[**1**] []**1** 1

Specifiability and Computability of functions by equations on partial algebras

By

Li Luo, B.Sc.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Science
Department of Computing and Software
McMaster University

MASTER OF SCIENCE (2003)　　McMaster University
(Computing and Software)　　Hamilton, Ontario

TITLE:
Specifiability and Computability of functions by equations on partial algebras

AUTHOR: Li Luo, B.Sc.(Beijing Institute of Technology, China)

SUPERVISOR: Dr. Jeffery I. Zucker

NUMBER OF PAGES: v, 95

Abstract

The aim of this research is to compare and contrast specifiability and computability of functions on many-sorted partial algebras A by systems of equations and conditional equations. As our model of computability, we take the system $\boldsymbol{\mu PR}^*(A)$ of primitive recursive schemes over A with added array sorts and the μ (least number) operator. We show:

(1) Any $\boldsymbol{\mu PR}^*$-computable function is specifiable (i.e. uniquely definable) by a finite set of conditional equations over A, using either Kleene's semantics, or a "strict" semantics, for the equality relation between partially defined terms; but not conversely, i.e., not all conditionally equationally specifiable functions are computable.

(2) If however we replace "unique definability" by "definability as a minimal solution" in Kleene equational logic, and if we consider only equations, not conditional equations, then we obtain the class of functions $\boldsymbol{ED}^*(A)$, which is shown to be equal to $\boldsymbol{\mu PR}^*(A)$. This equivalence provides added support for a Generalized Church-Turing Thesis. However the class $\boldsymbol{CED}^*(A)$ of minimal solutions of *conditional* equations goes beyond $\boldsymbol{\mu PR}^*(A)$ computability. In fact such functions are in $\boldsymbol{\mu PR}^*(A^{\mathsf{eq}})$, i.e., $\boldsymbol{\mu PR}^*$ over A extended by equality operators at all sorts.

Acknowledgements

I would like to thank Dr. J.I. Zucker, my supervisor, for his invaluable guidance and help in the preparation of my thesis and all the way in my studies.

I am appreciative of Dr. A. Wassyng and Dr. M. Soltys for their careful review and useful comments on my thesis. Thanks to Dr. W.M. Farmer, Dr. M.von Mohrenschildt and Dr. S. Qiao and all the other professors for their help in my studies. Thanks to my friends, especially the Ward 105 family, and all the graduate students in this department for their wonderful friendship and support during these two years.

Many thanks to my parents for their love, understanding and support.

Contents

Chapter 1

Introduction

Computability theory of functions over abstract algebra is a central concern in mathematics and computer science. Since the 1960s, many computation models, such as *imperative programming language* and *function schemes*, have been developed to describe ways of computing functions on many-sorted algebras.

All of our investigation in computability and specification is based on an abstract data model: a partial algebra A, which is given by a finite family of non-empty sets $A_{s_1}, \ldots, A_{s_n}$ called *carriers* of the algebra; and finite sets of *constants* $\mathsf{c}_1, \ldots, \mathsf{c}_n$ and *partial functions* $\mathsf{F}_1, \ldots, \mathsf{F}_n$ with a type:

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

We are mostly interested in *N-standard partial algebras* which are formed by including the set $\mathbb{B}$ of booleans and the set $\mathbb{N}$ of naturals, as well as the standard operations such as equalities on these sets.

In this chapter, we will first give a brief introduction to the background and our research objective, and then give an overview of the chapters in this thesis.

1.1 Background

1.1.1 $\boldsymbol{\mu PR^*}$ computability

Schemes for inductive definability over abstract structure have been developed by Platek [Pla66], Moschovakis [Mos84, Mos89] and Feferman [Fef96]. Tucker and Zucker generalized Kleene's schemes over $\mathbb{N}$ [Kle52] to $\boldsymbol{\mu PR}$ schemes over N-standard algebras [TZ88, TZ00].

$\boldsymbol{\mu PR}$ schemes define functions by starting with basic functions and applying *composition*, *definition by cases*, *simultaneous primitive recursion* and the *constructive least number operator* μ to these functions. A function on A is $\boldsymbol{\mu PR}^*(\Sigma)$ computable if it is defined by a $\boldsymbol{\mu PR}$ scheme over Σ^*, where Σ^* expands Σ by including the new starred (array) sorts s^* for each sort s of Σ as well as standard array operations. These define a broader class of functions than $\boldsymbol{\mu PR}$, providing a better generalization of Kleene's schemes, as we will see below.

By [TZ00], generalizing a classical result over $\mathbb{N}$ [MR67, BL74], we have

$$\boldsymbol{\mu PR}^*(A) = \boldsymbol{While}^*(A) \tag{1.1}$$

where $\boldsymbol{\mu PR}^*(A)$ denotes the class of $\boldsymbol{\mu PR}^*$ computable functions on a many-sorted N-standard partial algebra A and $\boldsymbol{While}^*(A)$ denotes the class of $\boldsymbol{While}^*$ (i.e. $\boldsymbol{While}$ with arrays) computable functions on A. $\boldsymbol{While}$ is an imperative programming language constructed from concurrent assignments, sequential composition, the conditional and the 'while' command. The *Generalized Church-Turing Thesis* [TZ88, TZ00] states that the the class of functions computable by finite deterministic algorithms on A are precisely the class given in (1.1).

1.1.2 Algebraic specification

The theory of algebraic specification, i.e, specification by formulae of a restricted syntactic class, such as equations or conditional equations, is a well-established field interesting for theoreticians and practitioners in both mathematics and computer science. It originated in the mid-seventies and has been developed in various areas, such as pure mathematics, abstract data types and software systems.

Algebraic specification of abstract data types as a rigorous mathematical approach was introduced by the ADJ-group [GTW77, GTW78], and has been further investigated, extended and defended by others [GH78, EM85]. Most writers are interested in *initial algebra* semantics, i.e. definability of an initial Σ-algebra A by a given set of equations or conditional equations. Our viewpoint is rather: given a Σ-algebra A, to consider the functions on A which are specified by a system of (conditional) equations, and show that

$$\textit{Computability} \;\Rightarrow\; \textit{Specifiability} \tag{1.2}$$

This has already been done in [TZ02] for *total algebras*. In this thesis, we extend (1.2) to *partial algebras*.

Equational definability as a model of *computation* has been investigated by Kleene over $\mathbb{N}$ [Kle52], and Moldestad and Tucker over many-sorted total algebras [MT81]. We investigated equational and conditional equational definability on many-sorted partial algebras from two viewpoints:

(a) *unique* definability, i.e. *specifiability*, and

(b) definability as a *minimal solution*, which provides a model of *computability*.

1.1.3 Computability of minimal solutions of systems of equations or conditional equations

Minimality of solutions of systems of recursive equations is connected with the *fixed point* (or denotational) semantics of a recursive formalism, given as the *least fixed point* of a continuous higher order functional Φ, i.e. a function f such that

$$\Phi(f) = f$$

which, by the Knaster-Tarski theorem [Kna28, Tar55], is obtained as the limit of a sequence of "approximation from below", i.e.

$$f = \bigcup_{i=0}^{\infty} f_i$$

where f_0 is the completely undefined function and $f_{i+1} = \Phi(f_i)$. Fixed point semantics have been investigated by Kleene for his recursive schemes [Kle52, §66], Stoy and De Bakker for the semantics of programming language [Sto77, dB80], Platek, Moschovakis and Feferman in connection with their inductive schemes (see §1.1.1 above), and also by Moldestad *et al.* [MSHT80].

We use this technique in a new setting, i.e. *conditional equations on many sorted partial algebras.*

1.2 Objectives

We compare *specifiability* and *computability* of functions on abstract partial algebras of a given signature Σ by systems of *equations* and *conditional equations*. Here, "specifiable" means uniquely definable, and "computable" corresponds (as we will see) to definable as the *minimal solution* of a set of equations.

Our work consists of two parts:

(1) To show:

$$\boldsymbol{\mu PR}^*(\Sigma)\textit{-computable on } A \;\Rightarrow\; \textit{conditional equational specifiable on } A$$

This was already shown in [TZ02] for total algebras. The new feature here is working on *partial algebras* (which are important for topological consideration [TZ99]). It is necessary to choose a logic for equations between partially defined terms. We consider and compare two kinds of logics: one based on *Kleene equality* "$\simeq$" [Kle52], and the other based on *strict equality* "$=$" [Far90, Par93, Fef95].

Note that the reverse of the above implication does not hold, i.e., *specification* goes beyond *computability*. (A counterexample is given in Remark 5.15.)

(2) To characterize a form of equational definability which *does* correspond to computability. We find that the existence of *minimal solutions* of a set of equations (using Kleene equality) gives rise to a new model of computability $\boldsymbol{ED}^*(\Sigma)$. We show that

$$\boldsymbol{ED}^*(A) = \boldsymbol{\mu PR}^*(A) \tag{1.3}$$

by proving a circle of relations

$$\boldsymbol{\mu PR}^*(A) \subseteq \boldsymbol{ED}^*(A) \subseteq \boldsymbol{Rec}^*(A) \subseteq \boldsymbol{While}^*(A) \subseteq \boldsymbol{\mu PR}^*(A)$$

where $\boldsymbol{Rec}^*(A)$ is a class of functions which can be computed by procedures with recursive procedure calls (the stars '$*$' refer to presence of array sorts). Then, (1.3) together with (1.1) gives a further confirmation to the *Generalized Church-Turing Thesis.*

We will also see (Theorem 7.19) that the class $\boldsymbol{CED}^*(A)$ of minimal solutions of *conditional* equations (with Kleene equality in the consequent and strict equality in the antecedent) takes us beyond $\boldsymbol{\mu PR}^*$ computability on A. However, if we expand A to A^{eq} by adding equality at all sorts, we get

$$\boldsymbol{CED}^*(A) \subseteq \boldsymbol{\mu PR}^*(A^{\mathsf{eq}}).$$

1.3 Overview of the thesis

This thesis consists of seven chapters.

Chapter 1 gives a brief introduction to the background and research aims, and then outlines the structure of the thesis.

Chapter 2 supplies the reader with some basic concepts and notations on *many-sorted partial algebras*, especially three kinds of expansions of such algebras: *standard algebras*, *N-standard algebras* and *starred algebras* (i.e. with array sorts) which have significant use in the later chapters.

Chapter 3 introduces the specification languages used in our research, and the two semantics (*Kleene* and *strict*) for equations between partially defined terms.

In Chapter 4, we investigate $\boldsymbol{\mu PR}^*$ computability on many-sorted N-standard partial algebras A, with the syntax and semantics of $\boldsymbol{\mu PR}^*$ schemes.

In Chapter 5, we investigate specification theories for $\boldsymbol{\mu PR}^*$ computable functions on A. We show that $\boldsymbol{\mu PR}^*$ computable functions on A are specifiable in both Kleene and strict equational logic (Theorems 5.6 and 5.12).

In Chapter 6, we prove (Theorem 6.10) the existence of a minimal solution for any system of equations, and also conditional equations (with Kleene equality in the consequent and strict equality in the antecedent) on A, by extending Kleene's approach used in his investigation of recursive functionals on $\mathbb{N}$.

In Chapter 7, we prove (Theorem 7.7) that the minimal solutions on A of equations (with Kleene equality) are computable by recursive programs. Hence, we derive (Theorem 7.18) the equivalence of the models $\boldsymbol{ED}^*(\Sigma)$, $\boldsymbol{Rec}^*(\Sigma)$, $\boldsymbol{While}^*(\Sigma)$ and $\boldsymbol{\mu PR}^*(\Sigma)$. We also show (Theorem 7.19) that class $\boldsymbol{CED}^*(A)$ of minimal solutions of conditional equations goes beyond $\boldsymbol{\mu PR}^*(A)$, but lies in $\boldsymbol{\mu PR}^*(A^{\mathsf{eq}})$. We conjecture the equivalence of $\boldsymbol{CED}^*(A)$ and $\boldsymbol{\mu PR}^*(A^{\mathsf{eq}})$ when 'eq' is interpreted as semi-equality (see Definition 2.17) at all sorts.

Chapter 2

Signatures and algebras

We start with some basic concepts and important notations to supply the readers with necessary fundamentals. Adding new data sets and operators is a key activity in our research, so, expansions and reducts of algebras are defined to track this change. We will introduce three kinds of expansions of algebras: *standard algebras*, *N-standard algebras* and *starred algebras*, which are formed by equipping algebras with Booleans, counters and arrays respectively. These three algebras have significant use in our research.

In this thesis, we are particularly interested in partial algebras, so we have:

Assumption 2.1 (Partial functions and algebras). All functions and algebras discussed below are partial except where specified as total.

Much of the content in this chapter is taken from [TZ00, §2], except for making relevant changes from total algebras to partial algebras.

2.1 Signature

Definition 2.2 (Many-sorted signatures). A signature Σ for a many-sorted algebra is a pair consisting of:

- a finite set $\boldsymbol{Sort}(\Sigma)$ of sorts
- a finite set $\boldsymbol{Func}(\Sigma)$ of primitive function symbols. Each symbol F has a type $s_1 \times \cdots \times s_m \to s$, where $m \geq 0$ is the arity of F, and $s_1, \ldots, s_m \in \boldsymbol{Sort}(\Sigma)$ are the *domain sorts* and $s \in \boldsymbol{Sort}(\Sigma)$ is the *range sort*; in such a case we write

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

  The case $m = 0$ corresponds to constant symbols; we write $\mathsf{F}\!: \to s$ or just $\mathsf{F} : s$. For convenience, we often consider constant c separately from F in inductive proof.

Remark 2.3. Our signatures do not explicitly include relation symbols; relation will be interpreted as Boolean-valued functions.

Definition 2.4 (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 $s_i \in \boldsymbol{Sort}(\Sigma)$, called its component sorts. We define $\boldsymbol{ProdType}(\Sigma)$ to be the set of Σ-product types, with elements $u,\ v,\ w$,...

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

Definition 2.5 (Partial Algebras). A partial algebra A for Σ is given by:

- a non-empty set A_s for each sort in $\boldsymbol{Sort}(\Sigma)$, called the carrier of sort s
- a partial function F^A: $A^u \to A_s$ for each Σ-function symbol F: $u \to s$, where $A^u =_{df} A_{s_1} \times \cdots \times A_{s_m}$ for a Σ-product type $u = s_1 \times \cdots \times s_m$.

Remarks 2.6.

1. We use $f : A \to B$ to denote a partial function f from A to B. if $a \notin \boldsymbol{dom}(f)$, we say $f(a)$ is *undefined* (or *divergent*), written $f(a) \uparrow$; if $a \in \boldsymbol{dom}(f)$, we say $f(a)$ is *defined* (or *convergent*), written $f(a) \downarrow$; if $a \in \mathbf{dom}(f)$ and $(a, b) \in f$, we say $f(a)$ *converges* to b, written $f(a) \downarrow b$ or $f(a) = b$.

2. If u is empty, then F is a *constant symbol* and F^A is an element of A_s.

3. Total functions are special cases of partial functions. A Σ-algebra A is a total algebra if F^A is total for each Σ-function F. In this thesis, all the functions and algebras are partial by default.

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

We will use the following notation for signatures Σ:

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

and for Σ-algebras A:

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

Examples 2.7.

1. The ring of reals $\mathcal{R}_0 = (\mathbb{R}; 0, 1, +, -, \times)$ has a signature containing the sort

real and the function symbols $0, 1 : \to \mathsf{real}$, $+, -, \times : \mathsf{real}^2 \to \mathsf{real}$. Since the signature can be inferred from the algebra, we only display the algebra in the following examples:

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

All the functions in this algebra are total, so this is a total algebra.

2. The field $\mathcal{R}_{\mathsf{inv}} = (\mathbb{R}; 0, 1, +, -, \times, \mathsf{inv})$ is formed by adding inv to the ring $\mathcal{R}_0$ where inv is the multiplicative inverse:

$$\mathsf{inv}(x) = \begin{cases} 1/x & \text{if } x \neq 0 \\ \uparrow & \text{otherwise.} \end{cases}$$

algebra	$\mathcal{R}_{\mathsf{inv}}$
import	$\mathcal{R}_0$
functions	$\mathsf{inv} : \mathbb{R} \to \mathbb{R}$
end	

This is a partial algebra since inv is a partial function.

Definition 2.8 (Reducts and expansions). Let Σ and Σ' be signatures.

1. We write $\Sigma \subseteq \Sigma'$ to mean $\textbf{\textit{Sort}}(\Sigma) \subseteq \textbf{\textit{Sort}}(\Sigma')$ and $\textbf{\textit{Func}}(\Sigma) \subseteq \textbf{\textit{Func}}(\Sigma')$.

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

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

Example 2.9. $\mathcal{R}_{\mathsf{inv}}$ (Example 2.7.(2)) is an expansion of $\mathcal{R}_0$ (Example 2.7.(1)) .

2.2 Σ-Terms

2.2.1 Syntax of terms

Let $\boldsymbol{Var}(\Sigma)$ be the class of Σ-variables $\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}\ldots$, and for each Σ-sort s, let $\boldsymbol{Var_s}(\Sigma)$ be the class of variables of sort s. Then $\mathtt{x}:s$ means $\mathtt{x}$ is a variable of sort s. For $u = s_1 \times \cdots \times s_m$, $\vec{\mathtt{x}}:u$ means $\mathtt{x}$ is a tuple of distinct variables of sorts $s_1, \ldots, s_m$.

We define the set $\boldsymbol{Term}_s(\Sigma)$ of Σ-terms of sort s by an *inductive definition*:

Base clauses:

(i) Every variable $\mathtt{x}:s$ is in $\boldsymbol{Term}_s(\Sigma)$

(ii) Every Σ-constant $\mathsf{c}:s$ is in $\boldsymbol{Term}_s(\Sigma)$

Inductive clauses:

(iii) If $\mathsf{F} \in \boldsymbol{Func}_{u\to s}(\Sigma)$, $u = s_1 \times \cdots \times s_m$ $(m > 0)$

and $t_i \in \boldsymbol{Term}_{s_i}(\Sigma)$ $(1 \leq i \leq m)$, then $\mathsf{F}(t_1, \ldots, t_m)$ is in $\boldsymbol{Term}_s(\Sigma)$.

(iv) If $\mathsf{bool} \in \boldsymbol{Sort}(\Sigma)$, $t_1 \in \boldsymbol{Term}_{\mathsf{bool}}(\Sigma)$, $t_2, t_3 \in \boldsymbol{Term}_s(\Sigma)$, then

$\mathsf{if}\ \ t_1\ \ \mathsf{then}\ \ t_2\ \ \mathsf{else}\ \ t_3\ \ \mathsf{fi}$ is in $\boldsymbol{Term}_s(\Sigma)$.

Let

$$\boldsymbol{Term}(\Sigma) \ = < \boldsymbol{Term}_s(\Sigma) |\ s \in \boldsymbol{Sort}(\Sigma) >$$

be the set of Σ-terms. We write $t^s,\ t_1^s, \ldots$ for Σ-terms of sort s and $t,\ t',\ t_1, \ldots$ for Σ-terms.

Note that $\mathsf{if} \cdots \mathsf{then} \cdots \mathsf{else} \cdots \mathsf{fi}$ is not a Σ-function but a term generation rule.

2.2.2 Semantics of terms

Definition 2.10.(States) For each Σ-algebra A, a state on A is a family of functions

$< \sigma_s |\ s \in \boldsymbol{Sort}(\Sigma) >$, where

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

Let ***State****(A)* be the set of states σ on A. Note that ***State****(A)* is the product of $\boldsymbol{State_s}(A)$ for all $s \in \boldsymbol{Sort}(\Sigma)$, where each $\boldsymbol{State_s}(A)$ is the set of all states over A_s.

For notational simplicity, we write $\sigma(\mathtt{x})$ for $\sigma_s(\mathtt{x})$ where $\mathtt{x} \in \boldsymbol{Var}$.

For $t \in \boldsymbol{Term_s}(\Sigma)$, we define the function

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

$[\![t]\!]^A\sigma$ is the value of t in A at state σ and the definition is given by structural induction on t:

(i) $t \equiv \mathtt{x}$

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

(ii) $t \equiv \mathsf{c}$

$$[\![\mathsf{c}]\!]^A\sigma = c$$

(iii) $t \equiv \mathsf{F}(t_1, \ldots, t_m)$

$$[\![\mathsf{F}(t_1,\ldots,t_m)]\!]^A\sigma \simeq \begin{cases} \mathsf{F}^A([\![t_1]\!]^A\sigma, \ldots, [\![t_m]\!]^A\sigma) & \text{if } [\![t_i]\!]^A\sigma \downarrow a_i \ (m > 0, 1 \leq i \leq m) \text{ and} \\ & \qquad \mathsf{F}^A(a_1, \ldots, a_m) \downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

(iv) $t \equiv \mathsf{if}\ \ t_1\ \ \mathsf{then}\ \ t_2\ \ \mathsf{else}\ \ t_3\ \ \mathsf{fi}$

$$[\![\mathsf{if}\ \ t_1\ \ \mathsf{then}\ \ t_2\ \ \mathsf{else}\ \ t_3\ \ \mathsf{fi}]\!]^A\sigma \simeq \begin{cases} [\![t_2]\!]^A\sigma & \text{if} \quad [\![t_1]\!]^A\sigma \downarrow \mathsf{tt} \\ [\![t_2]\!]^A\sigma & \text{if} \quad [\![t_1]\!]^A\sigma \downarrow \mathsf{ff} \\ \uparrow & \text{if} \quad [\![t_1]\!]^A\sigma \uparrow \end{cases}$$

Remarks 2.11.

1. We introduce a new equality " $\simeq$ " for partial algebra; it means both sides of the equation converge and are equal, or both sides diverge.

2. In clause (iii), terms have a *strict* valuation rule, i.e., for every term t in this clause, if the value of any of its subterms diverges, then the value of t diverges. However, in clause (iv), terms have a *non-strict* valuation rule, for example, when $[\![t_1]\!]^A\sigma \downarrow$, $[\![t_2]\!]^A\sigma \downarrow$, even though $[\![t_3]\!]^A\sigma \uparrow$, $[\![\mathsf{if}\ \ t_1\ \ \mathsf{then}\ \ t_2\ \ \mathsf{else}\ \ t_3\ \ \mathsf{fi}]\!]^A\sigma \downarrow [\![t_2]\!]^A\sigma$.

