \boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma), \boldsymbol{S}' \in \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)\}.$$

Lemma 3.13

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

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

Proof. By simple induction on n and Lemma 3.11 (see Appendix 5 for details). ■

Another important result expresses the i/o semantics of $\boldsymbol{S}$ in terms of leaf states:

Lemma 3.14

$$[\![\boldsymbol{S}]\!]^{\boldsymbol{A}} = \bigcup_{n=0}^{\infty} \boldsymbol{LS}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n) \cup \{\uparrow \mid \text{there is an infinite path in } \boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)\}.$$

Lemma 3.15 (Functionality lemma for *ND* statements).

Suppose $\boldsymbol{var}(\boldsymbol{S}) \subseteq \boldsymbol{M}$. If $\sigma_1 \approx \sigma_2$ (rel $\boldsymbol{M}$), then

$$[\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma_1 \approx [\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma_2 \text{ (rel } \boldsymbol{M}).$$

Proof. The result clearly follows from Lemma 3.13 and 3.14. ■

3.6 Semantics of *ND* procedures

Now if

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

is an ***ND*** procedure of type $u \rightarrow v$, then its meaning in $\boldsymbol{A}$ is a function

$$[\![\boldsymbol{P}]\!]^{\boldsymbol{A}} : \boldsymbol{A}^u \rightarrow \boldsymbol{P}(\boldsymbol{A}^v \cup \{\uparrow\}),$$

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

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

For $[\![\boldsymbol{P}]\!]^{\boldsymbol{A}}$ to be well defined, we need the fact that the procedure $\boldsymbol{P}$ is functional, i.e., $[\![\boldsymbol{P}]\!]^{\boldsymbol{A}}(a)$ is independent of the state σ.

Lemma 3.16 (Functionality lemma for *ND* procedures).

Suppose

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

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

$$[\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma_1 \approx [\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma_2 \text{ (rel } \mathbf{b}).$$

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

$$[\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma_1 = \bigcup \{ [\![\boldsymbol{S}']\!]^{\boldsymbol{A}}\tau_1 \mid \tau_1 \in [\![\boldsymbol{S}_{init}]\!]^{\boldsymbol{A}}\sigma_1 \},$$

$$[\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma_2 = \bigcup \{ [\![\boldsymbol{S}']\!]^{\boldsymbol{A}}\tau_2 \mid \tau_2 \in [\![\boldsymbol{S}_{init}]\!]^{\boldsymbol{A}}\sigma_2 \},$$

it is easy to see that

$$[\![\boldsymbol{S}_{init}]\!]^{\boldsymbol{A}}\sigma_1 \approx [\![\boldsymbol{S}_{init}]\!]^{\boldsymbol{A}}\sigma_2 \text{ (rel } \mathbf{a}, \mathbf{b}, \mathbf{c}).^{2}$$

Then, the result follows from the Lemma 3.15 and Definition 3.10. ■

---

[2] See Appendix 4 for the proof for concurrent assignment in Lemma 3.11.

The functionality lemma for procedures amount to saying that there are *no side effects* from the output variables or auxiliary variables.

We can now define ***ND*** computability of functions on $\boldsymbol{A}$. We deal with two types of functions on $\boldsymbol{A}$:

(i) *multi-valued functions*, i.e., functions

$$\boldsymbol{F} : \boldsymbol{A}^u \to \boldsymbol{P}(\boldsymbol{A}^v \cup \{\uparrow\}),$$

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

$$f : \boldsymbol{A}^u \xrightarrow{\cdot} \boldsymbol{A}^v$$

Note that a *single-valued function f* can be represented as a special case of a *multi-valued function* $\boldsymbol{F}$, where for all $a \in \boldsymbol{A}^u$,

$$\boldsymbol{F}(a) = \begin{cases} \{f(a)\} & \text{if } f(a)\downarrow, \\ \{\uparrow\} & \text{if } f(a)\uparrow. \end{cases}$$

Definition 3.14 (*ND* computable functions).

Let $\boldsymbol{P} : u \to v$ be an $\boldsymbol{ND}(\Sigma)$ procedure.

(a) A multi-valued function $f : \boldsymbol{A}^u \to \boldsymbol{P}(\boldsymbol{A}^v \cup \{\uparrow\})$ is *computable* on $\boldsymbol{A}$ by $\boldsymbol{P}$ if $f = [\![\boldsymbol{P}]\!]^{\boldsymbol{A}}$.

(b) A single-valued partial function $f : \boldsymbol{A}^u \xrightarrow{\cdot} \boldsymbol{A}^v$ is *computable* on $\boldsymbol{A}$ by $\boldsymbol{P}$ if

$$\forall a \in \boldsymbol{A}^u, \quad \boldsymbol{P}^A(a) = \begin{cases} \{f(a)\} & \text{if } f(a)\downarrow, \\ \{\uparrow\} & \text{if } f(a)\uparrow. \end{cases}$$

Definition 3.15 (single-valued *ND* function and procedure).

$\boldsymbol{P} : u \to v$ is called *single-valued* on $\boldsymbol{A}$, if for all $a \in \boldsymbol{A}^u$, $\boldsymbol{P}^A(a)$ is a singleton set (which could be either $\{d\}$ for $d \in \boldsymbol{A}^v$, or $\{\uparrow\}$).

Hence, a *single-valued partial* ***ND*** computable function is computed by a *single-valued* ***ND*** procedure.

Remark 3.16

Similarly, we can define ***While***$^{\boldsymbol{RA}}$ computability and ***GC*** computability. In fact, we will focus on ***GC*** computability in the next chapter.

Remark 3.17 (Interpretability of *GC* in *While*$^{\boldsymbol{RA}}$ and vice versa).

Two interesting questions are:

(a) Can ***GC*** be interpreted in ***While***$^{\boldsymbol{RA}}$?

(b) Can ***While***$^{\boldsymbol{RA}}$ be interpreted in ***GC***?

The answer to (a) is yes. To show this, consider the simple case of a guarded command conditional construct of the form

$$\textbf{if } b_1 \rightarrow \boldsymbol{S}_1 \mid b_2 \rightarrow \boldsymbol{S}_2 \textbf{ fi}$$

This can be interpreted with the help of a random assignment to an auxiliary boolean variable as the follows,

```
x : bool
if $b_1 \wedge \neg b_2$ then $S_1$
else if $\neg b_1 \wedge b_2$ then $S_2$
  else if $b_1 \wedge b_2$ then
      x := ?;
      if x then $S_1$
      else $S_2$ fi
    else skip
    fi
  fi
fi
```

The answer to (b), however, is no.

For example, consider the following $\boldsymbol{\mathit{While}}^{RA}$ procedure

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

At first glance, we might try to simulate this by the ***GC*** procedure

```
P′ ≡ proc in n : nat
        aux b: bool
     begin
        b := tt;
        if b → n++ | b → b := ff fi
     end
```

However, the semantics of ***P′*** includes non-termination, since its semantic computation tree has an infinite path. Therefore, ***P′*** is not semantically equivalent to ***P***.

In fact we can see that no ***GC*** procedure could simulate ***P***. For any such ***GC*** procedure would have to be total, i.e., its semantic computation tree could not have any infinite path. Therefore, since this semantic computation tree is finitely branching, its set of total possible output would have to be finite, by König's Lemma.[3]

---

[3] This states that a finitely branching tree without any infinite path is finite.

CHAPTER FOUR

REPRESENTATIONS OF SEMANTIC FUNCTIONS AND UNIVERSALITY[1]

In this section, we will investigate whether there is *a universal **GC** procedure that can compute all the **GC** computable functions on **A***. To do that, we need the techniques of Gödel numbering, state and set of state (and state set) representations, and symbolic computations on terms. Specifically, for Gödel numbering to be possible, we must work with ***N***-standard algebras, which includes the sort **nat**.

Since the *term evaluation function* is ***While*** computable in most commonly used algebras, it is reasonable to assume the *term evaluation property* [7, Definition 4.4 and Examples 4.5]. Then, by means of "local representation" of the semantics of computation, we will show that

*for any given Σ-type and Σ-algebra **A**, there is a universal **GC** procedure for that type over **A**.*

---

[1] Cf. [7, section 4].

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^{N}, i.e., a family ***gn*** of effective mappings from expressions $\boldsymbol{E}$ to natural numbers $\ulcorner \boldsymbol{E} \urcorner = \boldsymbol{gn}(\boldsymbol{E})$, which satisfy certain basic properties:

- $\ulcorner \boldsymbol{E} \urcorner$ increases strictly with ***compl***($\boldsymbol{E}$), and in particular, the code of an expression is larger than those of its subexpressions.
- Sets of codes of the various syntactic classes, and of their respective subclasses, such as $\{ \ulcorner t \urcorner \mid t \in \boldsymbol{Term}\}$, $\{ \ulcorner \boldsymbol{S} \urcorner \mid \boldsymbol{S} \in \boldsymbol{Stmt}\}$, etc. are primitive recursive;
- We can go primitive recursively from codes of expressions to codes of their immediate subexpressions, and vice versa; thus, e.g., $\ulcorner \boldsymbol{S}_1 \urcorner$ and $\ulcorner \boldsymbol{S}_2 \urcorner$ are primitive recursive in $\ulcorner \boldsymbol{S}_1; \boldsymbol{S}_2 \urcorner$, and conversely, $\ulcorner \boldsymbol{S}_1; \boldsymbol{S}_2 \urcorner$ is primitive recursive in $\ulcorner \boldsymbol{S}_1 \urcorner$ and $\ulcorner \boldsymbol{S}_2 \urcorner$.
- We will use the notation $\ulcorner \boldsymbol{Term} \urcorner =_{df} \{ \ulcorner t \urcorner \mid t \in \boldsymbol{Term}\}$, etc., for sets of Gödel numbers of syntactic expressions.

In short, *we can primitive recursively simulate all operations involved in processing the syntax of the programming language*. This means that the syntactic classes form a computable (in fact, primitive recursive) algebra.

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

4.2 Representation of states

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

The *state representing function*

$$\boldsymbol{Rep}_{\mathbf{x}}^{A} : \boldsymbol{State}(\boldsymbol{A}) \cup \{\uparrow\} \to \boldsymbol{A}^u \cup \{\uparrow\},$$

is defined by

$$\boldsymbol{Rep}_{\mathbf{x}}^{A}(\sigma) = \sigma[\mathbf{x}].$$

Note that ↑ is represented by ↑. I.e. $\boldsymbol{Rep}_{\mathbf{x}}^{A}(\uparrow) = \uparrow$.

Similiarly, a set $\boldsymbol{D}$ of states or '↑' on $\boldsymbol{A}$ is represented (relative to **x**) by a set $\boldsymbol{E} \in \boldsymbol{P}(\boldsymbol{A}^u \cup \{\uparrow\})$ of tuples of elements, if $\boldsymbol{E} = \{\tau[\mathbf{x}] \mid \tau \in \boldsymbol{D}\}$. The *set of states representing function*

$$\boldsymbol{RepSet}_{\mathbf{x}}^{A} : \boldsymbol{P}(\boldsymbol{State}(\boldsymbol{A}) \cup \{\uparrow\}) \to \boldsymbol{P}(\boldsymbol{A}^{\boldsymbol{u}} \cup \{\uparrow\}),$$

is defined by

$$\boldsymbol{RepSet}_{\mathbf{x}}^{A}(\boldsymbol{D}) = \{\tau[\mathbf{x}] \mid \tau \in \boldsymbol{D}\} = \{\boldsymbol{Rep}_{\mathbf{x}}^{A}(\tau) \mid \tau \in \boldsymbol{D}\}.$$

4.3 Representation of term evaluation

Let $\mathbf{x}$ be a u-tuple of program variables. Let $\boldsymbol{Term}_{\mathbf{x}} = \boldsymbol{Term}_{\mathbf{x}}(\Sigma)$ be the class of all Σ-terms with variables among $\mathbf{x}$ only, and for all sort s of Σ, let $\boldsymbol{Term}_{\mathbf{x},s} = \boldsymbol{Term}_{\mathbf{x},s}(\Sigma)$ be the class of such terms of sort s. Similarly, we write $\boldsymbol{TermTup}_{\mathbf{x}}$ for the class of all term tuples with variables among $\mathbf{x}$ only, $\boldsymbol{TermTup}_{\mathbf{x},v}$ for the class of all v-tuples of such terms.

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

$$\boldsymbol{TE}^{A}_{\mathbf{x},s} : \boldsymbol{Term}_{\mathbf{x},s} \times \boldsymbol{State}(\boldsymbol{A}) \to \boldsymbol{A}_s,$$

defined by

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

is *represented* by the function

$$\boldsymbol{te}^{A}_{\mathbf{x},s} : \ulcorner \boldsymbol{Term}_{\mathbf{x},s} \urcorner \times \boldsymbol{A}^{u} \to \boldsymbol{A}_s,$$

defined by

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

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

$$\begin{tikzcd}
\boldsymbol{Term}_{\mathbf{x},s} \times \boldsymbol{State}(\boldsymbol{A}) \arrow[d, "\langle \boldsymbol{gn}, \boldsymbol{Rep}^{A}_{\mathbf{x}} \rangle"'] \arrow[dr, "\boldsymbol{TE}^{A}_{\mathbf{x},s}"] & \\
\ulcorner \boldsymbol{Term}_{\mathbf{x},s} \urcorner \times \boldsymbol{A}^{u} \arrow[r, "\boldsymbol{te}^{A}_{\mathbf{x},s}"] & \boldsymbol{A}_s
\end{tikzcd}$$

Strictly speaking, if $\boldsymbol{gn}$ is not surjective on $\mathbf{N}$, then $\boldsymbol{te}^{\boldsymbol{A}}_{\mathbf{x},s}$ is not uniquely specified by the above definition, or by the diagram. However, we may assume that for n not a Gödel number (of the required sort), $\boldsymbol{te}^{\boldsymbol{A}}_{\mathbf{x},s}(n,a)$ takes the default value of sort s, i.e. $\boldsymbol{\delta}^{s}$. Similar remarks apply to the other representing functions given below.

