Models of Computation on Abstract Data Types based on Recursive Schemes

By
Jian Xu, 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: Models of Computation on Abstract Data Types based on Recursive Schemes

AUTHOR: Jian Xu, B.Sc. (McMaster University, Canada)

SUPERVISOR: Dr. Jeffery I. Zucker

NUMBER OF PAGES: vi, 80

Abstract

This thesis compares two scheme-based models of computation on abstract many-sorted algebras A: Feferman's system $\boldsymbol{ACP}(A)$ of "abstract computational procedures" based on a least fixed point operator, and Tucker and Zucker's system $\mu\boldsymbol{PR}(A)$ based on primitive recursion on the naturals together with a least number operator. We prove a conjecture of Feferman that (assuming A contains sorts for natural numbers and arrays of data) the two systems are equivalent. The main step in the proof is showing the equivalence of both systems to a system $\boldsymbol{Rec}(A)$ of computation by an imperative programming language with recursive calls. The result provides a confirmation for a Generalized Church-Turing Thesis for computation on abstract data types.

Acknowledgements

I would first like to express my sincere thanks and appreciation to my supervisor, Dr. J. I. Zucker, for his insight, thoughtful guidance and constant encouragement throughout my study.

I am grateful to the members of my Examination Committee: Dr. J. Carette, Dr. W. M. Farmer and Dr. W. Kahl, for their careful review and valuable comments. Thanks to all tmy other professors for their help in these two years.

Thanks to friends in ITC 206 for the time we shared together. Last but not least, many thanks to my parents for their support and my wife, Yan Li, for her love and encouragement.

Contents

Chapter 1

Introduction

Schemes for recursive definitions of functions form an important component of computability theory. Their theory is fully developed over the natural numbers $\mathbb{N}$. A well known recursive definition scheme is Kleene's schemes [Kle52] for general recursive functions on $\mathbb{N}$ based on the primitive recursion schemes of Dedekind and Gödel, and the least number operator of Kleene. Another group of schemes [MSHT80, Mos84, Mos89, Fef77, Fef92a, Fef92b, Fef96] employs the concept of least fixed points. In such schemes, functions are defined as the least fixed points of second-order functionals.

Recent research concerns not only the computability of functions on $\mathbb{N}$, but also that of functions on arbitrary structures, modelled as many-sorted algebras. A many-sorted algebra A consists of a finite family of non-empty sets $A_{s_1}, \ldots, A_{s_n}$ called the *carriers* of the algebra; and a finite family of *functions* on these sets with types like

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

We are interested in *N-standard partial algebras* whose carriers include the set $\mathbb{B}$ of booleans and the set $\mathbb{N}$ of naturals, and whose functions include the standard operations on these carriers.

Recursion schemes are also generalized to work over many-sorted algebras. A generalization of Kleene's scheme is Tucker and Zucker's $\mu\boldsymbol{PR}$ scheme, which generates functions by starting from some basic functions and applying to these *composition*, *simultaneous primitive recursion* on $\mathbb{N}$ and the *least number operator*. Feferman's *abstract computation procedures* ($\boldsymbol{ACP}$) for functionals of type level 2 over abstract algebras, characterized by using the LFP (*least fixed point*) scheme, is developed in [Fef96]. A natural question is the following.

$$\begin{tikzcd}
\boldsymbol{ACP}^{*1}(A) \arrow[r] & \boldsymbol{Rec}^{*}(A) \arrow[d] \\
\mu\boldsymbol{PR}^{*}(A) \arrow[u] & \boldsymbol{While}^{*}(A) \arrow[l]
\end{tikzcd}$$

Figure 1.1: Implication cycle

> *What is the relation between the sets of functions defined by these two schemes?*

In order to compare the schemes $\mu\boldsymbol{PR}$ and $\boldsymbol{ACP}$, since $\boldsymbol{ACP}$ (unlike $\mu\boldsymbol{PR}$) deals with functionals of type level 2, we first make some definitions.

A function on A is $\mu\boldsymbol{PR}^*(A)$ computable if it is defined by a $\mu\boldsymbol{PR}$ scheme over A^*, which expands A by including new starred (array) sorts s^* for each sort s of Σ as well as standard array operations. Similarly, $\boldsymbol{ACP}^{*1}(A)$ is the set of $\boldsymbol{ACP}^*$ computable functions (type level ≤ 1) on A.

The above question can now be re-stated more precisely:

> *For any abstract many-sorted algebra A, is $\mu\boldsymbol{PR}^*(A) \;=\; \boldsymbol{ACP}^{*1}(A)$?*

S. Feferman raised this question in [Fef96] and conjectured that the answer is “Yes”.

Inspired by the denotational (or “fixed point”) semantics of recursive procedures in [Sto77, dB80], we prove the following circle of inclusions:

Rec is an imperative language employed to simulate the least fixed points of second-order functionals by properly chosen recursive procedure calls. ***Rec**** is the extension of ***Rec*** with arrays. $\boldsymbol{Rec}^*(A)$ is the set of ***Rec**** computable functions on A. Similarly, $\boldsymbol{While}^*(A)$ is the set of ***While**** computable functions on A, where ***While*** is another imperative programming language characterized by the ‘while’ construct. (Precise definitions are given in Chapter 4.)

The equivalence between $\boldsymbol{While}^*(A)$ and $\mu\boldsymbol{PR}^*(A)$ was proved in [TZ88]. We need to prove the following relations.

$$\mu\boldsymbol{PR}^*(A) \subseteq \boldsymbol{ACP}^{*1}(A) \tag{1.1}$$

$$\boldsymbol{ACP}^{*1}(A) \subseteq \boldsymbol{Rec}^*(A) \tag{1.2}$$

$$\boldsymbol{Rec}^*(A) \subseteq \boldsymbol{While}^*(A) \tag{1.3}$$

Of the above three inclusions, (1.1) is quite straightforward, and (1.3) can be derived from the semantic investigation of ***While*** programs in [TZ00]. The really interesting new result is (1.2), which forms the core of the thesis (Chapter 6).

In the proof of (1.2), even if we are considering functions of type level ≤ 1, we nevertheless have to deal with functionals of type level 2, since functions are defined as the least fixed points of functionals of type level 2. To simulate functionals of type level 2, we therefore develop a second-order version of ***Rec***, namely $\boldsymbol{Rec}_2$, and prove that

$$\boldsymbol{ACP}(A) \subseteq \boldsymbol{Rec}_2(A)$$

for functionals of type level ≤ 2. Then (1.2) follows as a corollary.

We should point out that we have modified Feferman's schemes by replacing his *simple* LFP scheme by a *simultaneous* LFP scheme. However this seems a very reasonable modification of Feferman's system.

Our proof gives further confirmation to the *Generalized Church-Turing Thesis* [TZ88, TZ00], which states that the class of functions computable by finite deterministic algorithms on A are precisely $\mu\boldsymbol{PR}^*(A)$ (or equivalently $\boldsymbol{While}^*(A)$).

The thesis is organized as follows. In Chapter 2, we introduce the basic concepts of abstract many-sorted algebras that we will need. In particular, we will define the first-order many-sorted algebras with booleans and natural numbers, possibly extended by auxiliary array structures. We will also investigate second-order version of these algebras. In Chapter 3, we define the two computational models based on recursive schemes discussed above, namely ***ACP*** and $\mu\boldsymbol{PR}$. In Chapter 4, we define two computational models based on imperative languages, ***Rec*** and ***While***. The semantics of ***Rec*** is fully discussed, while the ***While*** language is presented briefly (details being given in [TZ88, TZ00]). Chapters 5, 6 and 7 prove (1.1), (1.2) and (1.3) respectively. Chapter 6, we believe, is the core of this thesis, which proves that any function computable by an ***ACP*** scheme is computable by some ***Rec*** procedures. Chapter 8 concludes this thesis with a short summary and future work. In the Appendix, we develop the denotational semantics of statements of the ***Rec*** language and prove their equivalence with the operational semantics. This is not essential to our main results, but we believe it is interesting in its own right.

Chapter 2

Basic concepts

In this chapter, we will introduce some basic concepts concerning signatures and algebras, which will be used in the following chapters. In particular, we have two groups of concepts extracted from [TZ88, TZ00] and [Fef96] respectively. We will use the definitions in [TZ88, TZ00] as the framework, and introduce the differences and connections between that and [Fef96] in the last section 2.5. We present this chapter to make the thesis self-contained, and to simplify the presentation. Interested readers can refer to [TZ88, TZ00, Fef96] for detailed discussions.

2.1 Signatures

Definition 2.1.1 (Many-sorted signatures)**.** A many-sorted *signature* Σ is a pair $\langle \boldsymbol{Sort}(\Sigma), \boldsymbol{Func}\,(\Sigma)\rangle$ where

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

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

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

Each symbol F has a *type* $s_1 \times \cdots \times s_m \to s$, where $m \geq 0$ is the *arity* of F, and $s_1, \ldots, s_m \in \boldsymbol{Sort}(\Sigma)$ are the *domain sorts* and $s \in \boldsymbol{Sort}(\Sigma)$ is the *range sort* of F. The case $m = 0$ corresponds to *constant symbols*, we then write $\mathsf{F} : \to s$.