2.2.3 Default term, default value

Definition 2.12 (Closed terms over Σ). We define the set $\boldsymbol{CT}(\Sigma)_s$ of closed terms of sort s by inductive definition:

- if $\mathsf{c} : s$, then $\mathsf{c} \in \boldsymbol{CT}(\Sigma)_s$
- if $\mathsf{F} \in \boldsymbol{Func}(\Sigma)_{u \to s}$, $u = s_1 \times \ldots, s_m$ $(m > 0)$ and $t_i \in \boldsymbol{CT}(\Sigma)_{s_i}$ for $i = 1, ..., m$, then $\mathsf{F}(t_1, \ldots, t_m) \in \boldsymbol{CT}(\Sigma)_s$

We also define the set

$$\boldsymbol{CT}(\Sigma) = < \boldsymbol{CT}(\Sigma)_s \mid s \in \boldsymbol{Sort}(\Sigma) >$$

of closed terms over Σ.

In our research, we are only concerned with the situation where $\boldsymbol{CT}(\Sigma)_s$ is non-empty for each $s \in Sort(\Sigma)$. So, we make the following assumption throughout this thesis:

Assumption 2.13 (Instantiation). For each $s \in Sort(\Sigma)$, $\boldsymbol{CT}(\Sigma)_s$ is non-empty.

We need to introduce *default term* and *default values* for the construction of arrays in (§2.6).

Definition 2.14 (Default terms, default values).

- For each sort s, we pick a closed term (there is at least one by the instantiation assumption) as the *default term* of sort s, written δ^s. Further, for each product type $u = s_1 \times \ldots \times s_m$ of Σ, the *default term tuple* of type u, written δ^u, is the tuple of default terms$(\delta^{s_1}, \ldots, \delta^{s_m})$.
- Given a Σ-algebra A, for any sort s, the *default value* of sort s in A is the valuation $\delta^s_A \in A_s$ of the default term δ^s; and for any product type $u = s_1 \times \ldots \times s_m$, the *default value tuple* of type u in A is the tuple of default values $\delta^u_A = (\delta^{s_1}_A, \ldots, \delta^{s_m}_A) \in A_u$.

2.3 Homomorphisms and isomorphisms

Definition 2.15 (Homomorphism). A Σ-homomorphism from A to B is a total function $h : A \to B$ such that for all $f \in Func(\Sigma)$, $a_1, \ldots, a_n \in A$:

$$f^A(a_1, \ldots, a_n) \downarrow \Leftrightarrow f^B(h(a_1), \ldots, h(a_n)) \downarrow$$

and when $f^A(a_1, \ldots, a_n) \downarrow$,

$$h(f^A(a_1, \ldots, a_n)) = f^B(h(a_1), \ldots, h(a_n))$$

Notes: For $n = 0$ (i.e., Σ-constants c), $h(c^A) = c^B$

Definition 2.16 (Isomorphism). A homomorphism h is a (Σ)-isomorphism from A to B if and only if h has an inverse homomorphism $h^{-1} : B \to A$, with $h^{-1} \circ h = I_A$ and $h \circ h^{-1} = I_B$, written $A \cong B$.

2.4 Adding Booleans: Standard signatures and algebras

The signature of booleans plays an essential role in our work:

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

with algebra:

$$\begin{array}{|ll|}\hline \mathsf{algebra} & \mathcal{B} \\ \mathsf{carriers} & \mathbb{B} \\ \mathsf{functions} & \mathsf{tt}, \mathsf{ff} : \to \mathbb{B} \\ & \mathsf{and}^{\mathcal{B}}, \mathsf{or}^{\mathcal{B}} : \mathbb{B}^2 \to \mathbb{B} \\ & \mathsf{not}^{\mathcal{B}} : \mathbb{B} \to \mathbb{B} \\ \mathsf{end} & \\ \hline \end{array}$$

The algebra $\mathcal{B}$ has the carrier $\mathbb{B} = \{\mathsf{tt}, \mathsf{ff}\}$ of sort bool and the standard interpretations of the function and constant symbols of $\Sigma(\mathcal{B})$. For example, $\mathsf{true}^{\mathcal{B}} = \mathsf{tt}$ and $\mathsf{false}^{\mathcal{B}} = \mathsf{ff}$.

We are particularly interested in those signatures and algebras which contain $\Sigma(\mathcal{B})$ and $\mathcal{B}$, called *standard* signatures and algebras.

Definition 2.17 (Standard signatures and algebras).

- A signature Σ is *standard* if:

  1. $\Sigma(\mathcal{B}) \subseteq \Sigma$, and
  2. it has function symbols of the *equality operator*

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

  for certain sorts s of Σ, called *equality sorts.*

- Given a standard signature Σ, a Σ-algebra A is *standard* if:

  1. it is an expansion of $\mathcal{B}$
  2. the equality operator eq_s is interpreted on each Σ-equality sort s in one of the following three ways:

(a) total equality in A_s.

(b) semi-equality in A_s, i.e.

$$\mathsf{eq}_s^A(x,y) \simeq \begin{cases} \mathsf{tt} & \text{if } x = y \\ \uparrow & \text{if } x \neq y \end{cases}$$

(c) co-semi-equality, i.e.

$$\mathsf{eq}_s^A(x,y) \simeq \begin{cases} \uparrow & \text{if } x = y \\ \mathsf{ff} & \text{if } x \neq y \end{cases}$$

Remark 2.18. *Semi-equality* arises typically in term models, when a semi-decidable test for equality of closed terms is given by reducing them to normal form. Semi-equality is also important for our later work (cf. Remark 7.17 and Conjecture 7.20). *Co-semi-equality* arises e.g. in algebras of reals (see Example 2.20(2) below) and infinite streams.

Remark 2.19. Any many-sorted signature Σ can be *standardized* to a signature $\Sigma^{\mathcal{B}}$ by adjoining the sort bool together with the standard boolean operations; and, correspondingly, any algebra A can be *standardized* to an algebra $A^{\mathcal{B}}$ by adjoining the algebra $\mathcal{B}$ as well as equality operators.

Examples 2.20.

1. The simplest standard algebra is the algebra $\mathcal{B}$ of the Booleans.

2. A standard algebra $\mathcal{R}^{\mathcal{B}}$ is formed by standardizing $\mathcal{R}_{\mathsf{inv}}$ of example 2.7(2) with partial equality operation on $\mathbb{R}$:

$\mathsf{algebra}$	$\mathcal{R}^{\mathcal{B}}$
import	$\mathcal{R}_{\mathsf{inv}}, \mathcal{B}$
$\mathsf{functions}$	$\mathsf{eq}_{\mathsf{real}} : \mathbb{R}^2 \to \mathbb{B}$
end	

where

$$\mathsf{eq}_{\mathsf{real}}(x, y) = \begin{cases} \uparrow & \text{if } \ x = y \\ \mathsf{ff} & \text{if } \ x \neq y. \end{cases}$$

Here, $\mathsf{eq}_{\mathsf{real}}$ is a partial function, because intuitively given two reals x, y represented with infinite decimal extensions, if $x \neq y$, we will discover this in finitely many steps; but if they are equal, we may not be able to, so $\mathsf{eq}_{\mathsf{real}}$ diverges. In terms of computability, this equality operator is *co-semicomputable*. It also has connections with continuity: If we accept the principle

$$Computable \ \Rightarrow \ Continuous$$

the total equality operation on $\mathcal{R}$ is *not continuous*; and therefore not computable. But the partial equality operation defined above *is* continuous and co-semicomputable.

(See [TZ03, §2] for a thorough discussion of these issues).

2.5 Adding counters: N-standard signatures and algebras

The algebra of naturals $\mathcal{N}_0 \ = \ (\mathbb{N}; 0, \mathsf{S})$ is also important for our work:

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

Then, we can standardize it to $\mathcal{N}$:

algebra	$\mathcal{N}$
import	$\mathcal{N}_0, \mathcal{B}$
functions	$\mathsf{eq}_{\mathsf{nat}}, \mathsf{less}_{\mathsf{nat}} : \mathbb{N}^2 \to \mathbb{B}$
end	

Definition 2.21.

- A standard signature Σ is called *N-standard* if it includes the numerical sort nat, as well as function symbols 0, S, $\mathsf{eq}_{\mathsf{nat}}$, $\mathsf{less}_{\mathsf{nat}}$.
- The corresponding Σ-algebra A is *N-standard* if the carrier A_{nat} is the set of natural numbers $\mathbb{N} = 0, 1, 2, \ldots$, and the standard operations have their *standard interpretations* on $\mathbb{N}$.

Remark 2.22. Any standard signature Σ can be N-standardized to a signature Σ^N by adjoining the sort nat and the operations 0,S,$\mathsf{eq}_{\mathsf{nat}}$, $\mathsf{less}_{\mathsf{nat}}$. Accordingly, any standard Σ-algebras A can be N-standardized to an algebra A^N by adjoining the carrier $\mathbb{N}$ together with corresponding standard operations on $\mathbb{N}$.

Examples 2.23.

1. The simplest *N-standard* algebra is the algebra $\mathcal{N}$.

2. We can N-standardize the standard real field $\mathcal{R}^{\mathcal{B}}$ to form the algebra $\mathcal{R}^{N}$

Assumption 2.24 (N-standardness). From now on, we will assume that the signatures and algebras both are N-standard throughout this thesis.

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

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

1. N-standardize these to form Σ^N and A^N, as in section 2.5.

2. Define a "starred sort" s^* for each sort $s \in Sort(\Sigma)$ and let carrier A_s^* be the set of finite sequences or arrays a^* over A_s

The resulting algebras A^* have signature Σ^*, which expands Σ^N by including the new starred sort s^* for each sort s of Σ as well as the following new function symbols:

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

(ii) The application operator $\mathsf{Ap}_s : \ s^* \times \mathsf{nat} \to s$, where

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

where δ^s is the default value at sort s (Instantiation Assumption 2.13);

(iii) the null array $\mathsf{Null}_s : \ s^*$ of zero length;

(iv) the operator $\mathsf{Update}_s :\ s^* \times \mathsf{nat} \times s \to s^*$, where $\mathsf{Update}_s^A(a^*, n, x)$ is the array $b^* \in A_s^*$ of length $\mathsf{Lgth}(b^*) = \mathsf{Lgth}(a^*)$, such that for all $k < \mathsf{Lgth}_s^A(a^*)$

$$b^*[k] = \begin{cases} a^*[k] & \text{if } k \neq n \\ x & \text{if } k = n \end{cases}$$

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

$$b^*[k] = \begin{cases} a^*[k] & \text{if } k < \mathsf{Lgth}_s^A(a^*) \\ \delta^s & \text{if } \mathsf{Lgth}_s^A(a^*) \leq k < m \end{cases}$$

(vi) the equality operator $\mathsf{eq}_s^* : s^* \times s^* \to \mathsf{bool}$ for each equality sort s, where

$$\mathsf{eq}_s^{A^*}(a_1^*, a_2^*) \simeq \begin{cases} \mathtt{tt} & \text{if } \ \mathsf{Lgth}_s^A(a_1^*) = \mathsf{Lgth}_s^A(a_2^*) \text{ and } \forall i < \mathsf{Lgth}_s^A(a_1^*)\ (\mathsf{eq}_s^A(a_1^*[i], a_2^*[i]) = \mathtt{tt}) \\ \mathtt{ff} & \text{if } \ (\,\mathsf{Lgth}_s^A(a_1^*) \neq \mathsf{Lgth}_s^A(a_2^*)\,) \\ & \quad \text{or } (\,\exists i < \mathsf{Lgth}_s^A(a_1^*)\ (\mathsf{eq}_s^A(a_1^*[i], a_2^*[i]) = \mathtt{ff}) \\ & \qquad \textsf{and } \forall j < i\ (\mathsf{eq}_s^A(a_1^*[j], a_2^*[j]) = \mathtt{tt})\,) \\ \uparrow & \text{otherwise} \end{cases}$$

Remarks 2.25.

1. The introduction of starred sorts provides an effective coding of finite sequences within abstract algebra.

2. A^* is an N-standard Σ^*-expansion of A.

Remark 2.26 (Equality of arrays). By clause (vi), if a sort s is an equality sort, then so is the sort s^*, since testing equality on s^* amounts to testing equality of finitely many pairs of objects of sort s. Note that all the array operators are total except possibly for the array equality operator eq_s^*, since equality on sort s may be partial. In clause (vi), $\mathsf{eq}_s^{A^*}(a_1^*, a_2^*)$ is defined by testing the equality of pairs of elements of these two arrays *from left to right*, i.e, from $a^*[0]$ to $a^*[\mathsf{Lgth}_s^A(a^*) - 1]$. When $\mathsf{Lgth}_s^A(a_1^*) = \mathsf{Lgth}_s^A(a_2^*) = l$, for the minimal i $(0 \leq i \leq l)$ such that $\mathsf{eq}_s^A(a_1^*[i], a_2^*[i]) \neq \mathsf{tt}$, if $\mathsf{eq}_s^A(a_1^*[i], a_2^*[i]) = \mathsf{ff}$, then the test for the equality of the whole array returns value ff; if $\mathsf{eq}_s^A(a_1^*[i], a_2^*[i]) \uparrow$, then the test for the equality of the whole array diverges, no matter what is $\mathsf{eq}_s^A(a_1^*[j], a_2^*[j])$ for all $i < j < l$. If we test the equality of every pair of elements of these two arrays *simultaneously*, then we can get a different definition:

$$\mathsf{eq}_s^{A^*}(a_1^*, a_2^*) \simeq \begin{cases} \mathsf{tt} & \text{if } \mathsf{Lgth}_s^A(a_1^*) = \mathsf{Lgth}_s^A(a_2^*) \text{ and } \forall i < \mathsf{Lgth}_s^A(a_1^*)\ (\mathsf{eq}_s^A(a_1^*[i], a_2^*[i]) = \mathsf{tt}) \\ \mathsf{ff} & \text{if } \quad (\mathsf{Lgth}_s^A(a_1^*) \neq \mathsf{Lgth}_s^A(a_2^*)) \\ & \quad \text{or } (\exists i < \mathsf{Lgth}_s^A(a_1^*)\ (\mathsf{eq}_s^A(a_1^*[i], a_2^*[i]) = \mathsf{ff}) \\ \uparrow & \text{otherwise} \end{cases}$$

This definition means that when $\mathsf{Lgth}_s^A(a_1^*) = \mathsf{Lgth}_s^A(a_2^*) = l$, if only the test for the equality of any pair of elements returns value ff, then the test for the equality of the whole array returns value ff. The reason for our choice of definition of $\mathsf{eq}_s^{A^*}$ is the simplicity of the equational specification for arrays. (see §5.1.3 and Remark 5.7)

Chapter 3

Specifiability of functions by theories

3.1 Theories for Σ-algebras

For specification and reasoning about algebras, we use a first order language with equality based on Σ as a specification language. The equality predicate in formulae is different from the equality operator eq_s (§2.4). The former is, in general, not computable or testable and will be used at all sorts; while the latter is used for tests in computation and only applied to the *equality sorts* s. Note that the equality predicate in the specification language does not form part of the signature. Intuitively, think of the equality operation as a *computable* boolean test, but the equality predicate as a *provable* assertion of equality between two terms.

Section 3.1 is essentially taken from [TZ01, §2] which dealt with total algebras since partial and total algebras have much in common in connection with specification theories.

Let $\boldsymbol{Form}(\Sigma)$ be the set of first order formulae over the signature Σ, with the equality predicate at all sorts. It is built up by the following inductive definition:

Base

(i) $t_1^s \ = t_2^s$ is in $\boldsymbol{Form}(\Sigma)$ where $t_i \in \boldsymbol{Term_s}(\Sigma),\ s \in \boldsymbol{Sort}(\Sigma)$.

Inductive clauses

(ii) If P is in $\boldsymbol{Form}(\Sigma)$, then so is $\neg P$.

(iii) – (v) If P, Q are in $\boldsymbol{Form}(\Sigma)$, then so are $P \wedge Q$, $P \vee Q$, $P \supset Q$.

(vi), (vii) If P is in $\boldsymbol{Form}(\Sigma)$ and $\mathtt{x} \in \boldsymbol{Var}_s(\Sigma)$, then $\forall \mathrm{x} P$ and $\exists \mathrm{x} P$ are in $\boldsymbol{Form}(\Sigma)$

Then, $\boldsymbol{Form}(\Sigma)$ constitutes an specification language. A Σ-theory is a set $\boldsymbol{T} \ \subseteq \boldsymbol{Form}(\Sigma)$. In our "algebraic approach", we are only interested in three kinds of formulae: *equations*, *conditional equations* and *conditional BU equations* .

3.1.1 Equational theories over Σ

An equation is a formula of the form:

$$t_1^s \ = t_2^s$$

where $t_i \in \boldsymbol{Term_s}(\Sigma),\ s \in \mathbf{Sort}(\Sigma)$. A equational theory is a set of such formulae.

In the next chapter, we will discuss two kinds of 2-valued logic for specification theory: one based on Kleene equality "$\simeq$", called *Kleene equational logic*, and the other based on strict equality "=", called *strict equational logic.* The former was used by Kleene in [Kle52]; The latter has been investigated independently by a number of researchers including Farmer, Parnas and Feferman [Far90, Par93, Fef95].

The semantics of Kleene equality is given by:

$$[\![t_1 \simeq t_2]\!]^A\sigma = \begin{cases} \mathsf{tt} & \text{if } [\![t_1]\!]^A\sigma\downarrow \text{ and } [\![t_2]\!]^A\sigma\downarrow \text{ and } [\![t_1]\!]^A\sigma = [\![t_2]\!]^A\sigma \\ & \quad\text{or } [\![t_1]\!]^A\sigma\uparrow \text{ and } [\![t_2]\!]^A\sigma\uparrow \\ \mathsf{ff} & \text{otherwise} \end{cases}$$

For strict equality:

$$[\![t_1 = t_2]\!]^A\sigma = \begin{cases} \mathsf{tt} & \text{if } [\![t_1]\!]^A\sigma\downarrow \text{ and } [\![t_2]\!]^A\sigma\downarrow \text{ and } [\![t_1]\!]^A\sigma = [\![t_2]\!]^A\sigma \\ \mathsf{ff} & \text{otherwise.} \end{cases}$$

These two different semantics for equality give rise to two kinds of specification theories, as we will see in §5.

3.1.2 Conditional equational theories over Σ

A *conditional equation* is a formula of the form:

$$P_1 \wedge \ldots \wedge P_n \supset P \tag{3.1}$$

where $n \geq 0$ and P_i and P are equations. A *conditional equational theory* is a set of such formulae. An *equational sequent* is a sequent of the form:

$$P_1, \ldots, P_n \to P \tag{3.2}$$

where $n \geq 0$ and P_i and P are equations. This sequent corresponds to the conditional equation (3.1).

3.1.3 Conditional BU equational theories over Σ

A BU (bounded universal) quantifier is a quantifier of the form $\forall \mathsf{z} < t$, where z:nat and t:nat. A Σ-BU *equation* is formed by prefixing an equation by a string of 0 or more bounded universal quantifiers. *A conditional* BU *equation* is a formula of the form of (3.1) where P_i and P are BU equations. *A conditional* BU *equational theory* is a set of such formulae (or their universal closures). A BU *equational sequent* is a sequent of the form of (3.2) where P_i and P are BU equations. This sequent corresponds to the conditional BU equation of the form of (3.1).

3.2 Specification over algebras

Assume that Σ, Σ' and Σ'' are N-standard partial signatures with $\Sigma \subset \Sigma' \subset \Sigma''$. Let A be an N-standard Σ-algebra, A' an N-standard Σ'-algebra and A'' an N-standard Σ''-algebra. Also, Let T be a Σ-theory, T' a Σ'-theory and T'' a Σ''-theory. We use 'f' as symbol for the function f. The following definitions will be used in Chapter 5 for specification.

Definition 3.1(Relative isomorphism). Let A_1' and A_2' be two Σ'-algebras with $A_1'|_\Sigma = A_2'|_\Sigma$. Then A_1' and A_2' are Σ'/Σ *isomorphic*, written $A_1' \cong_{\Sigma'/\Sigma} A_2'$, if there is a Σ'-isomorphism from A_1' to A_2' whose restriction to Σ is the identity on $A_1'|_\Sigma$.

Definition 3.2 (Subfunction).

- Given two functions $f, g : A^u \to A_s$, we write $f \subseteq g$ (f is a *subfunction* of g) to mean for all $\vec{x} \in A^u$,

$$f(\vec{x})\downarrow \;\Rightarrow\; (\; g(\vec{x})\downarrow \text{ and } f(\vec{x}) = g(\vec{x}) \;)$$

- Given two function tuples: $\vec{f} \equiv f_1, \ldots, f_m$ and $\vec{g} \equiv g_1, \ldots, g_m$ of matching types, we write $\vec{f} \subseteq \vec{g}$ to mean

$$f_i \subseteq g_i \text{ for } i = 1, \ldots, m$$

Remarks 3.3

1. The completely undefined function $\lambda x. \uparrow$ is a subfunction of every function of the same type.
2. $f = g \quad \Leftrightarrow \quad (\ f \subseteq g$ and $g \subseteq f\ )$

Definition 3.4. Suppose A' is a Σ'-expansion of A. We say that (Σ', T') *specifies* A' *over* A if and only if A' is the unique (up to Σ'/Σ isomorphism) Σ'-expansion of A satisfying T', in other words:

(i) $A' \models T'$ and

(ii) for any Σ' expansion B' of A, if $B' \models T'$, then $B' \cong_{\Sigma'/\Sigma} A'$

An important special case of Definition 3.4 is as the following.

Definition 3.5. Suppose $\Sigma' = \Sigma \cup \{\mathsf{f}\}$. We say that (Σ', T') *specifies* f *over* A if and only if it uniquely defines f over A, i.e., f is the unique function on A (of the type of f) such that $(A, f) \models T'$, i.e.

(i) $(A, f) \models T'$ and

(ii) for any function f', if $(A, f') \models T'$, then $f = f'$.

There is a *minimal definability* version for Theorem 3.5:

Definition 3.6. Suppose $\Sigma' = \Sigma \cup \{\mathsf{f}\}$. We say that (Σ', T') *minimally defines* f *over* A if and only if f is the minimal function on A (of the type of f) such that $(A, f) \models T'$, i.e.

(i) $(A, f) \models T'$ and

(ii) for any function f', if $(A, f') \models T'$, then $f \subseteq f'$.

Definition 3.7. Suppose A' is a Σ'-expansion of A. We say that (Σ'', T'') *specifies A' over A with hidden sorts and/or functions* if and only if A' is the unique (up to Σ'/Σ isomorphism) Σ'-expansion of A such that some Σ''-expansion of A' satisfies T''; in other words:

(i) A' is a Σ'-reduct of a Σ''-model of T'', and

(ii) for all Σ'-expansions B' of A, if B' is a Σ'-reduct of a standard Σ''-model of T'', then $B' \cong_{\Sigma'/\Sigma} A'$.