Further, for a product type v, we will define an evaluating function for *tuples of terms*

$$\boldsymbol{te}^{\boldsymbol{A}}_{\mathbf{x},\upsilon} : \ulcorner \boldsymbol{TermTup}_{\mathbf{x},v} \urcorner \times \boldsymbol{A}^{u} \rightarrow \boldsymbol{A}^{v},$$

similarly, defined by

$$\boldsymbol{te}^{\boldsymbol{A}}_{\mathbf{x},\upsilon}(\ulcorner t \urcorner, a) = [\![t]\!]^{\boldsymbol{A}}\sigma.$$

We will be interested in the computability of these term evaluation representing functions.

4.4 Representation of the atomic statement

Let $\boldsymbol{AtSt}_{\mathbf{x}}$ be the class of atomic statements with variables among $\mathbf{x}$ only. The *atomic statement evaluation function on* $\boldsymbol{A}$ *relative to* $\mathbf{x}$

$$\boldsymbol{AE}^{\boldsymbol{A}}_{\mathbf{x}} : \boldsymbol{AtSt}_{\mathbf{x}} \times \boldsymbol{State}(\boldsymbol{A}) \rightarrow \boldsymbol{P}(\boldsymbol{State}(\boldsymbol{A}))^{+},$$

defined by

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

is represented by the function

$$\boldsymbol{ae}^{A}_{\mathbf{x}} \ : \ \ulcorner \boldsymbol{AtSt}_{\mathbf{x}} \urcorner \times A^{u} \to \boldsymbol{P}(A^{u})^{+},$$

defined by

$$\boldsymbol{ae}^{A}_{\mathbf{x}} \,(\, \ulcorner \boldsymbol{S} \urcorner \,, a) = \bigcup \,\{\tau[\mathbf{x}] \mid \tau \in \lhd \boldsymbol{S} \rhd^{A} \sigma\},$$

where σ is any state on $\boldsymbol{A}$ such that $\sigma[\mathrm{x}] = a$ (again, this is well defined, by *Functionality Lemma for statements*). In other words, the following diagram commutes:

$$\begin{tikzcd}
\boldsymbol{AtSt}_{\mathbf{x}} \times \boldsymbol{State}(A) \arrow[r, "\boldsymbol{AE}^{A}_{\mathbf{x}}"] \arrow[d, "\langle\, \boldsymbol{gn}, \boldsymbol{Rep}^{A}_{\mathbf{x}} \rangle"'] & \boldsymbol{P}(\boldsymbol{State}(A))^{+} \arrow[d, "\boldsymbol{RepSet}^{A}_{\mathbf{x}}"] \\
\ulcorner \boldsymbol{AtSt}_{\mathbf{x}} \urcorner \times A^{u} \arrow[r, "\boldsymbol{ae}^{A}_{\mathbf{x}}"'] & \boldsymbol{P}(A^{u})^{+}
\end{tikzcd}$$

4.5 The *First* and *Rest* operations

Next, let $\boldsymbol{Stmt}_{\mathbf{x}}$ be the class of statements with variables among **x** only, and define

$$\boldsymbol{Rest}^{A}_{\mathbf{x}} \ =_{df} \ \boldsymbol{Rest}^{A} \upharpoonright (\, \boldsymbol{Stmt}_{\mathbf{x}} \times \boldsymbol{State}(A) \,),$$

Then ***First*** and $\boldsymbol{Rest}^{A}_{\mathbf{x}}$ are represented by the functions

$$\boldsymbol{first}\colon \ \ulcorner \boldsymbol{Stmt} \urcorner \ \to \ \ulcorner \boldsymbol{AtSt} \urcorner \,,$$

$$\boldsymbol{rest}^{A}_{\mathbf{x}} : \ \ulcorner \boldsymbol{Stmt}_{\mathbf{x}} \urcorner \ \times A^{u} \to \boldsymbol{P}(\, \ulcorner \boldsymbol{Stmt}_{\mathbf{x}} \urcorner \,),$$

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

$$\begin{tikzcd}
\boldsymbol{Stmt} \arrow[r, "\boldsymbol{First}"] \arrow[d, "\boldsymbol{gn}"'] & \boldsymbol{AtSt} \arrow[d, "\boldsymbol{gn}"] \\
\ulcorner \boldsymbol{Stmt} \urcorner \arrow[r, "\boldsymbol{first}"'] & \ulcorner \boldsymbol{AtSt} \urcorner
\end{tikzcd}$$

$$\begin{tikzcd}
\boldsymbol{Stmt}_{\mathbf{x}} \times \boldsymbol{State}(\boldsymbol{A}) \arrow[r, "\boldsymbol{Rest}^{\boldsymbol{A}}_{\mathbf{x}}"] \arrow[d, "\langle \boldsymbol{gn}, \boldsymbol{Rep}^{\boldsymbol{A}}_{\mathbf{x}} \rangle"'] & \boldsymbol{P}(\boldsymbol{Stmt}_{\mathbf{x}}) \arrow[d, "\boldsymbol{gn}"] \\
\ulcorner \boldsymbol{Stmt}_{\mathbf{x}} \urcorner \times \boldsymbol{A}^{u} \arrow[r, "\boldsymbol{rest}^{\boldsymbol{A}}_{\mathbf{x}}"'] & \boldsymbol{P}(\ulcorner \boldsymbol{Stmt}_{\mathbf{x}} \urcorner)
\end{tikzcd}$$

4.6 Representation of one step computation function

Let $\boldsymbol{Stmt}_{\mathbf{x}}$ be the class of statements with variables among **x** only. The *one step computation evaluation function on* ***A*** *relative to* **x**

$$\boldsymbol{CompStep}^{\boldsymbol{A}}_{\mathbf{x}} : \boldsymbol{Stmt}_{\mathbf{x}} \times \boldsymbol{State}(\boldsymbol{A}) \to \boldsymbol{P}(\boldsymbol{State}(\boldsymbol{A}))^{+},$$

defined by

$$\boldsymbol{CompStep}^{\boldsymbol{A}}_{\mathbf{x}}(\boldsymbol{S}, \sigma) = \lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}} \sigma,$$

is represented by the function

$$\boldsymbol{compstep}^{A}_{\mathbf{x}} : \ulcorner \boldsymbol{Stmt}_{\mathbf{x}} \urcorner \times A^{u} \rightarrow \boldsymbol{P}(A^{u})^{+},$$

defined by

$$\boldsymbol{compstep}^{A}_{\mathbf{x}}(\ulcorner S \urcorner, a) = \boldsymbol{ae}^{A}_{\mathbf{x}}(\boldsymbol{first}(\ulcorner S \urcorner), a),$$

where σ is any state on A such that $\sigma[\mathrm{x}] = a$. In other words, the following diagram commutes:

$$\begin{tikzcd}
\boldsymbol{Stmt}_{\mathbf{x}} \times \boldsymbol{State}(A) \arrow[r, "\boldsymbol{CompStep}^{A}_{\mathbf{x}}"] \arrow[d, swap, "\langle \boldsymbol{gn}, \boldsymbol{Rep}^{A}_{\mathbf{x}} \rangle"] & \boldsymbol{P}(\boldsymbol{State}(A))^{+} \arrow[d, "\boldsymbol{RepSet}^{A}_{\mathbf{x}}"] \\
\ulcorner \boldsymbol{Stmt}_{\mathbf{x}} \urcorner \times A^{u} \arrow[r, swap, "\boldsymbol{compstep}^{A}_{\mathbf{x}}"] & \boldsymbol{P}(A^{u})^{+}
\end{tikzcd}$$

Note that $\boldsymbol{compstep}^{A}_{\mathbf{x}}$ is defined by $\boldsymbol{ae}^{A}_{\mathbf{x}}$ and ***first***.

4.7 Representation of set of Leaf States function

Let $\boldsymbol{Stmt}_{\mathbf{x}}$ be the class of statements with variables among **x** only. The *set of Leaf States evaluation function on* ***A*** *relative to* **x**

$$\boldsymbol{LS}^{A}_{\mathbf{x}} : \boldsymbol{Stmt}_{\mathbf{x}} \times \boldsymbol{State}(A) \times \mathbf{N} \rightarrow \boldsymbol{P}(\boldsymbol{State}(A)),$$

defined by

$$\boldsymbol{LS}^{A}_{\mathbf{x}} =_{df} \boldsymbol{LS}^{A} \upharpoonright (\boldsymbol{Stmt}_{\mathbf{x}} \times \boldsymbol{State}(A) \times \mathbf{N}),$$

is represented by the function

$$\boldsymbol{ls}_{\mathrm{x}}^{A} : \ulcorner \boldsymbol{Stmt}_{\mathrm{x}} \urcorner \times A^{u} \times \mathbf{N} \to \boldsymbol{P}(A^{u}),$$

defined by a simple tail recursion on n as the follows (cf. Definition 3.12),

Base case: $\boldsymbol{ls}_{\mathrm{x}}^{A}(\ulcorner \boldsymbol{S} \urcorner, a, 0) = 0.$

Inductive step:

(i) for $\boldsymbol{S}$ atomic: $\boldsymbol{ls}_{\mathrm{x}}^{A}(\ulcorner \boldsymbol{S} \urcorner, a, n+1) = \boldsymbol{ae}_{\mathrm{x}}^{A}(\ulcorner \boldsymbol{S} \urcorner, a),$

(ii) for $\boldsymbol{S}$ not atomic: $\boldsymbol{ls}_{\mathrm{x}}^{A}(\ulcorner \boldsymbol{S} \urcorner, a, n+1) = \bigcup \{ \boldsymbol{ls}_{\mathrm{x}}^{A}(\ulcorner \boldsymbol{S}' \urcorner, a', n) \mid a' \in \boldsymbol{compstep}_{\mathrm{x}}^{A}(\ulcorner \boldsymbol{S} \urcorner, a), \ulcorner \boldsymbol{S}' \urcorner \in \boldsymbol{rest}_{\mathrm{x}}^{A}(\ulcorner \boldsymbol{S} \urcorner, a) \}.$

where σ is any state on $\boldsymbol{A}$ such that $\sigma[\mathrm{x}] = a$. In other words, the following diagram commutes:

$$\begin{tikzcd}
\boldsymbol{Stmt}_{\mathrm{x}} \times \boldsymbol{State}(A) \times \mathbf{N} \arrow[r, "\boldsymbol{LS}_{\mathrm{x}}^{A}"] \arrow[d, "\langle \boldsymbol{gn}, \boldsymbol{Rep}_{\mathrm{x}}^{A}, \boldsymbol{id}_{\mathrm{N}} \rangle"'] & \boldsymbol{P}(\boldsymbol{State}(A)) \arrow[d, "\boldsymbol{RepSet}_{\mathrm{x}}^{A}"] \\
\ulcorner \boldsymbol{Stmt}_{\mathrm{x}} \urcorner \times A^{u} \times \mathbf{N} \arrow[r, "\boldsymbol{ls}_{\mathrm{x}}^{A}"'] & \boldsymbol{P}(A^{u})
\end{tikzcd}$$

4.8 Representation of statement evaluation

Let $\boldsymbol{Stmt}_{\mathbf{x}}$ be the class of statements with variables among $\mathbf{x}$ only. The *statement evaluation function on **A** relative to* $\mathbf{x}$

$$\boldsymbol{SE}_{\mathbf{x}}^{A} : \boldsymbol{Stmt}_{\mathbf{x}} \times \boldsymbol{State}(\boldsymbol{A}) \to \boldsymbol{P}(\boldsymbol{State}(\boldsymbol{A}) \cup \{\uparrow\}),$$

defined by

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

is represented by the function

$$\boldsymbol{se}_{\mathbf{x}}^{A} : \ulcorner\boldsymbol{Stmt}_{\mathbf{x}}\urcorner \times \boldsymbol{A}^{u} \to \boldsymbol{P}(\boldsymbol{A}^{u} \cup \{\uparrow\}),$$

defined by

$$\boldsymbol{se}_{\mathbf{x}}^{A}(\ulcorner\boldsymbol{S}\urcorner, a) = \bigcup \{\tau[\mathbf{x}] \mid \tau \in [\![\boldsymbol{S}]\!]^{A}\sigma\},$$

where σ is any state on $\boldsymbol{A}$ such that $\sigma[\mathrm{x}] = a$. In other words, the following diagram commutes:

$$\begin{tikzcd}
\boldsymbol{Stmt}_{\mathbf{x}} \times \boldsymbol{State}(\boldsymbol{A}) \arrow[r, "\boldsymbol{SE}_{\mathbf{x}}^{A}"] \arrow[d, "\langle \boldsymbol{gn}, \boldsymbol{Rep}_{\mathbf{x}}^{A} \rangle"'] & \boldsymbol{P}(\boldsymbol{State}(\boldsymbol{A}) \cup \{\uparrow\}) \arrow[d, "\boldsymbol{RepSet}_{\mathbf{x}}^{A}"] \\
\ulcorner\boldsymbol{Stmt}_{\mathbf{x}}\urcorner \times \boldsymbol{A}^{u} \arrow[r, "\boldsymbol{se}_{\mathbf{x}}^{A}"'] & \boldsymbol{P}(\boldsymbol{A}^{u} \cup \{\uparrow\})
\end{tikzcd}$$

We will also be interested in the computability of $\boldsymbol{se}_{\mathbf{x}}^{A}$.

4.9 Representation of procedure evaluation

We will want later in section 4.11 a representation of the class $\boldsymbol{Proc}_{u\to v}$ of all $\boldsymbol{GC}$ procedures of type $u\to v$, in order to construct a universal procedure for that type. For now we consider a local version, for the subclass of $\boldsymbol{Proc}_{u\to v}$ of procedures with *auxiliary variables of a given fixed type*, which works for $\boldsymbol{ND}$ in general.