Definition 2.1.2 (Product types over Σ)**.** A *product type* over Σ, or Σ*-product type*, is a symbol of the form $s_1 \times \cdots \times s_m$ $(m \geq 0)$, where $s_1, \ldots, s_m$ are sorts of Σ, called its *component sorts.* We use $u, v, w, \ldots$ for Σ-product types.

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

Definition 2.1.3 (Function types)**.** Let A be a Σ-algebra. A *function type* over Σ, or *Σ-function type*, is a symbol of the form $u \to s$, with *domain type* u and *range type* s, where u is a Σ-product type. We use $\tau_1,\ \tau_2,\ \ldots$ for Σ-function types.

Definition 2.1.4 (Σ-algebras)**.** A *Σ-algebra* A has, for each sort s of Σ, a non-empty set A_s, called the *carrier of sort* s, and for each Σ-function symbol $\mathsf{F} : s_1 \times \cdots \times s_m \to s$, a (*partial*) function $\mathsf{F}^A : A_{s_1} \times \cdots \times A_{s_m} \to A_s$. (If $m = 0$, this is an element of A_s.)

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

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

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

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

Notation 2.1.5. We will write $\Sigma(A)$ to denote the signature of an algebra A.

Notation 2.1.6. (a) We will use the following notation for signatures Σ:

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

(b) We will use the following notation for Σ-algebras A:

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

Example 2.1.7 (Booleans)**.** The signature of booleans is important. It can be defined as

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

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

Example 2.1.8 (Naturals)**.** The signature of naturals can be defined as

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

Definition 2.1.9 (Reducts and expansions)**.** Let Σ and Σ' be signatures.

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

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

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

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

Definition 2.1.10 (Σ-variables)**.** Let $\boldsymbol{Var} = \boldsymbol{Var}(\Sigma)$ be the class of Σ-variables $\mathtt{x}$, $\mathtt{y}$, ..., and $\boldsymbol{Var}_s$ be the class of variables of sort s. For $u = s_1 \times \cdots \times s_m$, we write $\mathtt{x} : u$ to mean that $\mathtt{x}$ is a u-tuple of *distinct* variables.

Definition 2.1.11 (Σ-terms)**.** Let $\boldsymbol{Term} = \boldsymbol{Term}(\Sigma)$ be the class of Σ-terms $t, \ldots$, and $\boldsymbol{Term}_s$ be the class of terms of sort s, defined by

$$t^s ::= \mathsf{x}^s \mid \mathsf{F}(t_1^{s_1}, \ldots, t_m^{s_m})$$

where $\mathsf{F} \in \boldsymbol{Func}\,(\Sigma)_{u \to s}$ and $u = s_1 \times \cdots \times s_m$. We write $t : s$ to indicate that $t \in \boldsymbol{Term}_s$. Further, we write $t : u$ to indicate that t is a u-tuple of terms, *i.e.*, a tuple of terms of sorts $s_1, \ldots, s_m$. (Note that in standard signature Σ the definition of $\boldsymbol{Term}(\Sigma)$ is extended to include a conditional constructor, *cf.* Definition 2.2.3)

Definition 2.1.12 (Closed terms over Σ)**.** We define the class $\boldsymbol{T}(\Sigma)$ of *closed terms over Σ*, and for each Σ-sort s, the class $\boldsymbol{T}(\Sigma)_s$ of closed terms of sort s. These are generated inductively by the rule: if $\mathsf{F} \in \boldsymbol{Func}\,(\Sigma)_{u \to s}$ and $t_i \in \boldsymbol{T}(\Sigma)_{s_i}$ for $i = 1, \ldots, m$ where $u = s_1 \times \cdots \times s_m$, then $\mathsf{F}(t_1, \ldots, t_m) \in \boldsymbol{T}(\Sigma)_s$.

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

Assumption 2.1.13 (Instantiation)**.** In this thesis, we will assume:

$\boldsymbol{T}(\Sigma)_s$ is non-empty for each $s \in \boldsymbol{Sort}(\Sigma)$.

Definition 2.1.14 (Valuation of closed terms)**.** For a Σ-algebra A and $t \in \boldsymbol{T}(\Sigma)_s$, we define the *valuation* $t_A \in A_s$ *of* t *in* A by structural inductions on t:

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

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

Definition 2.1.15 (Default terms; Default values)**.** (a) For each sort s, we pick a closed term of sort s. (There is at least one, by the instantiation assumption.) We call this the *default term of sort* s, written $\boldsymbol{\delta}^s$. Further, for each product type $u = s_1 \times \cdots \times s_m$ of Σ, the *default (term) tuple of type* u, written $\boldsymbol{\delta}^u$, is the tuple of default terms $(\boldsymbol{\delta}^{s_1}, \ldots, \boldsymbol{\delta}^{s_m})$.

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

2.2 Standard signatures and algebras

Definition 2.2.1 (Standard signatures)**.** A signature Σ is *standard* if $\Sigma(\mathcal{B}) \subseteq \Sigma$.

Definition 2.2.2 (Standard algebras)**.** Given a standard signature Σ, a Σ-algebra A is a *standard algebra* if it is an expansion of $\mathcal{B}$, as defined in Example 2.1.7.

Definition 2.2.3 (Σ-terms for standard signatures)**.** We extend $\boldsymbol{Term}(\Sigma)$ to include a conditional constructor

$$t^s ::= \ \ldots \ |\mathsf{if}\ b\ \mathsf{then}\ t^s_1\ \mathsf{else}\ t^s_2\ \mathsf{fi}$$

where b is a (boolean) term of sort bool.

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 $\mathcal{B}$.

2.3 N-standard signatures and algebras

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

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

Definition 2.3.2 (N-standard algebra)**.** The corresponding Σ-algebra A is *N-standard* if the carrier A_{nat} is the set of natural numbers $\mathbb{N}=\{0,1,2,\ldots\}$, and the standard operations (listed above) have their *standard interpretations* on $\mathbb{N}$.

Definition 2.3.3 (N-standardization of Σ)**.** The N-*standardization* Σ^N of a standard signature Σ is formed by adjoining the sort nat and the operations 0, S, $\mathsf{eq}_{\mathsf{nat}}$, and $\mathsf{less}_{\mathsf{nat}}$.

Definition 2.3.4 (N-standardization of A)**.** The N-*standardization* A^N of a standard Σ-algebra A is the Σ^N-algebra formed by adjoining the carrier $\mathbb{N}$ together with certain standard operations to A, thus:

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

Assumption 2.3.5 (N-Standardness)**.** In this thesis, we will assume, unless stated otherwise:

All signatures Σ and Σ-algebras A are N-standard.

2.4 Algebras A^* of signature Σ^*

Definition 2.4.1 (Signature Σ^* and algebras A^*)**.** Given a signature Σ, and Σ-algebra A, we extend Σ and expand A in two stages:

(a) N-standardize these to form Σ^N and A^N.

(b) Extend Σ^N by including, for each sort s of Σ, a new *starred sort* s^*, and also the function symbols described below. Define, for each sort s of Σ, the carrier A^*_s of sort s^*, to be the set of finite sequences (or arrays) a^* over A_s.

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

(ii) $\mathsf{Null}_s \;:\; \to s^*$, where Null^A_s is the array in A^*_s of zero length;

(iii) $\mathsf{Ap}_s \;:\; s^* \times \mathsf{nat} \to s$, where

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

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

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

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

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

Definition 2.4.2. (a) A sort of Σ^* is called *simple* or *starred* according as it has the form s or s^*(respectively), for some $s \in \boldsymbol{Sort}(\Sigma)$.

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

Remarks 2.4.3. (a) The reason for introducing starred sorts is the lack of effective coding of finite sequences within abstract algebras in general.

(b) Starred sorts have significance in programming languages, since starred variables can be used to model arrays, and (hence) *finite but unbounded memory.* They give us the power of dynamic memory allocation.

(c) For signatures Σ and algebras A where we focus on the array signatures and algebras Σ^* and A^* (*e.g.* with the computation models $\mu\boldsymbol{PR}^*(A)$, $\boldsymbol{ACP}^*(A)$ discussed later) only standardness (not N-standardness) need really be assumed, since in any case Σ^* and A^* will be N-standard, as required by Assumption 2.3.5.

2.5 Second-order signatures and algebras

The algebras in [TZ88, TZ00] are first order algebras, since all functional symbols are interpreted as first-order functions within the algebras. In general, however, Feferman's $\boldsymbol{ACP}$ deals with second-order many-sorted algebras in [Fef96]. This section provides the background for Feferman's $\boldsymbol{ACP}$ schemes in the next chapter. The N-Standardness Assumption (Assumption 2.3.5) holds here as elsewhere throughout the thesis.

Definition 2.5.1 (Second-order signatures)**.** A *second-order signature* Σ is a pair $\langle \boldsymbol{Sort}(\Sigma), \boldsymbol{Func}(\Sigma)\rangle$ where

(a) $\boldsymbol{Sort}(\Sigma)$ is a finite set of *sorts*, where $\mathsf{bool} \in \boldsymbol{Sort}(\Sigma)$, *i.e.*, Σ is standard.