Again, as a special case and its "minimal" version, we have:

Definition 3.8. Suppose $\Sigma' = \Sigma \cup \{\mathsf{f}\}$. We say that (Σ'', T'') *specifies f over A with hidden sorts and/or functions* if and only if f is the unique function on A (of the type of f) such that some Σ''-expansion of (A, f) satisfies T''.

Definition 3.9. Suppose $\Sigma' = \Sigma \cup \{\mathsf{f}\}$. We say that (Σ'', T'') *minimally defines f over* A with hidden sorts and/or functions if and only if f is the minimal function on A (of the type of f) such that some Σ''-expansion of (A, f) satisfies T''.

Chapter 4

Computable functions

We will consider four notions of computability on N-standard algebras, formalized by *schemes*. Two computability classes, $\boldsymbol{PR}(\Sigma)$ and $\boldsymbol{PR}^*(\Sigma)$ are introduced, then two more classes are formed by adjoining the $\boldsymbol{\mu}$ operator to these. These models of computation were developed in [TZ88] by Tucker and Zucker as a generalization of Kleene's $\boldsymbol{PR}$ schemes over $\mathbb{N}$ [Kle52] to total many-sorted abstract algebras.

4.1 $\boldsymbol{PR}(\Sigma)$ and $\boldsymbol{PR}^*(\Sigma)$ computable functions

Given an N-standard signature Σ and Σ-algebra A, we define $\boldsymbol{PR}$ computable functions over A by starting with some initial functions (as in the base cases(i)-(ii) below) and applying *composition*, *definition by cases* and *simultaneous primitive recursion* to these functions (as in the inductive cases (iii)-(v)). Here, for the partial functions, we introduce another kind of equality symbol '$\simeq$', which means both sides of the equality converge with equal values, or both sides diverge. Let $\vec{x} \equiv x_1, \ldots, x_m$.

Base:

(i) *Primitive* Σ*-functions*:

$$f(\vec{x}) \simeq \mathsf{F}^A(\vec{x})$$

$$f(\vec{x}) = \mathsf{c}^A$$

of type $u \to s$, for all the primitive function symbols $\mathsf{F} : u \to s$ and constant symbols c of Σ, where $\vec{x} : u,\ u = s_1 \times \ldots \times s_m$.

(ii) *Projection*:

$$f(\vec{x}) = x_i$$

of type $u \to s_i$, where $\vec{x} \in A^u$ and is of type $u = s_1 \times \cdots \times s_m$.

Inductive clauses:

(iii) *Composition*:

$$f(\vec{x}) \simeq h(g_1(\vec{x}) \ldots g_m(\vec{x}))$$

of type $u \to s$, where $g_i : u \to s_i$ $(i = 1, \ldots, m)$ and $h : s_1 \times \cdots \times s_m \to s$. If $g_i(\vec{x}) \downarrow a_i \in A^u$ $(1 \leq i \leq m)$ and $h(a_1, \ldots, a_m) \downarrow a \in A_s$, then $f(\vec{x}) \downarrow a$; otherwise, $f(\vec{x}) \uparrow$. This is a *strict composition* rule, i.e, if any value occurring in this composition is undefined, then the final result is undefined.

(iv) *Definition by cases*:

$$f(\vec{x}) \quad \simeq \begin{cases} g_1(\vec{x}) & \text{if} \quad h(\vec{x}) \downarrow \mathsf{tt} \\ g_2(\vec{x}) & \text{if} \quad h(\vec{x}) \downarrow \mathsf{ff} \\ \uparrow & \text{if} \quad h(\vec{x}) \uparrow \end{cases}$$

of type $u \to s$. Note that when $h(\vec{x}) \downarrow \mathsf{tt}$, the value of $f(\vec{x})$ is determined only by $g_1(\vec{x})$, no matter whether $g_2(\vec{x})$ converges or not, i.e. if $g_1(\vec{x}) \downarrow$, then $f(\vec{x}) \downarrow g_1(\vec{x})$; if $g_1(\vec{x}) \uparrow$, then $f(\vec{x}) \uparrow$. Similarly, for the case of $h(\vec{x}) \downarrow \mathsf{ff}$, the value of $f(\vec{x})$ is independent of the convergence of $g_1(\vec{x})$. So, this is *not* a *strict computation* rule, unlike clauses (iii) and (v).

(v) *Simultaneous primitive recursion on* $\mathbb{N}$:

$$f_i(0, \vec{x}) \simeq g_i(\vec{x})$$
$$f_i(z+1, \vec{x}) \simeq h_i(z, \vec{x}, f_1(z, \vec{x}) \ldots f_m(z, \vec{x}))$$

where $g_i : u \to s_i$ and $h_i : \mathsf{nat} \times u \times v \to s_i$ $(i = 1, \ldots, m)$. This defines an m-tuple of functions $\vec{f} = (f_1, \ldots, f_m)$ with $f_i : \mathsf{nat} \times u \to s_i$, for fixed degree of simultaneity $m > 0$ and product types u and $v = s_1 \times \cdots \times s_m$. The strict composition rule still applies on the second scheme, that is to say, $f_i(z+1, \vec{x}) \downarrow$ if and only if $f_i(z, \vec{x})) \downarrow a_i \in A_{s_i}(1 \leq i \leq m)$ and $h_i(z, \vec{x}, a_1, \ldots, a_m) \downarrow$.

$\boldsymbol{PR}(\Sigma)$ *schemes* are notation symbols for $\boldsymbol{PR}(\Sigma)$ functions. We write σ, $\tau \ldots$ for these. Corresponding to every $\boldsymbol{PR}$ computable functions defined as above, there is a $\boldsymbol{PR}$ *scheme*. For example, for the case (i) *primitive functions*, the scheme is:

$$< \mathsf{P}, \mathsf{F}, n, u, s >$$

where 'P' means that it is a scheme for primitive function, 'F' is a Σ-function symbol, n is the arity of F and u, s mean that the type of F is $u \to s$. For case (iii) *composition*, the scheme is:

$$< \mathsf{C},\, n,\, m,\, \sigma_1, \ldots \sigma_m,\, \tau >$$

where 'C' means it is a scheme for composition, σ_i is the scheme for g_i $(1 \le i \le m)$, τ is the scheme for h, n is the arity of g_i and m is the arity of h. Note that a scheme for a $\boldsymbol{PR}$ function contains the schemes for all the auxiliary functions used in its definition. The actual details of the syntax of schemes is not important for our purpose; see [TZ88, §4.1.5] for details of a possible syntax for schemes, like the one above.

In the context of algebraic specification theory, it is more convenient to work with $\boldsymbol{PR}$ *derivations* [TZ01, §4.1] than with $\boldsymbol{PR}$ schemes. A $\boldsymbol{PR}$ derivation is a "linear version" of a $\boldsymbol{PR}$ scheme, in which all the auxiliary functions are displayed in a list. More precisely:

Definition 4.1 ($\boldsymbol{PR}$ derivation). A $\boldsymbol{PR}(\Sigma)$ *derivation*

$$\alpha = ((\mathsf{f}_0, \sigma_0),\ (\mathsf{f}_1, \sigma_1), \ldots, (\mathsf{f}_n, \sigma_n))$$

is a list of pairs of function symbols f_i and $\boldsymbol{PR}$ schemes $\sigma_i (i = 1, \ldots, n)$ where for each i, either f_i is an initial function, or f_i is defined by σ_i from functions f_j, for certain $j < i$. The derivation α is called a $\boldsymbol{PR}$ *derivation* of f_n, with auxiliary functions $\mathsf{f}_0, \ldots, \mathsf{f}_{n-1}$. The type of α is the type of f_n. We use $\alpha,\ \beta,\ \gamma, \ldots$ for derivation.

Remark 4.2. The formalism of $\boldsymbol{PR}(\Sigma)$ derivations is equivalent to that of $\boldsymbol{PR}(\Sigma)$ schemes: from a $\boldsymbol{PR}$ scheme we derive an equivalent $\boldsymbol{PR}$ derivation by 'linearizing' the subschemes, and conversely, given an derivation, the scheme σ_n is equivalent to it.

Notation 4.3. A $\boldsymbol{PR}\,(\Sigma)_{u\to s}$ scheme (or derivation) is a $\boldsymbol{PR}(\Sigma)$ scheme (or derivation) of type $u \to s$. It defines, or computes, in each N-standard algebra A, a function $\mathsf{f}_\alpha^A : A^u \to A_s$.

Now we introduce another, broader class of functions providing a better generalization of the notion of primitive recursiveness, namely $\boldsymbol{PR}^*$ computability. A function on A is $\boldsymbol{PR}^*(\Sigma)$ *computable* if it is defined by a $\boldsymbol{PR}$ derivation over Σ^*, i.e, this function have sorts in Σ while the auxiliary functions used in its definition may be of the starred sorts.

4.2 $\boldsymbol{\mu PR}(\Sigma)$ and $\boldsymbol{\mu PR}^*\,(\Sigma)$ computable functions

The $\boldsymbol{\mu PR}$ schemes over Σ are formed by adding to the $\boldsymbol{PR}$ schemes in §4.1 a new scheme for the following least number functions:

(vi) *Least number* or μ *operator*:

$$
\begin{aligned}
f(\vec{x}) &\simeq \mu z[g(\vec{x},z) = \mathtt{tt}] \\
&\simeq \begin{cases} z & \text{if } \forall y < z(g(\vec{x},y) \downarrow \mathtt{ff}) \text{ and } g(\vec{x},z) \downarrow \mathtt{tt} \\ \uparrow & \text{otherwise} \end{cases}
\end{aligned}
$$

of type $u \to \mathsf{nat}$, where $g :\ u \times \mathsf{nat} \to \mathsf{bool}$.

Remark 4.4. This is a "constructive" version of the least number operator. For example, if $g(\vec{x},0) \downarrow \mathtt{ff}$, $g(\vec{x},1) \uparrow$ and $g(\vec{x},2) \downarrow \mathtt{tt}$, $f(\vec{x}) \uparrow$ (it does not converge to 2).

A function on A is $\boldsymbol{\mu PR}^*(\Sigma)$ *computable* if it is defined by a $\boldsymbol{\mu PR}$derivation over Σ^*.

Remark 4.5. The notion of $\boldsymbol{\mu PR}^*(\Sigma)$ computation is important in our computability theory, in connection with the *Generalized Church-Turing Thesis*(§1.1.1)

Chapter 5

Algebraic specifications for computable functions

Computable functions can be specified by algebraic formulae which consist of equations, conditional equations and conditional BU equations. The specifications for computable functions in total algebra have been discussed in [TZ02]. Both in theory and in practice, there is also an interest in the specification for computable functions in *partial* algebras. In this chapter, we will consider functions f computable over partial algebras by $\boldsymbol{PR}$, $\boldsymbol{PR}^*$, $\boldsymbol{\mu PR}$, $\boldsymbol{\mu PR}^*$ derivations, and show that they are specifiable by algebraic formulae.

We will define Σ-theories E that specify computable functions in two kinds of 2-valued logic: one using Kleene equality [Kle52], and the other using strict equality [Far90, Par93, Fef95]. These two logics give rise to different interpretations of equations between two terms.

For Kleene equality:

$$[\![t_1 \simeq t_2]\!]^A \sigma = \begin{cases} \mathsf{tt} & \text{if } \ ([\![t_1]\!]^A\sigma \downarrow \ \text{and } [\![t_2]\!]^A\sigma \downarrow \ \text{and } [\![t_1]\!]^A\sigma = [\![t_2]\!]^A\sigma) \\ & \qquad \text{or } ([\![t_1]\!]^A\sigma \uparrow \ \text{and } [\![t_2]\!]^A\sigma \uparrow) \\ \mathsf{ff} & \text{otherwise} \end{cases}$$

For strict equality:

$$[\![t_1 = t_2]\!]^A \sigma = \begin{cases} \mathsf{tt} & \text{if } \ [\![t_1]\!]^A\sigma \downarrow \ \text{and } [\![t_2]\!]^A\sigma \downarrow \ \text{and } [\![t_1]\!]^A\sigma = [\![t_2]\!]^A\sigma \\ \mathsf{ff} & \text{otherwise.} \end{cases}$$

So, it is easy to see that these two kinds of logic will give different specification theories for computable functions.

Remarks 5.1.

1. One may wonder why we need to discuss specifications in two kinds of logic. That is because each of them has its own advantages for our work. As we will see, Kleene equality provides simpler specification theories than strict equality does. However, strict equality (at least for closed terms with effective normalization) is semicomputable, while the Kleene equality is not. (How could it be tested if both sides of an equality diverge?)

2. One may also ask why we don't discuss specification theories in 3-valued logic. The reason is that we get equivalent conditional equational specification theories in strict 2-valued logic and in 3-valued logic. This is discussed further in Remark 5.16 below.

In the specification theories below, f, g, h are function symbols corresponding to functions $f,\ g,\ h$.

5.1 Algebraic specification for computable functions in Kleene equational logic

The semantics for equations, conditional equations and conditional BU equations in Kleene equational logic are:

1. $A \models_\sigma t_1 \simeq t_2$ iff
   $(\llbracket t_1 \rrbracket^A \sigma \downarrow$ and $\llbracket t_2 \rrbracket^A \sigma \downarrow$ and $\llbracket t_1 \rrbracket^A \sigma = \llbracket t_2 \rrbracket^A \sigma)$ or $(\llbracket t_1 \rrbracket^A \sigma \uparrow$ and $\llbracket t_2 \rrbracket^A \sigma \uparrow)$

2. $A \models_\sigma \forall \mathtt{y} < t(t_1(\vec{\mathtt{x}}, \mathtt{y}) \simeq t_2(\vec{\mathtt{x}}, \mathtt{y}))$ iff
   for all $i < k$, $A \models_\sigma t_1(\vec{\mathtt{x}}, \bar{i}) \simeq t_2(\vec{\mathtt{x}}, \bar{i})$
   (where $\llbracket t \rrbracket^A \sigma = k$, $\bar{i}$ is the numeral of i.)

3. $A \models_\sigma P_1, \ldots, P_n \to P$ iff
   $A \models_\sigma P_i$ for $i = 1, \ldots, n \Rightarrow A \models_\sigma P$

4. $A \models E$ iff for all σ, $A \models_\sigma E$

Remark 5.2. Clause 2 is *not* a "constructive interpretation" of bounded qualification. (Cf. Remarks 4.4 and 5.1(1).)

5.1.1 Algebraic specification for *PR* computable functions

For each ***PR*** derivation α and N-standard Σ-algebra A, let f_α^A be the function on A computed by α, and $\vec{\mathsf{g}}_\alpha^A$, $\vec{\mathsf{h}}_\alpha^A$ be the corresponding auxiliary functions on A.

For each ***PR***(Σ) derivation α, there is a finite set E_α of specifying equations for the function f_α^A. The set E_α consist of conditional equations in an expanded signature $\Sigma_\alpha = \Sigma \cup \{\vec{\mathsf{g}}_\alpha, \mathsf{f}_\alpha, \vec{\mathsf{h}}_\alpha\}$ where $\vec{\mathsf{g}}_\alpha \equiv \mathsf{g}_{\alpha_1}, \ldots, \mathsf{g}_{\alpha_m}$, $\vec{\mathsf{h}}_\alpha \equiv \mathsf{h}_{\alpha_1}, \ldots, \mathsf{h}_{\alpha_m}$ are the auxiliary functions in the derivation of f_α. The set E_α is defined by CV (course of values) induction on the length of the derivation α (see §4.1 and Definition 4.1)

(i) *Primitive Σ-functions:* $\alpha \equiv (\mathsf{f}, \sigma)$ where σ is the scheme for a primitive Σ-function F. Then

$$E_\alpha \;=\; \{\mathsf{f}(\vec{\mathsf{x}}) \simeq \mathsf{F}(\vec{\mathsf{x}})\}$$

Constant: $\alpha \equiv (\mathsf{f}, \sigma)$ where σ is the scheme for a Σ-constant c. Then

$$E_\alpha \;=\; \{\mathsf{f}(\vec{\mathsf{x}}) \simeq \mathsf{c}\}$$

(ii) *Projection:* $\alpha \equiv (\mathsf{f}, \sigma)$ where σ is the scheme for projection. Then

$$E_\alpha \;=\; \{\mathsf{f}(\vec{\mathsf{x}}) \simeq \mathsf{x}_i\}$$

(iii) *Composition:* $\alpha \equiv ((\mathsf{g}_1, \sigma_1), \ldots, (\mathsf{g}_m, \sigma_m), (\mathsf{h}, \sigma_{m+1}), (\mathsf{f}, \sigma_{m+2}))$ where σ_{m+2} is the scheme for composition. Suppose the derivation for $\mathsf{g}^A_{\alpha i}$ is β_i $(1 \le i \le m)$, and that for h^A_α is γ. Then

$$E_\alpha \;= E_{\beta_1} \cup \cdots \cup E_{\beta_m} \cup E_\gamma \cup \{\mathsf{f}(\vec{\mathsf{x}}) \simeq \mathsf{h}(\mathsf{g}_1(\vec{\mathsf{x}}), \ldots, \mathsf{g}_m(\vec{\mathsf{x}}))\}$$

(iv) *Definition by cases:* $\alpha = ((\mathsf{h}, \sigma_1)(\mathsf{g}_1, \sigma_2), (\mathsf{g}_2, \sigma_3), (\mathsf{f}, \sigma_4))$ where σ_4 is the scheme for definition by cases. Suppose the derivation for $\mathsf{g}^A_{\alpha 1}$, $\mathsf{g}^A_{\alpha 2}$, h^A_α are β_1, β_2, γ respectively. Then

$$E_\alpha = E_{\beta_1} \cup E_{\beta_2} \cup E_\gamma \cup \;\{\mathsf{f}(\vec{\mathsf{x}}) \simeq \mathsf{if}\;\; \mathsf{h}(\vec{\mathsf{x}})\;\; \mathsf{then}\;\; \mathsf{g}_1(\vec{\mathsf{x}})\;\; \mathsf{else}\;\; \mathsf{g}_2(\vec{\mathsf{x}})\;\; \mathsf{fi}\}$$

(v) *Simultaneous primitive recursion:* $\alpha = ((\mathsf{g}_i, \sigma_{1i}), (\mathsf{h}_i, \sigma_{2i}), (\mathsf{f}_i, \sigma_{3i}))$ where σ_{3i} is the scheme for simultaneous primitive recursion $(i = 1, \ldots, m)$. Suppose the derivation for $\mathsf{g}^A_{\alpha i}$ is β_i, and that for $\mathsf{h}^A_{\alpha i}$ is γ_i. Then

$$
\begin{aligned}
E_\alpha = \quad & E_{\beta_1} \cup \cdots \cup E_{\beta_2} \cup E_{\gamma_1} \cup \cdots \cup E_{\gamma_m} \cup \\
\{ \quad & \mathsf{f}_i(0, \vec{\mathsf{x}}) \simeq \mathsf{g}_i(\vec{\mathsf{x}}), \\
& \mathsf{f}_i(\mathsf{z}+1, \vec{\mathsf{x}}) \simeq \mathsf{h}_i(\mathsf{z}, \vec{\mathsf{x}}, \mathsf{f}_1(\mathsf{z}, \vec{\mathsf{x}}), \ldots, \mathsf{f}_m(\mathsf{z}, \vec{\mathsf{x}})) \\
& (1 \le i \le m) \\
\} \quad &
\end{aligned}
$$

Remark 5.3. The specifications E_α for partial functions are similar to those for total functions [TZ02,§5].

Theorem 5.4 (Kleene equational specification of *PR* functions). For each $\boldsymbol{PR}(\Sigma)$ derivation α and N-standard Σ-algebra A, the Kleene equational specification $(\Sigma_\alpha, E_\alpha)$ specifies the $\boldsymbol{PR}$ computable function f_α^A with hidden functions.

Proof: By CV induction on the length of the derivation α. We do not give a complete proof here since it is simpler than the proof described in detail for the strict conditional equational specifications of $\boldsymbol{PR}$ functions (Theorem 5.12. below). $\square$

Remark 5.5. The specifications E_α specify the auxiliary functions $\vec{\mathsf{g}}_\alpha^A$, $\vec{\mathsf{h}}_\alpha^A$ as well as f_α^A.

5.1.2 Algebraic specification for $\boldsymbol{\mu PR}$ computable functions

Now, we will consider $\boldsymbol{\mu PR}$ derivations. For each such derivation, there is a finite set E'_α of "specifying conditional BU equations" for the function f_α^A. This set is constructed by CV induction on α, like E_α in §5.1.1, except adding some conditional BU equations to E'_α for the new case, i.e, the scheme for the μ-operator.

Suppose the derivation for g_α^A is β, then the E'_α for the function f_α^A with $\boldsymbol{\mu}$-operator is as follows:

$$\begin{aligned} E'_\alpha = E'_\beta \ \cup \{ \quad & \forall \mathsf{y} < \mathsf{z}(\mathsf{g}(\vec{\mathsf{x}}, \mathsf{y}) \simeq \mathsf{false}), \mathsf{g}(\vec{\mathsf{x}}, \mathsf{z}) \simeq \mathsf{true} \rightarrow \mathsf{f}(\vec{\mathsf{x}}) \simeq \mathsf{z}, \\ & \mathsf{f}(\vec{\mathsf{x}}) \simeq \mathsf{z} \rightarrow \mathsf{g}(\vec{\mathsf{x}}, \mathsf{z}) \simeq \mathsf{true}, \\ & \mathsf{f}(\vec{\mathsf{x}}) \simeq \mathsf{z} \rightarrow \forall \mathsf{y} < \mathsf{z}(\mathsf{g}(\vec{\mathsf{x}}, \mathsf{y}) \simeq \mathsf{false}) \ \} \end{aligned}$$

Next, we will reduce conditional BU equations to conditional equations by the method discussed in [TZ02,§3], i.e. eliminate the bounded quantifiers by incorporating into the signature, for each BU quantifier occurring in the theory, a function which computes that quantifier.