So let **a**, **b**, **c** be pariwise disjoint lists of variables, with types $\mathbf{a} : u$, $\mathbf{b} : v$ and $\mathbf{c} : w$. Let $\boldsymbol{Proc}_{\mathbf{a,b,c}}$ be the class of $\boldsymbol{ND}$ procedures of type $u\to v$, with declaration **in a out b aux c**. The *procedure evaluation function on* $\boldsymbol{A}$ *relative to* **a**, **b**, **c**

$$\boldsymbol{PE}^{\boldsymbol{A}}_{\mathbf{a,b,c}} : \boldsymbol{Proc}_{\mathbf{a,b,c}} \times \boldsymbol{A}^u \to \boldsymbol{P}(\boldsymbol{A}^v \cup \{\uparrow\}),$$

defined by

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

is represented by the function

$$\boldsymbol{pe}^{\boldsymbol{A}}_{\mathbf{a,b,c}} : \ulcorner \boldsymbol{Proc}_{\mathbf{a,b,c}} \urcorner \times \boldsymbol{A}^u \to \boldsymbol{P}(\boldsymbol{A}^v \cup \{\uparrow\}),$$

defined by

$$\boldsymbol{pe}^{\boldsymbol{A}}_{\mathbf{a,b,c}}(\ulcorner \boldsymbol{P} \urcorner, a) = \boldsymbol{P}^{\boldsymbol{A}}(a).$$

In other words, the following diagram commutes:

$$\begin{tikzcd}
\boldsymbol{Proc}_{\mathbf{a,b,c}} \times \boldsymbol{A}^u \arrow[d, "\langle \boldsymbol{gn},\, \boldsymbol{id}_{\boldsymbol{A}^u} \rangle"'] \arrow[dr, "\boldsymbol{PE}^{\boldsymbol{A}}_{\mathbf{a,b,c}}"] & \\
\ulcorner \boldsymbol{Proc}_{\mathbf{a,b,c}} \urcorner \times \boldsymbol{A}^u \arrow[r, "\boldsymbol{pe}^{\boldsymbol{A}}_{\mathbf{a,b,c}}"'] & \boldsymbol{P}(\boldsymbol{A}^v \cup \{\uparrow\})
\end{tikzcd}$$

4.10 Computability of semantic representing functions

To study the computability of the representing functions we stated early, we need the *term evaluation property*.

Definition 4.1 (Term evaluation).

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

In fact, this definition is exactly the same as that in [7, Definition 4.4], referring to ***While*** rather than ***ND*** computation. The reason is that the term evaluation function $\boldsymbol{te}^{\boldsymbol{A}}_{\mathbf{x},s}$ is only a *single-valued* function, (which is different from the other *multi-valued* representing functions), and it does not depend on non-determinism. Therefore, ***While*** computation is more appropriate here.

The term evaluation function is not always computable. However, it is ***While*** computable in most commonly used algebras such as: semi-groups, groups, rings, boolean algebras, and subalgebras [7, Examples 4.5]. So, it is reasonable to assume the *term evaluation property*, and study the computability of the other semantic representing functions (what we are very interested in) by assuming it.

From now on therefore, we assume,

Assumption 4.2 (Term evaluation Property).

The algebra $\boldsymbol{A}$ has the *term evaluation property* (***TEP***), i.e., for all $\mathbf{x}$ and s, the term evaluation representing function $\boldsymbol{te}^{A}_{\mathbf{x},s}$ is ***While*** computable on $\boldsymbol{A}^{N}$.

Remark 4.3

The ***TEP*** can be proved to hold for the array algebra $\boldsymbol{A}^{*}$ (see [7, Proposition 4.6]).

To study the computability of the semantic representing functions, we also need the following lemmas,

Lemma 4.4

(a) Given a ***While***$^{\boldsymbol{RA}}$ procedure $P : \mathbf{nat} \times u \to v$, we can construct another ***While***$^{\boldsymbol{RA}}$ procedure $Q : u \to v$ so that for all $\mathbf{x} \in A^{u}$,

$$Q^{A}(\mathbf{x}) = \bigcup_{n=0}^{\infty} P^{A}(n, \mathbf{x}).$$

(b) If P is a ***GC*** procedure, Q can also be constructed as a ***GC*** procedure.

Proof. (a) Consider the ***ND*** procedure P:

```
proc    in a : u
        in n: nat
        out b : v
```

```
begin
        S;
end
```

Q can then be constructed as follows,

```
proc    in a : u
        aux n: nat
        out b : v
begin
        n := ?;
        S;
End
```

(b) For ***GC***, the construction of *Q* from *P* is more complicated. We need to use a subroutine $\boldsymbol{notover}^{A}_{\mathbf{x}}$ ($\ulcorner \boldsymbol{S} \urcorner$, *a*, *n*) (see Appendix 6), which tells us whether the computation of $\ulcorner \boldsymbol{S} \urcorner$ with input *a*, is *over* by step *n* (see Appendix 7 for details). ■

Remark 4.5

Lemma 4.4 (b) is needed in section 4.11 for proof of Theorem 4.12.

Lemma 4.6

The function ***first*** : $\mathbf{N} \to \mathbf{N}$ is *primitive recursive*, and hence ***While*** *computable* on A^N, for *any standard* Σ*-algebra* A.[2]

Now, we give the computability theorem for the semantic representing functions.

Starting with Assumption 4.2 (the term evaluation property), we can prove the following, uniformly for all $A \in \mathbf{StdAlg}(\Sigma)$ and all $\mathbf{x}$.

Theorem 4.7

(i) The atomic statement evaluation representing function $ae_{\mathbf{x}}^{A}$, and the representing function $rest_{\mathbf{x}}^{A}$, are ***ND*** computable on A^N.

(ii) The set of leaf states representing function $ls_{\mathbf{x}}^{A}$ is ***ND*** computable on A^N.

(iii) The statement evaluation representing function $se_{\mathbf{x}}^{A}$ is ***ND*** computable on A^N.

(iv) For all $\mathbf{a},\mathbf{b},\mathbf{c}$, the procedure evaluation representing function $pe_{\mathbf{a},\mathbf{b},\mathbf{c}}^{A}$ is ***ND*** computable on A^N.

---

[2] As shown in e.g. [10], a ***PR*** (i.e., *primitive recursive*) function is ***While*** computable.

Proof. We construct ***ND*** procedures to compute the semantic representing functions as the follows (we only give the general ideas here, please refer to Appendix 8 for details).

(i) By using $\boldsymbol{te}^{A}_{\mathbf{x},s}$ as a subroutine, we construct an ***ND*** procedure $\mathbf{P}_{\mathbf{ae}}$ to compute $\boldsymbol{ae}^{A}_{\mathbf{x}}$. For $\boldsymbol{rest}^{A}_{\mathbf{x}}$, we use $\boldsymbol{te}^{A}_{\mathbf{x},\mathrm{bool}}$ to compute the boolean test in the ***ND*** procedure $\mathbf{P}_{\mathbf{rest}}$.

(ii) We construct an ***ND*** procedure $\mathbf{P}_{\mathbf{ls}}$ to compute $\boldsymbol{ls}^{A}_{\mathbf{x}}$ by using $\mathbf{P}_{\mathbf{ae}}$, ***first*** (to compute ***compstep***) and $\mathbf{P}_{\mathbf{rest}}$ as subroutines.

(iii) By Lemmas 3.14 and 4.4, we can give an ***ND*** procedure $\mathbf{P}_{\mathbf{se}}$ to compute $\boldsymbol{se}^{A}_{\mathbf{x}}$ from $\mathbf{P}_{\mathbf{ls}}$ as a subroutine.

(iv) Finally, we can give an ***ND*** procedure to compute $\boldsymbol{pe}^{A}_{\mathbf{a,b,c}}$ via $\mathbf{P}_{\mathbf{se}}$ and $\boldsymbol{te}^{A}_{\mathbf{x},v}$ as two subroutines. ■

4.11 Universal procedure for *GC*[3]

It is important to note that the procedure representing function $\boldsymbol{pe}^{A}_{\mathbf{a,b,c}}$ of section 4.9 is not *universal* for ***Proc***$(\Sigma)_{u\to v}$, (where $\mathbf{a} : u$ and $\mathbf{b} : v$). It is only 'universal' for ***ND*** procedures of type *u→v with auxiliary variables of type* ***type***(**c**). In this subsection, we will

---

[3] Cf. [7, section 4.8].

construct a universal procedure $\mathbf{Univ}^{A}_{u,v}(\ulcorner \boldsymbol{P} \urcorner, a)$ for all ***GC*** procedures $\boldsymbol{P} \in \boldsymbol{Proc}_{u\to v}$ and $a \in \boldsymbol{A}^u$. This incorporates not only the auxiliary variables of ***P***, but also *representations of their values* as (Gödel numbers of) *terms in the input variables* **a** (using localization of computation). These can then all be coded by a single number variable.

By the nature of ***GC*** statements, the semantic computation tree for ***GC*** statements is only *finitely branching*. Thus we have the following properties for the semantic computation tree of ***GC*** statements:

(i) at each step, we only have finitely many leaves, which can all be coded by a single Gödel number,

(ii) localization of computation: the output is always in the subalgebra generated from the input.

Remark 4.8

Property (ii) is also true for $\boldsymbol{While}^{\boldsymbol{RA}}$ over minimal algebras.

We will, assuming the ***TEP*** for $\boldsymbol{A}$, construct *a universal procedure* for $\boldsymbol{Proc}_{u\to v}$ on $\boldsymbol{A}$. For this, we need another representation of the "set of Leaf States" function $\boldsymbol{LS}^{\boldsymbol{A}}$ which differs in two ways from $\boldsymbol{ls}^{\boldsymbol{A}}$ in section 4.8:

(i) it is defined relative to a tuple **a** of program variables ('input variables'), which does not necessarily include all the variables in $\boldsymbol{S}$,

(ii) it has as output not a tuple of *values* in $\boldsymbol{A}$, but a tuple of *terms* in the input variables, or rather, the Gödel number of such a tuple of terms.

More precisely, given a product type $u = s_1 \times \cdots \times s_m$ and a u-tuple of variables $\mathbf{a} : u$, we define

$$\boldsymbol{lsu}_{\mathbf{a}}^{\boldsymbol{A}} : \ulcorner \boldsymbol{VarTup} \urcorner \times \ulcorner \boldsymbol{Stmt}_{\mathbf{x}} \urcorner \times \boldsymbol{A}^{u} \times \mathbf{N} \to \ulcorner \boldsymbol{TermTup} \urcorner$$

as follows: for any product type w extending u, i.e., $w = s_1 \times \cdots \times s_p$ for some $p \geq m$, and for any $\mathbf{x} : w$ extending $\mathbf{a}$ (i.e., $\mathbf{x} \equiv \mathbf{a}, \mathbf{x}_{s_{m+1}}, \ldots, \mathbf{x}_{s_p}$), and for any $\boldsymbol{S} \in \boldsymbol{Stmt}_{\mathbf{x}}$, $a \in \boldsymbol{A}^{u}$ and $n \in \mathbf{N}$,

$$\boldsymbol{lsu}_{\mathbf{a}}^{\boldsymbol{A}} (\ulcorner \mathbf{x} \urcorner , \ulcorner \boldsymbol{S} \urcorner , a, n) = \ulcorner t_n \urcorner$$

where $t_n \in \boldsymbol{TermTup}_{\mathbf{x},w}$ and $\ulcorner \boldsymbol{te}_{\mathbf{x},w}^{\boldsymbol{A}} (\ulcorner t_n \urcorner , (a, \boldsymbol{\delta}_A)) \urcorner = \ulcorner \boldsymbol{ls}_{\mathbf{x}}^{\boldsymbol{A}} (\ulcorner \boldsymbol{S} \urcorner , (a, \boldsymbol{\delta}_A), n) \urcorner$.

where $\boldsymbol{\delta}_A$ is the default tuple of type $s_{m+1} \times \cdots \times s_p$. This use of default values follows from the *initialisation condition* for output and auxiliary variables in procedures (see section 3.1 (e), (iv)). (This is also what lies behind the functionality lemma 3.16 for procedures.)

Now, consider the fact that the set of all the leaves of the semantic computation tree of $\boldsymbol{S}$ is just $\bigcup\limits_{n=0}^{\infty} \boldsymbol{LS}^{A}(\boldsymbol{S}, \sigma, n)$ (see Lemma 3.14), then we can code this from $\boldsymbol{lsu}_{\mathbf{a}}^{A}$, which is the Gödel number of the set of leaf states accumulated by a certain step.

Besides this, we also need the following definition (cf. [7, Definition 4.11 and Remark 4.12]).

Definition 4.9

For any term or term tuple t and variable tuple a, $\boldsymbol{subex}(t, \mathbf{a})$ is the result of substitute the default term $\boldsymbol{\delta}^{s}$ for all variables $\mathbf{x}^{s}$ in t *except* for the variables in $\mathbf{a}$.

Remark 4.10

(a) For all $t \in \boldsymbol{TermTup}$, $\boldsymbol{subex}(t, \mathbf{a}) \in \boldsymbol{TermTup}_{\mathbf{a}}$.

(b) $\boldsymbol{subex}$ is primitive recursive in Gödel numbers.

(c) Suppose $t : w$ and $\boldsymbol{var}(t) \subseteq \mathbf{x} \equiv \mathbf{a}, \mathbf{z}$ where $\mathbf{a} : u$. Then for $a \in \boldsymbol{A}^{u}$,

$$\boldsymbol{te}_{\mathbf{a},w}^{A}(\ulcorner \boldsymbol{subex}(t, \mathbf{a}) \urcorner, a) = \boldsymbol{te}_{\mathbf{x},w}^{A}(\ulcorner t \urcorner, (a, \boldsymbol{\delta}_A))$$

where $\boldsymbol{\delta}_A$ is the default tuple of type $\boldsymbol{type}(\mathbf{z})$. This follows the 'Substitution Lemma' in logic [4].