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

$$\mathsf{F}: \ \tau_1 \times \cdots \times \tau_m \times s_1 \times \cdots \times s_n \to s.$$

Each symbol F has a type $\tau_1 \times \cdots \times \tau_m \times s_1 \times \cdots \times s_n \to s$, where $m \geq 0$ and $n \geq 0$, $s_1, \ldots, s_m, s \in \boldsymbol{Sort}(\Sigma)$, and $\tau_1, \ldots, \tau_m$ are Σ-function types (see Definition 2.1.3). When $m = 0$, a symbol F is *first-order*, *i.e.* a function symbol.

Definition 2.5.2 (second-order algebras)**.** A *second-order Σ-algebra A* has:

(a) for each sort s of Σ, a non-empty set A_s, called the *carrier of sort s*. In particular, we have $\mathbb{B}$ as the carrier of sort bool. Then, for each $\tau = u \to s$, we take $A_\tau = \{\varphi \mid \varphi : A^u \to A_s\}$.

(b) for each functional symbol $\mathsf{F} : \tau_1 \times \cdots \times \tau_m \times s_1 \times \cdots \times s_n \to s$, a (partial) functional $\mathsf{F}^A : A_{\tau_1} \times \cdots \times A_{\tau_m} \times A_{s_1} \times \cdots \times A_{s_n} \to A_s$. (Again, if $m = n = 0$, this is an element of A_s.)

Notation 2.5.3. We will write $\pi, \ldots$ for *function product types* $\tau_1 \times \cdots \times \tau_m$ ($m \geq 0$).

Notation 2.5.4. If $\pi = \tau_1 \times \cdots \times \tau_m$, we write $A^\pi = A_{\tau_1} \times \cdots \times A_{\tau_m}$.

Remarks 2.5.5. (a) Given a second-order signature Σ, a Σ-function symbol $\mathsf{F}: \ \tau_1 \times \cdots \times \tau_m \times s_1 \times \cdots \times s_n \to s$ is of type level 2, 1, or 0, according as $m > 0$, $m = 0$ and $n > 0$, or $m = n = 0$.

(b) Σ is said to be first-order if each $\mathsf{F} \in \boldsymbol{Func}(\Sigma)$ is of type level ≤ 1, in that it is equivalent to the standard (first-order) signature defined in §2.2.

(c) Corresponding to each $\mathsf{F} \in \boldsymbol{Func}(\Sigma)$, F^A is of type level 2, 1 or 0; and corresponding to Σ, a Σ-algebra A is of second or first order.

Chapter 3

Models of computation based on recursive schemes

In this chapter, we will introduce two models of computation based on recursive schemes, $\boldsymbol{ACP}$ and $\mu\boldsymbol{PR}$. The contents are taken from [Fef96] and [TZ88] respectively with necessary modification.

3.1 Feferman's $\boldsymbol{ACP}$ schemes

In general, *abstract computational procedures* ($\boldsymbol{ACP}$) deal with many-sorted algebras A with objects of type level ≤ 2 (see Remark 2.5.5). With each signature Σ are associated the following formal schemes for computation procedures on Σ-algebras.

I.	(Initial functionals)	$\mathsf{F}(\varphi, x) \simeq \mathsf{F}_k(\varphi, x)$ $\;$ (*for each* $\mathsf{F}_k \in \boldsymbol{Func}\,(\Sigma)$);
II.	(Identity)	$\mathsf{F}(x) = x$;
III.	(Application)	$\mathsf{F}(\varphi, x) \simeq \varphi(x)$;

IV. (Conditional) $\mathsf{F}(\varphi, x, b) \simeq$ $[\mathsf{if}\ b\ \mathsf{then}\ \mathsf{G}(\varphi, x)\ \mathsf{else}\ \mathsf{H}(\varphi, x)];$

V. (Structural) $\mathsf{F}(\varphi, x) \simeq \mathsf{G}(\varphi_f, x_g);$

VI. (Individual substitution) $\mathsf{F}(\varphi, x) \simeq \mathsf{G}(\varphi, x, \mathsf{H}(\varphi, x)));$

VII. (Function substitution) $\mathsf{F}(\varphi, x) \simeq \mathsf{G}(\varphi, \lambda y \cdot \mathsf{H}(\varphi, x, y), x);$

VIII. (Least fixed point)

$$
\begin{aligned}
&\mathsf{F}_1(\varphi, x, y_1) \simeq \varrho_1^{\varphi,x}(y_1)\\
&\ldots,\\
&\mathsf{F}_n(\varphi, x, y_n) \simeq \varrho_n^{\varphi,x}(y_n)\\
&\text{where}\\
&(\varrho_1^{\varphi,x}, \ldots, \varrho_n^{\varphi,x}) \equiv\\
&\mathrm{LFP}((\lambda\varrho_1 \cdot \ldots \cdot \lambda\varrho_n \cdot \lambda z_1 \cdot\\
&\qquad \mathsf{G}_1(\varphi, \varrho_1, \ldots, \varrho_n, x, z_1)),\\
&\ldots,\\
&(\lambda\varrho_1 \cdot \ldots \cdot \lambda\varrho_n \cdot \lambda z_n \cdot\\
&\qquad \mathsf{G}_n(\varphi, \varrho_1, \ldots, \varrho_n, x, z_n))).
\end{aligned}
$$

The partial equality "$\simeq$" above means that either both sides of the equation converge and are equal, or both sides diverge. In scheme V, $f : \{1, \ldots, m'\} \to \{1, \ldots, m\}$, $g : \{1, \ldots, n'\} \to \{1, \ldots, n\}$ and the scheme itself abbreviates

$$\mathsf{F}(\varphi_1, \ldots, \varphi_m, x_1, \ldots, x_n) \simeq \mathsf{G}(\varphi_{f(1)}, \ldots, \varphi_{f(m')}, x_{g(1)}, \ldots, x_{g(n')}).$$

As shown in [Fef96], the schemes are invariant under isomorphism.

Definition 3.1.1. (a) $\boldsymbol{ACP}(\Sigma)$ is the collection of all F generated by the schemes for signature Σ.

(b) For any particular A of signature Σ, we take $\boldsymbol{ACP}(A)$ to be the collection of all F^A for $\mathsf{F} \in \boldsymbol{ACP}(\Sigma)$, and say that a functional F is $\boldsymbol{ACP}$ computable over A if $F = \mathsf{F}^A$ for some such F.

(c) $\boldsymbol{ACP}^1(A)$ is the collection of all functions of type level ≤ 1 in $\boldsymbol{ACP}(A)$.

Definition 3.1.2. (a) $\boldsymbol{ACP}^*(\Sigma)$ is the collection of all F in $\boldsymbol{ACP}(\Sigma^*)$, with the restriction that the domain and range types of F are simple (see Definition 2.4.2).

(*b*) For any particular A of signature Σ, we take $\boldsymbol{ACP}^*(A)$ to be the collection of all F^A for $\mathsf{F} \in \boldsymbol{ACP}^*(\Sigma)$.

(*c*) $\boldsymbol{ACP}^{*1}(A)$ is the collection of all functions of type level ≤ 1 in $\boldsymbol{ACP}^*(A)$.

Notation 3.1.3. In the above context, we use, for $1 \leq i \leq n$,

(*a*) $\hat{\mathsf{G}}_i^{\varphi,x}$ as abbreviations of $\lambda\varrho_1 \cdot \ldots \cdot \lambda\varrho_n \cdot \lambda z_i \cdot \mathsf{G}_i(\varphi, \varrho_1, \ldots, \varrho_n, x, z_i)$;

(*b*) $\hat{\mathsf{G}}_i^{x}$ as abbreviations of $\lambda\varrho_1 \cdot \ldots \cdot \lambda\varrho_n \cdot \lambda z_i \cdot \mathsf{G}_i(\varrho_1, \ldots, \varrho_n, x, z_i)$.

Notation 3.1.4. We define, for $1 \leq i \leq n$,

(*a*) $\hat{G}_i^{\varphi,x}$ is the interpretation of $\hat{\mathsf{G}}_i^{\varphi,x}$ in A defined by

$$\lambda\varrho_1 \cdot \ldots \cdot \lambda\varrho_n \cdot \lambda z_i \cdot \mathsf{G}_i^A(\varphi, \varrho_1, \ldots, \varrho_n, x, z_i);$$

(*b*) $\hat{G}_i^{x}$ is the interpretation of $\hat{\mathsf{G}}_i^{x}$ in A defined by

$$\lambda\varrho_1 \cdot \ldots \cdot \lambda\varrho_n \cdot \lambda z_i \cdot \mathsf{G}_i^A(\varrho_1, \ldots, \varrho_n, x, z_i).$$

Remark 3.1.5 (Simultaneous LFP)**.** In the least fixed points scheme VIII, we diverge from [Fef96] by using *simultaneous least fixed points*, in the sense that

$$\begin{aligned}
\varrho_1^0 &\equiv \perp \\
\ldots & \\
\varrho_n^0 &\equiv \perp \\
\varrho_1^1 &\equiv \hat{G}_1^{\varphi,x}(\varrho_1^0, \ldots, \varrho_n^0) \\
\ldots & \\
\varrho_n^1 &\equiv \hat{G}_n^{\varphi,x}(\varrho_1^0, \ldots, \varrho_n^0) \\
\varrho_1^{k+1} &\equiv \hat{G}_1^{\varphi,x}(\varrho_1^k, \ldots, \varrho_n^k) \\
\ldots & \\
\varrho_n^{k+1} &\equiv \hat{G}_n^{\varphi,x}(\varrho_1^k, \ldots, \varrho_n^k)
\end{aligned}$$

and $\varrho_i^{\varphi,x} = \bigcup_{k=0}^{\infty} \varrho_i^k$ for $i = 1, \ldots, n$.