In the theory E'_α, there is a conditional BU equation of the form

$$\forall \mathsf{y} < \mathsf{z} \ (\mathsf{g}(\vec{\mathsf{x}}, \mathsf{y}) \simeq \mathsf{false}) \tag{5.1}$$

So, we will first define a boolean valued function symbol:

$$\mathsf{h} : u \times \mathsf{nat} \rightarrow \mathsf{bool}$$

which is interpreted in A as:

$$\mathsf{h}_\alpha^A(\vec{x}, z) = \mathsf{tt} \Leftrightarrow \forall y < z \ (\mathsf{g}_\alpha^A(\vec{x}, y) = \mathsf{ff})$$

and then adjoin to the specifying theory the following axioms giving the inductive definition for h_α^A:

$$\mathsf{h}(\vec{\mathsf{x}}, 0) \simeq \mathsf{true}$$

$$\mathsf{h}(\vec{\mathsf{x}}, \mathsf{z}+1) \simeq \mathsf{h}(\vec{\mathsf{x}}, \mathsf{z}) \ \mathsf{and} \ (\mathsf{not}(\mathsf{g}(\vec{\mathsf{x}}, \mathsf{z})))$$

and replacing (5.1) in the theory F_α by

$$\mathsf{h}(\vec{\mathsf{x}}, \mathsf{z}) \simeq \mathsf{true}$$

By this method, we can eliminate the BU quantifiers in E'_α and replace E'_α by a conditional equational theory E_α as the following:

$$
\begin{aligned}
E_\alpha = E_\beta \cup \ \{ \ & \rightarrow \ \mathsf{h}(\vec{\mathsf{x}}, 0) \simeq \mathsf{true}, \\
& \rightarrow \ \mathsf{h}(\vec{\mathsf{x}}, \mathsf{z}+1) \simeq \mathsf{h}(\vec{\mathsf{x}}, \mathsf{z}) \ \mathsf{and} \ (\mathsf{not}(\mathsf{g}(\vec{\mathsf{x}}, \mathsf{z}))), \\
& \mathsf{h}(\vec{\mathsf{x}}, \mathsf{z}) \simeq \mathsf{true}, \mathsf{g}(\vec{\mathsf{x}}, \mathsf{z}) \simeq \mathsf{true} \ \rightarrow \ \mathsf{f}(\vec{\mathsf{x}}) \simeq \mathsf{z}, \\
& \mathsf{f}(\vec{\mathsf{x}}) \simeq \mathsf{z} \ \rightarrow \ \mathsf{g}(\vec{\mathsf{x}}, \mathsf{z}) \simeq \mathsf{true}, \\
& \mathsf{f}(\vec{\mathsf{x}}) \simeq \mathsf{z} \ \rightarrow \ \mathsf{h}(\vec{\mathsf{x}}, \mathsf{z}) \simeq \mathsf{true} \\
\} &
\end{aligned}
$$

Theorem 5.6 (Kleene conditional equational specification of $\boldsymbol{\mu PR}$ functions). For each $\boldsymbol{\mu PR}(\Sigma)$ derivation α and N-standard Σ-algebra A, the Kleene conditional equational specification $(\Sigma_\alpha,\ E_\alpha)$ specifies the $\boldsymbol{\mu PR}$ computable function f_α^A.

Proof: By CV induction on the length of α. The reader can refer to the (more complicated) proof of Theorem 5.12 for details. □

5.1.3 Algebraic specification for $\boldsymbol{\mu PR^*}$ computable functions

Recall from §4.2 that a $\boldsymbol{\mu PR}^*(\Sigma)$ function is defined by a $\boldsymbol{\mu PR}$ derivation over Σ^*. Let the Σ-array axioms $\boldsymbol{ArrAx}(\Sigma)$ be the following theory (dropping sort subscripts):

$$
\begin{aligned}
\boldsymbol{ArrAx}(\Sigma) = \{\ & \mathsf{Lgth}(\mathsf{Null}) \simeq 0, && (1)\\
& \mathsf{Ap}(\mathsf{a}^*, \mathsf{z}) \simeq \mathsf{if}\ \ \mathsf{less_{nat}}(\mathsf{z}, \mathsf{Lgth}(\mathsf{a}^*))\ \ \mathsf{then}\ \ \mathsf{Ap}(\mathsf{a}^*, \mathsf{z})\ \ \mathsf{else}\ \ \delta\ \ \mathsf{fi}, && (2)\\
& \mathsf{Lgth}(\mathsf{Update}(\mathsf{a}^*, \mathsf{z}, \mathtt{x})) \simeq \mathsf{Lgth}(\mathsf{a}^*), && (3)\\
& \mathsf{Ap}(\mathsf{Update}(\mathsf{a}^*, \mathsf{z}_1, \mathtt{x}), \mathsf{z}) \simeq \mathsf{if}\ \ \mathsf{eq_{nat}}(\mathsf{z}, \mathsf{z}_1)\ \ \mathsf{then}\ \ \mathtt{x}\ \ \mathsf{else}\ \mathsf{Ap}(\mathsf{a}^*, \mathsf{z})\ \ \mathsf{fi}, && (4)\\
& \mathsf{Ap}(\mathsf{Update}(\mathsf{a}^*, \mathsf{z}, \mathtt{x}), \mathsf{z}) \simeq \mathsf{if}\ \ \mathsf{less_{nat}}(\mathsf{z}, \mathsf{Lgth}(\mathsf{a}^*))\ \ \mathsf{then}\ \ \mathtt{x}\ \ \mathsf{else}\ \ \delta\ \ \mathsf{fi}, && (5)\\
& \mathsf{Lgth}(\mathsf{Newlength}(\mathsf{a}^*, \mathsf{z})) \simeq \mathsf{z}, && (6)\\
& \mathsf{Ap}(\mathsf{Newlength}(\mathsf{a}^*, \mathsf{z}_1), \mathsf{z})) \simeq \mathsf{if}\ \ \mathsf{less_{nat}}(\mathsf{z}, \mathsf{z}_1)\ \ \mathsf{then}\ \ \mathsf{Ap}(\mathsf{a}^*, \mathsf{z})\ \ \mathsf{else}\ \ \delta\ \ \mathsf{fi}, && (7)\\
& \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, 0) \simeq \mathsf{ture}, && (8)\\
& \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, \mathsf{z}+1) \simeq \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, \mathsf{z})\ \ \mathsf{cand}\ \ \mathsf{eq}(\mathsf{Ap}(\mathsf{a}_1^*, \mathsf{z}), \mathsf{Ap}(\mathsf{a}_2^*, \mathsf{z})), && (9)\\
& \mathsf{eq}^*(\mathsf{a}_1^*, \mathsf{a}_2^*) \simeq \mathsf{eq_{nat}}(\mathsf{Lgth}(\mathsf{a}_1^*), \mathsf{Lgth}(\mathsf{a}_2^*))\ \ \mathsf{cand}\ \ \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, \mathsf{Lgth}(\mathsf{a}_1^*)) && (10)\\
& \}
\end{aligned}
$$

Where the boolean operator ‘cand’ (“conditional and”) can be defined as an abbreviation:

$$t_1\ \ \mathsf{cand}\ \ t_2\ \ \equiv\ \ \mathsf{if}\ \ t_1\ \ \mathsf{then}\ \ t_2\ \ \mathsf{else}\ \ \mathsf{false}\ \ \mathsf{fi}$$

It has the following truth table:

$t_1 \setminus t_2$	tt	ff	$\uparrow$
tt	tt	ff	$\uparrow$
ff	ff	ff	ff
$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$

Equations (8), (9) specify an auxiliary function equpto, which is interpreted in A as:

$$
\mathsf{equpto}^A(a_1^*, a_2^*, k) \simeq \begin{cases} \mathsf{tt} & \text{if } \forall i < k\ (\mathsf{eq}^A(a_1^*[i]), a_2^*[i]) = \mathsf{tt}) \\ \mathsf{ff} & \text{if } \exists i < k\ (\mathsf{eq}^A(a_1^*[i]), a_2^*[i]) = \mathsf{ff})\ \mathsf{and}\ \forall j < i\ (\mathsf{eq}^A(a_1^*[j], a_2^*[j]) = \mathsf{tt}) \\ \uparrow & \text{otherwise} \end{cases}
$$

We introduced this function to reduce BU equations in array axioms to equations.

Remark 5.7. As shown in Remark 2.26, there is another reasonable interpretation for array equality eq^*. To specify this function, we need a new boolean operator sand ("strict and"), which has the truth table:

$t_1 \setminus t_2$	tt	ff	$\uparrow$
tt	tt	ff	$\uparrow$
ff	ff	ff	ff
$\uparrow$	$\uparrow$	ff	$\uparrow$

Compared to the truth table for 'cand' above, this operator can not be defined by the term construction rules in (§2.2). So, in order to specify eq^*, we would have to redefine the term construction rules to include 'sand'. for the sake of simplicity, we choose the definition in §2.6.

Theorem 5.8. Let T be the Σ-theory, A be an N-standard Σ-algebra and

$$T^* = T \ \cup \ \boldsymbol{ArrAx}(\Sigma)$$

Then the specification$(\Sigma^*,\ T^*)$ specifies A^* over A.

Proof: According to the definition of the array and $A \models T$, it is obvious that $A^* \models T^*$. Now, we need to prove for all Σ^*-expansions A' of A, if $A' \models T^*$ then $A^* \cong_{\Sigma^*/\Sigma} A'$.

Suppose $A' \models T^*$ and the corresponding operators in A' are Null', Lgth', Ap', Update', $\mathsf{Newlength}'$ and eq'.

Suppose $a' \in A'$, $\mathsf{Ap}'(a', i) = a_i$ $(i < \mathsf{Lgth}'(a'))$ and write $a'(l+1)$ for a' whose length is $l+1$.

Define $[a_0, \ldots, a_l]'$ by recursion:

Base:

$$[\ ]' = \mathsf{Null}'$$

Induction:

$$[a_0, \ldots, a_l]' = \mathsf{Update}'(\mathsf{Newlength}'([a_0, \ldots, a_{l-1}]', l+1), l, a_l)$$

Then we will show $a'(l+1) = [a_0, \ldots, a_l]'$ by simple induction on l.

Base: $l = 0$

$$a' = \mathsf{Null}' = [\ ]'$$

Inductive step:

Suppose for l, $a'(l) = [a_0, \ldots, a_{l-1}]'$, then for $l+1$,

$$\begin{aligned}
\mathsf{Lgth}'([a_0, \ldots, a_l]') &= \mathsf{Lgth}'(\mathsf{Update}'(\mathsf{Newlength}'([a_0, \ldots, a_{l-1}]', l+1), l, a_l)) \\
&= \mathsf{Lgth}'(\mathsf{Update}'(\mathsf{Newlength}'(a'(l), l+1), l, a_l)) && \text{(by i.h.)} \\
&= \mathsf{Lgth}'(\mathsf{Newlength}'(a'(l), l+1)) && \text{(by (3))} \\
&= l+1 && \text{(by (6))}
\end{aligned}$$

and for $i < l+1$,

$$
\begin{aligned}
\mathsf{Ap}'([a_1,\ldots,a_l]',i) &= \mathsf{Ap}'(\mathsf{Update}'(\mathsf{Newlength}'([a_0,\ldots,a_{l-1}]',l+1),l,a_l),i) \\
&= \mathsf{Ap}'(\mathsf{Update}'(\mathsf{Newlenth}'(a'(l),l+1)l,a_l),i) && \text{(by i.h.)} \\
&= \begin{cases} \mathsf{Ap}'(\mathsf{Newlenth}'(a'(l),l+1),i) & \text{if } \ i<l \\ a_l & \text{if } \ i=l \end{cases} && \text{by (4), (5)} \\
&= \begin{cases} \mathsf{Ap}'(a'(l),i) & \text{if } \ i<l \\ a_l & \text{if } \ i=l \end{cases} && \text{by (7)} \\
&= \mathsf{Ap}'(a'(l),i)
\end{aligned}
$$

Then, by the array equality axiom (10),

$$a'(l+1) = [a_0,\ldots,a_l]'$$

Now, we can define a function $h : A^* \mapsto A'$ by:

$$h(a^*) = a'$$

where

$$\mathsf{Lgth}(a^*) = \mathsf{Lgth}'(a') = l+1$$

$$\forall i < l+1 \ (\mathsf{Ap}(a^*,i) = \mathsf{Ap}'(a',i))$$

and the reduct of h on A is the identity. Then prove that h is the isomorphism from A^* to A'.

First prove h preserves the functions: Null, Lgth, Ap, Update, $\mathsf{Newlength}$ and eq^*.

When $\mathsf{Lgth}(a^*) = \mathsf{Lgth}(a') = 0$, $a^* = \mathsf{Null}$, $a' = \mathsf{Null}'$, then

$h(\mathsf{Null}) = \mathsf{Null}'$

$$\begin{aligned}
h(\mathsf{Lgth}(a^*)) &= h(l+1)\\
&= l+1\\
&= \mathsf{Lgth}'(a')
\end{aligned}$$

$$\begin{aligned}
h(\mathsf{Newlength}(a^*,k)) &= \begin{cases} h[a_0,\dots,a_{k-1}] & \text{if } k < \mathsf{Lgth}(a^*),\\ h[a_0,\dots,a_l,\underbrace{\delta,\dots,\delta}_{k-l-1}] & \text{otherwise.}\end{cases}\\
&= \begin{cases} [a_0,\dots,a_{k-1}]' & \text{if } k < \mathsf{Lgth}(a^*),\\ [a_0,\dots,a_l,\underbrace{\delta,\dots,\delta}_{k-l-1}]' & \text{otherwise.}\end{cases}\\
&= \mathsf{Newlength}'(a',k)
\end{aligned}$$

$$\begin{aligned}
h(\mathsf{Ap}(a^*,k)) &= \begin{cases} h(\mathsf{Ap}(a^*,k)) & \text{if } k < \mathsf{Lgth}(a^*),\\ \delta & \text{otherwise}\end{cases}\\
&= \begin{cases} \mathsf{Ap}(a^*,k) & \text{if } k < \mathsf{Lgth}(a^*)\\ \delta & \text{otherwise}\end{cases}\\
&= \begin{cases} \mathsf{Ap}(a',k) & \text{if } k < \mathsf{Lgth}(a^*)\\ \delta & \text{otherwise}\end{cases}\\
&= \mathsf{Ap}'(a',k)
\end{aligned}$$

$$\begin{aligned}
h(\mathsf{Update}(a^*,k,x)) &= h[a_0,\dots,x\dots,a_l]\\
&= [a_0,\dots,x\dots,a_l]'\\
&= \mathsf{Update}'(a',k,x)
\end{aligned}$$

We will prove h preserves equpto by simple induction on z.

Base:

$$h(\mathsf{equpto}(a_1^*, a_2^*, 0)) = \mathsf{tt} = \mathsf{equpto}'(a_1', a_2', 0)$$

Inductive step:

Suppose

$$\mathsf{equpto}(a_1^*, a_2^*, z) \downarrow \Leftrightarrow \ \mathsf{equpto}'(a_1', a_2', z) \downarrow$$

and when $\mathsf{equpto}(a_1^*, a_2^*, z) \downarrow$

$$h(\mathsf{equpto}(a_1^*, a_2^*, z)) = \mathsf{equpto}'(a_1', a_2', z)$$

then for $z + 1$:

$$\begin{aligned}
& \mathsf{equpto}(a_1^*, a_2^*, z+1) \downarrow \\
\Rightarrow\ & \mathsf{equpto}(a_1^*, a_2^*, z+1) = \mathsf{tt} \ \text{ or } \ \mathsf{equpto}(a_1^*, a_2^*, z+1) = \mathsf{ff} \\
\Rightarrow\ & (\mathsf{equpto}(a_1^*, a_2^*, z) = \mathsf{tt} \ \text{ and } \ \mathsf{eq}(\mathsf{Ap}(a_1^*, z), \mathsf{Ap}(a_2^*, z)) = \mathsf{tt}) \\
& \text{or } \ \mathsf{equpto}(a_1^*, a_2^*, z+1) = \mathsf{ff} \\
\Rightarrow\ & (h(\mathsf{equpto}(a_1^*, a_2^*, z)) = \mathsf{tt} \ \text{ and } \ \mathsf{eq}(h(\mathsf{Ap}(a_1^*, z)), h(\mathsf{Ap}(a_2^*, z))) = \mathsf{tt}) \\
& \text{or } \ h(\mathsf{equpto}(a_1^*, a_2^*, z+1)) = \mathsf{ff}
\end{aligned}$$

So, $h(\mathsf{equpto}(a_1^*, a_2^*, z+1)) = \mathsf{tt}$ or $h(\mathsf{equpto}(a_1^*, a_2^*, z+1)) = \mathsf{ff}$

Also, by i.h. and h preserves Ap, we can get:

$$(\mathsf{equpto}'(a_1', a_2', z) = \mathsf{tt} \text{ and } \mathsf{eq}(\mathsf{Ap}'(a_1', z), \mathsf{Ap}(a_2', z)) = \mathsf{tt}) \text{ or } \mathsf{equpto}'(a_1', a_2', z) = \mathsf{ff}$$

and then

$$\mathsf{equpto}'(a_1', a_2', z+1) = \mathsf{tt} \ \text{ or } \mathsf{equpto}'(a_1', a_2', z+1) = \mathsf{ff}$$

Thus

$$h(\mathsf{equpto}(a_1^*, a_2^*, z+1)) = \mathsf{equpto}'(a_1', a_2', z+1)$$

If $\mathsf{equpto}'(a'_1, a'_2, z+1)\downarrow$, we can similarly get

$$h(\mathsf{equpto}(a^*_1, a^*_2, z+1)) = \mathsf{equpto}'(a'_1, a'_2, z+1)$$

So, we proved h preserves equpto.

Then, we can prove h preserves eq^* by the method similar to that for equpto.

Now, we will show that h is 1-1 and onto.

Suppose $h(a^*_1) = a'_1,\ h(a^*_2) = a'_2$,

$$\begin{aligned} h(a^*_1) = h(a^*_2) &\Rightarrow\ a'_1 = a'_2 \\ &\Rightarrow\ \mathsf{Lgth}'(a'_1) = \mathsf{Lgth}'(a'_2),\ \mathsf{equpto}'(a'_1, a'_2, z) = \mathsf{tt} \\ &\Rightarrow\ \mathsf{Lgth}(a^*_1) = \mathsf{Lgth}(a^*_2),\ \mathsf{equpto}(a^*_1, a^*_2, z) = \mathsf{tt} \\ &\Rightarrow\ a^*_1 = a^*_2 \end{aligned}$$

So, h is 1-1.

For any $a' \in A'$,

$$\begin{aligned} a' = &\ [a_0, \ldots, a_l]' \\ &= \mathsf{Update}'(\mathsf{Newlength}([a_0, \ldots, a_{l-1}]', l+1), l, a_l)) \\ &= \mathsf{Update}'(\mathsf{Newlength}'(\cdots \mathsf{Newlength}'(\mathsf{Null}', 1), 0, a_0 \cdots), l, a_l) \\ &= \mathsf{Update}'(\mathsf{Newlength}'(\cdots \mathsf{Newlength}'(h(\mathsf{Null}), 1), 0, a_0 \cdots), l, a_l) \\ &= \mathsf{Update}'(\mathsf{Newlength}'(\cdots h(\mathsf{Newlength}(\mathsf{Null}, 1), 0, a_0 \cdots)), l, a_l) \\ &= h(\mathsf{Update}(\mathsf{Newlength}(\cdots (\mathsf{Newlength}(\mathsf{Null}, 1), 0, a_0 \cdots)), l, a_l)) \\ &= h(a^*) \end{aligned}$$

So, for any $a' \in A'$, we can find an $a^* \in A^*$ such that $h(a^*) = a'$. i.e., h is onto.

So, $A^* \cong_{\Sigma'/\Sigma} A'$.

According to Definition 3.4 for specification, $(\Sigma^*,\ T^*)$ specifies A^* □

For an N-standard Σ-algebra A and a $\boldsymbol{\mu PR}^*$ derivation α, let f^A_α be the function on A defined by α and let $\vec{\mathsf{g}}^A_\alpha$, $\vec{\mathsf{h}}^A_\alpha$ be the corresponding auxiliary function tuple on A^*.

Corollary 5.9. For each $\boldsymbol{\mu PR}^*$ derivation and N-standard Σ-algebra A, let

$$\Sigma^*_\alpha = \Sigma^* \cup \{\mathsf{f}_\alpha, \vec{\mathsf{g}}_\alpha\, \vec{\mathsf{h}}_\alpha\}$$

and

$$E^*_\alpha = \boldsymbol{ArrAx}(\Sigma) \cup E_\alpha$$

Then the Kleene conditional equational specification$(\Sigma^*_\alpha, E^*_\alpha)$ specifies f^A_α with hidden functions and sorts.

Proof: By Definition 3.8 and Theorems 5.6 and 5.8. $\square$

Remark 5.10 (Minimal definability of $\boldsymbol{\mu PR}$ functions).

For each $\boldsymbol{\mu PR}$ derivation α, we can also give sets of equations F_α which minimally define f^A_α. They are the same as the specifications E_α shown in §5.1.1 except for the case of the $\boldsymbol{\mu}$-operator. Suppose the derivation for g^A_α is β, then the minimal definition for the μ-operator is (using an auxiliary function h):

$$\begin{aligned} F_\alpha = F_\beta \;\cup \{\; & \mathsf{h}(\vec{\mathsf{x}}, \mathsf{z}) \simeq \mathsf{if}\;\; \mathsf{g}(\vec{\mathsf{x}}, \mathsf{z}) \;\; \mathsf{then} \;\; \mathsf{z} \;\; \mathsf{else} \;\; \mathsf{h}(\vec{\mathsf{x}}, S(\mathsf{z})) \;\; \mathsf{fi}, \\ & \mathsf{f}(\vec{\mathsf{x}}) \simeq \mathsf{h}(\vec{\mathsf{x}}, 0) \\ \} & \end{aligned}$$

This defines the $\boldsymbol{\mu}$-operator minimally (see Theorem 6.10) but not uniquely. So, as we can see, the minimal definition theory for $\boldsymbol{\mu PR}^*$ computable functions is a set of *equations*, i.e, an *equational* minimal definition theory, which is simpler than unique definition theory, and its relation to computability will be discussed in Chapter 6 and 7.

5.2 Algebraic specification in strict equational logic for computable functions

The semantics for *equations*, *conditional equations* and *conditional* BU *equations* in *strict equalitional logic* are:

1. $A \models_\sigma t_1 = t_2$ iff $[\![t_1]\!]^A\sigma\downarrow$ and $[\![t_2]\!]^A\sigma\downarrow$ and $[\![t_1]\!]^A\sigma = [\![t_2]\!]^A\sigma$

2. $A \models_\sigma \forall \mathtt{y} < t(t_1(\vec{\mathtt{x}}, \mathtt{y}) = t_2(\vec{\mathtt{x}}, \mathtt{y}))$ iff
   for all $i < k$, $A \models_\sigma t_1(\vec{\mathtt{x}}, \bar{i}) = t_2(\vec{\mathtt{x}}, \bar{i})$
   (where $[\![t]\!]^A\sigma = k,$ and $\bar{i}$ is the numeral of i.)