Lemma 4.11

The function $\boldsymbol{lsu}^A_{\mathbf{a}}$ is ***ND*** computable on A^N, for *any standard* Σ-*algebra* $\boldsymbol{A}$ (cf. [7, Lemma 4.13]).

Proof. (Outline.) We essentially redo part (i) and (ii) of Theorem 4.7 using the definition of $\boldsymbol{LS}^A$, and localised versions of $\boldsymbol{ae}^A_{\mathbf{x}}$ and $\boldsymbol{rest}^A_{\mathbf{x}}$,

$$\boldsymbol{aeu}^A : \ulcorner \boldsymbol{VarTup} \urcorner \times \ulcorner \boldsymbol{AtSt} \urcorner \rightarrow \ulcorner \boldsymbol{TermTup} \urcorner$$

where for any $\mathbf{x} : w$ and $\boldsymbol{S} \in \boldsymbol{AtSt}_{\mathbf{x}}$. We have

$$\boldsymbol{aeu}^A(\ulcorner \mathbf{x} \urcorner , \ulcorner \boldsymbol{S} \urcorner) \in \ulcorner \boldsymbol{TermTup}_{\mathbf{x},w} \urcorner ,$$

such that for any $x \in \boldsymbol{A}^w$,

$$\ulcorner \boldsymbol{te}^A_{\mathbf{x},w} (\boldsymbol{aeu}^A(\ulcorner \mathbf{x} \urcorner , \ulcorner \boldsymbol{S} \urcorner), x) \urcorner = \ulcorner \boldsymbol{ae}^A_{\mathbf{x}} (\ulcorner \boldsymbol{S} \urcorner , x) \urcorner ;$$

and (2) the function,

$$\boldsymbol{restu}^A_{\mathbf{a}} : \ulcorner \boldsymbol{VarTup} \urcorner \times \ulcorner \boldsymbol{Stmt} \urcorner \times \boldsymbol{A}^u \rightarrow \ulcorner \boldsymbol{Stmt} \urcorner$$

where for any $\mathbf{x} : w$ extending $\mathbf{a} : u$, $\boldsymbol{S} \in \boldsymbol{Stmt}$ and $a \in \boldsymbol{A}^u$,

$$\boldsymbol{restu}^{A}_{\mathbf{a}}(\ulcorner \mathbf{x} \urcorner, \ulcorner \boldsymbol{S} \urcorner, a) = \ulcorner \boldsymbol{rest}^{A}_{\mathbf{x}}(\ulcorner \boldsymbol{S} \urcorner, (a, \boldsymbol{\delta}_{A})) \urcorner$$

We can then show that,

(i) $\boldsymbol{aeu}^{A}$ is primitive recursive,

(ii) $\boldsymbol{restu}^{A}_{\mathbf{a}}$ is ***ND*** computable (by using subroutines $\langle \boldsymbol{te}^{A}_{\mathbf{a},s} \mid s \in \boldsymbol{Sort}(\Sigma) \rangle$),

(iii) $\boldsymbol{lsu}^{A}_{\mathbf{a}}$ is ***ND** computable* on ***A*** by using $\boldsymbol{aeu}^{A}$ and $\boldsymbol{restu}^{A}_{\mathbf{a}}$ as subroutines.

Note that, in (iii), the term evaluation function $\boldsymbol{te}^{A}_{\mathbf{a},s}$ is used to evaluate boolean tests in the course of defining $\boldsymbol{restu}^{A}_{\mathbf{a}}$. The one tricky point is this: how do we evaluate, using $\boldsymbol{te}^{A}_{\mathbf{a},s}$, a (Gödel number of) a term $\boldsymbol{t} \in \boldsymbol{Term}_{\mathrm{x},s}$, which contains variables in **x** other than **a**? The answer is that by Remark 4.10 (c) the evaluation of t is given by $\boldsymbol{te}^{A}_{\mathbf{a},s}(\ulcorner \boldsymbol{subex}(t, \mathbf{a}) \urcorner, a)$. ■

Theorem 4.12 (Universality characterization theorem for *GC*(Σ) computations).[4]

If ***A*** has *TEP*, then for all Σ-product types type u, v, there is a $\boldsymbol{GC}(\Sigma^{N})$ procedure

$$\mathbf{Univ}_{u,v} : \ulcorner \boldsymbol{Proc}_{u \to v} \urcorner \times u \to v$$

---

[4] Cf. [7, Theorem 4.14].

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

$$\mathbf{Univ}^{\boldsymbol{A}}_{u,v}(\ulcorner \boldsymbol{P} \urcorner, a) = \boldsymbol{P}^{\boldsymbol{A}}(a).$$

Proof. We give an informal description of the algorithm represented by the procedure $\mathbf{Univ}^{\boldsymbol{A}}_{u,v}$. With input $(\ulcorner \boldsymbol{P} \urcorner, a)$, where $\boldsymbol{P} \in \boldsymbol{Proc}_{u\to v}$ and $a \in \boldsymbol{A}^u$, suppose

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

where $\mathbf{a} : u$, $\mathbf{b} : v$ and let $\mathbf{x} \equiv \mathbf{a}, \mathbf{b}, \mathbf{c}$.

By the techniques of Lemma 4.4 (b), we can then define a ***GC*** procedure

$$Q : \mathbf{nat} \times u \to \mathbf{nat},$$

where

$$Q^{\boldsymbol{A}} : \ulcorner \boldsymbol{Proc}_{u\to v} \urcorner \times \boldsymbol{A}^u \to \boldsymbol{P}(\mathbf{N} \cup \{\uparrow\})^+,$$

with

$$Q^{\boldsymbol{A}}(\ulcorner \boldsymbol{P} \urcorner, a) = \bigcup_{n=0}^{\infty} \{\boldsymbol{lsu}^{\boldsymbol{A}}_{\mathbf{a}}(\ulcorner \mathbf{x} \urcorner, \ulcorner \boldsymbol{S} \urcorner, a, n)\}.$$

Here we use the subroutine $\boldsymbol{notoveru}^{\boldsymbol{A}}_{\mathbf{a}}$ (cf. the definition for $\boldsymbol{lsu}^{\boldsymbol{A}}_{\mathbf{a}}$), which is a “localized” version of $\boldsymbol{notover}^{\boldsymbol{A}}_{\mathbf{x}}$ (see Appendix 6).

Write the elements of the output set of Q^A as

$$\ulcorner t, t', t'' \urcorner \in Q^A(\ulcorner \boldsymbol{P} \urcorner, a),$$

where the term tuples t, t' and t'' represent the current values of **a**, **b** and **c**, respectively. The function Q^A is ***GC***-computable by Lemma 4.4 (b) and 4.11, and the *TEP* Assumption.

Finally, we get the desired output values in $\boldsymbol{A}^v$ from t' as

$$\boldsymbol{te}^A_{\mathbf{a},v}(\ulcorner \boldsymbol{subex}(t', \mathbf{a}) \urcorner, a)$$

which is ***GC***-computable by the *TEP*. ■

Note 4.13

The universal procedure at a Σ-type $u \to v$ is constructed uniformly over ***StdAlg***(Σ) relative to a *term evaluation subroutine* (or “oracle”).

Moreover, we have,

Corollary 4.14 (Universality for A^*)

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

$$\mathbf{Univ}^{*}_{u,v} : \ulcorner \boldsymbol{Proc}_{u\to v} \urcorner \times u \to v$$

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

$$\mathbf{Univ}^{*,\boldsymbol{A}}_{u,v}(\ulcorner \boldsymbol{P} \urcorner, a) = \boldsymbol{P}^{\boldsymbol{A}}(a).$$

Proof. By Remark 4.3, $\boldsymbol{A}^{*}$ has ***TEP*** (cf. [7, Corollary 4.15]). ■

Corollary 4.15 (Universal $\boldsymbol{GC}^{\boldsymbol{N}}$ procedure for $\boldsymbol{GC}^{*}$)[5]

If ***A*** has *TEP*, then for all Σ-product types type *u*, *v*, there is a $\boldsymbol{GC}(\Sigma^{\boldsymbol{N}})$ procedure

$$\mathbf{Univ}_{u,v} : \ulcorner \boldsymbol{Proc}_{u\to v} \urcorner \times u \to v$$

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

$$\mathbf{Univ}^{\boldsymbol{A}}_{u,v}(\ulcorner \boldsymbol{P} \urcorner, a) = \boldsymbol{P}^{\boldsymbol{A}}(a).$$

Proof. The result follows from Theorem 4.12 by using a Σ^{*} / Σ conservativity theorem (see [7, Theorem 3.63]). ■

---

[5] Cf. [7, Theorem 4.17].

CONCLUSION

We investigated the semantics and computation theories of two *non-deterministic* programming languages over many-sorted signatures Σ, and Σ-algebras ***A***, extending the ***While***(Σ) language studied in [7]: (a) ***GC***(Σ), the *Guarded Command* language of Dijkstra [3], and (b) $\boldsymbol{While}^{\boldsymbol{RA}}$(Σ), which contains *random assignments*. These two languages were also combined into a single language ***ND***.

It was found that the *algebraic operational semantics* used in [7] for ***While*** could be generalized smoothly to the whole of ***ND***, mainly by replacing computation sequences by *semantic computation trees*.

However, when the possibility of generalizing the *Universal Function Theorem* (*UFT*) in [7] to ***ND*** was investigated, a sharp distinction was found between ***GC*** and $\boldsymbol{While}^{\boldsymbol{RA}}$. The crucial issues here seem to be (i) *finite nondeterminism*, which says that the semantic computation tree is finitely branching, and (ii) *localization of computation*, which says that the output is always in the Σ-subalgebra of ***A*** generated from the input. It was found that the techniques of [7] could be adapted to proving a UFT for ***GC***, which satisfies both these properties, but not for $\boldsymbol{While}^{\boldsymbol{RA}}$, which satisfies neither.

Thus the UFT was proved for ***GC***, assuming a *term evaluation property* on ***A***.

Future investigations in this area should include:

- investigating the UFT for ***While***^{RA}, and
- studying semicomputability properties of ***GC*** and ***While***^{RA}.

BIBLIOGRAPHY

[1] Dirk van Dalen. Logic and Structure. Springer-Verlag, 1997.

[2] Martin D. Davis and Elaine J. Weyuker. Computability, Complexity and Languages. *Fundamentals of Theoretical Computer Science*. Academic Press Inc., Orlando, Florida, 1983.

[3] Edsger W. Dijkstra. A discipline of programming. *Series in Automatic Computation*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1976.

[4] V. Sperschneider and G. Antoniou. Logic: A Foundation for Computer Science. Addison-Wesley Publishing Company Inc, 1991.

[5] J. V. Tucker and J. I. Zucker. Program Correctness over Abstract Data Types with Error-State Semantics, North Holland, Amsterdam, 1988.

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

[7] J. V. Tucker and J. I. Zucker. Computable functions and semicomputable sets on many-sorted algebras. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 5, pages 317 – 523. Oxford University Press, 2000.

[8] J. V. Tucker and J. I. Zucker. Abstract versus Concrete Computation on Metric Partial Algebras. Technical report CAS–01–01–JZ, Department of Computing and Software, McMaster University, 2001.

[9] A. M. Turing. On computable numbers, with an application to the Entscheidungs problem, *Proceedings of the London Mathematical Society* 42: pp. 230 – 265; correction [1937], *ibid*. 42, pp. 544 – 546. Reprinted [1965], *The Undecidable*, M. Davis, ed., Raven Press, 1936.

[10] J. I. Zucker and L. Pretorius. Introduction to computability theorey, *South African Computer Journal*, 9. April 1993.

APPENDIX

In this appendix, we give the proof of the important theorems and lemmas stated in the previous chapters.

Firstly, we give the proof of the functionality lemma for terms. This lemma together with the functionality lemma for statements and procedures, which are stated and proved in section 3, are crucial to ensure the semantics of the terms, statements and procedures are well defined from the states. Lemma 3.13 and 3.14 are important to prove the functionality lemma for statements.

Lemma 3.9 is crucial to prove theorem 3.8, which shows the i/o semantics of ***ND*** statements, derived from our algebraic operational semantics.

By proving Theorem 4.7, we get a weaker ***UFT*** for fixed input, output and auxiliary variables.

$\boldsymbol{notover}_{\mathbf{x}}^{\boldsymbol{A}}$, the representation function of ***NotOver***, is used as a subroutine in the proof of Lemma 4.4 (b) and Theorem 4.12 (***UFT*** for ***GC***).

1. Lemma 3.5 (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 $[\![t]\!]^{\boldsymbol{A}}\sigma_1 = [\![t]\!]^{\boldsymbol{A}}\sigma_2$.

Proof. By structural induction on t.

Base case: $t \equiv \mathbf{x}$

By definition, it's trivial to have $[\![\mathbf{x}]\!]^{\boldsymbol{A}}\sigma_1 = [\![\mathbf{x}]\!]^{\boldsymbol{A}}\sigma_2$.