This seems necessary to prove the equivalence of $\boldsymbol{ACP}^1(A)$ with $\mu\boldsymbol{PR}(A)$ which uses *simultaneous primitive recursion* [TZ88, TZ00].

Note that if our type structure incorporated *product types*, then the simultaneous LFP scheme could be replaced (or coded) by a *simple* LFP scheme in an obvious way.

Remarks 3.1.6. (a) The types of the schemes and their arguments are not specified but should be evident.

(b) Since we consider only first-order algebras, *i.e.* all primitive functions F_k are objects of type level ≤ 1, by [Fef96, Theorem 4] all F^A are *continuous*, hence, *monotonic* (see Definition A.1.5 and A.1.7). This justifies the use of scheme VIII, *i.e.* the existence of the least fixed points.

Notation 3.1.7. We write $\boldsymbol{ACP}_0$ for $\boldsymbol{ACP}$ minus scheme VII.

Remark 3.1.8. By [Fef96, Theorem 3], $\boldsymbol{ACP}_0(A)$ is closed under scheme VII for first-order algebras A, *i.e.* if A is first-order, then

$$\boldsymbol{ACP}_0(A) = \boldsymbol{ACP}(A).$$

Therefore, in the rest of thesis, we will not distinguish $\boldsymbol{ACP}$ and $\boldsymbol{ACP}_0$.

3.2 $\mu\boldsymbol{PR}$ schemes

We give the definitions of $\mu\boldsymbol{PR}$ computability in this section. Most of the contents are taken from [TZ88] with some necessary modifications. We avoid excessive formality. Interested readers can refer to [TZ88] for more details.

For each Σ, we have the following induction schemes which specify a common structure for functions over all N-standard algebras A of signature Σ.

I. (Primitive functions) $\mathsf{f}(x) \simeq \mathsf{F}_k(x)$ $(for\ each\ \mathsf{F}_k \in \boldsymbol{Func}\,(\Sigma))$;

II. (Projection) $\mathsf{f}(x) = x_i$;

III. (Definition by cases)
$$\mathsf{f}(x) \simeq \begin{cases} \mathsf{g}_1(x) & \text{if } \mathsf{h}(\mathrm{x}) = \mathtt{tt} \\ \mathsf{g}_2(x) & \text{if } \mathsf{h}(\mathrm{x}) = \mathtt{ff} \\ \uparrow & \text{if } \mathsf{h}(\mathrm{x}) \uparrow; \end{cases}$$

IV. (Composition) $\mathsf{f}(x) \simeq \mathsf{h}(\mathsf{g}_1(x), \ldots, \mathsf{g}_m(x))$;

V. (Simultaneous primitive recursion)

$$\mathsf{f}_1(x, 0) \simeq \mathsf{g}_1(x)$$
$$\ldots,$$
$$\mathsf{f}_n(x, 0) \simeq \mathsf{g}_n(x)$$

$$\begin{aligned}
&\mathsf{f}_1(x, z+1) \simeq \mathsf{h}_1(x, z, \mathsf{f}_1(x,z), \ldots, \mathsf{f}_n(x,z))\\
&\ldots,\\
&\mathsf{f}_n(x, z+1) \simeq \mathsf{h}_n(x, z, \mathsf{f}_1(x,z), \ldots, \mathsf{f}_n(x,z));
\end{aligned}$$

VI. (Least number operator) $\mathsf{f}(x) \simeq \mu z[\mathsf{g}(x,z) = \mathtt{tt}]$.

Similar to ***ACP***, the schemes are invariant under isomorphism.

Remarks 3.2.1. (*a*) The types of the schemes and their arguments are not specified but should be evident.

(*b*) The semantics of the schemes should be clear from their formal presentation. (Formal semantics can be found in [TZ88].) We should however point out that the least number or μ operator in scheme VI is the *constructive* μ-operator, with the operational semantics: "Test $g(z,0)$, $g(z,1)$, $g(z,2), \ldots$ in turn until you find k such that $g(z,k)$ is true; then halt with output k." This is a *partial* operator; *e.g.* if $g(z,0)\downarrow\mathtt{ff}$, $g(z,1)\uparrow$ and $g(z,2)\downarrow\mathtt{tt}$, then $f(z)\uparrow$ (*i.e.*, it does not converge to 2).

(*c*) $\mu\boldsymbol{PR}(A)$ is the set of all partial functions obtained from the basic functions defined in I-III by means of operations defined in IV-VI.

(*d*) We can see, from schemes V and VI, the reason that we need to assume that natural numbers $\mathbb{N}$ is built into our algebras A, *i.e.* the N-standardness assumption.

Definition 3.2.2. (*a*) $\mu\boldsymbol{PR}(\Sigma)$ is the collection of all f generated by the schemes for signature Σ.

(*b*) For any particular A of signature Σ, we take $\mu\boldsymbol{PR}(A)$ to be the collection of all f^A for $\mathsf{f} \in \mu\boldsymbol{PR}(\Sigma)$, and say that a function f is $\mu\boldsymbol{PR}$ computable over A if $f = \mathsf{f}^A$ for some such f.

Definition 3.2.3. (*a*) $\mu\boldsymbol{PR}^*(\Sigma)$ is the collection of f in $\mu\boldsymbol{PR}(\Sigma^*)$, with the restriction that the domain and range types of f are simple.

(*b*) For any particular A of signature Σ, we take $\mu\boldsymbol{PR}^*(A)$ to be the collection of all f^A for $\mathsf{f} \in \mu\boldsymbol{PR}^*(\Sigma)$.

Remark 3.2.4 (***PR*** schemes)**.** We denote by $\boldsymbol{PR}(\Sigma)$ the collection of all f generated by the schemes I-V for signature Σ. Then, we can define $\boldsymbol{PR}(A)$, $\boldsymbol{PR}^*(A)$, and $\boldsymbol{PR}(A^*)$ in same way as for $\mu\boldsymbol{PR}(A)$, $\mu\boldsymbol{PR}^*(A)$, and $\mu\boldsymbol{PR}(A^*)$. We will say that f is *primitive recursive* on A to mean that $f \in \boldsymbol{PR}(A)$.

Chapter 4

Models of computation based on imperative languages

In this chapter, we will study two imperative programming models of computation based on imperative programming languages, $\boldsymbol{Rec}$ and $\boldsymbol{While}$. $\boldsymbol{Rec}$ is of particular interest, since we will use it to bridge $\boldsymbol{ACP}$ and $\mu\boldsymbol{PR}$. $\boldsymbol{While}$ is presented briefly in the last section (4.11) to make this thesis self-contained.

First, we define an imperative programming language $\boldsymbol{Rec} = \boldsymbol{Rec}(\Sigma)$ on standard Σ-algebras. Then, we will define the abstract syntax and semantics of this language.

4.1 Syntax

We define five syntactic classes: *variables*, *procedure name*, *terms*, *statements*, and *procedures.*

(a) $\boldsymbol{Var} = \boldsymbol{Var}(\Sigma)$ is the class of Σ-variables $\mathtt{x}$, $\mathtt{y}$, ... (see Definition 2.1.10).

(b) $\boldsymbol{ProcName} = \boldsymbol{ProcName}(\Sigma)$ is the class of procedure names $P_1,\ P_2, \ldots$. We write $\boldsymbol{ProcName}_{u \to v}$ for all procedure names of type $u \to v$.

(c) $\boldsymbol{Term} = \boldsymbol{Term}(\Sigma)$ is the class of Σ-terms $t, \ldots$ (see Definition 2.2.3).

(d) $\boldsymbol{Stmt} = \boldsymbol{Stmt}(\Sigma)$ is the class of statements $S, \ldots$, defined by:
$S ::= \mathsf{skip} \mid \mathtt{x}^u := t^u \mid S_1; S_2 \mid \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \mid \mathtt{x}^v := P(t^u)$

where $\mathtt{x}^u := t^u$ is a *concurrent assignment* and $\mathtt{x}^v := P(t^u)$ is a *procedure call*, with $P \in \boldsymbol{ProcName}_{u \to v}$ for some product types u, v.

We will write $\boldsymbol{Stmt}^*$ for $\boldsymbol{Stmt}(\Sigma^*)$.

(*e*) $\boldsymbol{Proc} = \boldsymbol{Proc}(\Sigma)$ is the class of procedures $R, \ldots,$ defined by

$$R ::= \langle D^{\mathsf{p}} : D^{\mathsf{v}} : S \rangle,$$

where D^{p} is a *procedure declaration*, D^{v} is a *variable declaration*, and S is the *body*.

D^{p} is defined by

$$D^{\mathsf{p}} ::= P_1 \Longleftarrow R_1,\ \ldots,\ P_m \Longleftarrow R_m, \qquad (m \geq 0)$$

where $R_i ::= \langle D_i^{\mathsf{p}} : D_i^{\mathsf{v}} : S_i \rangle$, for $i = 1, \ldots, m$; D_i^{p} and D_i^{v} are defined like D^{p} and D^{v}.