3. $A \models_\sigma P_1, \ldots, P_n \to P$ iff
   $A \models_\sigma P_i$ for $i = 1, \ldots, n \Rightarrow A \models_\sigma P$

4. $A \models E$ iff for all σ, $A \models_\sigma E$

Note that clauses (2)–(4) are the same as the corresponding clauses in §5.1.

5.2.1 Algebraic specification for $\boldsymbol{\mu PR}$ computable functions

For each $\boldsymbol{\mu PR}(\Sigma)$ derivation α, there is a finite set E_α of specifying conditional equation for the function f^A_α defined by α, as well as the auxiliary functions $\vec{\mathsf{g}}^A_\alpha$ and $\vec{\mathsf{h}}^A_\alpha$. The set E_α is defined by CV induction on the length of the derivation α as shown in cases (i)-(vi) below:

(i) *Primitive* Σ*-functions:* $\alpha = (\mathsf{f}, \sigma)$ where σ is the scheme for a Σ-primitive function F. Then

$$\begin{aligned} E_\alpha \;=\; \{ \;\; & \mathsf{F}(\vec{\mathsf{x}}) = \mathsf{F}(\vec{\mathsf{x}}) \;\rightarrow\; \mathsf{f}(\vec{\mathsf{x}}) = \mathsf{F}(\vec{\mathsf{x}}), && (1)\\ & \mathsf{f}(\vec{\mathsf{x}}) = \mathsf{f}(\vec{\mathsf{x}}) \;\rightarrow\; \mathsf{F}(\vec{\mathsf{x}}) = \mathsf{F}(\vec{\mathsf{x}}) && (2)\\ \} \end{aligned}$$

Note for a function f^A_α which is the Σ-primitive function F, if F^A is total, then f^A_α can be specified by the single equation

$$\mathsf{f}(\vec{\mathsf{x}}) = \mathsf{F}(\vec{\mathsf{x}})$$

but if F^A is partial, the equation can not specify f^A_α since when $\mathsf{F}^A(\vec{x})$ diverges, the equation does not hold. So, we need conditional equations here.

Constants: $\alpha = (\mathsf{f}, \sigma)$ where σ is the scheme for a constant c. Then

$$E_\alpha \;=\; \{\mathsf{f}(\vec{\mathsf{x}}) = \mathsf{c}\}$$

(ii) *Projection:* $\alpha = (\mathsf{f}, \sigma)$ where σ is the scheme for projection. Then

$$E_\alpha \;=\; \{\mathsf{f}(\vec{\mathsf{x}}) = \vec{\mathsf{x}}_i\}$$

(iii) *Composition:* $\alpha = ((\mathsf{g}_1, \sigma_1), \ldots, (\mathsf{g}_m, \sigma_m), (\mathsf{h}, \sigma_{m+1}), (\mathsf{f}, \sigma_{m+2}))$ where σ_{m+2} is the scheme for composition. Suppose the derivation for $\mathsf{g}^A_{\alpha i}$ is β_i $(1 \le i \le m)$, for h^A_α is γ. Then

$$\begin{aligned} E_\alpha \;=\; & E_{\beta_1} \cup \cdots \cup E_{\beta_m} \cup E_\gamma \cup \\ & \{ \;\; \mathsf{g}_1(\vec{\mathsf{x}}) = \mathsf{y}_1, \cdots \mathsf{g}_m(\vec{\mathsf{x}}) = \mathsf{y}_m, \mathsf{h}(\mathsf{y}_1, \ldots, \mathsf{y}_m) = \mathsf{y} \;\rightarrow\; \mathsf{f}(\vec{\mathsf{x}}) = \mathsf{y}, && (1)\\ & \quad\;\; \mathsf{f}(\vec{\mathsf{x}}) = \mathsf{f}(\vec{\mathsf{x}}) \;\rightarrow\; \mathsf{g}_i(\vec{\mathsf{x}}) = \mathsf{g}_i(\vec{\mathsf{x}}) \quad (1 \le i \le m), && (2)\\ & \quad\;\; \mathsf{f}(\vec{\mathsf{x}}) = \mathsf{y} \;\rightarrow\; \mathsf{h}(\mathsf{g}_1(\vec{\mathsf{x}}) \ldots \mathsf{g}_m(\vec{\mathsf{x}})) = \mathsf{y} && (3)\\ & \} \end{aligned}$$

(iv) *Definition by cases:* $\alpha = ((\mathsf{h}, \sigma_1)(\mathsf{g}_1, \sigma_2), (\mathsf{g}_2, \sigma_3), (\mathsf{f}, \sigma_4))$ where σ_4 is the scheme for definition by cases. Suppose the derivation for $\mathsf{g}^A_{\alpha 1}$, $\mathsf{g}^A_{\alpha 2}$, h^A_α are β_1, β_2, γ respectively. Then

$$
\begin{aligned}
E_\alpha \;=\;\; & E_{\beta_1} \cup E_{\beta_2} \cup E_\gamma \cup \\
\{\;\; & \mathsf{h}(\vec{\mathsf{x}}) = \mathsf{true}\ ,\ \mathsf{g}_1(\vec{\mathsf{x}}) = \mathsf{y} \;\rightarrow\; \mathsf{f}(\vec{\mathsf{x}}) = \mathsf{y}, && (1)\\
& \mathsf{h}(\vec{\mathsf{x}}) = \mathsf{false},\ \mathsf{g}_2(\vec{\mathsf{x}}) = \mathsf{y} \;\rightarrow\; \mathsf{f}(\vec{\mathsf{x}}) = \mathsf{y}, && (2)\\
& \mathsf{f}(\vec{\mathsf{x}}) = \mathsf{f}(\vec{\mathsf{x}}) \;\rightarrow\; \mathsf{h}(\vec{\mathsf{x}}) = \mathsf{h}(\vec{\mathsf{x}}), && (3)\\
& \mathsf{f}(\vec{\mathsf{x}}) = \mathsf{y},\ \mathsf{h}(\vec{\mathsf{x}}) = \mathsf{true} \;\rightarrow\; \mathsf{g}_1(\vec{\mathsf{x}}) = \mathsf{y}, && (4)\\
& \mathsf{f}(\vec{\mathsf{x}}) = \mathsf{y},\ \mathsf{h}(\vec{\mathsf{x}}) = \mathsf{false} \;\rightarrow\; \mathsf{g}_2(\vec{\mathsf{x}}) = \mathsf{y} && (5)\\
\}\;\; &
\end{aligned}
$$

(v) *Simultaneous primitive recursion:* $\alpha = ((\mathsf{g}_i, \sigma_{1i}), (\mathsf{h}_i, \sigma_{2i}), (\mathsf{f}_i, \sigma_{3i}))$ where σ_{3i} is the scheme for simultaneous primitive recursion ($i = 1, \ldots, m$). Suppose the derivation for $\mathsf{g}^A_{\alpha i}$ is β_i, and that for $\mathsf{h}^A_{\alpha i}$ is γ_i. Then

$$
\begin{aligned}
E_\alpha = \;\; & E_{\beta_1} \cup \cdots \cup E_{\beta_2} \cup E_{\gamma_1} \cup \cdots \cup E_{\gamma_m} \\
\{\;\; & \mathsf{g}_i(\vec{\mathsf{x}}) = \mathsf{y} \;\rightarrow\; f_i(0, x) = \mathsf{y}, && (1)\\
& \mathsf{f}_i(0, \vec{\mathsf{x}}) = \mathsf{f}_i(0, \vec{\mathsf{x}}) \;\rightarrow\; \mathsf{g}_i(\vec{\mathsf{x}}) = \mathsf{g}_i(\vec{\mathsf{x}}), && (2)\\
& \mathsf{f}_1(\mathsf{z}, \vec{\mathsf{x}}) = \mathsf{y}_1, \cdots \mathsf{f}_m(\mathsf{z}, \vec{\mathsf{x}}) = \mathsf{y}_m,\ \mathsf{h}_i(\mathsf{z}+1, \vec{\mathsf{x}}, \mathsf{y}_1, \ldots, \mathsf{y}_m) = \mathsf{y} \\
& \qquad \rightarrow\; \mathsf{f}_i(\mathsf{z}+1, \vec{\mathsf{x}}) = \mathsf{y}, && (3)\\
& \mathsf{f}_i(\mathsf{z}+1, \vec{\mathsf{x}}) = \mathsf{f}_i(\mathsf{z}+1, \vec{\mathsf{x}}) \;\rightarrow\; \mathsf{f}_j(\mathsf{z}, \vec{\mathsf{x}}) = \mathsf{f}_j(\mathsf{z}, \vec{\mathsf{x}}), && (4)\\
& \mathsf{f}_i(\mathsf{z}+1, \vec{\mathsf{x}}) = \mathsf{y} \;\rightarrow\; \mathsf{h}_i(\mathsf{z}, \vec{\mathsf{x}}, \mathsf{f}_1(\mathsf{z}, \vec{\mathsf{x}}) \ldots \mathsf{f}_m(\mathsf{z}, \vec{\mathsf{x}})) = \mathsf{y} && (5)\\
& (i, j = 1, \ldots, m) \\
\}\;\; &
\end{aligned}
$$

(vi) $\boldsymbol{\mu}$*-operator:* $\alpha = ((\mathsf{g}, \sigma_1), (\mathsf{f}, \sigma_2))$ where σ_2 is the scheme for the $\boldsymbol{\mu}$-operator.

Define a boolean-valued function symbol:

$$\mathsf{h} : u \times \mathsf{nat} \ \rightarrow \ \mathsf{bool}$$

that satisfies in A:

$$\mathsf{h}_\alpha^A(x,n) = \mathtt{tt} \ \Leftrightarrow \ \forall y < z \ (\mathsf{g}_\alpha^A(x,z) = \mathtt{ff})$$

by which we can reduce BU conditional equation to conditional equations according to the method in §5.1.2.

Suppose the derivation for g_α^A is β, then

$$\begin{aligned} E_\alpha = E_\beta \cup \ \{ \ & \rightarrow \ \mathsf{h}(\mathtt{x},0) = \mathsf{true}, \\ & \mathsf{h}(\mathtt{x},\mathsf{z}) = \mathsf{true}, \mathsf{g}(\mathtt{x},\mathsf{z}) = \mathsf{false} \ \rightarrow \ \mathsf{h}(\mathtt{x},\mathsf{z}+1)) = \mathsf{true}, \\ & \mathsf{h}(\mathtt{x},\mathsf{z}+1) = \mathsf{true} \ \rightarrow \ \mathsf{h}(\mathtt{x},\mathsf{z}) = \mathsf{true}, \\ & \mathsf{h}(\mathtt{x},\mathsf{z}+1) = \mathsf{true} \ \rightarrow \ \mathsf{g}(\mathtt{x},\mathsf{z}) = \mathsf{false}, \\ & \mathsf{h}(\mathtt{x},\mathsf{z}) = \mathsf{true}, \mathsf{g}(\mathtt{x},\mathsf{z}) = \mathsf{true} \ \rightarrow \ \mathsf{f}(\mathtt{x}) = \mathsf{z}, \\ & \mathsf{f}(\mathtt{x}) = \mathsf{z} \ \rightarrow \ \mathsf{g}(\mathtt{x},\mathsf{z}) = \mathsf{true}, \\ & \mathsf{f}(\mathtt{x}) = \mathsf{z} \ \rightarrow \ \mathsf{h}(\mathtt{x},\mathsf{z}) = \mathsf{true} \\ \} & \end{aligned}$$

Remark 5.11. Note how complicated the specifications for $\boldsymbol{\mu PR}$ computable functions are in strict equational logic, compared to Kleene equational logic (cf. Remark 5.1(1)).

Theorem 5.12 (Strict conditional equational specification of $\boldsymbol{\mu PR}$ functions). For each $\boldsymbol{\mu PR}(\Sigma)$ derivation α and N-standard Σ-algebra A, the strict conditional equational specification $(\Sigma_\alpha, \ E_\alpha)$ specifies the $\boldsymbol{\mu PR}$ computable function f_α^A with hidden functions.

Proof: By course of values induction on the length of α. (The equation numbers below refer to the definition of E_α above).

Base case:

(i) *Primitive* Σ- *functions*:

It is clear that $(A, \mathsf{f}_\alpha^A) \models E_\alpha$ (see definition of E_α above).

Suppose that $(A, f) \models E_\alpha$, then for any $\vec{x}$:

$$\begin{aligned} f(\vec{x})\downarrow \;\; &\Rightarrow \mathsf{F}^A(\vec{x})\downarrow && \text{(by (2))} \\ &\Rightarrow f(\vec{x}) = \mathsf{F}^A(\vec{x}),\ \mathsf{f}_\alpha^A(\vec{x}) = \mathsf{F}^A(\vec{x}) && \text{(by (1))} \\ &\Rightarrow f(\vec{x}) = \mathsf{f}_\alpha^A(\vec{x}) \end{aligned}$$

So, $f \subseteq \mathsf{f}_\alpha^A$.

Similarly, we can get $\mathsf{f}_\alpha^A \subseteq f$, then $f = \mathsf{f}_\alpha^A$, i.e. f_α^A is unique.

Hence, $(\Sigma_\alpha, E_\alpha)$ specifies f_α^A (by Definition 3.5)

Constants:

It is obvious that $(A, \mathsf{f}_\alpha^A) \models E_\alpha$ and f_α^A is unique

Hence, $(\Sigma_\alpha,\ E_\alpha)$ specifies f_α^A

(ii) *Projection*:

It is obvious that $(A, \mathsf{f}_\alpha^A) \models E_\alpha$ and f_α^A is unique

Hence, $(\Sigma_\alpha,\ E_\alpha)$ specifies f_α^A

Induction steps:

(iii) *Composition:*

Clearly, $(A, \vec{\mathsf{g}}_\alpha^A, \mathsf{h}_\alpha^A, \mathsf{f}_\alpha^A) \models E_\alpha$.

Suppose $(A, \vec{g}, h, f) \models E_\alpha$, then for any $\vec{x}$:

$f(\vec{x})\downarrow \Rightarrow\ g_i(\vec{x})\downarrow,\ h(g_1(\vec{x}), \ldots, g_m(\vec{x}))\downarrow$ (by (2),(3)) Suppose $g_i(\vec{x}) = y_i$ and $h(y_1, \ldots, y_m) = y$,

then $f(\vec{x}) = y$ (by (1))

By i.h., $\vec{g} = \vec{\mathsf{g}}_\alpha^A$, $h = \mathsf{h}_\alpha^A$, then:

$\mathsf{f}_\alpha^A(\vec{x}) = y$ (by (1))

So, $f \subseteq \mathsf{f}_\alpha^A$.

Similarly, we can get $\mathsf{f}_\alpha^A \subseteq f$, then $f = \mathsf{f}_\alpha^A$, i.e. f_α^A is unique.

Hence, $(\Sigma_\alpha,\ E_\alpha)$ specifies f_α^A with hidden functions.

(iv) *Definition by cases:*

Clearly, $(A, \vec{\mathsf{g}}_\alpha^A, \mathsf{h}_\alpha^A, \mathsf{f}_\alpha^A) \models E_\alpha$.

Suppose $(A, g_1, g_2, h, f) \models E_\alpha$, then

$$
\begin{aligned}
f(\vec{x}) \downarrow \ &\Rightarrow\ h(\vec{x}) \downarrow && \text{(by (3))}\\
&\Rightarrow\ (h(\vec{x}) = \mathtt{tt} \text{ and } g_1(\vec{x}) \downarrow) \text{ or } (h(\vec{x}) = \mathtt{ff} \text{ and } g_2(\vec{x}) \downarrow) && \text{(by (4), (5))}\\
&\Rightarrow\ f(\vec{x}) = g_1(\vec{x}) \text{ or } f(\vec{x}) = g_2(\vec{x}) && \text{(by (1), (2))}
\end{aligned}
$$

by i.h., $h = \mathsf{h}_\alpha^A$, $\vec{g} = \vec{\mathsf{g}}_\alpha^A$, then:

$$
\begin{aligned}
&(\mathsf{h}_\alpha^A(\vec{x}) = \mathtt{tt} \text{ and } \mathsf{g}_{\alpha 1}^A(\vec{x}) \downarrow) \text{ or } (\mathsf{h}_\alpha^A(\vec{x}) = \mathtt{ff} \text{ and } \mathsf{g}_{\alpha 2}^A(\vec{x}) \downarrow)\\
\Rightarrow\ &\mathsf{f}_\alpha^A(\vec{x}) = \mathsf{g}_{\alpha 1}^A(\vec{x}) \text{ or } \mathsf{f}_\alpha^A(\vec{x}) = \mathsf{g}_{\alpha 2}^A(\vec{x}) && \text{(by (1), (2))}\\
\Rightarrow\ &\mathsf{f}_\alpha^A(\vec{x}) = g_1(\vec{x}) \text{ or } \mathsf{f}_\alpha^A(\vec{x}) = g_2(\vec{x})
\end{aligned}
$$

So, $f \subseteq \mathsf{f}_\alpha^A$.

Similarly, we can get $\mathsf{f}_\alpha^A \subseteq f$, then $f = \mathsf{f}_\alpha^A$, i.e. f_α^A is unique.

Hence, $(\Sigma_\alpha,\ E_\alpha)$ specifies f_α^A with hidden functions.

(v) Simultaneous primitive recursion on $\mathbb{N}$:

Clearly, $(A, \vec{\mathsf{g}}_\alpha^A, \vec{\mathsf{h}}_\alpha^A, \mathsf{f}_\alpha^A) \models E_\alpha$.

Suppose $(A, \vec{g}, \vec{h}, f) \models E_\alpha$, then prove for any $\vec{x}$,

$$f_i(n, \vec{x}) \downarrow \;\Rightarrow\; \mathsf{f}^A_{\alpha i}(n, \vec{x}) = f_i(n, \vec{x})$$

by simple induction on n.

Base: $n = 0$

$f_i(0, \vec{x}) \downarrow \Rightarrow\; f_i(0, \vec{x}) = g_i(\vec{x}) = \mathsf{g}^A_{\alpha i}(\vec{x}) = \mathsf{f}^A_{\alpha i}(0, \vec{x})$.

Induction step:

Suppose $f_i(n, \vec{x}) \downarrow \Rightarrow\; \mathsf{f}^A_{\alpha i}(n, \vec{x}) = f_i(n, \vec{x})$, then

$f_i(n+1, \vec{x}) \downarrow \Rightarrow\quad f_j(n, \vec{x}) \downarrow (j = 1, \ldots, m),\; h_i(n, \vec{x}, f_1(n, \vec{x}) \ldots, f_m(n, \vec{x})) \downarrow$

(by(4), (5))

Suppose $f_j(n, \vec{x}) = y_j\ (1 \leq j \leq m)$, $h_i(n, \vec{x}, y_1, \ldots, y_m) = y$,

then $f_i(n+1, \vec{x}) = y$ (by (1))

By i.h. $h_i = \mathsf{h}^A_{\alpha i}$ and $\mathsf{f}^A_{\alpha i}(n, \vec{x}) = f_i(n, \vec{x})$, then

$$\mathsf{f}^A_{\alpha j}(n, \vec{x}) = y_j\ (1 \leq j \leq m),\; \mathsf{h}^A_{\alpha i}(n, \vec{x}, y_1, \ldots, y_m) = y$$

So, $\mathsf{f}^A_{\alpha}(n+1, \vec{x}) = y$ (by (3)

Therefore, $f_i(n+1, \vec{x}) \downarrow \Rightarrow\; \mathsf{f}^A_{\alpha i}(n+1, \vec{x}) = f_i(n+1, \vec{x})$

So, we can get $f \subseteq \mathsf{f}^A_\alpha$.

Similarly, we can get $\mathsf{f}^A_\alpha \subseteq f$, then $f = \mathsf{f}^A_\alpha$, i.e. f^A_α is unique.

Hence, $(\Sigma_\alpha,\ E_\alpha)$ specifies f^A_α with hidden functions.

(vi) $\boldsymbol{\mu}$-operator:

It is clear that $(A, \mathsf{g}^A_\alpha, \mathsf{f}^A_\alpha) \models E_\alpha$.

Suppose $(A, g, f) \models E_\alpha$. By i.h., g^A_α is unique and we can prove h^A_α is unique by simple induction on $\mathbb{N}$.

Then for any $\vec{x}$

$$\begin{aligned} f(\vec{x}) \downarrow z \quad &\Rightarrow \quad h(\vec{x}, z) = \mathsf{tt},\ g(\vec{x}, z) = \mathsf{tt} && \text{(by (4), (5))} \\ &\Rightarrow \quad \mathsf{h}^A_\alpha(\vec{x}, z) = \mathsf{tt},\ \mathsf{g}^A_\alpha(\vec{x}, z) = \mathsf{tt} \\ &\Rightarrow \quad \mathsf{f}^A_\alpha \downarrow z && \text{(by (3'))} \end{aligned}$$

Similarly, we can get $\mathsf{f}^A_\alpha \subseteq f$, then $f = \mathsf{f}^A_\alpha$, i.e. f^A_α is unique.