Inductive step: $t \equiv \boldsymbol{F}(t_1, \ldots, t_m)$, where $\boldsymbol{F} \in \boldsymbol{Func}(\Sigma)_{u \to s}$ for $u = s_1 \times \cdots \times s_m$ and $t_i \in \boldsymbol{Term}_{s_i}$ for $i = 1, \ldots, m$.

$$\begin{aligned} \text{By the definition, } [\![t]\!]^{\boldsymbol{A}}\sigma_1 &= [\![\boldsymbol{F}(t_1, \ldots, t_m)]\!]^{\boldsymbol{A}}\sigma_1 \\ &= \boldsymbol{F}^{\boldsymbol{A}}([\![t_1]\!]^{\boldsymbol{A}}\sigma_1, \ldots, [\![t_m]\!]^{\boldsymbol{A}}\sigma_1) \qquad (1.1) \\ [\![t]\!]^{\boldsymbol{A}}\sigma_2 &= [\![\boldsymbol{F}(t_1, \ldots, t_m)]\!]^{\boldsymbol{A}}\sigma_2 \\ &= \boldsymbol{F}^{\boldsymbol{A}}([\![t_1]\!]^{\boldsymbol{A}}\sigma_2, \ldots, [\![t_m]\!]^{\boldsymbol{A}}\sigma_2) \qquad (1.2) \end{aligned}$$

By $\sigma_1 \approx \sigma_2$ (rel $\boldsymbol{var}(t)$), we have $\sigma_1 \approx \sigma_2$ (rel $\boldsymbol{var}(t_i)$), for $i = 1, \ldots, m$.

Then, by the base case, we have $[\![t_i]\!]^{\boldsymbol{A}}\sigma_1 = [\![t_i]\!]^{\boldsymbol{A}}\sigma_2$, for $i = 1, \ldots, m$.

So, (1.1) = (1.2); i.e., $[\![t]\!]^{\boldsymbol{A}}\sigma_1 = [\![t]\!]^{\boldsymbol{A}}\sigma_2$. ■

2. Lemma 3.9

Assume $n > 0$.

(a) If $\boldsymbol{S}_{\text{at}} \in \boldsymbol{AtSt}$, $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_{\text{at}}, \sigma, n)$ is formed by *attaching to* the root $\{\sigma\}$, the leaf $\{\tau\}$, for each $\tau \in \lhd \boldsymbol{S}_{\text{at}} \rhd^{\boldsymbol{A}} \sigma$.

Proof. (Trivially) For $\boldsymbol{S}_{\text{at}} \in \boldsymbol{AtSt}$, by defintion[1], $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_{\text{at}}, \sigma, n)$ is formed by *attaching to* the root $\{\sigma\}$, the leaf $\{\tau\}$, for each $\tau \in \lhd \boldsymbol{S}_{\text{at}} \rhd^{\boldsymbol{A}} \sigma$. ■

(b) If $\boldsymbol{S} \equiv \boldsymbol{S}_1; \boldsymbol{S}_2$, $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n)$ is formed by *attaching subtree('s)* $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_2, \tau, n\text{-}d')$ to each leaf $\{\tau\}$ of $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma, n)$, where d' is the depth of $\{\tau\}$ in $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma, n)$.

Proof. We split the proof into 2 cases on whether $\boldsymbol{S}_1$ is atomic or not.

Case 1: If $\boldsymbol{S}_1$ is atomic, then by definition, $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n)$ is formed by attaching to the root $\{\sigma\}$, the subtree $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}', \sigma', n\text{-}1)$, for each $\sigma' \in \boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$ and $\boldsymbol{S}' \in \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$ (2.1)

Since $\boldsymbol{S}_1$ is atomic, in this case,

$$\boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma) = \lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}} \sigma = \lhd \boldsymbol{First}(\boldsymbol{S}_1) \rhd^{\boldsymbol{A}} \sigma = \lhd \boldsymbol{S}_1 \rhd^{\boldsymbol{A}} \sigma,$$

---

[1] Refer to the definition of $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}$ in section 3.4.4.

$\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma) = \{ \boldsymbol{S}_2 \}$.

Then, (2.1) turns to be, $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n)$ is formed by attaching to the root $\{\sigma\}$, the subtree $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_2, \tau, n\text{-}1)$, for each $\tau \in \lhd \boldsymbol{S}_1 \rhd^{\boldsymbol{A}} \sigma$.

From the result of (a), $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma, n)$ is a one-step tree with each leaf $\tau \in \lhd \boldsymbol{S}_1 \rhd^{\boldsymbol{A}} \sigma$, with a depth of 1.

So, (b) is proved for this case.

Case 2: (*Interesting case*) $\boldsymbol{S}_1$ is not atomic.

We use simple induction on *n* to prove (b).

Base case: n = 1.

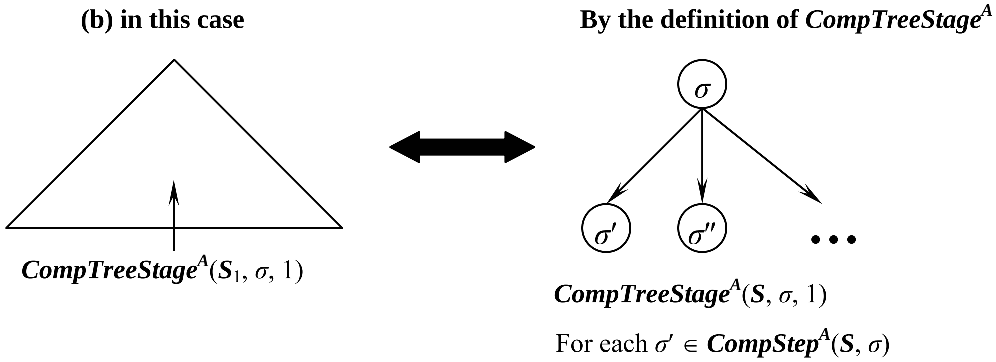
By definition, $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, 1)$ is formed by attaching to the root $\{\sigma\}$, the subtree $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}', \sigma', 0)$, for each $\sigma' \in \boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S},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

Since $\boldsymbol{S}_1$ is not atomic, $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma, 1)$ has no leaf. Then, (b) amounts to saying that $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, 1)$ is formed by $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma, 1)$. I.e., attach to the root $\{\sigma\}$, the node $\{\sigma'\}$, for each $\sigma' \in \boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma)$.

$$\text{Since } \boldsymbol{S} \equiv \boldsymbol{S}_1;\boldsymbol{S}_2,\ \boldsymbol{CompStep}^{A}(\boldsymbol{S}, \sigma) = \lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{A} \sigma = \lhd \boldsymbol{First}(\boldsymbol{S}_1) \rhd^{A} \sigma$$

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

So, (b) is proved for the base case. The following diagram might help understand the proof for this case.



Inductive step: Assume, $\boldsymbol{CompTreeStage}^{A}(\boldsymbol{S}, \sigma, n)$ is formed by *attaching subtree('s)* $\boldsymbol{CompTreeStage}^{A}(\boldsymbol{S}_2, \tau, n\text{-}d')$ to each leaf $\{\tau\}$ of $\boldsymbol{CompTreeStage}^{A}(\boldsymbol{S}_1, \sigma, n)$, where d' is the depth of $\{\tau\}$ in $\boldsymbol{CompTreeStage}^{A}(\boldsymbol{S}_1, \sigma, n)$. (Induction Hypothesis)

We want to prove: $\boldsymbol{CompTreeStage}^{A}(\boldsymbol{S}, \sigma, n+1)$ is formed by *attaching subtree('s)* $\boldsymbol{CompTreeStage}^{A}(\boldsymbol{S}_2, \tau, n+1\text{-}d')$ to each leaf $\{\tau\}$ of $\boldsymbol{CompTreeStage}^{A}(\boldsymbol{S}_1, \sigma, n+1)$, where d' is the depth of $\{\tau\}$ in $\boldsymbol{CompTreeStage}^{A}(\boldsymbol{S}_1, \sigma, n+1)$. (2.2)

By definition, $\boldsymbol{CompTreeStage}^{A}(\boldsymbol{S}, \sigma, n+1)$ is formed by attaching to the root $\{\sigma\}$, the subtree $\boldsymbol{CompTreeStage}^{A}(\boldsymbol{S}', \sigma', n)$, for each $\sigma' \in \boldsymbol{CompStep}^{A}(\boldsymbol{S}, \sigma)$ and $\boldsymbol{S}' \in \boldsymbol{Rest}^{A}(\boldsymbol{S}, \sigma)$

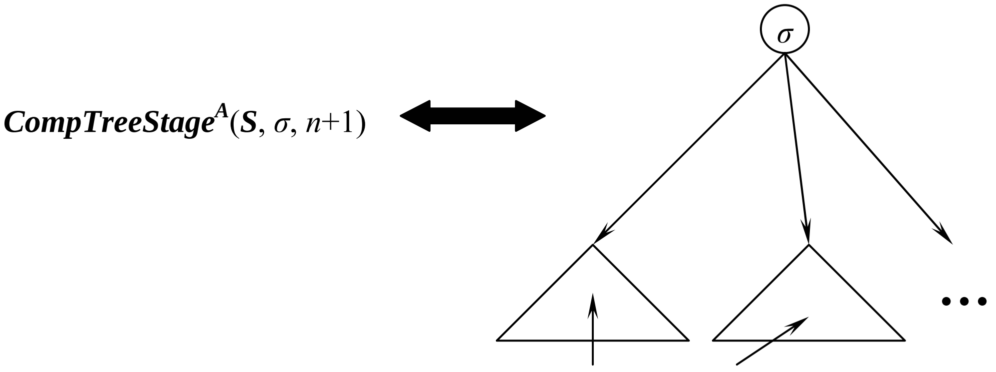
And, $\boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma) = \lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}} \sigma = \lhd \boldsymbol{First}(\boldsymbol{S}_1) \rhd^{\boldsymbol{A}} \sigma$,

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

Since $\boldsymbol{S}_1$ is not atomic, $\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma) = \bigcup \{ \boldsymbol{S}_1'; \boldsymbol{S}_2 \mid \boldsymbol{S}_1' \in \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma) \}$

Then, we can change the above result as, $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n+1)$ is formed by attaching to the root $\{\sigma\}$, the subtree $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1'; \boldsymbol{S}_2, \sigma', n)$, for each $\sigma' \in \boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma)$ and $\boldsymbol{S}_1' \in \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma)$. (2.3)

By the definition of *CompTreeStage*^{A}



$\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1'; \boldsymbol{S}_2, \sigma', n)$

For each $\sigma' \in \boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma)$ and $\boldsymbol{S}_1' \in \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma)$

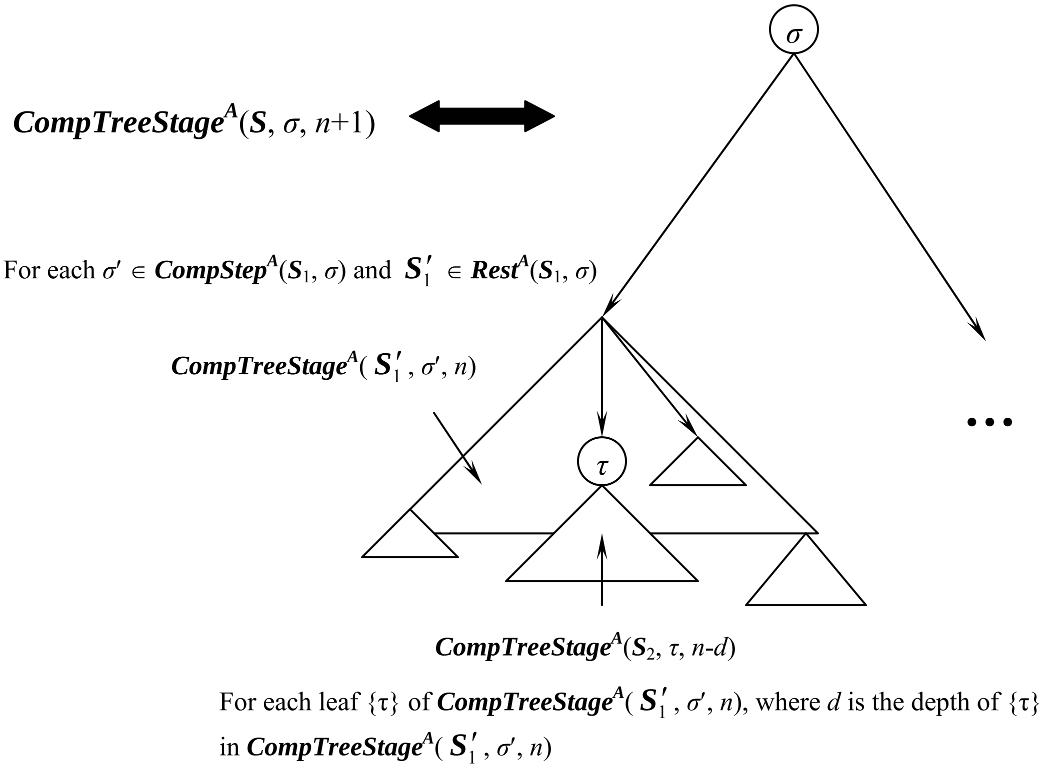
By induction hypothesis, for each $\sigma' \in \boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma)$ and $\boldsymbol{S}_1' \in \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma)$, $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1'; \boldsymbol{S}_2, \sigma', n)$ is formed by attaching $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_2, \tau, n\text{-}d)$ to each leaf {τ} of ***CompTreeStage***^{A}($\boldsymbol{S}_1'$, σ', n), where d is the depth of {τ} in ***CompTreeStage***^{A}($\boldsymbol{S}_1'$, σ', n).