D^{v} is defined by

$$D^{\mathsf{v}} ::= \mathsf{in}\ \mathsf{a}\ \mathsf{out}\ \mathsf{b}\ \mathsf{aux}\ \mathsf{c},$$

where a, b, and c are lists of input variables, output variables, and auxiliary variables respectively, subject to the conditions:

- a, b, and c are pairwise disjoint;
- every variable occurring in S must be declared in D^{v};
- the *input variables* must not occur on the left hand side of assignments in S.

4.2 Closed programs

Notation 4.2.1. For a procedure declaration D^{p}, we use following notation to indicate its depth in the main procedure:

(*a*) If D^{p} is the main procedure declaration, we write D^{p_0} for D^{p}.

(*b*) Let $D^{\mathsf{p}} \equiv \langle P_i \Longleftarrow R_i \rangle_{i=1}^m$ and $R_i \equiv \langle D_i^{\mathsf{p}} : D_i^{\mathsf{v}} : S_i \rangle$ for $i = 1, \ldots, m$. If $D^{\mathsf{p}} \equiv D^{\mathsf{p}_k}$, we write $D_i^{\mathsf{p}_{k+1}}$ for D_i^{p}, for $i = 1, \ldots, m$.

So k is the *depth* of the procedure declaration. When $k = 0$, D^{p_k} is the main procedure declaration; when $k > 0$, D^{p_k} is an intermediate procedure declaration.

Definition 4.2.2. $\boldsymbol{ProcSet}(D^{\mathsf{p}_k})$ is the set of procedure variables defined as follows:

(a) for $D^{\mathsf{p}_0} \equiv \langle P_i \Longleftarrow R_i \rangle_{i=1}^m$,

$$\boldsymbol{ProcSet}(D^{\mathsf{p}_0}) \equiv \{P_1, \ldots, P_m\}$$

(b) for $D^{\mathsf{p}_k} \equiv \langle P_i \Longleftarrow R_i \rangle_{i=1}^m$, where $R_i \equiv \langle D_i^{\mathsf{p}_{k+1}} : D_i^{\mathsf{v}} : S_i \rangle$, and $D_i^{\mathsf{p}_{k+1}} \equiv \langle P_{ij} \Longleftarrow R_{ij} \rangle_{j=1}^n$,

$$\boldsymbol{ProcSet}(D_i^{\mathsf{p}_{k+1}}) \equiv \boldsymbol{ProcSet}(D^{\mathsf{p}_k}) \cup \{P_{i1}, \ldots, P_{in}\}$$

Note that the definition is by recursion on the depth k of the declaration, *i.e.* "top-down". $\boldsymbol{ProcSet}(D^{\mathsf{p}_k})$ consists of all procedure variables currently declared in D^{p_k}, as well as those declared in the "prior" declarations $D^{\mathsf{p}_0}, \ldots, D^{\mathsf{p}_{k-1}}$. Thus the definition depends implicitly on a main declaration D^{p_0} as a global context.

Notation 4.2.3. Let $\boldsymbol{ProcVar}(S)$ be the set of procedure names occurring in the statement S (as procedure calls).

Definition 4.2.4 (Closed declaration)**.** A procedure declaration $D^{\mathsf{p}} \equiv \langle P_i \Longleftarrow R_i \rangle_{i=1}^m$, where $R_i \equiv \langle D_i^{\mathsf{p}} : D_i^{\mathsf{v}} : S_i \rangle$, is *closed* if

(a) D_i^{p} is *closed* for $i = 1, \ldots, m$,

(b) for $i = 1, \ldots, m$, $\boldsymbol{ProcVar}(S_i) \subseteq \boldsymbol{ProcSet}(D_i^{\mathsf{p}})$

Again, this is a recursive definition, but unlike Definition 4.2.2, it is "bottom-up", *i.e.* structural recursion on D^{p}, and the base case occurs whenever $m = 0$.

Definition 4.2.5 (Closed procedure)**.** A procedure $R \equiv \langle D^{\mathsf{p}} : D^{\mathsf{v}} : S \rangle$ is *closed* if

(a) D^{p} is *closed* and

(b) $\boldsymbol{ProcVar}(S) \subseteq \boldsymbol{ProcSet}(D^{\mathsf{p}})$

Assumption 4.2.6 (Closure)**.** In this thesis, we will assume:

All procedure declarations and procedures are closed.

4.3 States

Definition 4.3.1 (State)**.** For each standard Σ-algebra A, a *state* on A is a family $\langle \sigma_s \mid s \in \boldsymbol{Sort}(\Sigma) \rangle$ of functions

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

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

Notation 4.3.2. For $\mathtt{x} \in \boldsymbol{Var}_s$, we often write $\sigma(\mathtt{x})$ for $\sigma_s(\mathtt{x})$. Also, for a tuple $\mathtt{x} \equiv (\mathtt{x}_1, \dots, \mathtt{x}_m)$, we write $\sigma[\mathtt{x}]$ for $(\sigma(\mathtt{x}_1), \dots, \sigma(\mathtt{x}_m))$.

Definition 4.3.3 (Variant of a state)**.** Let σ be a state over A, $\mathtt{x} \equiv (\mathtt{x}_1, \dots, \mathtt{x}_n) : u$ and $a = (a_1, \dots, a_n) \in A^u$ (for $n \geq 1$). We define $\sigma\{\mathtt{x}/a\}$ to be the state over A formed from σ by replacing its value at $\mathtt{x}_i$ by a_i for $i = 1, \dots, n$. That is, for all variables $\mathtt{y}$:

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

4.4 Semantics of terms

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

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

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

The definition is by structural induction on t:

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

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

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

Definition 4.4.1. For any $M \subseteq \boldsymbol{Var}$, and states σ_1 and σ_2, $\sigma_1 \underset{M}{\approx} \sigma_2$ means $\sigma_1 \restriction M = \sigma_2 \restriction M$, *i.e.*, for all $\mathtt{x} \in M$, $\sigma_1(\mathtt{x}) = \sigma_2(\mathtt{x})$.

Lemma 4.4.2 (Functionality lemma for terms)**.** *For any term t and states σ_1 and σ_2, if $\sigma_1 \underset{M}{\approx} \sigma_2$ $(M = \boldsymbol{var}(t))$, then $[\![t]\!]^A \sigma_1 \simeq [\![t]\!]^A \sigma_2$.*

Proof. By structural induction on t. □

4.5 Algebraic operational semantics

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

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

We will present an outline of this approach following [TZ00]. Interested readers can refer to [TZ00] for details.

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

$$\langle\!| S |\!\rangle^A : \boldsymbol{State}(A) \stackrel{\cdot}{\longrightarrow} \boldsymbol{State}(A),$$

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

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

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

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

$$\boldsymbol{Comp}_1^A : \boldsymbol{Stmt} \times \boldsymbol{State}(A) \stackrel{\cdot}{\longrightarrow} \boldsymbol{State}(A)$$

by

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

Finally, the definition of *the computation step* function

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

follows by a simple recursion on n:

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

Note that for $n = 1$, this yields

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

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

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

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

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

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

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

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

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

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

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

by

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

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

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

4.6 Operational semantics of statements

We now apply the above theory to the language $\boldsymbol{Rec}(\Sigma)$. Even if the original statement concerns only algebras A, we nevertheless have to work over A^* (see Case 4 and Remark 4.6.6 below). Therefore, in what follows, $\sigma \in \boldsymbol{State}(A^*)$, and we define the semantic functions over A^*.

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

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

Note that $\mathtt{x}$ can be a starred variable and t a starred term. We will see later that, even if the original statement contains only unstarred atomic statements, $\boldsymbol{Rest}^{A^*}$ will generate starred atomic statements (see Case 4 and Remark 4.6.6 below).

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

Case 1. S is atomic.

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

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

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

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

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

Case 4. $S \equiv \mathtt{x} := P_i(t) \qquad (i = 1, \ldots, m)$

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

where $\hat{S}_i$ is the statement defined in Figure 4.1.

Here $\hat{S}_i$ looks complicated; however, the idea is simple. We want $\hat{S}_i$ to have the same functionality as P_i without any side effects. In other words, we want x to get its required value via the computation of $\hat{S}_i$, but all other variables in a, b, and c left unchanged, which is crucial for the proof of Lemma 4.7.3. Therefore, as is customary in most recursive procedure semantics, we first store the current values in some temporary storage; then execute the body of the procedure; and finally restore the values of the variables. We now give some details.

- We use array structures for temporary storage. In most compilers, stacks are used, and in this case, stacks would also be the better choice in principle; however, we want to avoid introducing too many concepts in this thesis. Actually, we simulate stacks by our array variables in $\hat{S}_i$. It is here that starred variables are introduced in the definition of $\boldsymbol{Rest}^{A^*}$ (see Remark 4.6.6).
- In the construction of $\hat{S}_i$, we assume a, b, and c are single variables in order to keep the notation manageable. It is, however, not hard to generalize this to the case that a, b, and c are tuples of variables.
- We introduce an extra temporary variable $\mathtt{b}_{\mathsf{tmp}}$ to avoid erasing the output of S_i when restoring b.
- Before the execution of the body S_i, we need to initialize the *local* variables a, b, and c.
- s_a, s_b, and s_c are sorts corresponding to the variables a, b, and c. Then $\boldsymbol{\delta}^{s_b}$ and $\boldsymbol{\delta}^{s_c}$ are the corresponding default values for b and c.
- The expressions '$t+1$' and '$t-1$' (for term $t : \mathsf{nat}$) must be interpreted in the language of N-standard signatures (§2.3). Now '$t+1$' can be simply interpreted as '$\mathsf{S}\ t$', and '$t-1$' can be interpreted as '$\mathsf{Pd}\ t$', where 'Pd' is a procedure name for the predecessor function which is easily defined by a ***Rec*** procedure.