Hence, $(\Sigma_\alpha,\ E_\alpha)$ specifies f^A_α with hidden functions. $\Box$

5.2.2 Algebraic specification for $\boldsymbol{\mu PR^*}$ computable functions

For the specification of $\boldsymbol{\mu PR}^*$ functions, we need a conditional equational theory in *strict equational logic* for the array operators added to Σ.

$$
\begin{aligned}
\boldsymbol{ArrAx}(\Sigma) = \{ \quad & \rightarrow \ \mathsf{Lgth}(\mathsf{Null}) = 0, && (1)\\
& \rightarrow \ \mathsf{Ap}(\mathsf{a}^*, \mathsf{z}) = \mathsf{if}\ \ \mathsf{less_{nat}}(\mathsf{z}, \mathsf{Lgth}(\mathsf{a}^*))\ \ \mathsf{then}\ \ \mathsf{Ap}(\mathsf{a}^*, \mathsf{z})\ \ \mathsf{else}\ \ \delta\ \ \mathsf{fi}, && (2)\\
& \rightarrow \ \mathsf{Lgth}(\mathsf{Update}(\mathsf{a}^*, \mathsf{z}, \mathtt{x})) = \mathsf{Lgth}(\mathsf{a}^*), && (3)\\
& \rightarrow \ \mathsf{Ap}(\mathsf{Update}(\mathsf{a}^*, \mathsf{z}_1, \mathtt{x}), \mathsf{z}) = \mathsf{if}\ \ \mathsf{eq_{nat}}(\mathsf{z}, \mathsf{z}_1)\ \ \mathsf{then}\ \ \mathtt{x}\ \ \mathsf{else}\ \mathsf{Ap}(\mathsf{a}^*, \mathsf{z})\ \ \mathsf{fi}, && (4)\\
& \rightarrow \ \mathsf{Ap}(\mathsf{Update}(\mathsf{a}^*, \mathsf{z}, \mathtt{x}), \mathsf{z}) = \mathsf{if}\ \ \mathsf{less_{nat}}(\mathsf{z}, \mathsf{Lgth}(\mathsf{a}^*))\ \ \mathsf{then}\ \ \mathtt{x}\ \ \mathsf{else}\ \ \delta\ \ \mathsf{fi}, && (5)\\
& \rightarrow \ \mathsf{Lgth}(\mathsf{Newlength}(\mathsf{a}^*, \mathsf{z})) = \mathsf{z}, && (6)\\
& \rightarrow \ \mathsf{Ap}(\mathsf{Newlength}(\mathsf{a}^*, \mathsf{z}_1), \mathsf{z})) = \mathsf{if}\ \mathsf{less_{nat}}(\mathsf{z}, \mathsf{z}_1)\ \mathsf{then}\ \mathsf{Ap}(\mathsf{a}^*, \mathsf{z})\ \mathsf{else}\ \delta\ \mathsf{fi}, && (7)\\
& \rightarrow \ \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, 0) = \mathsf{ture}, && (8)\\
& \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, \mathsf{z}) = \mathsf{true}, \mathsf{eq}(\mathsf{Ap}(\mathsf{a}_1^*, \mathsf{z}), \mathsf{Ap}(\mathsf{a}_2^*, \mathsf{z})) = \mathtt{x} \\
& \qquad \rightarrow \ \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, \mathsf{z}+1) = \mathtt{x}, && (9)\\
& \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, \mathsf{z}) = \mathsf{false} \ \rightarrow \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, \mathsf{z}+1) = \mathsf{false}, && (10)\\
& \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, \mathsf{z}+1) = \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, \mathsf{z}+1) \\
& \qquad \rightarrow \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, \mathsf{z}) = \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, \mathsf{z}), && (11)\\
& \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, \mathsf{z}+1) = \mathtt{x},\ \ \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, \mathsf{z}) = \mathsf{true} \\
& \qquad \rightarrow \mathsf{eq}(\mathsf{Ap}(\mathsf{a}_1^*, \mathsf{z}), \mathsf{Ap}(\mathsf{a}_2^*, \mathsf{z})) = \mathtt{x}, && (12)\\
& \mathsf{Lgth}(\mathsf{a}_1^*) = \mathsf{Lgth}(\mathsf{a}_2^*),\ \ \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, \mathsf{Lgth}(\mathsf{a}_1^*)) = \mathtt{x} \ \rightarrow\ \mathsf{eq}^*(\mathsf{a}_1^*, \mathsf{a}_2^*) = \mathtt{x}, && (13)\\
& \mathsf{Lgth}(\mathsf{a}_1^*) \neq \mathsf{Lgth}(\mathsf{a}_2^*) \rightarrow \mathsf{eq}^*(\mathsf{a}_1^*, \mathsf{a}_2^*) = \mathsf{false}, && (14)\\
& \mathsf{eq}^*(\mathsf{a}_1^*, \mathsf{a}_2^*) = \mathtt{x},\ \ \mathsf{Lgth}(\mathsf{a}_1^*) = \mathsf{Lgth}(\mathsf{a}_2^*) \ \rightarrow\ \mathsf{equpto}(\mathsf{a}_1^*, \mathsf{a}_2^*, \mathsf{Lgth}(\mathsf{a}_1^*)) = \mathtt{x} && (15)\\
\}
\end{aligned}
$$

Note that equations (1)–(7) here are the same as corresponding ones in $\boldsymbol{ArrAx}(\Sigma)$ (§5.1.3) in *Kleene equational logic* except for the substitution of '=' for '≃'. But the axioms for the auxiliary function equpto and the array equality operator eq^* are different, since they may be partial.

Theorem 5.13. For any N-standard Σ-algebra A, the specification$(\Sigma^*, \boldsymbol{ArrAx}(\Sigma))$

specifies A^* over A.

Proof: Similar to the proof of Theorem 5.8. □

Corollary 5.14. For each $\boldsymbol{\mu PR}^*(\Sigma)$ derivation α and N-standard Σ-algebra A, let

$$\Sigma^*_\alpha = \Sigma^* \cup \{\vec{\mathsf{g}}_\alpha, \vec{\mathsf{h}}_\alpha, \mathsf{f}_\alpha\}$$

$$E^*_\alpha = \boldsymbol{ArrAx}(\Sigma) \cup E_\alpha$$

Then the strict conditional equational specification $(\Sigma^*_\alpha, E^*_\alpha)$ specifies f^A_α with hidden functions and sorts.

Proof: Immediate from Definition 3.8 and Theorems 5.12 and 5.13. □

Remark 5.15 (Converse to Theorem 5.12 fails). The functions specifiable by conditional equations in strict equational logic need not be computable. The following are counterexamples.

1. In the algebra $\mathcal{N}_1 = (\mathbb{N}, 0, \mathsf{S}, +, *)$, define

$$f(x) = \begin{cases} 1 & \text{if } x \in K \\ 0 & \text{otherwise} \end{cases}$$

   where K is a recursively enumerable, non-recursive subset of $\mathbb{N}$. Then f is clearly not computable over $\mathbb{N}$, but is uniquely specifiable in the first order language with equality over $\mathcal{N}_1$ (i.e. $\boldsymbol{Form}(\Sigma(\mathcal{N}_1))$). This follows from the expressibility of K in this language, which can be seen by taking K to be the set given by the Matiyasevich/Davis/Putnam/Robinson proof of the unsolvability of Hilbert's Tenth problem. [MR75].

2. For a simpler example, take an algebra A of signature Σ with only one sort s and no function symbols. Let A^N be the N-standardization of A. Assume s is not an equality sort. Then, the total equality function on A

$$f : A^2 \;\rightarrow\; B$$

defined by

$$f(x,y) = \begin{cases} \mathtt{tt} & \text{if} \quad x = y \\ \mathtt{ff} & \text{if} \quad x \neq y \end{cases}$$

can be specified by the following set of conditional equations in strict equational logic:

$$\begin{aligned} \{ \quad &\rightarrow\; \mathsf{f}(\mathtt{x},\mathtt{y}) = \mathsf{f}(\mathtt{x},\mathtt{y}), \\ &\rightarrow\; \mathsf{f}(\mathtt{x},\mathtt{x}) = \mathsf{true}, \\ &\mathsf{f}(\mathtt{x},\mathtt{y}) = \mathsf{true} \;\rightarrow\; \mathtt{x} = \mathtt{y}\} \end{aligned}$$

But, this is not computable on A^N.

Remark 5.16 (Three-valued logical specification). There is another kind of specification logic for partial algebra: *3-valued logic* based on strict equality

$$[\![t_1 = t_2]\!]^A\sigma = \begin{cases} \mathtt{tt} & \text{if} \;\; [\![t_1]\!]^A\sigma\downarrow \;\text{ and }\; [\![t_2]\!]^A\sigma\downarrow \;\text{ and }\; [\![t_1]\!]^A\sigma = [\![t_2]\!]^A\sigma \\ \mathtt{ff} & \text{if} \;\; [\![t_1]\!]^A\sigma\downarrow \;\text{ and }\; [\![t_2]\!]^A\sigma\downarrow \;\text{ and }\; [\![t_1]\!]^A\sigma \neq [\![t_2]\!]^A\sigma \\ \mathtt{uu} & \text{otherwise} \end{cases}$$

Define

$$A \models_\sigma t_1 = t_2 \;\Leftrightarrow\; [\![t_1 = t_2]\!]^A\sigma = \mathtt{tt}$$

and

$$A \models_\sigma P_1, \ldots, P_n \rightarrow P \quad \text{iff} \quad \text{for all } i,\; A \models_\sigma P_i \; (i = 1, \ldots, n) \;\Rightarrow\; A \models_\sigma P$$

Then, it is clear that conditional equations and equations in this logic have the same semantics as those in strict 2-valued equality logic, and thus they have the same specification theories.

Chapter 6

Minimal solution for equations and conditional equations

The algebraic specification method characterizes functions as the solutions of systems of algebraic formulae. In the last chapter, we discussed algebraic specification for computable functions and showed that $\boldsymbol{\mu PR}^*$ computable functions can be specified, i.e. *uniquely defined* by conditional equational theories in each of two equational logics, and also defined as *minimal solutions* of sets of equations in Kleene equational logics. In this chapter, we will be interested in the reverse direction, i.e., given a conditional equation, can we find a solution which is unique or (failing that) minimal?

Remark 6.1. Unique solutions of conditional equations are relevant to *specification theories*, while minimal solution are relevant to *computation theories*, cf. Kleenee's theories of recursive functionals [Kle52] and the denotational semantics of recursive procedures [TZ88]. We will investigate this in Chapter 7.

Notation 6.2.

(a) Given a function tuple $\vec{\mathsf{g}} \equiv \mathsf{g}_1, \ldots, \mathsf{g}_m$ and variables $\vec{\mathsf{x}} : u$, let Σ be an N-standard signature and $\Sigma' = \Sigma \cup \{\vec{\mathsf{g}}\}$, then we write a Σ'-term t as $t[\vec{\mathsf{g}}, \vec{\mathsf{x}}]$ to indicate that t is generated from the function symbols $\vec{\mathsf{g}}$ as well as primitive Σ-functions F and the variable tuple $\vec{\mathsf{x}}$ only.

(b) We then write $t^{A'}[\vec{g}, \vec{x}]$ to mean the interpretation of t in the Σ'-algebra A' when $\vec{\mathsf{g}}$ is interpreted as a tuple of functions $\vec{g} \equiv g_1, \ldots, g_m$ of the same type, and $\vec{\mathsf{x}}$ is interpreted as $\vec{x} \in A^u$. For simplicity, we write $t^A[\vec{g}, \vec{x}]$ for $t^{A'}[\vec{g}, \vec{x}]$.

Notation 6.3.

(a) For fixed $\vec{g}$, t defines a function

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

by

$$f(\vec{x}) \simeq t^A[\vec{g}, \vec{x}] \text{ for all } \vec{x} \in A^u$$

We write f as $t^A[\vec{g}, \cdot]$.

Given two functions $f,\ g : A^u \to A_s$ and $\vec{x} \in A^u$, we write

$$f(\vec{x}) \sqsubseteq g(\vec{x})$$

to mean

$$f(\vec{x}) \downarrow \;\Rightarrow\; g(\vec{x}) \downarrow \text{ and } f(\vec{x}) = g(\vec{x})$$

and also write $t_1^A[\vec{g}, \vec{x}] \sqsubseteq t_2^A[\vec{g}, \vec{x}]$ to mean:

$$t_1^A[\vec{g}, \vec{x}] \downarrow \;\Rightarrow\; t_2^A[\vec{g}, \vec{x}] \downarrow \text{ and } t_1^A[\vec{g}, \vec{x}] = t_2^A[\vec{g}, \vec{x}]$$

Remarks 6.4

1. $f(x) \simeq g(x) \quad \Leftrightarrow \quad f(x) \sqsubseteq g(x)$ and $g(x) \sqsubseteq f(x)$.

2. $f \subseteq g$ (see Definition 3.2) iff $\forall \vec{x} \in A^u$ $(f(\vec{x}) \sqsubseteq g(\vec{x})$ and $g(\vec{x}) \sqsubseteq f(\vec{x}))$.

Theorem 6.5 (Monotonicity). In an N-standard algebra A, let
$f = t^A[\vec{g}, \cdot],\ f' = t^A[\vec{g'}, \cdot]$. If $\vec{g} \subseteq \vec{g'}$, then $f \subseteq f'$.

Proof: Structural induction on t.

(i) $t \equiv \mathsf{x}_i$

Then for any $\vec{x}$, $f(\vec{x}) = x_i = f'(\vec{x})$

and so $f \subseteq f'$

(ii) $t \equiv \mathsf{c}$

Then for any $\vec{x}$, $f(\vec{x}) = \mathsf{c} = f'(\vec{x})$

and so $f \subseteq f'$

(iii) $t \equiv \mathsf{g}_j(t_1, \ldots, t_n)$ $(j = 1, \ldots, m)$

By i.h. for any $\vec{x}$, $t_i^A[\vec{g}, \vec{x}] \sqsubseteq t_i^A[\vec{g'}, \vec{x}]($ $(i = 1, \ldots, n)$, then :

$$\begin{aligned}
t^A[\vec{g}, \vec{x}] \downarrow z \ &\Rightarrow \ g_j(t_1^A[\vec{g}, \vec{x}], \ldots, t_n^A[\vec{g}, \vec{x}]) \downarrow z \\
&\Rightarrow \ \text{for some } y_i, \ldots, y_m \in A, \\
&\qquad t_1^A[\vec{g}, \vec{x}] = y_1, \ldots, t_m^A[\vec{g}, \vec{x}] = y_m \text{ and } g_j(y_1, \ldots, y_m) \downarrow z \\
&\Rightarrow \ t_1^A[\vec{g'}, \vec{x}] = y_1, \ldots, t_m^A[\vec{g'}, \vec{x}] = y_m \text{ and } g'_j(y_1, \ldots, y_m) \downarrow z \\
&\Rightarrow \ g'_j(t_1^A[\vec{g'}, \vec{x}], \ldots, t_n^A[\vec{g'}, \vec{x}]) \downarrow z \\
&\Rightarrow \ t_i^A[\vec{g'}, \vec{x}] \downarrow z
\end{aligned}$$

So, for any $\vec{x}$, $f(\vec{x}) \downarrow z \Rightarrow f'(\vec{x}) \downarrow z$

Hence, $f \subseteq f'$

(iv) $t \equiv \mathsf{F}(t_1, \ldots, t_n)$ $(j = 1, \ldots, m)$

Similar to case (iii).

(v) $t \equiv \mathsf{if}\ \ t_1\ \ \mathsf{then}\ \ t_2\ \ \mathsf{else}\ \ t_3\ \ \mathsf{fi}$

By i.h. for any $\vec{x}$, $t_i^A[\vec{g}, \vec{x}] \sqsubseteq t_i^A[\vec{g'}, \vec{x}]$ $(i = 1, 2, 3)$, then :

$$\begin{aligned} t^A[\vec{g}, \vec{x}] \downarrow z \ &\Rightarrow\ (\, t_1^A[\vec{g}, \vec{x}] \downarrow \mathtt{tt} \text{ and } t_2^A[\vec{g}, \vec{x}] \downarrow z \,) \text{ or } (\, t_1^A[\vec{g}, \vec{x}] \downarrow \mathtt{ff} \text{ and } t_3^A[\vec{g}, \vec{x}] \downarrow z \,) \\ &\Rightarrow\ (\, t_1^A[\vec{g'}, \vec{x}] \downarrow \mathtt{tt} \text{ and } t_2^A[\vec{g'}, \vec{x}] \downarrow z \,) \text{ or } (\, t_1^A[\vec{g'}, \vec{x}] \downarrow \mathtt{ff} \text{ and } t_3^A[\vec{g'}, \vec{x}] \downarrow z \,) \\ &\Rightarrow\ t^A[\vec{g'}, \vec{x}] \downarrow z \end{aligned}$$

So, for any $\vec{x}$, $f(\vec{x}) \downarrow z \Rightarrow f'(\vec{x}) \downarrow z$

Hence, $f \subseteq f'$ $\square$

6.1 Minimal solutions of equations

Our major task in this chapter is to discuss solutions of algebraic formulae. Since equations are special cases of conditional equations, we will start by considering equations of the form:

$$\mathsf{f}(\vec{\mathsf{x}}) \simeq t[\mathsf{f}, \vec{\mathsf{g}}, \vec{\mathsf{x}}]$$

Examples 6.6.

1. $\mathsf{f}(\vec{\mathsf{x}}) \simeq \mathsf{f}(\vec{\mathsf{x}})$

   In any algebra, any partial function satisfies this equation and the completely undefined function is the minimal function.

2. $\mathsf{f}(\vec{\mathsf{x}}) \simeq \mathsf{S}(\mathsf{f}(\vec{\mathsf{x}}))$

   In the algebra $\mathcal{N}_0 = (\mathbb{N}; 0, \mathsf{S})$, only the completely undefined function satisfies this equation.

3. $\mathsf{f}(\mathtt{n}, \vec{\mathtt{x}}) \simeq \mathsf{if}\ \ \mathsf{eq}_{\mathsf{nat}}(\mathtt{n}, 0)\ \ \mathsf{then}\ \ \mathsf{g}(\vec{\mathtt{x}})\ \ \mathsf{else}\ \ \mathsf{h}(\mathtt{n}-1, \vec{\mathtt{x}}, \mathsf{f}(\mathtt{n}-1, \vec{\mathtt{x}}))\ \ \mathsf{fi}$

   The primitive recursive function defined by

$$f(0, \vec{x}) \simeq g(\vec{x})$$
$$f(\mathsf{S}(n), \vec{x}) \simeq h(n, \vec{x}, f(n, \vec{x}))$$

   is the unique solution of this equation in A.

From these examples, we can see that not all equations have unique solutions. But, in fact, we can find a *unique minimal solution* among all the solutions of the same equation, i.e. a function which is a subset of all the functions which satisfy the same equation.

Kleene [Kle52] considers the minimal solution for equations over $\mathbb{N}$ in his investigation of *recursive functionals* on $\mathbb{N}$. We will extend Kleene's approach to prove the existence of a minimal solution for any system of equations and also conditional equations in any many-sorted partial N-standard Σ-algebra.

Theorem 6.7. Let A be an N-standard Σ-algebra. Given an equation

$$\mathsf{f}(\mathsf{x}) \simeq t[\mathsf{f}, \vec{\mathsf{g}}, \vec{\mathsf{x}}] \tag{6.1}$$

in an expanded signature $\Sigma' = \Sigma\ \cup \{\mathsf{f}, \vec{\mathsf{g}}\}$, there is a minimal solution on A for this equation, i.e. there is a (unique) function f which satisfies this equation and is a subset of any function which satisfies this equation.

Proof: Let f_0 be the completely undefined function of the type of f, and then define $f_1, f_2, f_3 \ldots$ successively by:

$$f_1(\vec{x}) \simeq t^A[f_0, \vec{g}, \vec{x}]$$
$$f_2(\vec{x}) \simeq t^A[f_1, \vec{g}, \vec{x}]$$
$$f_3(\vec{x}) \simeq t^A[f_2, \vec{g}, \vec{x}]$$
$$\vdots$$

Note that $f_0 \subseteq f_1$, since f_0 is completely undefined.

By induction on i and the Monotonicity Theorem, we can then get:

$$f_i \subseteq f_{i+1} \text{ for all } i$$

Let

$$f = \bigcup_{i=0}^{\infty} f_i \tag{6.2}$$

This means that for any $\vec{x}$, if there exists some i such that $f_i(\vec{x})$ is defined, then the value of $f(\vec{x})$ is the common value of $f_j(\vec{x})$ for all $j \geq i$; $f(\vec{x}) \uparrow$ iff there is no such i. Note that for all i, $f_i \subseteq f$.

We first prove that f satisfies the equation (6.1) by proving for all $\vec{x}$,

$$f(\vec{x}) \sqsubseteq t^A[f, \vec{g}, \vec{x}] \tag{6.3}$$

and

$$t^A[f, \vec{g}, \vec{x}] \sqsubseteq f(\vec{x}) \tag{6.4}$$

First prove (6.3):

$$\begin{array}{lll} f(\vec{x}) \downarrow y & \Rightarrow \; f_{i+1}(\vec{x}) \downarrow y \text{ for some } i & (\text{Definition } of \; f(\vec{x})) \\ & \Rightarrow \; t^A[f_i, \vec{g}, \vec{x}] \downarrow y & (\text{Definition } of \; f_{i+1}) \\ & \Rightarrow \; t^A[f, \vec{g}, \vec{x}] \downarrow y & (f_i \subseteq f \text{ and Monotonicity Theorem}) \end{array}$$

In order to prove the opposite inclusion (6.4), we need the following lemma:

Lemma. For all $\vec{x} \in A^u$ and $y \in A_s$, if $t^A[f, \vec{g}, \vec{x}] \downarrow y$, then there exists $k \in \mathbb{N}$ such that $t^A[f_k, \vec{g}, \vec{x}] \downarrow y$.

Proof: Structural induction on t:

(i) $t \equiv \mathtt{x}_i$

Then for all $\vec{x}$

$$t^A[f_0, \vec{g}, \vec{x}] = t^A[f_1, \vec{g}, \vec{x}] = \cdots = t^A[f, \vec{g}, \vec{x}] = x_i$$

and so for all $k \geq 0$, $t^A[f, \vec{g}, \vec{x}] = t^A[f_k, \vec{g}, \vec{x}]$.

(ii) $t \equiv \mathsf{c}$

Then for all $\vec{x}$

$$t^A[f_0, \vec{g}, \vec{x}] = t^A[f_1, \vec{g}, \vec{x}] \cdots = t^A[f, \vec{g}, \vec{x}] = c$$

and so for all $k \geq 0$, $t^A[f, \vec{g}, \vec{x}] = t^A[f_k, \vec{g}, \vec{x}]$.

(iii) $t \equiv \mathsf{f}(t_1, \ldots, t_n)$

$t^A[f, \vec{g}, \vec{x}] =_{df} f(t_1^A[f, \vec{g}, \vec{x}], \ldots, t_m^A[f, \vec{g}, \vec{x}])$.