Then, (2.3) turns to be, ***CompTreeStage***^{A}($\boldsymbol{S}$, σ, n+1) is formed by attaching to the root {σ}, (in 2 steps) (2.4)

(i) ***CompTreeStage***^{A}($\boldsymbol{S}_1'$, σ', n), for each $\sigma' \in$ ***CompStep***^{A}($\boldsymbol{S}_1$, σ) and $\boldsymbol{S}_1' \in$ ***Rest***^{A}($\boldsymbol{S}_1$, σ)

(ii) attach ***CompTreeStage***^{A}($\boldsymbol{S}_2$, τ, n-d) to each leaf {τ} of ***CompTreeStage***^{A}($\boldsymbol{S}_1'$, σ', n), where d is the depth of {τ} in ***CompTreeStage***^{A}($\boldsymbol{S}_1'$, σ', n)

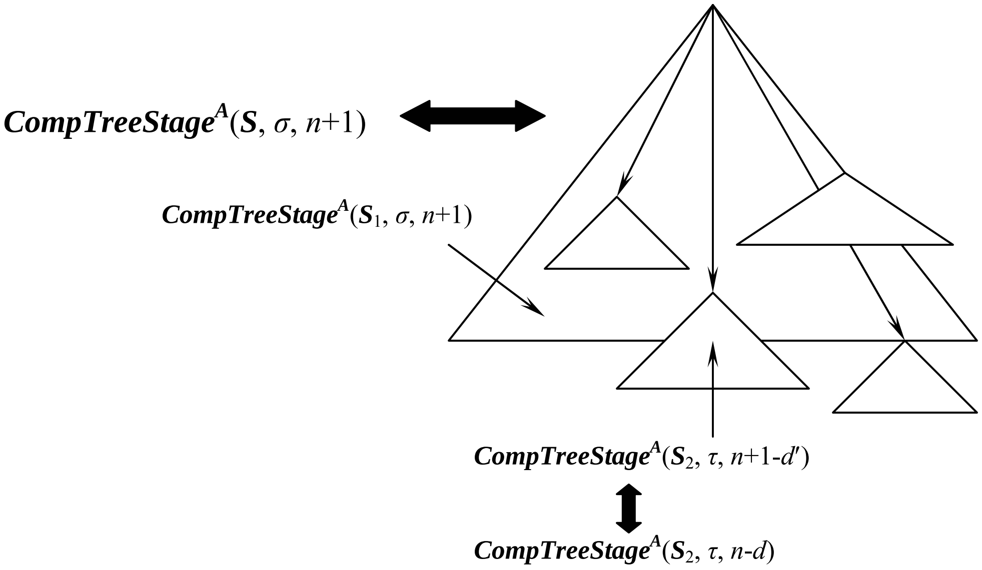
By induction hypothesis



Reversely use the definition of $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}$, then step (i) is just $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma, n+1)$. Let the depth of leaf {τ} in $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma, n+1)$ to be d'. We have $d' = d+1$, where d is the depth of {τ} in $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1', \sigma', n)$.

Then, (2.4) is just saying that, $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n+1)$ is formed by attaching $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_2, \tau, n-(d'-1))$ (= $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_2, \tau, n+1-d')$), to each leaf {τ} of $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma, n+1)$, where d' is the depth of {τ} in $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma, n+1)$.

By the definition of $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}$**, (reversely)**



For each leaf {τ} of $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma, n+1)$, where d' is the depth of {τ} in $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma, n+1)$. It's easy to see that $d' = d+1$.

The above result is just (2.2), what we want to prove. ■

(c) If $\boldsymbol{S} \equiv \textbf{if } b_1 \rightarrow \boldsymbol{S}_1 \mid \ldots \mid b_k \rightarrow \boldsymbol{S}_k \textbf{ fi}$. $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n)$ is formed by *attaching to* the root $\{\sigma\}$, the subtree('s) $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_i, \sigma, n\text{-}1)$, where $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \textbf{tt}$, for all $i = 1, \ldots, k$.

Proof. By definition, $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n)$ is formed by attaching to the root $\{\sigma\}$, the subtree $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}', \sigma', n\text{-}1)$, for each $\sigma' \in \boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$ and $\boldsymbol{S}' \in \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$.

And, $\boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma) = \lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}} \sigma = \lhd \textbf{skip} \rhd^{\boldsymbol{A}} \sigma = \{\sigma\}$

$$\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma) = \bigcup_{i=1}^{k} \{ \boldsymbol{S}_i \mid [\![b_i]\!]^{\boldsymbol{A}}\sigma = \textbf{tt} \}$$

So, it's trivial to see that, $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n)$ is formed by *attaching to* the root $\{\sigma\}$, the subtree('s) $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_i, \sigma, n\text{-}1)$, where $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \textbf{tt}$, for all $i = 1, \ldots, k$.

■

(d) If $\boldsymbol{S} \equiv \textbf{do } b_1 \rightarrow \boldsymbol{S}_1 \mid \ldots \mid b_k \rightarrow \boldsymbol{S}_k \textbf{ od}$. $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n)$ is formed by *attaching to* the root $\{\sigma\}$,

(i) the subtree('s) $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}_i;\boldsymbol{S}, \sigma, n\text{-}1)$, where $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \textbf{tt}$, if for some $i = 1, \ldots, k$,

(ii) the leaf $\{\sigma\}$ otherwise.

Proof. By definition, $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma, n)$ is formed by attaching to the root $\{\sigma\}$, the subtree $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}(\boldsymbol{S}', \sigma', n\text{-}1)$, for each $\sigma' \in \boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$ and $\boldsymbol{S}' \in \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$.

And, $\boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma) = \lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}} \sigma = \lhd \mathbf{skip} \rhd^{\boldsymbol{A}} \sigma = \{\sigma\}$

$$\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma) = \begin{cases} \displaystyle\bigcup_{i=1}^{k} \{\, \boldsymbol{S}_i ; \boldsymbol{S} \mid [b_i]^{\boldsymbol{A}} \sigma = \mathbf{tt} \} & \text{if for some } i, [b_i]^{\boldsymbol{A}} \sigma = \mathbf{tt} \\ \{\mathbf{skip}\} & \text{otherwise} \end{cases}$$

So, it is easy to see (d) is true. ■

3. Proof of Theorem 3.8 from Lemma 3.9

(a) For $\boldsymbol{S}_{\text{at}}$ atomic, $[\![\boldsymbol{S}_{\text{at}}]\!]^{\boldsymbol{A}} = \lhd \boldsymbol{S}_{\text{at}} \rhd^{\boldsymbol{A}}$.

Proof. From Lemma 3.9 (a), take the 'limit' over n for all $\boldsymbol{CompTreeStage}^{\boldsymbol{A}}$, then we have, $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}_{\text{at}}, \sigma)$ is formed by attaching to the root $\{\sigma\}$, the leaf $\{\tau\}$, for each $\tau \in \lhd \boldsymbol{S}_{\text{at}} \rhd^{\boldsymbol{A}} \sigma$, $\boldsymbol{S}_{\text{at}} \in \boldsymbol{AtSt}$.

By definition[2], $[\![\boldsymbol{S}]\!]^{\boldsymbol{A}} \sigma$ is the set of states at all leaves in $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$. I.e., $[\![\boldsymbol{S}_{\text{at}}]\!]^{\boldsymbol{A}} \sigma = \lhd \boldsymbol{S}_{\text{at}} \rhd^{\boldsymbol{A}} \sigma$ ■

---

[2] Refer to the definition of $[\![\boldsymbol{S}]\!]^{\boldsymbol{A}}$ in section 3.5.

(c) $\boldsymbol{S} \equiv \textbf{if } b_1 \to \boldsymbol{S}_1 \mid \ldots \mid b_k \to \boldsymbol{S}_k \textbf{ fi}$. Then, $[\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma = \bigcup_{i=1}^{k} \{ [\![\boldsymbol{S}_i]\!]^{\boldsymbol{A}}\sigma \mid [\![b_i]\!]^{\boldsymbol{A}}\sigma = \textbf{tt} \}$.

Proof. From Lemma 3.9 (c), take the 'limit' over n for all ***CompTreeStage***$^{\boldsymbol{A}}$, then we have, ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S},\sigma)$ is formed by attaching to the root $\{\sigma\}$, the subtree ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S}_i,\sigma)$, where $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \textbf{tt}$, for all $i = 1, \ldots, k$.

So, the leaves of ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S},\sigma)$ are formed from all the leaves of ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S}_i,\sigma)$, where $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \textbf{tt}$, for all $i = 1, \ldots, k$.

Also (trivially), if there exists an infinite path in any possible ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S}_i,\sigma)$, where $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \textbf{tt}$, for all $i = 1, \ldots, k$, there must be an infinite path in ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S},\sigma)$, by extending the infinite path in ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S}_i,\sigma)$ one step up to the root $\{\sigma\}$.

By the definition (of semantics of ***ND*** statements), $[\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma$ is the set of states at all leaves in ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$, together with '↑' if ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$ has an infinite path.

$\bigcup_{i=1}^{k} \{ [\![\boldsymbol{S}_i]\!]^{\boldsymbol{A}}\sigma \mid [\![b_i]\!]^{\boldsymbol{A}}\sigma = \textbf{tt} \}$ is the set of states at all leaves in any possible ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S}_i,\sigma)$, together with '↑' if there is an infinite path in any ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S}_i,\sigma)$, where $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \textbf{tt}$, for all $i = 1, \ldots, k$.

So, $[\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma = \bigcup_{i=1}^{k} \{ [\![\boldsymbol{S}_i]\!]^{\boldsymbol{A}}\sigma \mid [\![b_i]\!]^{\boldsymbol{A}}\sigma = \textbf{tt} \}$. ■

(d) $\boldsymbol{S} \equiv \textbf{do}\ b_1 \rightarrow \boldsymbol{S}_1 \mid \ldots \mid b_k \rightarrow \boldsymbol{S}_k\ \textbf{od}$. Then,

$$[\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma = \begin{cases} \bigcup_{i=1}^{k} \{[\boldsymbol{S}_i;\boldsymbol{S}]^{\boldsymbol{A}}\sigma \mid [b_i]^{\boldsymbol{A}}\sigma = \textbf{tt}\} & \text{if for some } i, [b_i]^{\boldsymbol{A}}\sigma = \textbf{tt} \\ \{\sigma\} & \text{otherwise} \end{cases}$$

Proof. From Lemma 3.9 (d), take the 'limit' over n for all ***CompTreeStage***$^{\boldsymbol{A}}$, then we have, ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S},\sigma)$ is formed by attaching to the root $\{\sigma\}$,

(i) the subtree ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S}_i;\boldsymbol{S},\sigma)$, for those i for which $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \textbf{tt}$,

if for some i, $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \textbf{tt}$

(ii) the leaf $\{\sigma\}$ otherwise

So, the leaves of ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S},\sigma)$ are formed from,

(i) the leaves of the subtree ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S}_i;\boldsymbol{S},\sigma)$, for those i for which $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \textbf{tt}$,

if for some i, $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \textbf{tt}$

(ii) the leaf $\{\sigma\}$ otherwise

Also (trivially), if there exists an infinite path in any possible ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S}_i;\boldsymbol{S},\sigma)$, where $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \textbf{tt}$, for all $i = 1, \ldots, k$, there must be an infinite path in ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S},\sigma)$, by extending the infinite path in ***CompTree***$^{\boldsymbol{A}}(\boldsymbol{S}_i,\sigma)$ one step up to the root $\{\sigma\}$.

By the definition (of semantics of ***ND*** statements), $[\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma$ is the set of states at all leaves in $\boldsymbol{CompTree^A}(\boldsymbol{S}, \sigma)$, together with '↑' if $\boldsymbol{CompTree^A}(\boldsymbol{S}, \sigma)$ has an infinite path.

$\bigcup_{i=1}^{k}$ { $[\![\boldsymbol{S_i};\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma \mid [\![b_i]\!]^{\boldsymbol{A}}\sigma = \mathbf{tt}$ } is the set of states at all leaves in any possible $\boldsymbol{CompTree^A}(\boldsymbol{S_i};\boldsymbol{S},\sigma)$, together with '↑' if there is an infinite path in any $\boldsymbol{CompTree^A}(\boldsymbol{S_i},\sigma)$, where $[\![b_i]\!]^{\boldsymbol{A}}\sigma = \mathbf{tt}$, for all $i = 1, \ldots, k$.

$$\text{So, } [\![\boldsymbol{S}]\!]^{\boldsymbol{A}}\sigma = \begin{cases} \bigcup_{i=1}^{k} \{[\boldsymbol{S}_i;\boldsymbol{S}]^{A}\sigma \mid [b_i]^{A}\sigma = \mathbf{tt}\} & \text{if for some } i, [b_i]^{A}\sigma = \mathbf{tt} \\ \{\sigma\} & \text{otherwise} \end{cases} \qquad \blacksquare$$

4. Lemma 3.13

Suppose $\boldsymbol{var}(\boldsymbol{S}) \subseteq \boldsymbol{M}$. If $\sigma_1 \approx \sigma_2$ (rel $\boldsymbol{M}$), then

$$\lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}}\sigma_1 \approx \lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}}\sigma_2 \text{ (rel } \boldsymbol{M}), \qquad (4.1)$$

$$\text{and, } \boldsymbol{Rest^A}(\boldsymbol{S}, \sigma_1) = \boldsymbol{Rest^A}(\boldsymbol{S}, \sigma_2). \qquad (4.2)$$

Proof. Firstly, we prove (4.1) by structural induction on ***S***.

Base case: ***S*** is atomic. By definition of ***First***, $\boldsymbol{First}(\boldsymbol{S}) = \boldsymbol{S}$, and

$$\lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}}\sigma_1 = \lhd \boldsymbol{S} \rhd^{\boldsymbol{A}}\sigma_1$$

$$\lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}} \sigma_2 = \lhd \boldsymbol{S} \rhd^{\boldsymbol{A}} \sigma_2$$

(i) $\boldsymbol{S} \equiv$ **skip**. $\{\sigma_1\} \approx \{\sigma_2\}$ (rel $\boldsymbol{M}$).