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

Theorem 4.6.1. (a) *For S atomic,* $[\![S]\!]^{A^*} = \langle\!| S |\!\rangle^{A^*}$, *i.e.,*

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

(b)

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

$$
\begin{array}{|lll|}
\hline
\mathtt{a}^* & := & \mathsf{Newlength}_{s_a}(\mathtt{a}^*, \mathsf{Lgth}_{s_a}(\mathtt{a}^*) + 1); \\
\mathtt{b}^* & := & \mathsf{Newlength}_{s_b}(\mathtt{b}^*, \mathsf{Lgth}_{s_b}(\mathtt{b}^*) + 1); \\
\mathtt{c}^* & := & \mathsf{Newlength}_{s_c}(\mathtt{c}^*, \mathsf{Lgth}_{s_c}(\mathtt{c}^*) + 1); \\
\mathtt{a}^* & := & \mathsf{Update}_{s_a}(\mathtt{a}^*, \mathsf{Lgth}_{s_a}(\mathtt{a}^*) - 1, \mathtt{a}); \\
\mathtt{b}^* & := & \mathsf{Update}_{s_b}(\mathtt{b}^*, \mathsf{Lgth}_{s_b}(\mathtt{b}^*) - 1, \mathtt{b}); \\
\mathtt{c}^* & := & \mathsf{Update}_{s_c}(\mathtt{c}^*, \mathsf{Lgth}_{s_c}(\mathtt{c}^*) - 1, \mathtt{c}); \\
\mathtt{a} & := & t; \\
\mathtt{b} & := & \boldsymbol{\delta}^{s_b}; \\
\mathtt{c} & := & \boldsymbol{\delta}^{s_c}; \\
 & S_i & \\
\mathtt{b}_{\mathsf{tmp}} & := & \mathtt{b}; \\
\mathtt{a} & := & \mathsf{Ap}_{s_a}(\mathtt{a}^*, \mathsf{Lgth}_{s_a}(\mathtt{a}^*) - 1); \\
\mathtt{b} & := & \mathsf{Ap}_{s_b}(\mathtt{b}^*, \mathsf{Lgth}_{s_b}(\mathtt{b}^*) - 1); \\
\mathtt{c} & := & \mathsf{Ap}_{s_c}(\mathtt{c}^*, \mathsf{Lgth}_{s_c}(\mathtt{c}^*) - 1); \\
\mathtt{a}^* & := & \mathsf{Newlength}_{s_a}(\mathtt{a}^*, \mathsf{Lgth}_{s_a}(\mathtt{a}^*) - 1); \\
\mathtt{b}^* & := & \mathsf{Newlength}_{s_b}(\mathtt{b}^*, \mathsf{Lgth}_{s_b}(\mathtt{b}^*) - 1); \\
\mathtt{c}^* & := & \mathsf{Newlength}_{s_c}(\mathtt{c}^*, \mathsf{Lgth}_{s_c}(\mathtt{c}^*) - 1); \\
\mathtt{x} & := & \mathtt{b}_{\mathsf{tmp}}; \\
\mathtt{b}_{\mathrm{tmp}} & := & \boldsymbol{\delta}^{s_b}; \\
\hline
\end{array}
$$

Figure 4.1: Content of $\hat{S}_i$

(c)

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

(d)

$$[\![\mathsf{x} := P_i(t)]\!]^{A^*}\sigma \ \simeq \ [\![\hat{S}_i]\!]^{A^*}\sigma.$$

Proof. The results follow from Lemmas 4.6.2, 4.6.3, 4.6.4, and 4.6.5 below. We omit details. □

Lemma 4.6.2. *For S atomic,* $\boldsymbol{Comp}^{A^*}(S,\sigma,n) \ \simeq \begin{cases} \langle\!| S |\!\rangle^{A^*}\sigma & \text{if } \ n=1 \\ * & \text{otherwise} \end{cases}$

Lemma 4.6.3. $\boldsymbol{Comp}^{A^*}(S_1;S_2,\sigma,n) \ \simeq$

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

Lemma 4.6.4. $\boldsymbol{Comp}^{A^*}(\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi},\sigma,n+1) \ \simeq$

$$\begin{cases} \boldsymbol{Comp}^{A^*}(S_1,\sigma,n) & \text{if } \ [\![b]\!]^{A^*}\sigma = \mathtt{tt} \\ \boldsymbol{Comp}^{A^*}(S_2,\sigma',n) & \text{if } \ [\![b]\!]^{A^*}\sigma = \mathtt{ff} \\ \uparrow & \text{if } \ [\![b]\!]^{A^*}\sigma\!\uparrow. \end{cases}$$

Lemma 4.6.5. $\boldsymbol{Comp}^{A^*}(\mathsf{x} := P_i(t),\ \sigma,\ n+1) \simeq \boldsymbol{Comp}^{A^*}(\hat{S}_i,\ \sigma,\ n)$.

Remark 4.6.6. In case A is an N-standard Σ-algebra without starred sorts, we still need starred variables to define the semantic functions (see Case 4 in the definition of $\boldsymbol{Rest}^{A^*}$). Thus, we have to work with A^* for these semantic functions. An intuitive explanation is the following:

> *For a **Rec** procedure, we may need finite but arbitrarily large memory, since a recursive procedure can be called arbitrarily many times and we have to store information for all callers in order to make the caller work properly when the callee terminates and returns. This requires dynamic memory allocation, which is simulated by an array structure.*

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

Lemma 4.6.7. *Suppose* $\boldsymbol{var}(S) \subseteq M$. *If* $\sigma_1 \underset{M}{\approx} \sigma_2$, *then for all* $n \geq 0$,

$$\boldsymbol{Comp}^{A^*}(S, \sigma_1, n) \underset{M}{\approx} \boldsymbol{Comp}^{A^*}(S, \sigma_2, n).$$

Proof. By induction on n. Use the functionality lemma (4.4.2) for terms. □

Lemma 4.6.8 (Functionality lemma for statements)**.** *Suppose* $\boldsymbol{var}(S) \subseteq M$. *If* $\sigma_1 \underset{M}{\approx} \sigma_2$, *then either*

(i) $[\![S]\!]^A \sigma_1 \downarrow \sigma_1'$ *and* $[\![S]\!]^A \sigma_2 \downarrow \sigma_2'$ *(say), where* $\sigma_1' \underset{M}{\approx} \sigma_2'$, *or*

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

Proof. From Lemma 4.6.7. □

4.7 Semantics of procedures

Assumption 4.7.1 (Initialization)**.** Before the execution of procedures, we assume:

> *All but the input variables are initialized to the default values of the same sort.*

Definition 4.7.2 (Semantics of procedures)**.** Let

$$R \equiv \langle D^{\mathsf{p}} : D^{\mathsf{v}} : S \rangle, \text{ where } D^{\mathsf{v}} \equiv \mathsf{in\ a\ out\ b\ aux\ c}$$

be a procedure of type $u \to v$. Then its meaning is a function

$$[\![R]\!]^A : A^u \to A^v$$

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

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

Note, this is well defined by the functionality lemma (4.6.8) for statements.

Lemma 4.7.3 (Procedure assignment lemma). *Consider a statement* $\mathtt{x} := P_i(t)$*, where* P_i *has the declaration* $P_i \Longleftarrow R_i$*, and* $R_i \equiv \langle D_i^{\mathsf{p}} : D_i^{\mathsf{v}} : S_i \rangle$*. Then*

$$[\![\mathtt{x} := P_i(t)]\!]^{A^*}\sigma \quad \simeq \quad \sigma\{\mathtt{x}/[\![R_i]\!]^A([\![t]\!]^{A^*}\sigma)\}$$

Note that this lemma amounts to saying that the semantics of a procedure call statement is a state transformation which transforms a state to its *variant* in which the tuple $\mathtt{x}$ gets the required values while all other variables are left unchanged; in other words, there are *no side effects*.

Proof. Consider $\hat{S}_i$ (*cf.* Figure 4.1), let $\sigma' = [\![\mathtt{a} := t; \mathtt{b} := \boldsymbol{\delta}^{s_b}; \mathtt{c} := \boldsymbol{\delta}^{s_c}]\!]^{A^*}\sigma$. Clearly, $\sigma'[\mathtt{a}] = [\![t]\!]^{A^*}\sigma$, $\sigma'[\mathtt{b}] = \boldsymbol{\delta}^{s_b}$, and $\sigma'[\mathtt{c}] = \boldsymbol{\delta}^{s_c}$. By Definition 4.7.2,

$$[\![R_i]\!]^A([\![t]\!]^{A^*}\sigma) \ \simeq \ ([\![S_i]\!]^{A^*}\sigma')[\mathtt{b}]. \tag{4.1}$$

By Theorem 4.6.1 (d),

$$[\![\mathtt{x} := P_i(t)]\!]^{A^*}\sigma \simeq [\![\hat{S}_i]\!]^{A^*}\sigma. \tag{4.2}$$

We will show

$$[\![\hat{S}_i]\!]^{A^*}\sigma \simeq \sigma\{\mathtt{x}/([\![S_i]\!]^{A^*}\sigma')[\mathtt{b}]\}. \tag{4.3}$$

The result will then follow from (4.1), (4.2) and (4.3).