For any $\vec{x}$ and y, suppose $f(t_1^A[f, \vec{g}, \vec{x}], \ldots, t_m^A[f, \vec{g}, \vec{x}]) \downarrow y$,

then there exists some y_i such that $t_i^A[f, \vec{g}, \vec{x}] \downarrow y_i \, (1 \leq i \leq m)$

By i.h. $\exists k_1, \ldots, k_m$ such that

$$t_1^A[f_{k_1}, \vec{g}, \vec{x}] \downarrow y_1$$

$$\vdots$$

$$t_m^A[f_{k_m}, \vec{g}, \vec{x}] \downarrow y_m$$

By definition of f (6.2), $\exists k_{m+1}$ such that

$$f_{k_{m+1}}(y_1, \ldots, y_m) = f(y_1, \ldots, y_m) = y$$

Let $k = max(k_1, \ldots, k_{m+1})$. Then

$$\begin{aligned} f(t_1^A[f,\vec{g},\vec{x}],\ldots,t_m^A[f,\vec{g},\vec{x}]) \downarrow y &\Rightarrow f(y_1,\ldots,y_m) \downarrow y \\ &\Rightarrow f_k(y_1,\ldots,y_m) \downarrow y \\ &\Rightarrow f_k((t_1^A[f_k,\vec{g},\vec{x}],\ldots,t_m^A[f_k,\vec{g},\vec{x}]) \downarrow y \end{aligned}$$

and so $t^A[f,\vec{g},\vec{x}] \downarrow y \;\Rightarrow\; t^A[f_k,\vec{g},\vec{x}] \downarrow y$.

(iv) $t \equiv \mathsf{g}_j(t_1, \ldots, t_m)$

The proof is similar to that of case (iii) but simpler.

(v) $t \equiv \mathsf{F}(t_1, \ldots, t_m)$

The proof is also similar to that of case (iii) but simpler.

(vi) $t \equiv \mathsf{if}\ \ t_1\ \ \mathsf{then}\ \ t_2\ \ \mathsf{else}\ \ t_3\ \ \mathsf{fi}$

For any $\vec{x}$ and y, suppose $t^A[f,\vec{g},\vec{x}] \downarrow y$,

then either

$$t_1^A[f,\vec{g},\vec{x}] \downarrow \mathsf{tt} \text{ and } t_2^A[f,\vec{g},\vec{x}] \downarrow y$$

or

$$t_1^A[f,\vec{g},\vec{x}] \downarrow \mathsf{ff} \text{ and } t_3^A[f,\vec{g},\vec{x}] \downarrow y$$

By i.h. $\exists\, k_1,\ k_2,\ k_3$ such that

$$t_1^A[f_{k_1},\vec{g},\vec{x}] \downarrow \mathsf{tt} \text{ and } t_2^A[f_{k_2},\vec{g},\vec{x}] \downarrow y$$

or

$$t_1^A[f_{k_1},\vec{g},\vec{x}] \downarrow \mathsf{ff} \text{ and } t_3^A[f_{k_3},\vec{g},\vec{x}] \downarrow y$$

Let $k = max(k_1, k_2, k_3)$. Then

$$(\mathsf{if}\ \ t_1^A[f,\vec{g},\vec{x}]\ \ \mathsf{then}\ \ t_2^A[f,\vec{g},\vec{x}]\ \ \mathsf{else} t_3^A[f,\vec{g},\vec{x}])\ \ \mathsf{fi}) \downarrow y$$

$$\Rightarrow \quad (\mathsf{if}\ \ t_1^A[f_k,\vec{g},\vec{x}]\ \ \mathsf{then}\ \ t_2^A[f_k,\vec{g},\vec{x}]\ \ \mathsf{else}\ \ t_3^A[f_k,\vec{g},\vec{x}]\ \ \mathsf{fi}) \downarrow y$$

and so, $t^A[f,\vec{g},\vec{x}] \downarrow y \;\Rightarrow\; t^A[f_k,\vec{g},\vec{x}] \downarrow y$

This completes the proof of the Lemma. $\square$

Now, suppose $t^A[f,\vec{g},\vec{x}] \downarrow y$, then by this Lemma, for some k

$$\begin{aligned} t^A[f,\vec{g},\vec{x}] \downarrow y &\Rightarrow t^A[f_k,\vec{g},\vec{x}] \downarrow y \\ &\Rightarrow f_{k+1}(\vec{x}) \downarrow y \\ &\Rightarrow f(\vec{x}) \downarrow y \end{aligned}$$

So, we have proved (6.4).

We have shown that f satisfies the equation (6.1). The next step is to prove f is a subfunction of all the solutions of (6.1). So, suppose f' satisfies (6.1), we must show $f \subseteq f'$.

Since

$$f = \bigcup_{i=0}^{\infty} f_i$$

we will completed the proof by showing that

$$f_i \subseteq f' \text{ for all } i$$

by simple induction on i:

Base: $i = 0$

$$f_0 \subseteq f'$$

since f_0 is completely undefined.

Induction step: Suppose for i, $f_i \subseteq f'$, then for any $\vec{x}$ and y:

$$\begin{aligned} f_{i+1}(\vec{x}) \downarrow y &\Rightarrow t^A[f_i,\vec{g},\vec{x}] \downarrow y \quad \text{(by definition of } f_{i+1}) \\ &\Rightarrow t^A[f',\vec{g},\vec{x}] \downarrow y \quad \text{(by i.h. and Monotonicity Theorem)} \\ &\Rightarrow f'(\vec{x}) \downarrow y \end{aligned}$$

So, $f_{i+1} \subseteq f'$, and hence $f \subseteq f'$ $\square$

Remark 6.8. The minimal solution of (6.1), proved to exist by the above theorem, is computable, as we will see in Chapter 7.

Now, we look at another simple example:

Example 6.9. Consider the two equations:

$$\mathsf{f}_1(\mathtt{n}, \vec{\mathtt{x}}) \simeq \mathsf{if}\ \ \mathtt{n} = 0\ \ \mathsf{then}\ \ c_1\ \ \mathsf{else}\ \ \mathsf{h}_1(\mathtt{n} - 1, \vec{\mathtt{x}}, \mathsf{f}_1(\mathtt{n} - 1, \vec{\mathtt{x}}), \mathsf{f}_2(\mathtt{n} - 1, \vec{\mathtt{x}}))\ \ \mathsf{fi}$$

$$\mathsf{f}_2(\mathtt{n}, \vec{\mathtt{x}}) \simeq \mathsf{if}\ \ \mathtt{n} = 0\ \ \mathsf{then}\ \ c_2\ \ \mathsf{else}\ \ \mathsf{h}_2(\mathtt{n} - 1, \vec{\mathtt{x}}, \mathsf{f}_1(\mathtt{n} - 1, \vec{\mathtt{x}}), \mathsf{f}_2(\mathtt{n} - 1, \vec{\mathtt{x}}))\ \ \mathsf{fi}$$

Here, f_1 and f_2 are defined by mutual primitive recursion. So, in order to find the solutions for these two equations, we need to solve them together. In the following content, we will discuss the general case in which functions are defined by mutual recursion.

Theorem 6.10. Let A be an N-standard Σ-algebra. Given a finite set of equations:

$$\begin{array}{rcl} \mathsf{f}_1(\vec{\mathtt{x}}) & \simeq & t_1[\vec{\mathsf{f}}, \vec{\mathsf{g}}, \vec{\mathtt{x}}] \\ & \vdots & \\ \mathsf{f}_l(\vec{\mathtt{x}}) & \simeq & t_l[\vec{\mathsf{f}}, \vec{\mathsf{g}}, \vec{\mathtt{x}}] \end{array} \tag{6.5}$$

in an expanded signature $\Sigma' = \Sigma \cup \{\vec{\mathsf{f}}, \vec{\mathsf{g}}\}$ where $\vec{\mathsf{f}} \equiv \mathsf{f}_1, \ldots, \mathsf{f}_l$, there is a minimal solution for it on A, i.e., we can find a tuple of functions $\vec{f}$, which satisfies these equations and is a subset of any solution.

Proof: Let $f_{10}, f_{20}, \ldots, f_{l0}$ be completely undefined functions of type $\mathsf{f}_1, \cdots, \mathsf{f}_l$. Then define

$f_{11}, \ldots f_{l1}, f_{12}, \ldots, f_{l2} \ldots$ successively by:

$$f_{11}(\vec{x}) \simeq t_1^A[f_{10}, f_{20}, \ldots, f_{l0}, \vec{g}, \vec{x}]$$

$$\vdots$$

$$f_{l1}(\vec{x}) \simeq t_l^A[f_{10}, f_{20}, \ldots, f_{l0}, \vec{g}, \vec{x}]$$

$$\vdots$$

$$f_{12}(\vec{x}) \simeq t_1^A[f_{11}, f_{21}, \ldots, f_{l1}, \vec{g}, \vec{x}]$$

$$\vdots$$

$$f_{l2}(\vec{x}) \simeq t_l^A[f_{11}, f_{21}, \ldots, f_{l1}, \vec{g}, \vec{x}]$$

$$\vdots$$

$f_{i0} \subseteq f_{i1}$ since f_{i0} is completely undefined.

By the Monotonicity Theorem, we get: $f_{ij} \subseteq f_{i(j+1)}$ for all $j \in \mathbb{N}$ and $i = 1, \ldots, l$.

Define:

$$f_1 = \bigcup_{j=0}^{\infty} f_{1j}$$

$$\vdots$$

$$f_l = \bigcup_{j=0}^{\infty} f_{lj}$$

Just as in Theorem 6.7, we can prove that every function in the tuple $f_1, \ldots, f_l$ is the minimal solution of the corresponding equation in (6.5) in A, and thus we can conclude that this tuple is the minimal solution of (6.5) in A. $\square$

6.2 Minimal solutions of conditional equations

We have shown equations of form (6.1) have minimal solution in an N-standard Σ-algebra. What about conditional equations?

Theorem 6.11. Let A be an N-standard Σ-algebra. Given a conditional equation:

$$t_1[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \simeq t'_1[\vec{\mathsf{g}}, \vec{\mathsf{x}}], \ldots, t_n[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \simeq t'_n[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \ \rightarrow \ \mathsf{f}(\vec{\mathsf{x}}) \simeq t[\mathsf{f}, \vec{\mathsf{g}}, \vec{\mathsf{x}}] \tag{6.6}$$

where $t \in \boldsymbol{Term}(\Sigma \cup \{\vec{\mathsf{g}}, \mathsf{f}\})$, and $t_i, t'_i \in \boldsymbol{Term}(\Sigma \cup \{\vec{\mathsf{g}}\})$, there is a minimal solution f on A for it.

Proof: Let f_0 be the completely undefined function of type f, then define $f_1, f_2 \ldots$ as following:

$$f_1(x) \simeq \begin{cases} t^A[f_0, \vec{g}, \vec{x}] & \text{if} \quad t_1^A[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \simeq t_1'^A[\vec{\mathsf{g}}, \vec{\mathsf{x}}], \ldots, t_n^A[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \simeq \mathbf{t}_n'^A[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \\ \uparrow & \text{otherwise} \end{cases}$$

$$f_2(x) \simeq \begin{cases} t^A[f_1, \vec{g}, \vec{x}] & \text{if} \quad t_1^A[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \simeq t_1'^A[\vec{\mathsf{g}}, \vec{\mathsf{x}}], \ldots, t_n^A[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \simeq \mathbf{t}_n'^A[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \\ \uparrow & \text{otherwise} \end{cases}$$

$$\vdots$$

Since f does not occur in the antecedent, as in Theorem 6.7, we can derive:

$$f_i \subseteq f_{i+1} \text{ for all } i \in \mathbb{N}$$

Let

$$f \simeq \bigcup_{i=0}^{\infty} f_i$$

The rest of the proof is similar to that of Theorem 6.7. $\square$

Remarks 6.12.

1. In equation (6.6), f does not occur in the antecedent.

2. Although equation (6.6) has a minimal solution, it does not give a good model of *computability* since the equations $t_i \simeq t'_i$ $(i = 1, \ldots, n)$ in the antecedent are highly non-computable. (See Remark 5.1(1).)

3. More interesting from viewpoint of computability is the variant

$$t_1[\vec{\mathsf{g}}, \vec{\mathsf{x}}] = t'_1[\vec{\mathsf{g}}, \vec{\mathsf{x}}], \ldots, t_n[\vec{\mathsf{g}}, \vec{\mathsf{x}}] = t'_n[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \;\rightarrow\; \mathsf{f}(\vec{\mathsf{x}}) \simeq t[\mathsf{f}, \vec{\mathsf{g}}, \vec{\mathsf{x}}] \tag{6.7}$$

with $t_i \simeq t'_i$ replaced by strict equality $t_i = t'_i$ (see Remark 5.1(1)). Theorem 6.11 still holds for (6.7). However, we will see that the minimal solution of (6.7) is not, in general, computable (Remark 7.17(1)).

4. More interesting still from viewpoint of computability is the case in which the sorts s_i $(i = 1, \ldots, n)$ of all the terms in the antecedent of (6.7) are *equality sorts*, where $\mathsf{eq}^A_{s_i}$ is total equality or semi-equality (Definition 2.17), so that the conditional equation (6.7) can be replace by the equation

$$\begin{aligned} \mathsf{f}(\vec{\mathsf{x}}) \simeq \mathsf{if}\;\; & \mathsf{eq}_{s_1}(t_1[\vec{\mathsf{g}}, \vec{\mathsf{x}}], t'_1[\vec{\mathsf{g}}, \vec{\mathsf{x}}]) \;\; \mathsf{and} \cdots \mathsf{and} \;\; \mathsf{eq}_{s_n}(t_n[\vec{\mathsf{g}}, \vec{\mathsf{x}}], t'_n[\vec{\mathsf{g}}, \vec{\mathsf{x}}]) \\ & \mathsf{then}\;\; t[\mathsf{f}, \vec{\mathsf{g}}, \vec{\mathsf{x}}] \\ & \mathsf{else}\;\; \mathsf{f}(\vec{\mathsf{x}}) \\ \mathsf{fi} & \end{aligned} \tag{6.8}$$

The minimal solution of this equation *is* computable. (See Remark 6.8.)

What about the *minimal* or *unique* solutions of conditional equations in *strict equational logic*:

$$t_1[\vec{\mathsf{g}}, \vec{\mathsf{x}}] = t'_1[\vec{\mathsf{g}}, \vec{\mathsf{x}}], \ldots, t_n[\vec{\mathsf{g}}, \vec{\mathsf{x}}] = \mathbf{t}'_n[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \rightarrow \mathsf{f}(\vec{\mathsf{x}}) = t[\mathsf{f}, \vec{\mathsf{g}}, \vec{\mathsf{x}}] \;?$$

Let us look at some simple examples.

Examples 6.13.

1. $\mathsf{f}(\vec{\mathsf{x}}) = \mathsf{f}(\vec{\mathsf{x}})$

   Any total function functions in any algebra satisfies this equation and there is no minimal function among them.

2. $\mathsf{f}(\vec{\mathsf{x}}) = \mathsf{S}(\mathsf{f}(\vec{\mathsf{x}}))$

   There is no function that satisfies this in $\mathcal{N}$.

3. $\mathsf{x} = 1 \ \rightarrow\ \mathsf{f}(\vec{\mathsf{x}}) = \mathsf{S}(\mathsf{f}(\vec{\mathsf{x}}))$

   There is no function that satisfies this in $\mathcal{N}$.

4. $\mathsf{x} = 1 \ \rightarrow\ \mathsf{f}(\vec{\mathsf{x}}) = \mathsf{f}(\vec{\mathsf{x}})$

   All the functions which converge when $x = 1$ in any algebra whose carrier includes a closed term '1' satisfy this equation and there is no minimal among them.

These examples shows that conditional equations in strict equational logic do not necessarily have unique, or minimal, or indeed *any* solutions!

Chapter 7

Computability for the minimal solutions of algebraic specification

We have found in Chapter 6 that there is a minimal solution for a given finite set of equations, as well as of conditional equations in Kleene equationl logic. Further, considering that any $\boldsymbol{\mu PR}^*$ computable function can be defined as a minimal solution of a set of equations (cf. Chapter 5), what can we say about the computability of the minimal solutions of these algebraic formulae? In this chapter, we will discuss computability (1) in terms of a simple *imperative programming* model: a *recursive programming* language $\boldsymbol{Rec}(\Sigma)$ whose programs are constructed from *assignments*, *procedure calls* (possibly recursive), *sequential composition* and the *conditional*; and also (2) in terms of the *schematic* model $\boldsymbol{\mu PR}^*$.

7.1 The recursive programming language *Rec*

We define four syntactic classes for $\boldsymbol{Rec}(\Sigma)$: *variables*, *terms*, *statements* and *procedures*.

1. $\boldsymbol{Var}(\Sigma)$ is the class of Σ-*variables* $\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}\ldots$

   For each $s \in \boldsymbol{Sort}(\Sigma)$, we write $\mathtt{x}\!:s$ to indicate $\mathtt{x}$ is a variable of sort s, and for a product type $u = s_1 \times \cdots \times s_n$, write $\vec{\mathtt{x}}\!:u$ to mean $\vec{\mathtt{x}}$ is a n-tuple of distinct variables of sorts $s_1, \ldots, s_n$.

2. $\boldsymbol{Term}(\Sigma)$ is the class of Σ-*terms* $t, \ldots$ (defined as in §2.2).

   For each $s \in \boldsymbol{Sort}(\Sigma)$, we use $t : s$ to mean t is a term in sort s, and for a product type $u = s_1 \times \cdots \times s_n$, $\vec{t}\!:u$ means $\vec{t}$ is a n-tuple of terms of sorts $s_1, \ldots, s_n$.

3. $\boldsymbol{Stmt}(\Sigma)$ is the class of *statements* S,...

   They are generated by the rules:

   $$S ::= \mathsf{skip}|\ \mathtt{x}\ :=\ t|\ \mathtt{x}\ := P(\vec{t})|\ S_1\ ;\ S_2|\ \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}$$

   where $\mathtt{x} := P(\vec{t})$ is a procedure call which calls a procedure with parameters $\vec{t}$ by its name P_i and returns its value to $\mathtt{x}$. This procedure call can be recursive. In fact, we can regard procedure calls as a sort of assignment. Atomic statements include 'skip' and the assignments $\mathtt{x}\ :=\ t$, $\mathtt{x}\ := P(\vec{t})$. The two sides of an assignment must have the same sort.

4. $\boldsymbol{Proc}(\Sigma)$ is the class of *procedures* $E, \ldots$ which have the form:

$$
\begin{aligned}
E \;\equiv\; & \mathsf{proc} \;\; P_1 \Leftarrow E_1, \ldots, P_n \Leftarrow E_n \\
& \quad \mathsf{in} \;\; \vec{\mathtt{x}} : u \\
& \quad \mathsf{out} \;\; \mathtt{y} : s \\
& \quad \mathsf{aux} \; \vec{\mathtt{z}} : w \\
& \mathsf{begin} \\
& \quad S \\
& \mathsf{end}
\end{aligned}
$$

where for $i = 1, \ldots, n$ $(n \geq 0)$, E_i is a *procedure* with *name* P_i and $\vec{\mathtt{x}}$, $\mathtt{y}$, $\vec{\mathtt{z}}$ are *input* variables, *output* variable and *auxiliary* variables respectively. We say that the procedure is of type $u \to s$. Note that all the procedure names occurring in S must either be declared by $P_i \Leftarrow E_i$, or be the name of E, corresponding to a *recursive call*.

Remark 7.1 (Semantics of $\boldsymbol{Rec}(\Sigma)$). We do not give formal semantics for $\boldsymbol{Rec}(\Sigma)$, since that would be a major undertaking which would take us too far from the main focus of this thesis. Informal semantics of $\boldsymbol{Rec}(\Sigma)$ is enough for our purpose. Formal semantics (operational and denotational) for a recursive programming language without parameters on abstract data types have been given in [TZ88]. Formal semantics for a language like $\boldsymbol{Rec}(\Sigma)$ is one of the topics under investigation in [Xu03].

Definition 7.2 ($\boldsymbol{Rec}(\Sigma)$-computable functions). A function f is $\boldsymbol{Rec}(\Sigma)$- computable on A if it is computable by a $\boldsymbol{Rec}(\Sigma)$-procedure on A. Let $\boldsymbol{Rec}(A)$ denotes the class of $\boldsymbol{Rec}(\Sigma)$-computable functions on A.

Let $\vec{\mathtt{g}} \equiv \mathtt{g}_1, \ldots, \mathtt{g}_m$ be a tuple of function symbols of given functions $\vec{g}$ ("oracles"). An *oracle* statement in $\mathtt{g}_i$ has the form:

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

where $\mathtt{x} : s$, $\vec{t} : u$ and $\mathtt{g}_i : u \to s$. Then, we can expand the recursive language $\boldsymbol{Rec}(\Sigma)$ to the *relative recursive* language $\boldsymbol{Rec}(\Sigma,\, \vec{\mathtt{g}})$ by including *oracle* statements in $\vec{\mathtt{g}}$.

Thus the notion of $\boldsymbol{Rec}(\Sigma)$-computable can be relativized to obtain the notion $\boldsymbol{Rec}(\Sigma)$-*computable in* $\vec{g}$.

Definition 7.3 (Relative Rec-computable functions). A function f is $\boldsymbol{Rec}(\Sigma)$-computable in $\vec{g}$ on A if it can be computed by a $\boldsymbol{Rec}(\Sigma,\, \vec{\mathtt{g}})$-procedure on A with $\vec{g} = \vec{\mathtt{g}}^A$. Let $\boldsymbol{Rec}(A, \vec{g})$ denotes the class of functions which are $\boldsymbol{Rec}(\Sigma)$-computable in $\vec{g}$ on A.

Lemma 7.4 (Transitivity of relative *Rec* computability). If a function f is $\boldsymbol{Rec}(\Sigma)$-computable in $\vec{g}$ on A and $\vec{g}$ are $\boldsymbol{Rec}(\Sigma)$-computable on A, then f is $\boldsymbol{Rec}(\Sigma)$-computable on A. More generally, if f is $\boldsymbol{Rec}(\Sigma)$-computable in $\vec{g}$ on A and $\vec{g}$ are $\boldsymbol{Rec}(\Sigma)$-computable in $\vec{h}$ on A, then f is $\boldsymbol{Rec}(\Sigma)$-computable in $\vec{h}$ on A.

Proof: Since $g_i \in \boldsymbol{Rec}(A, \vec{h})$ and $f \in \boldsymbol{Rec}(A, \vec{g})$, we can construct relative $\boldsymbol{Rec}(\Sigma)$-procedures P_{g_i} and P_f to compute them. Then we can replace the statement $\mathtt{x} := \mathtt{g}_i(\vec{t})$ in P_f with the procedure call $\mathtt{x} := P_{g_i}(\vec{t})$ and produce a new $\boldsymbol{Rec}(\Sigma)$-procedure for f relative to $\vec{h}$. $\square$

Definition 7.5. A $\boldsymbol{Rec}^*(\Sigma)$-procedure is a $\boldsymbol{Rec}(\Sigma^*)$-procedure in which the *input* and *output* variables have sorts in Σ, while the *auxiliary* variables may have starred sorts.