(ii) $\boldsymbol{S} \equiv \mathbf{x} := t$. $\lhd \boldsymbol{S} \rhd^{\boldsymbol{A}} \sigma_1 = \{ \sigma_1\{\mathbf{x} / [\![t]\!]^{\boldsymbol{A}}\sigma_1\} \}$

$$\lhd \boldsymbol{S} \rhd^{\boldsymbol{A}} \sigma_2 = \{ \sigma_2\{\mathbf{x} / [\![t]\!]^{\boldsymbol{A}}\sigma_2\} \}$$

$$\forall \mathbf{y} \in \boldsymbol{M},\ \{ \sigma_1\{\mathbf{x} / [\![t]\!]^{\boldsymbol{A}}\sigma_1\} \}(\mathbf{y}) = \begin{cases} [t]^{\boldsymbol{A}} \sigma_1 & \mathbf{y} \equiv \mathbf{x} \\ \sigma_1(\mathbf{y}) & \mathbf{y} \neq \mathbf{x} \end{cases}$$

$$\{ \sigma_2\{\mathbf{x} / [\![t]\!]^{\boldsymbol{A}}\sigma_2\} \}(\mathbf{y}) = \begin{cases} [t]^{\boldsymbol{A}} \sigma_2 & \mathbf{y} \equiv \mathbf{x} \\ \sigma_2(\mathbf{y}) & \mathbf{y} \neq \mathbf{x} \end{cases}$$

Because $\boldsymbol{var}(t) \subseteq \boldsymbol{var}(\boldsymbol{S}) \subseteq \boldsymbol{M}$, and $\sigma_1 \approx \sigma_2$ (rel $\boldsymbol{M}$), by functionality lemma for terms, $[\![t]\!]^{\boldsymbol{A}}\sigma_1 = [\![t]\!]^{\boldsymbol{A}}\sigma_2$, and $\forall \mathbf{y} \in \boldsymbol{M}$, $\sigma_1(\mathbf{y}) = \sigma_2(\mathbf{y})$.

(iii) $\boldsymbol{S} \equiv \mathbf{x} := \mathbf{?}$.

$\lhd \boldsymbol{S} \rhd^{\boldsymbol{A}} \sigma_1 = \{ \sigma_1' \mid \sigma_1'$ agrees with σ_1 on all variables, except $\mathbf{x}\}$

$\lhd \boldsymbol{S} \rhd^{\boldsymbol{A}} \sigma_2 = \{ \sigma_2' \mid \sigma_2'$ agrees with σ_2 on all variables, except $\mathbf{x}\}$

Since $\sigma_1 \approx \sigma_2$ (rel $\boldsymbol{M}$), then we have $\forall \sigma_1' \in \lhd \boldsymbol{S} \rhd^{\boldsymbol{A}} \sigma_1$, $\exists \sigma_2' \in \lhd \boldsymbol{S} \rhd^{\boldsymbol{A}} \sigma_2$, $\sigma_1'(\mathbf{x}) = \sigma_2'(\mathbf{x})$. (4.3)

$\forall \mathbf{y} \in \boldsymbol{M}$, let $\sigma_1' \in \lhd \boldsymbol{S} \rhd^{\boldsymbol{A}} \sigma_1$, $\sigma_2' \in \lhd \boldsymbol{S} \rhd^{\boldsymbol{A}} \sigma_2$,

$$\sigma_1'(\mathbf{y}) = \begin{cases} \sigma_1'(\mathbf{x}) & \mathbf{y} \equiv \mathbf{x} \\ \sigma_1(\mathbf{y}) & \mathbf{y} \neq \mathbf{x} \end{cases} \quad \text{and} \quad \sigma_2'(\mathbf{y}) = \begin{cases} \sigma_2'(\mathbf{x}) & \mathbf{y} \equiv \mathbf{x} \\ \sigma_2(\mathbf{y}) & \mathbf{y} \neq \mathbf{x} \end{cases}$$

By (4.3), we have $\forall \sigma_1' \in \lhd \boldsymbol{S} \rhd^{\boldsymbol{A}}\sigma_1, \exists \sigma_2' \in \lhd \boldsymbol{S} \rhd^{\boldsymbol{A}}\sigma_2, \sigma_1'(\mathbf{y}) = \sigma_2'(\mathbf{y})$.

Similarly, $\forall \sigma_2' \in \lhd \boldsymbol{S} \rhd^{\boldsymbol{A}}\sigma_2, \exists \sigma_1' \in \lhd \boldsymbol{S} \rhd^{\boldsymbol{A}}\sigma_1, \sigma_1'(\mathbf{y}) = \sigma_2'(\mathbf{y})$.

So, finally by definition 3.10, $\lhd \boldsymbol{S} \rhd^{\boldsymbol{A}}\sigma_1 \approx \lhd \boldsymbol{S} \rhd^{\boldsymbol{A}}\sigma_2$ (rel $\boldsymbol{M}$) is proved (i.e., $\lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}}\sigma_1 \approx \lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}}\sigma_2$ (rel $\boldsymbol{M}$)). I.e., base case is proved.

Inductive step: if $\boldsymbol{S}$ is not atomic, since $\boldsymbol{First}(\boldsymbol{S})$ is atomic, by base case, we have

$$\lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}}\sigma_1 \approx \lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}}\sigma_2 \text{ (rel } \boldsymbol{M}\text{)}$$

Secondly, we prove (4.2) by structural induction on $\boldsymbol{S}$.

Base case: $\boldsymbol{S}$ is atomic. $\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_1) = \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_2) = \{\mathbf{skip}\}$

Inductive step: if $\boldsymbol{S}$ is not atomic, we will prove (4.2) as the follows,

(i) $\boldsymbol{S} \equiv \boldsymbol{S}_1 ; \boldsymbol{S}_2$,

(a) If $\boldsymbol{S}_1$ is atomic, $\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_1) = \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_2) = \{\boldsymbol{S}_2\}$

(b) If $\boldsymbol{S}_1$ is not atomic,

$$\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_1) = \{\boldsymbol{S}_1'; \boldsymbol{S}_2 \mid \boldsymbol{S}_1' \in \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma_1)\}$$

$$\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_2) = \{\boldsymbol{S}_1''; \boldsymbol{S}_2 \mid \boldsymbol{S}_1'' \in \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma_2)\}$$

By base case, $\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma_1) = \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}_1, \sigma_2)$.

So, $\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_1) = \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_2)$

(ii) $\boldsymbol{S} \equiv \textbf{if}\ b_1 \rightarrow \boldsymbol{S}_1 \mid \ldots \mid b_k \rightarrow \boldsymbol{S}_k\ \textbf{fi}$. Then,

$$\boldsymbol{Rest}^{A}(\boldsymbol{S},\sigma_1) = \bigcup_{i=1}^{k} \{\ \boldsymbol{S}_i \mid [\![b_i]\!]^{\boldsymbol{A}}\sigma_1 = \textbf{tt}\ \}$$

$$\boldsymbol{Rest}^{A}(\boldsymbol{S},\sigma_2) = \bigcup_{i=1}^{k} \{\ \boldsymbol{S}_i \mid [\![b_i]\!]^{\boldsymbol{A}}\sigma_2 = \textbf{tt}\ \}$$

Since $\boldsymbol{var}(b_i) \subseteq \boldsymbol{var}(\boldsymbol{S}) \subseteq \boldsymbol{M}$, and $\sigma_1 \approx \sigma_2$ (rel $\boldsymbol{M}$), by Lemma 3.5 (the functionality lemma for terms), we have $[\![b_i]\!]^{\boldsymbol{A}}\sigma_1 = [\![b_i]\!]^{\boldsymbol{A}}\sigma_2$, for all $i = 1,\ldots,$ k.

So, $\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_1) = \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_2)$ for this case.

(iii) $\boldsymbol{S} \equiv \textbf{do}\ b_1 \rightarrow \boldsymbol{S}_1 \mid \ldots \mid b_k \rightarrow \boldsymbol{S}_k\ \textbf{od}$. Then,

$$\boldsymbol{Rest}^{A}(\boldsymbol{S},\ \sigma_1) = \begin{cases} \bigcup_{i=1}^{k} \{\ \boldsymbol{S}_i ; \boldsymbol{S} \mid [b_i]^{A}\sigma_1 = \textbf{tt}\} & \text{if for some } i, [b_i]^{A}\sigma_1 = \textbf{tt} \\ \{\textbf{skip}\} & \text{otherwise} \end{cases}$$

$$\boldsymbol{Rest}^{A}(\boldsymbol{S},\ \sigma_2) = \begin{cases} \bigcup_{i=1}^{k} \{\ \boldsymbol{S}_i ; \boldsymbol{S} \mid [b_i]^{A}\sigma_2 = \textbf{tt}\} & \text{if for some } i, [b_i]^{A}\sigma_2 = \textbf{tt} \\ \{\textbf{skip}\} & \text{otherwise} \end{cases}$$

Similarly to (ii), we can prove $\boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_1) = \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_2)$ by Lemma 3.5. ■

5. Lemma 3.14

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

$$\boldsymbol{LS}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_1, n) \approx \boldsymbol{LS}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_2, n) \text{ (rel } \boldsymbol{M}\text{)}, \qquad (5.1)$$

where $\boldsymbol{LS^A}$ stands for “leaf states”, and $\boldsymbol{LS^A}(\boldsymbol{S}, \sigma, n)$ means the set of states at all leaves of $\boldsymbol{CompTree^A}(\boldsymbol{S}, \sigma)$ in $\boldsymbol{CompTreeStage^A}(\boldsymbol{S}, \sigma, n)$.

Proof. By simple induction on *n*.

Base case: n=0, $\boldsymbol{LS^A}(\boldsymbol{S}, \sigma_1, 0) = 0$,

$\boldsymbol{LS^A}(\boldsymbol{S}, \sigma_2, 0) = 0$.

And trivially, $0 \approx 0$ (rel $\boldsymbol{M}$).

Inductive step: Suppose (5.1) is true for *n*. (induction hypothesis)

Now, we want to prove $\boldsymbol{LS^A}(\boldsymbol{S}, \sigma_1, n+1) \approx \boldsymbol{LS^A}(\boldsymbol{S}, \sigma_2, n+1)$ (rel $\boldsymbol{M}$)

(i) If $\boldsymbol{S}$ is atomic,

$$\boldsymbol{LS^A}(\boldsymbol{S}, \sigma_1, n+1) = \lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}} \sigma_1 = \lhd \boldsymbol{S} \rhd^{\boldsymbol{A}} \sigma_1$$

$$\boldsymbol{LS^A}(\boldsymbol{S}, \sigma_2, n+1) = \lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}} \sigma_2 = \lhd \boldsymbol{S} \rhd^{\boldsymbol{A}} \sigma_2$$

Then, by lemma 3.8, $\boldsymbol{LS^A}(\boldsymbol{S}, \sigma_1, n+1) \approx \boldsymbol{LS^A}(\boldsymbol{S}, \sigma_2, n+1)$ (rel $\boldsymbol{M}$)

(ii) If $\boldsymbol{S}$ is not atomic,

$$\boldsymbol{LS^A}(\boldsymbol{S}, \sigma_1, n+1) = \bigcup \{ \boldsymbol{LS^A}(\boldsymbol{S}_1', \sigma_1', n) \mid \boldsymbol{S}_1' \in \boldsymbol{Rest^A}(\boldsymbol{S}, \sigma_1), \sigma_1' \in \boldsymbol{CompStep^A}(\boldsymbol{S}, \sigma_1) \},$$

$$\boldsymbol{LS^A}(\boldsymbol{S}, \sigma_2, n+1) = \bigcup \{ \boldsymbol{LS^A}(\boldsymbol{S}_2', \sigma_2', n) \mid \boldsymbol{S}_2' \in \boldsymbol{Rest^A}(\boldsymbol{S}, \sigma_2), \sigma_2' \in \boldsymbol{CompStep^A}(\boldsymbol{S}, \sigma_2) \}.$$

By lemma 3.8, $\boldsymbol{Rest^A}(\boldsymbol{S}, \sigma_1) = \boldsymbol{Rest^A}(\boldsymbol{S}, \sigma_2)$, and

$$\lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}} \sigma_1 \approx \lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}} \sigma_2 \text{ (rel } \boldsymbol{M}\text{)}.$$

Also since $\boldsymbol{CompStep^A}(\boldsymbol{S}, \sigma_1) = \lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}} \sigma_1$, and

$\boldsymbol{CompStep^A}(\boldsymbol{S}, \sigma_2) = \lhd \boldsymbol{First}(\boldsymbol{S}) \rhd^{\boldsymbol{A}} \sigma_2$,

we have $\boldsymbol{CompStep^A}(\boldsymbol{S}, \sigma_1) \approx \boldsymbol{CompStep^A}(\boldsymbol{S}, \sigma_2)$ (rel $\boldsymbol{M}$).

By induction hypothesis,

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

where $\boldsymbol{S}' \in \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_1)$ ($= \boldsymbol{Rest}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_2)$) and $\sigma_1' \approx \sigma_2'$ (rel $\boldsymbol{M}$), for $\sigma_1' \in \boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_1)$, $\sigma_2' \in \boldsymbol{CompStep}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_2)$.

Then, we have $\boldsymbol{LS}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_1, n+1) \approx \boldsymbol{LS}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma_2, n+1)$ (rel $\boldsymbol{M}$). ■

6. Representation of *NotOver*^{A} in *GC*

Let $\boldsymbol{Stmt}_{\mathbf{x}}$ be the class of statements with variables among **x** only. The function ***NotOver***$^{\boldsymbol{A}}$ *on* **A** *relative to* **x**

$$\boldsymbol{NotOver}_{\mathbf{x}}^{\boldsymbol{A}} : \boldsymbol{Stmt}_{\mathbf{x}} \times \boldsymbol{State}(\boldsymbol{A}) \times \mathbf{N} \to \mathbf{boolean},$$

which tests whether or not the *semantic computation tree* of **S** at σ is not over by step n, is represented by the function

$$\boldsymbol{notover}_{\mathbf{x}}^{\boldsymbol{A}} : \ulcorner \boldsymbol{Stmt}_{\mathbf{x}} \urcorner \times \boldsymbol{A}^{u} \times \mathbf{N} \to \mathbf{boolean},$$

defined by a simple tail recursion on n as the follows,

Base case: $\boldsymbol{notover}_{\mathbf{x}}^{\boldsymbol{A}}(\ulcorner \boldsymbol{S} \urcorner, a, 0) = \mathbf{tt}$.