To show (4.3), note that $[\![\hat{S}_i]\!]^{A^*}$ is a state transformation involving only variables $\mathtt{a}^*$, $\mathtt{b}^*$, $\mathtt{c}^*$, $\mathtt{a}$, $\mathtt{b}$, $\mathtt{c}$, $\mathtt{b}_{\mathsf{tmp}}$, and $\mathtt{x}$ (*cf.* Figure 4.1). We will investigate the behavior of these variables to show that $\mathtt{a}^*$, $\mathtt{b}^*$, $\mathtt{c}^*$, $\mathtt{a}$, $\mathtt{b}$, $\mathtt{c}$ are unchanged, and $\mathtt{x}$ gets the desired values, *i.e.* $([\![S_i]\!]^{A^*}\sigma')[b]$. A formal proof will be tedious, since we need to record many state transformations carefully. An informal proof, however, is easy to provide, and, we believe, clear enough.

(a) $\mathtt{a}^*$, $\mathtt{b}^*$, and $\mathtt{c}^*$ are extended by one at the beginning of $\hat{S}_i$ and trimmed by one at the end. Within the execution of $\hat{S}_i$, only the last locations of $\mathtt{a}^*$, $\mathtt{b}^*$, and $\mathtt{c}^*$, which are trimmed, are modified. Clearly, $\mathtt{a}^*$, $\mathtt{b}^*$, and $\mathtt{c}^*$ keep their original values.

(b) The original values of $\mathtt{a}$, $\mathtt{b}$, and $\mathtt{c}$ are stored in the last locations in $\mathtt{a}^*$, $\mathtt{b}^*$, and $\mathtt{c}^*$ respectively before the execution of S_i, and restored after the execution. So their original values are kept.

(c) The last line of $\hat{S}_i$ ensures that $\mathtt{b}_{\mathsf{tmp}}$ takes the default value.

(d) The atomic statements $\mathtt{b}_{\mathsf{tmp}} := \mathtt{b}$ and $\mathtt{x} := \mathtt{b}_{\mathsf{tmp}}$ in $\hat{S}_i$ guarantee that $\mathtt{x}$ takes the desired value $([\![S_i]\!]^{A^*}\sigma')[\mathtt{b}]$.

□

Remark 4.7.4. The importance of the procedure assignment lemma is that, by stating that the semantics of a procedure call assignment is a state variant (without side-effects), it justifies the replacement of such a call by an oracle call statement (see S4.8 for definition).

Definition 4.7.5 (***Rec*** computable functions)**.** (a) A function f on A is *computable on* A *by a* ***Rec*** *procedure* R if $f = [\![R]\!]^A$. It is ***Rec*** *computable on* A if it is computable on A by some ***Rec*** procedure.

(b) ***Rec***(A) is the class of functions ***Rec*** computable on A.

Definition 4.7.6. A ***Rec****(Σ) procedure is a ***Rec***(Σ^*) procedure in which the *input* and *output* variables are *simple*. (However the auxiliary variables may be starred.)

Definition 4.7.7 (***Rec**** computable functions)**.** (a) A function f on A is *computable on* A *by a* ***Rec**** *procedure* R if $f = [\![R]\!]^A$. It is ***Rec**** *computable on* A if it is computable on A by some ***Rec**** procedure.

(b) ***Rec****(A) is the class of functions ***Rec**** computable on A.

4.8 *RelRec* computability

Let $\varphi \equiv \varphi_1, \ldots, \varphi_n$ be a tuple of (partial) functions

$$\varphi_i : A^{u_i} \stackrel{\cdot}{\longrightarrow} A^{v_i}.$$

We define the programming language ***Rec***(ϕ) (or by abuse of notation, ***Rec***(φ)) which extends the language ***Rec*** by including a set of special function symbols $\phi_1, \ldots, \phi_n$. We can think of $\phi_1, \ldots, \phi_n$ as "oracles" for $\varphi_1, \ldots, \varphi_n$.

Notation 4.8.1. We will use ***RelRec*** for the class of all ***Rec***(ϕ) procedures without specifying the oracle names.

The atomic statements of ***Rec***(ϕ) include the *oracle calls*

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

where $t : u_i$ and $\mathtt{x} : v_i$.

The semantics of this is given by

$$\langle\!|\mathtt{x} := \phi_i(t)|\!\rangle^A \sigma \;\simeq\; \begin{cases} \sigma\{\mathtt{x}/b\} & \text{if } [\![t]\!]^A\sigma \downarrow a \text{ and also } \varphi_i(a) \downarrow b \\ \uparrow & \text{otherwise.} \end{cases}$$

Following is the general form for a $\boldsymbol{Rec}(\phi)$ procedure R. Note that the oracle list is global, hence it is not presented in the inner procedures $R_1, \ldots, R_n$.

oracles $\phi_1, \ldots, \phi_m$
$P_1 \Longleftarrow R_1, \ldots, P_n \Longleftarrow R_n$
in a out b aux c
S

Note that the semantic functions for statements as well as other related functions like the computation step functions will all depend on the interpretations of the oracles. Therefore we will have functions $\boldsymbol{Rest}_\varphi^{A^*}$, $\boldsymbol{Comp}_\varphi^{A^*}$, $\boldsymbol{CompLength}_\varphi^{A^*}$, and $[\![S]\!]_\varphi^{A^*}$ instead of $\boldsymbol{Rest}^{A^*}$, $\boldsymbol{Comp}^{A^*}$, $\boldsymbol{CompLength}^{A^*}$, and $[\![S]\!]^{A^*}$. The definitions of these functions follow along similar lines to those in §4.6.

Notation 4.8.2. We will use notation $[\![R]\!]_\varphi^A$ for the function defined by the $\boldsymbol{Rec}(\phi)$ procedure R on A when ϕ is interpreted as φ. We may drop the subscript φ when it is clear from the context.

Therefore, the semantics of a $\boldsymbol{RelRec}$ procedure $R: \; u \to v$, where $R \equiv$

oracles ϕ
$P_1 \Longleftarrow R_1, \ldots, P_n \Longleftarrow R_n$
in a out b aux c
S

(where $\phi : \pi$ and $\mathsf{a} : u$) is a function

$$[\![R]\!]_\varphi^A : \; A^u \;\stackrel{\cdot}{\longrightarrow}\; A^v$$

given by

$$[\![R]\!]^A_\varphi(a) \ \simeq \ \begin{cases} \sigma'[\mathsf{b}] & \text{if } \ [\![S]\!]^{A^*}_\varphi \sigma \downarrow \sigma' \\ \uparrow & \text{if } \ [\![S]\!]^{A^*}_\varphi \sigma\uparrow. \end{cases}$$

where σ can be any state on A^* such that $\sigma[\mathsf{a}] = a$.

In this way we can define the notion of $\boldsymbol{Rec}(\varphi)$ *computability* or $\boldsymbol{Rec}$ *computability relative to* φ, and $\boldsymbol{Rec}^*(\varphi)$ *computability* or $\boldsymbol{Rec}^*$ *computability relative to* φ.

The reason for introducing $\boldsymbol{Rec}(\varphi)$ computability is because we need oracle call statements to simulate functions as arguments in higher order functionals.

4.9 Monotonicity of *RelRec* procedures

Notation 4.9.1. For any functions φ and φ' of the same type, we write $\varphi \sqsubseteq \varphi'$ to mean that for any input x,

$$\varphi(x)\downarrow \quad \Longrightarrow \quad \varphi'(x)\downarrow \ \text{ and } \ \varphi(x) = \varphi'(x).$$

Note that $\sqsubseteq$ is a partial order over the set of partial functions of the same type, where the totally divergent function is the bottom element.

Notation 4.9.2. Let $\varphi \equiv \varphi_1, \ldots, \varphi_m$ and $\varphi' \equiv \varphi'_1, \ldots, \varphi'_m$ be tuples of functions. We write $\varphi \sqsubseteq \varphi'$ to mean that $\varphi_i \sqsubseteq \varphi'_i$ for $i = 1, \ldots, m$.

Below, φ and φ' are two interpretations of the oracle tuple ϕ.

Lemma 4.9.3. *Consider statement S with oracles ϕ. If $\varphi \sqsubseteq \varphi'$, then*

$$\langle\!| \boldsymbol{First}(S) |\!\rangle^{A^*}_\varphi \ \sqsubseteq \ \langle\!| \boldsymbol{First}(S) |\!\rangle^{A^*}_{\varphi'}.$$

Proof. By definition, $\boldsymbol{First}(S)$ is an atomic statement. We have three cases:

(a) $\boldsymbol{First}(S) \equiv \mathsf{skip}$.

(b) $\boldsymbol{First}(S) \equiv \mathtt{x} := t$.

(c) $\boldsymbol{First}(S) \equiv \mathtt{x} := \phi_i(t)$.