Definition 7.6 (*Rec-computable functions).** A function f is $\boldsymbol{Rec}^*(\Sigma)$-computable on A if it is computable by a $\boldsymbol{Rec}^*(\Sigma)$-procedure on A. Let $\boldsymbol{Rec}^*(A)$ denotes the class of $\boldsymbol{Rec}^*(\Sigma)$-computable functions on A.

We can expand $\boldsymbol{Rec}^*(\Sigma)$ to $\boldsymbol{Rec}^*(\Sigma, \vec{\mathsf{g}})$ by including oracle statements $\mathtt{x} := \mathsf{g}_i(\vec{t})$ and relativize the notion of *Rec**-computability to *Rec**-computability in $\vec{g}$ as well.

7.2 $\boldsymbol{Rec}(\Sigma)$-computability of minimal solutions of equations and conditional equations

Theorem 7.7. Let A be an N-standard Σ-algebra. For an equation

$$\mathsf{f}(\vec{\mathsf{x}}) \simeq t[\mathsf{f}, \vec{\mathsf{g}}, \vec{\mathsf{x}}] \tag{7.1}$$

in an expanded signature $\Sigma' = \Sigma \cup \{\mathsf{f}, \vec{\mathsf{g}}\}$, the minimal solution f of it is $\boldsymbol{Rec}(\Sigma)$-computable in $\vec{g}$ on A, i.e. $f \in \boldsymbol{Rec}(A, \vec{g})$. Hence if $\vec{g} \in \boldsymbol{Rec}(A)$, then $f \in \boldsymbol{Rec}(A)$.

Proof: We can use the following $\boldsymbol{Rec}(\Sigma, \vec{\mathsf{g}})$ procedure to compute the minimal solution f:

$$
\begin{array}{rl}
E_f \equiv \mathsf{proc} & P_t \Leftarrow E_t \\
 & \mathsf{in} \;\; \vec{\mathsf{x}} : u \\
 & \mathsf{out} \;\; \mathsf{y} : v \\
\mathsf{begin} & \\
 & \mathsf{y} := P_t(\vec{\mathsf{x}}) \\
\mathsf{end} &
\end{array}
$$

where E_t is the procedure for $t^A[\mathsf{f}, \vec{\mathsf{g}}, \vec{\mathsf{x}}]$ (recall Notation 6.1(b).) and is defined by structural induction on t:

(i) $t \equiv \mathsf{x}_i$

$$
\begin{aligned}
E_t \equiv \mathsf{proc}\quad & \mathsf{in}\ \ \vec{\mathsf{x}} : u \\
& \mathsf{out}\ \ \mathsf{y} : s \\
\mathsf{begin}\quad & \\
& \mathsf{y} := \mathsf{x}_i \\
\mathsf{end}\quad &
\end{aligned}
$$

(ii) $t \equiv \mathsf{c}$

$$
\begin{aligned}
E_t \equiv \mathsf{proc}\quad & \mathsf{in}\ \ \vec{\mathsf{x}} : u \\
& \mathsf{out}\ \ \mathsf{y} : s \\
\mathsf{begin}\quad & \\
& \mathsf{y} := \mathsf{c} \\
\mathsf{end}\quad &
\end{aligned}
$$

(iii) $t \equiv \mathsf{F}_i(t_1, \ldots, t_m)$, $(1 \leq i \leq m)$

By i.h. $t_1^A, \ldots, t_m^A$ can be computed with ***Rec***$(\Sigma,\ \vec{\mathsf{g}})$-procedures: $E_{t_1}, \ldots, E_{t_m}$. Then, we can define :

$$
\begin{aligned}
E_t \equiv \mathsf{proc}\quad & P_{t_1} \Leftarrow E_{t_1}, \ldots, P_{t_m} \Leftarrow E_{t_m} \\
& \mathsf{in}\ \ \vec{\mathsf{x}} : u \\
& \mathsf{out}\ \ \mathsf{y} : s \\
& \mathsf{aux}\ \ \vec{\mathsf{z}} : w \\
\mathsf{begin}\quad & \\
& \mathsf{z}_1 := P_{t_1}(\vec{\mathsf{x}}) \\
& \vdots \\
& \mathsf{z}_m := P_{t_m}(\vec{\mathsf{x}}) \\
& \mathsf{y} := \mathsf{F}(\mathsf{z}_1, \ldots, \mathsf{z}_m) \\
\mathsf{end}\quad &
\end{aligned}
$$

(iv) $t \equiv \mathsf{g}_i(t_1, \ldots, t_m)$, $(1 \le i \le m)$

By i.h. $t_1^A, \ldots, t_m^A$ can be computed with $\boldsymbol{Rec}(\Sigma,\ \vec{\mathsf{g}})$-procedures: $E_{t_1}, \ldots, E_{t_m}$. Then, we can define :

$$
\begin{aligned}
E_t \equiv \mathsf{proc}\quad & P_{t_1} \Leftarrow E_{t_1}, \ldots, P_{t_m} \Leftarrow E_{t_m} \\
& \mathsf{in}\ \ \vec{\mathsf{x}} : u \\
& \mathsf{out}\ \ \mathsf{y} : s \\
& \mathsf{aux}\ \ \vec{\mathsf{z}} : w \\
\mathsf{begin}\quad & \\
& \mathsf{z}_1 := P_{t_1}(\vec{\mathsf{x}}) \\
& \vdots \\
& \mathsf{z}_m := P_{t_m}(\vec{\mathsf{x}}) \\
& \mathsf{y} := \mathsf{g}_i(\mathsf{z}_1, \ldots, \mathsf{z}_m) \\
\mathsf{end}\quad &
\end{aligned}
$$

(v) $t \equiv \mathsf{if}\ \ t_1\ \ \mathsf{then}\ \ t_2\ \ \mathsf{else}\ \ t_3\ \ \mathsf{fi}$

By i.h. t_1^A, t_2^A, t_3^A can be computed with $\boldsymbol{Rec}(\Sigma,\ \vec{\mathsf{g}})$-procedures:

$E_{t_1},\ E_{t_2},\ E_{t_3}$ respectively. Then, we can define:

$E_t \equiv \mathsf{proc}\quad P_{t_1} \Leftarrow E_{t_1}, \ldots, P_{t_2} \Leftarrow E_{t_2}, P_{t_3} \Leftarrow E_{t_3}$

$\mathsf{in}\ \ \vec{\mathsf{x}} : u$

$\mathsf{out}\ \ \mathsf{y} : s$

$\mathsf{aux}\ \ \mathsf{z} : w$

begin

$\mathsf{z}_1 := P_{t_1}(\vec{\mathsf{x}})$

if z_1

then

$\mathsf{y} := P_{t_2}(\vec{\mathsf{x}})$

else

$\mathsf{y} := P_{t_3}(\vec{\mathsf{x}})$

fi

end

(vi) $t \equiv \mathsf{f}(t_1, \ldots, t_m)$

By i.h. $t_1^A, \ldots, t_m^A$ can be computed with $\boldsymbol{Rec}(\Sigma,\ \vec{\mathsf{g}})$-procedures: $E_{t_1}, \ldots, E_{t_m}$. Then, we can define:

$$
\begin{aligned}
E_t \equiv \mathsf{proc}\quad & P_{t_1} \Leftarrow E_{t_1}, \ldots, P_{t_m} \Leftarrow E_{t_m}, P_f \Leftarrow E_f \\
& \mathsf{in}\ \ \vec{\mathsf{x}} : u \\
& \mathsf{out}\ \ \mathsf{y} : s \\
& \mathsf{aux}\ \ \vec{\mathsf{z}} : w \\
\mathsf{begin}\quad & \\
& \mathsf{z}_1 := P_{t_1}(\vec{\mathsf{x}}) \\
& \vdots \\
& \mathsf{z}_m := P_{t_m}(\vec{\mathsf{x}}) \\
& \mathsf{y} := P_f(\mathsf{z}_1, \ldots, \mathsf{z}_m) \qquad\qquad (*\ \text{Note the recursive call !}\ *) \\
\mathsf{end}\quad &
\end{aligned}
$$

So, we have shown that the minimal solution of (7.1) f can be computed with a $\boldsymbol{Rec}(\Sigma,\ \vec{\mathsf{g}})$ procedure. Hence according to Lemma 7.4, if $\vec{g}$ is $\boldsymbol{Rec}(\Sigma)$-computable on A, so is f. □

We have defined the procedures for $t^A[\mathsf{f}, \vec{\mathsf{g}}, \vec{\mathsf{x}}]$ by structural induction on t. The function f is computed by calling these procedures. The case (vi) is the most interesting case with the recursive call. If the minimal solution f is undefined at $\vec{x}$, then the procedure will never halt; otherwise, the procedure will return the value of $f(\vec{x})$.

Corollary 7.8. Let A be an N-standard Σ-algebra. For a finite set of equations:

$$
\begin{gathered}
\mathsf{f}_1(\vec{\mathsf{x}}) \simeq t_1[\vec{\mathsf{f}}, \vec{\mathsf{g}}, \vec{\mathsf{x}}] \\
\vdots \\
\mathsf{f}_k(\vec{\mathsf{x}}) \simeq t_k[\vec{\mathsf{f}}, \vec{\mathsf{g}}, \vec{\mathsf{x}}]
\end{gathered}
\tag{7.2}
$$

in an expanded signature $\Sigma' = \Sigma \cup \{\mathsf{f}, \vec{\mathsf{g}}\}$, its minimal solution $\vec{f}$ is $\boldsymbol{Rec}(\Sigma)$-computable in $\vec{g}$. Hence if $\vec{g}$ are $\boldsymbol{Rec}$-computable, so are $\vec{f}$.

Proof: The proof is similar to that of Theorem 7.7 □.

7.3 *ED*-computability

The above results give us another model of computability.

Definitions 7.9 (Equational definability). $\boldsymbol{ED}(\Sigma, \vec{\mathsf{g}})$ consists of all finite sets of equations of the form (7.2).

Definition 7.10 (*ED-computability*).

(a) A function f on A is $\boldsymbol{ED}(A, \vec{g})$-*computable* if it is one of the tuple of minimal solutions of equations (7.2) in $\boldsymbol{ED}(\Sigma, \vec{\mathsf{g}})$ on A, where $\vec{\mathsf{g}}$ is interpreted as $\vec{g}$.

(b) $\boldsymbol{ED}(A)$-*computability* is the special case of (a) without any auxiliary functions.

Definition 7.11 (*ED*-computability*). A function f on A is $\boldsymbol{ED}^*(A)$-*computable* if $f \in \boldsymbol{ED}(A^*)$.

Proposition 7.12 (Transitivity of *ED*-computability).

$$f \in \boldsymbol{ED}(A, \vec{g}),\ \vec{g} \in \boldsymbol{ED}(A) \ \Rightarrow\ f \in \boldsymbol{ED}(A)$$

Proof: We combine the systems of equations for f and for $\vec{g}$ into a single system, and we have the fact that simultaneous least fixed points are equal to iterated least fixed points [dB80, Theorem 5.14].

Remark 7.13. By Theorem 7.7 (or Corollary 7.8),

$$\boldsymbol{ED}(A) \subseteq \boldsymbol{Rec}(A)$$

$$\boldsymbol{ED}^*(A) \subseteq \boldsymbol{Rec}^*(A)$$

Remark 7.14. The minimal solutions of conditional equations of the form

$$t_1[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \simeq t'_1[\vec{\mathsf{g}}, \vec{\mathsf{x}}], \ldots, t_n[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \simeq t'_n[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \rightarrow \mathsf{f}(\vec{\mathsf{x}}) \simeq t[\mathsf{f}, \vec{\mathsf{g}}, \vec{\mathsf{x}}] \tag{7.3}$$

is not computable since '$\simeq$' is not testable. (See Remark 6.12(2).)

For the variant of (7.3) with strict equality in the antecedent:

$$t_1[\vec{\mathsf{g}}, \vec{\mathsf{x}}] = t'_1[\vec{\mathsf{g}}, \vec{\mathsf{x}}], \ldots, t_n[\vec{\mathsf{g}}, \vec{\mathsf{x}}] = t'_n[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \ \rightarrow \ \mathsf{f}(\vec{\mathsf{x}}) \simeq t[\mathsf{f}, \vec{\mathsf{g}}, \vec{\mathsf{x}}]$$

the minimal solution is also not, in general, computable except for the case in which all the terms in the antecedent are of equality sorts and the equality operations on them are total equality or semi-equality. (See Remark 7.17 below)

Definition 7.15 (Conditional equational definability). $\boldsymbol{CED}(\Sigma, \vec{\mathsf{g}})$ consists of all finite sets of conditional equations

$$\begin{aligned} t_{11}[\vec{\mathsf{g}}, \vec{\mathsf{x}}] = t'_{11}[\vec{\mathsf{g}}, \vec{\mathsf{x}}], \ldots, t_{1n}[\vec{\mathsf{g}}, \vec{\mathsf{x}}] = t'_{1n}[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \ &\rightarrow \ \mathsf{f}_1(\vec{\mathsf{x}}) \simeq t_1[\vec{\mathsf{f}}, \vec{\mathsf{g}}, \vec{\mathsf{x}}] \\ &\vdots \\ t_{k1}[\vec{\mathsf{g}}, \vec{\mathsf{x}}] = t'_{k1}[\vec{\mathsf{g}}, \vec{\mathsf{x}}], \ldots, t_{kn}[\vec{\mathsf{g}}, \vec{\mathsf{x}}] = t'_{kn}[\vec{\mathsf{g}}, \vec{\mathsf{x}}] \ &\rightarrow \ \mathsf{f}_k(\vec{\mathsf{x}}) \simeq t_k[\vec{\mathsf{f}}, \vec{\mathsf{g}}, \vec{\mathsf{x}}] \end{aligned} \tag{7.4}$$

over an N-standard signature Σ.

Definition 7.16 ($\boldsymbol{CED}(A, \vec{g})$-definability). A function f is $\boldsymbol{CED}(A, \vec{g})$-*definable* if it is one of the tuple of minimal solutions of conditional equations (7.4) in $\boldsymbol{CED}(\Sigma, \vec{\mathsf{g}})$.

We can then define $\boldsymbol{CED}(A)$ and $\boldsymbol{CED}^*(A)$ similarly to Definitions 7.10, 7.11.

Remark 7.17 ($\boldsymbol{CED}$ definability $\not\Rightarrow$ Computability).

1. ***CED*** definability does not imply computability (in general), as the following counterexample shows.

   Given the same algebra A^N as in Remark 5.15, for the conditional equation

   $$\{\mathsf{x} = \mathsf{y} \;\rightarrow\; \mathsf{f}(\mathsf{x},\mathsf{y}) \simeq \mathsf{true}\}$$

   the minimal solution is the partial equality function:

   $$f : A^2 \;\rightarrow\; B$$

   where

   $$f(x,y) \simeq \begin{cases} \mathtt{tt} & \text{if } \; x = y \\ \uparrow & \text{otherwise} \end{cases}$$

   This is not computable on A^N (since A^N has no equality operation).

2. However, if the sorts s_{ij} $(i = 1, \ldots, k, \; j = 1, \ldots, n)$ of terms t_{ij} are equality sorts and $\mathsf{eq}^A_{s_{ij}}$ are total equality or semi-equality, then we can transform (7.4) to a set of equations (see Remark 6.12(4)), the minimal solution of which *is* computable, by Theorem 7.7.

3. In any case, it follows from (2) above that ***CED*** definability on A *does* imply computability on A expanded by total equality or semi-equality at all sorts.

7.4 Computability in schemes for minimal solution

In the preceding sections, we have discussed computability by means of an imperative programming language. Now, we are interested in another model of computability: $\boldsymbol{\mu PR}^*$ *schemes*.

Taking the results

$$\boldsymbol{Rec}^*(A) \subseteq \boldsymbol{While}^*(A) \qquad \text{from [Xu03]}^1$$

and

$$\boldsymbol{While}^*(A) \subseteq \boldsymbol{\mu PR}^*(A) \qquad \text{from [TZ00]}$$

and by Theorem 7.7 (or Corollary 7.8), we get

$$\boldsymbol{ED}^*(A) \subseteq \boldsymbol{Rec}^*(A) \subseteq \boldsymbol{While}^*(A) \subseteq \boldsymbol{\mu PR}^*(A).$$

Conversely, by Theorem 6.10, for an N-standard Σ-algebra A, any $\boldsymbol{\mu PR}^*(\Sigma)$ computable function can be defined as a minimal solution of a set of equations of the form (7.2) over A, i.e.

$$\boldsymbol{\mu PR}^*(A) \subseteq \boldsymbol{ED}^*(A).$$

Therefore, we close the circle to get the equivalence of *equational*, *schematic* and *imperative* models:

$$\boldsymbol{\mu PR}^*(A) \subseteq \boldsymbol{ED}^*(A) \subseteq \boldsymbol{Rec}^*(A) \subseteq \boldsymbol{While}^*(A) \subseteq \boldsymbol{\mu PR}^*(A).$$

Thus, we have the main result of this thesis:

Theorem 7.18.

$$\boxed{\boldsymbol{ED}^*(A) = \boldsymbol{\mu PR}^*(A) = \boldsymbol{Rec}^*(A) = \boldsymbol{While}^*(A)}$$

[1]This is actually proved for total algebras in [Xu03], but the extension to partial algebras should be routine.

This gives further confirmation to the generalized Church-Turing Thesis (as discussed in Chapter 1).

From Remark 7.17, on the other hand, we have:

Theorem 7.19.

(a) $\quad \boldsymbol{CED}^*(A) \supsetneqq \boldsymbol{ED}^*(A) = \boldsymbol{\mu PR}^*(A)$

(b) But $\quad \boldsymbol{CED}^*(A) \subseteq \boldsymbol{ED}^*(A^{\mathsf{eq}}) = \boldsymbol{\mu PR}^*(A^{\mathsf{eq}})$

where A^{eq} is an expansion of A by adding total equalities or semi-equalities on all sorts.

We conclude with a conjecture:

Conjecture 7.20. When the equality operations at all sorts are semi-equality, "$\subseteq$" can be replaced by "$=$" in Theorem 7.19(b).

The resolution of this conjecture is an interesting open problem.

Bibliography

[**dB80**] J.W. de Bakker. *Mathematical Theory of Program Correctness*, Prentic Hall, 1980.

[**BL74**] W.S. Brainerd and L.H. Landweber. *Theory of Computation*, John Wiley, 1974

[**EM85**] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1, Equational and Initial Semantics*, EATCS Monographs vol.6, Springer-Verlag, 1985.

[**Far90**] W.M. Farmer. A Partial Functions Version of Church's Simple Theory of Types, *Journal of Symbolic Logic* 55:1269–1291, 1990.

[**Fef95**] S. Feferman. Definedness, *Erkenntnis* 43:295–320, 1995.

[**Fef96**] S. Feferman. A new approach to abstract data types: The extensional approach, with an application to streams, *Annals of Pure and Applied Logic* 81: 75–113, 1996.

[**GH78**] J.V. Guttag and J.J. Horning. The Algebraic Specification of Abstract Data Types, *Acta Informatica* 10:27-52, Springer-Verlag, 1978.

[**GTW77**] J.A. Goguen, J.W. Thatcher and E.G. Wagner. Initial Algera Semantics and Continuous Algebras, *Journal of the Association for Computing Machinery* 24:68–95, 1977.

[**GTW78**] J.A. Goguen, J.W. Thatcher and E.G. Wagner. An initial approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology, vol. 4: Data Structuring*, ed. R.T. Yeh, pp. 80-149, Prentice Hall, 1978.

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

[**Kna28**] B. Knaster. Un théorème sur les fonctions d'ensembles. *Annales de la Societeé Polonaise de Mathematique*, 6:133-134, 1928.

[**Mos84**] Y.N. Moschovakis. Abstract recursion as a foundation for the theory of recursive algorithms. In *Computation and Proof Theory*, Lecture Notes in Mathematics 1104, pp.289-364, Springer-Verlag, 1984.

[**Mos89**] Y.N Moschovakis. The formal language of recursion, *Journal of Symbolic Logic* 54:1216-1252, 1989.

[**MR67**] A.R. Meyer and D.M. Ritchie. The Complexity of Loop Programs. In *Proceedings of the 22nd National Conference, Association for Computing Machinery*, pp.465–469, Thompson Book Company, 1967.

[**MR75**] Y.Matiyasevich and J.Robinson. Reduction of an arbitrary Diophantine equation to one in 13 unknowns, *Acta Arithmetica* 27:521-553, 1975.

[**MSHT80**] J. Moldestad, V. Stoltenberg-Hansen and J.V. Tucker. Finite algorithmic procedures and inductive definability, *Mathematica Scandinavica* 46:62–76, 1980.

[**MT81**] J. Moldestad and J.V. Tucker. On the classification of computable functions in an abstract setting. Unpublished manuscript, 1981.

[**Pla66**] R.A. Platek. *Foundations of recursion theory*. PhD thesis, Department of Mathematics, Stanford University, 1966.

[**Par93**] D.L.Parnas. Predicate Logic For Software Engineering, *IEEE Transactions on Software Engineering* 19:856–862, 1993

[**Sto77**] Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT press, 1977

[**Tar55**] A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific journal of Mathematics*, 5:285-309, 1955.

[**TZ88**] J.V.Tucker and J.I. Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics*, CWI Monograph 6, North Holland, 1988.

[**TZ99**] J.V. Tucker and J.I. Zucker. Computation by 'While' programs on topological partial algebras, *Theoretcal Computer Science* 219:379-420, 1999.

[**TZ00**] J.V. Tucker and J.I. Zucker. Computable Functions and Semicomputable Sets on Many Sorted Algebras. In *Handbook of Logic in Computer Science* vol. 5, ed. S. Abramsky, D. Gabbay, and T. Maibaum, pp. 317-523, Oxford University Press, 2000.

[**TZ02**] J.V. Tucker and J.I. Zucker. Abstract Computability and Algebraic Specification, *ACM Transactions on Computational Logic* 3:279–333, 2002.

[**TZ03**] J.V. Tucker and J.I. Zucker. Abstract Versus Concrete Computation on Metric Partial Algebras, to appear in *ACM Transactions on Computational Logic*, 2003.

[**Xu03**] Jian Xu, *Models of Computation on Abstract Data Types Based on Recursive Schemes*. Master's Thesis (in preparation), Dept. of Computing and Software, McMaster University, 2003.