Inductive step:

(i) for **S** atomic: $\boldsymbol{notover}_{\mathbf{x}}^{\boldsymbol{A}}(\ulcorner \boldsymbol{S} \urcorner, a, n+1) = \mathbf{ff}$,

(ii) for $\boldsymbol{S}$ not atomic: $\boldsymbol{notover}^{\boldsymbol{A}}_{\mathbf{x}}(\ulcorner \boldsymbol{S} \urcorner, a, n+1) = \quad \{\boldsymbol{notover}^{\boldsymbol{A}}_{\mathbf{x}}(\ulcorner \boldsymbol{S'} \urcorner, a', n) \mid a' \in \boldsymbol{compstep}^{\boldsymbol{A}}_{\mathbf{x}}(\ulcorner \boldsymbol{S} \urcorner, a),\ \ulcorner \boldsymbol{S'} \urcorner \in \boldsymbol{rest}^{\boldsymbol{A}}_{\mathbf{x}}(\ulcorner \boldsymbol{S} \urcorner, a)\}$.

where σ is any state on $\boldsymbol{A}$ such that $\sigma[\mathrm{x}] = a$. In other words, the following diagram commutes:

$$\begin{tikzcd}
\boldsymbol{Stmt}_{\mathbf{x}} \times \boldsymbol{State}(\boldsymbol{A}) \times \mathbf{N} \arrow[d, "\langle \boldsymbol{gn}, \boldsymbol{Rep}^{\boldsymbol{A}}_{\mathbf{x}}, \boldsymbol{id}_{\mathbf{N}} \rangle"'] \arrow[dr, "\boldsymbol{NotOver}^{\boldsymbol{A}}_{\mathbf{x}}"] & \\
\ulcorner \boldsymbol{Stmt}_{\mathbf{x}} \urcorner \times \boldsymbol{A}^{u} \times \mathbf{N} \arrow[r, "\boldsymbol{notover}^{\boldsymbol{A}}_{\mathbf{x}}"'] & \mathbf{boolean}
\end{tikzcd}$$

Note 5.1

(a) The disjunction in (ii) is finite because of the finite nondeterminism of $\boldsymbol{GC}$ and finiteness of $\boldsymbol{rest}^{\boldsymbol{A}}_{\mathbf{x}}$ (see Note 3.6 (a))[3]. Hence this function only works for $\boldsymbol{GC}$.

(b) $\boldsymbol{notover}^{\boldsymbol{A}}_{\mathbf{x}}$ is very similar to $\boldsymbol{ls}^{\boldsymbol{A}}_{\mathbf{x}}$ (see section 4.7). The difference is that the output of $\boldsymbol{ls}^{\boldsymbol{A}}_{\mathbf{x}}$ is a set of terms, but the output of $\boldsymbol{notover}^{\boldsymbol{A}}_{\mathbf{x}}$ is a boolean.

(c) This function $\boldsymbol{notover}^{\boldsymbol{A}}_{\mathbf{x}}$ is used as a subroutine in the proof of Lemma 4.4 (b) and Theorem 4.12.

---

[3] Note 3.6 (a) says that $\boldsymbol{Rest}^{\boldsymbol{A}}$ is finite, from section 4.5, hence $\boldsymbol{rest}^{\boldsymbol{A}}_{\mathbf{x}}$ is finite.

Remark 5.2

Although our syntax for ***GC*** does not allow procedures as subroutines of others, we freely use these as pseudo-code in the interest of readability. In practice, we could use macro-expansions and new auxiliary variables to get the same effect.

7. Lemma 4.4 (b): *GC* procedure for computing the *Union* function

We construct the following ***GC*** procedure *Q* to compute the *Union* function $\bigcup_{n=0}^{\infty} P^{\boldsymbol{A}}(n, \mathbf{x})$, by using ***notover***$^{\boldsymbol{A}}$ as a subroutine and a boolean auxiliary variable,

```
proc   in    $\mathbf{a} : u$
       out   $\mathbf{b} : v$
       aux   $\mathbf{gn} : \mathbf{nat}$
       aux   $n : \mathbf{nat}$
       aux   $\mathbf{continue} : \mathbf{bool}$
begin
       $\mathbf{a} := a$;
       $\mathbf{continue} := \mathbf{tt}$;
       $\mathbf{gn} := \ulcorner \boldsymbol{S} \urcorner$;
       do      $\mathbf{continue} \to n{+}{+};\ \boldsymbol{notover}^{\boldsymbol{A}}_{\mathbf{x}}(\mathbf{gn}, \mathbf{a}, \mathbf{continue})$;
```

```
                | continue → continue := ff; S;
            od
        end
```

By definition 3.14, if P is $\boldsymbol{ND}$ computable on $\boldsymbol{A}$, so is Q. Then together with the early proof for $\boldsymbol{While}^{\boldsymbol{RA}}$ (see Lemma 4.4 (a)), we finished the proof for Lemma 4.4. ■

8. Theorem 4.7

(i) The atomic statement evaluation representing function $\boldsymbol{ae}_{\mathbf{x}}^{\boldsymbol{A}}$, and the representing function $\boldsymbol{rest}_{\mathbf{x}}^{\boldsymbol{A}}$, are $\boldsymbol{ND}$ computable on $\boldsymbol{A}^{\boldsymbol{N}}$.

Proof. We give an informal description of the algorithm represented by the procedure $\mathbf{P}_{\mathbf{ae}}$, which computes $\boldsymbol{ae}_{\mathbf{x}}^{\boldsymbol{A}}$. With input ($\ulcorner \boldsymbol{S} \urcorner$, a), since Gödel numbers are primitive recursive (refer to [9]), we can judge what the atomic statement $\boldsymbol{S}$ is and thus, get the output $\mathbf{b}$ by using $\boldsymbol{te}_{\mathbf{x},s}^{\boldsymbol{A}}$ as a subroutine as follows.

(a) a, if **skip**.

(b) $\boldsymbol{te}_{\mathbf{x},u}^{\boldsymbol{A}}$ ($\ulcorner \mathbf{x} \urcorner$, a, $\mathbf{b}$), if $\mathbf{x} := t$, where for some product type u, $\mathbf{x} : u$ and $t : u$.

(c) $\boldsymbol{te}_{\mathbf{x},s}^{\boldsymbol{A}}$ ($\ulcorner \mathbf{x} \urcorner$, a, $\mathbf{b}$), if $\mathbf{x} := ?$, for $\mathbf{x} : s$.

Next, we give an informal description of the algorithm represented by the procedure $\mathbf{P}_{\mathbf{rest}}$, which computes $\boldsymbol{rest}_{\mathbf{x}}^{\boldsymbol{A}}$. And we define a “sequential operator” for Gödel numbers: $\boldsymbol{seq}(\ulcorner \boldsymbol{S}_1 \urcorner, \ulcorner \boldsymbol{S}_2 \urcorner) = \ulcorner \boldsymbol{S}_1; \boldsymbol{S}_2 \urcorner$.

Similarly with what we have done for $\boldsymbol{ae}_{\mathbf{x}}^{\boldsymbol{A}}$, since Gödel numbers are primitive recursive, we can judge what $\boldsymbol{S}$ is. Then, we can give the output in the following cases,

(a) $\ulcorner \mathbf{skip} \urcorner$, if $\boldsymbol{S}$ is atomic.

(b) $\ulcorner \boldsymbol{S}_2 \urcorner$, if $\boldsymbol{S} \equiv \boldsymbol{S}_1; \boldsymbol{S}_2$ and $\boldsymbol{S}_1$ is atomic,

$\mathbf{P}_{\mathbf{rest}}(\ulcorner \boldsymbol{S}_1 \urcorner, a, \mathbf{c}); \mathbf{b} := \boldsymbol{seq}(\mathbf{c}, \ulcorner \boldsymbol{S}_2 \urcorner);$ if $\boldsymbol{S} \equiv \boldsymbol{S}_1; \boldsymbol{S}_2$, but $\boldsymbol{S}_1$ is not atomic.

Note that: $\mathbf{c}$ is an auxiliary variable of type **nat**.

(c) If $\boldsymbol{S} \equiv \mathbf{if}\ b_1 \to \boldsymbol{S}_1 \mid \dots \mid b_k \to \boldsymbol{S}_k\ \mathbf{fi}$, then we need $\boldsymbol{te}_{\mathbf{x},\mathbf{bool}}^{\boldsymbol{A}}$ to do the boolean test, and we also need auxiliary boolean variables $\mathbf{y}_1, \dots, \mathbf{y}_k$ to construct the followings:

$\boldsymbol{te}_{\mathbf{x},\mathbf{bool}}^{\boldsymbol{A}}(\ulcorner b_1 \urcorner, a, \mathbf{y}_1);$

… …

$\boldsymbol{te}_{\mathbf{x},\mathbf{bool}}^{\boldsymbol{A}}(\ulcorner b_k \urcorner, a, \mathbf{y}_k);$

$\mathbf{if}\ \mathbf{y}_1 \to \mathbf{b} := \ulcorner \boldsymbol{S}_1 \urcorner \mid \dots \mid \mathbf{y}_k \to \mathbf{b} := \ulcorner \boldsymbol{S}_k \urcorner\ \mathbf{fi}$

(d) If $\boldsymbol{S} \equiv \mathbf{do}\ b_1 \to \boldsymbol{S}_1 \mid \dots \mid b_k \to \boldsymbol{S}_k\ \mathbf{od}$, then similarly with (c), we do the follows,

$\boldsymbol{te}_{\mathbf{x},\mathbf{bool}}^{\boldsymbol{A}}(\ulcorner b_1 \urcorner, a, \mathbf{y}_1);$

… …

$\boldsymbol{te}_{\mathbf{x},\mathbf{bool}}^{\boldsymbol{A}}(\ulcorner b_k \urcorner, a, \mathbf{y}_k);$

$\mathbf{if}\quad \mathbf{y}_1 \to \mathbf{b} := \boldsymbol{seq}(\ulcorner \boldsymbol{S}_1 \urcorner, \ulcorner \boldsymbol{S} \urcorner)$

$\mid$ … …

```
        | $\mathbf{y}_k \to \mathbf{b} := \boldsymbol{seq}(\ulcorner \mathbf{S}_k \urcorner, \ulcorner \boldsymbol{S} \urcorner)$
        | $\neg(\mathbf{y}_1 \wedge \ldots \wedge \mathbf{y}_k) \to \mathbf{b} := \ulcorner \mathbf{skip} \urcorner$
    fi
```

Note that: we use **if fi** to compute **do od**. ■

(ii) The set of leaf states representing function $\boldsymbol{ls}^{\boldsymbol{A}}_{\mathbf{x}}$ is ***ND*** computable on $\boldsymbol{A}^{\boldsymbol{N}}$.

Proof. With input ($\ulcorner \boldsymbol{S} \urcorner$, a, n_0), we construct the following ***ND*** procedure $\mathbf{P}_{\mathbf{ls}}$ to compute $\boldsymbol{ls}^{\boldsymbol{A}}_{\mathbf{x}}$ by using $\mathbf{P}_{\mathbf{ae}}$, ***first*** (to compute ***compstep***) and $\mathbf{P}_{\mathbf{rest}}$ as subroutines.

```
proc  in   a : $u$
      out  b : $v$
      aux  d : $u$
      aux  gn : nat
      aux  $n$ : nat
      aux  $m$ : nat
      aux  $l$ : nat
begin
      a := $a$;
      gn := $\ulcorner \boldsymbol{S} \urcorner$;
      $n := n_0$;
      while $n \neq 0$,
```

```
            do
                if gn is not atomic,
                    $\mathbf{P_{rest}}(\mathbf{gn}, \mathbf{a}, l)$;
                    $\boldsymbol{first}(\mathbf{gn}, m)$;
                    $\mathbf{P_{ae}}(m, \mathbf{a}, \mathbf{d})$;
                    $\mathbf{gn} := l$;
                    $\mathbf{a} := \mathbf{d}$;
                    $n := n-1$;
                else
                    $\mathbf{P_{ae}}(\mathbf{gn}, \mathbf{a}, \mathbf{b})$;
            od
        end
```

■

(iii) The statement evaluation representing function $\boldsymbol{se}_{\mathbf{x}}^{\boldsymbol{A}}$ is ***ND*** computable on $\boldsymbol{A}^{\boldsymbol{N}}$.

Proof. By Notation 3.11 and Lemma 4.4, we can give an ***ND*** procedure $\mathbf{P_{se}}$ to compute $\boldsymbol{se}_{\mathbf{x}}^{\boldsymbol{A}}$ from $\mathbf{P_{ls}}$ as a subroutine. Note that, there is an infinite path in $\boldsymbol{CompTree}^{\boldsymbol{A}}(\boldsymbol{S}, \sigma)$ if and only if, $\mathbf{P_{se}}$ diverges. ■

(iv) For all **a**,**b**,**c**, the procedure evaluation representing function $\boldsymbol{pe}^{\boldsymbol{A}}_{\mathbf{a,b,c}}$ is ***ND*** computable on $\boldsymbol{A}^{\boldsymbol{N}}$.

Proof. With input ($\ulcorner \boldsymbol{S} \urcorner$, a), we use the following ***ND*** procedure to compute $\boldsymbol{pe}^{\boldsymbol{A}}_{\mathbf{a,b,c}}$ via $\mathbf{P}_{\mathbf{se}}$ and $\boldsymbol{te}^{\boldsymbol{A}}_{\mathbf{x},v}$ as two subroutines.

```
proc  in   a : $u$
      out  b : $v$
      aux  d : $u$
      aux  gn : nat
begin
      a := $a$;
      gn := $\ulcorner S \urcorner$;
      $P_{se}$(gn, a, d);
      $te^{A}_{x,v}(\ulcorner b \urcorner, d, b)$;
end
```

■