Cases (a) and (b) are trivial, while Case (c) follows directly from condition $\varphi \sqsubseteq \varphi'$. $\square$

Lemma 4.9.4. *Consider statement S with oracles ϕ. If $\varphi \sqsubseteq \varphi'$, then*[1]

$$\boldsymbol{Rest}_{\varphi}^{A^*}(S, \cdot\,) \;\sqsubseteq\; \boldsymbol{Rest}_{\varphi'}^{A^*}(S, \cdot\,).$$

Proof. By induction on the complexity of S. Let σ be an arbitrary state. (Recall the definition of $\boldsymbol{Rest}^{A^*}$ in §4.6.)

(a) For S atomic,

$$\begin{aligned} \boldsymbol{Rest}_{\varphi}^{A^*}(S,\sigma) &\equiv \mathsf{skip} \\ &\equiv \boldsymbol{Rest}_{\varphi'}^{A^*}(S,\sigma). \end{aligned}$$

(b) $S \;\equiv\; S_1; S_2$. If $\boldsymbol{Rest}_{\varphi}^{A^*}(S,\sigma)\!\downarrow$ then

$$\begin{aligned} \boldsymbol{Rest}_{\varphi}^{A^*}(S,\sigma) \;&=\; \begin{cases} S_2 & \text{if } \ S_1 \text{ is atomic} \\ \boldsymbol{Rest}_{\varphi}^{A^*}(S_1,\sigma); S_2 & \text{otherwise} \end{cases} \\ &\qquad \text{(by definition of } \boldsymbol{Rest}_{\varphi}^{A^*}\text{)} \\ &=\; \begin{cases} S_2 & \text{if } \ S_1 \text{ is atomic} \\ \boldsymbol{Rest}_{\varphi'}^{A^*}(S_1,\sigma); S_2 & \text{otherwise} \end{cases} \\ &\qquad \text{(by i.h.)} \\ &=\; \boldsymbol{Rest}_{\varphi'}^{A^*}(S,\sigma) \\ &\qquad \text{(by definition of } \boldsymbol{Rest}_{\varphi'}^{A^*}\text{).} \end{aligned}$$

(c) $S \;\equiv\; \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}.$

Note that $[\![b]\!]_{\varphi}^{A^*}\sigma \simeq [\![b]\!]_{\varphi'}^{A^*}\sigma$. If $\boldsymbol{Rest}_{\varphi}^{A^*}(S,\sigma)\!\downarrow$ then

[1]Here we use the notation $f(x, \cdot\,)$ for $\lambda y \cdot f(x,y)$.

$$\begin{aligned}
\boldsymbol{Rest}_{\varphi}^{A^*}(S,\sigma) &= \begin{cases} S_1 & \text{if } [\![b]\!]_{\varphi}^{A^*}\sigma = \mathrm{tt} \\ S_2 & \text{if } [\![b]\!]_{\varphi}^{A^*}\sigma = \mathrm{ff} \end{cases} \\
&\quad \text{(by definition of } \boldsymbol{Rest}_{\varphi}^{A^*}\text{)} \\
&= \begin{cases} S_1 & \text{if } [\![b]\!]_{\varphi'}^{A^*}\sigma = \mathrm{tt} \\ S_2 & \text{if } [\![b]\!]_{\varphi'}^{A^*}\sigma = \mathrm{ff} \end{cases} \\
&\quad \text{(since } [\![b]\!]_{\varphi}^{A^*}\sigma = [\![b]\!]_{\varphi'}^{A^*}\sigma\text{)} \\
&= \boldsymbol{Rest}_{\varphi'}^{A^*}(S,\sigma) \\
&\quad \text{(by definition of } \boldsymbol{Rest}_{\varphi'}^{A^*}\text{).}
\end{aligned}$$

(*d*) $S \equiv \mathtt{x} := P_i(t)$
$(P_i \Longleftarrow R_i,\ \ R_i \equiv \langle D_i^{\mathsf{p}} : D_i^{\mathsf{v}} : S_i \rangle$
where $D_i^{\mathsf{v}} \equiv \mathsf{in\ a\ out\ b\ aux\ c}$.
If $\boldsymbol{Rest}_{\varphi}^{A^*}(S,\sigma)\downarrow$ then

$$\begin{aligned}
\boldsymbol{Rest}_{\varphi}^{A^*}(S,\sigma) &= \hat{S}_i \qquad \text{(by definition of } \boldsymbol{Rest}_{\varphi}^{A^*}\text{)} \\
&= \boldsymbol{Rest}_{\varphi'}^{A^*}(S,\sigma) \quad \text{(by definition of } \boldsymbol{Rest}_{\varphi'}^{A^*}\text{).}
\end{aligned}$$

□

Lemma 4.9.5. *Consider statement S with oracles ϕ. If $\varphi \sqsubseteq \varphi'$, then*

$$\boldsymbol{Comp}_{\varphi}^{A^*}(S,\cdot) \ \sqsubseteq\ \boldsymbol{Comp}_{\varphi'}^{A^*}(S,\cdot).$$

Proof. By induction on n. Let σ be arbitrary state.
Base case: $n = 0$,

$$\begin{aligned}
\boldsymbol{Comp}_{\varphi}^{A^*}(S,\sigma,n) &= \sigma \\
&= \boldsymbol{Comp}_{\varphi'}^{A^*}(S,\sigma,n)
\end{aligned}$$

Induction Step: suppose $\boldsymbol{Comp}_{\varphi}^{A^*}(S,\sigma,n+1)\downarrow$,

$$\begin{aligned}
\boldsymbol{Comp}_{\varphi}^{A^*}(S,\sigma,n+1) &= \boldsymbol{Comp}_{\varphi}^{A^*}(\boldsymbol{Rest}_{\varphi}^{A^*}(S,\sigma), \langle\!|\boldsymbol{First}(S)|\!\rangle_{\varphi}^{A^*}\sigma, n) \\
&= \boldsymbol{Comp}_{\varphi'}^{A^*}(\boldsymbol{Rest}_{\varphi'}^{A^*}(S,\sigma), \langle\!|\boldsymbol{First}(S)|\!\rangle_{\varphi'}^{A^*}\sigma, n) \\
&\quad \text{(by i.h. and Lemmas 4.9.4 and 4.9.3)} \\
&= \boldsymbol{Comp}_{\varphi'}^{A^*}(S,\sigma,n+1)
\end{aligned}$$

$\square$

Lemma 4.9.6. *Consider statement S with oracles ϕ. If $\varphi \sqsubseteq \varphi'$, then*

$$[\![S]\!]^{A^*}_{\varphi} \ \sqsubseteq \ [\![S]\!]^{A^*}_{\varphi'}.$$

Proof. From Lemma 4.9.5 $\square$

Theorem 4.9.7 (Monotonicity Theorem for ***RelRec*** procedures)**.** *Let R be a **RelRec** procedure with oracles ϕ. If $\varphi \sqsubseteq \varphi'$, then*

$$[\![R]\!]^{A}_{\varphi}(x) \sqsubseteq [\![R]\!]^{A}_{\varphi'}(x).$$

Proof. Suppose $R \equiv \langle D^{\mathsf{p}} : D^{\mathsf{v}} : S \rangle$.

Let σ be any state such that $\sigma[\mathsf{a}] = x$. By definition of semantics of procedures

$$[\![R]\!]^{A}_{\varphi}(x) \ = \ \begin{cases} \sigma_1[\mathsf{b}] & \text{if} \ \ [\![S]\!]^{A^*}_{\varphi}\sigma{\downarrow}\sigma_1 \\ \uparrow & \text{if} \ \ [\![S]\!]^{A^*}_{\varphi}\sigma{\uparrow}. \end{cases}$$

and

$$[\![R]\!]^{A}_{\varphi}(x) \ = \ \begin{cases} \sigma_2[\mathsf{b}] & \text{if} \ \ [\![S]\!]^{A^*}_{\varphi'}\sigma{\downarrow}\sigma_2 \\ \uparrow & \text{if} \ \ [\![S]\!]^{A^*}_{\varphi'}\sigma{\uparrow}. \end{cases}$$

If $[\![R]\!]^{A}_{\varphi}(x){\downarrow}$, by Lemma 4.9.6, $[\![S]\!]^{A^*}_{\varphi}\sigma \ = \ [\![S]\!]^{A^*}_{\varphi'}\sigma$, in other words $\sigma_1 \ = \ \sigma_2$. Hence, $\sigma_1[\mathsf{b}] = \sigma_2[\mathsf{b}]$, and $[\![R]\!]^{A}_{\varphi}(x) = [\![R]\!]^{A}_{\varphi'}(x)$. $\square$

4.10 $\boldsymbol{Rec}_2$ computability

We will extend ***Rec*** to a second-order programming language $\boldsymbol{Rec}_2$ with the following syntax extensions:

- A class of *function variables* $\phi_1, \ \phi_2, \ \ldots$, with corresponding types $\tau_1, \ \tau_2, \ \ldots$.
- A new program term constructor

  $$t^s ::= \ \ldots \ | \ \phi(t^u)$$

  where $\phi : u \to s$, $t^u : u$ and $t^s : s$.

- A *function variables declaration*

$$D^{\mathsf{f}} ::= \mathsf{functions}\ \phi$$

where $\phi \equiv \phi_1, \ldots, \phi_m$ is a tuple of function symbols and $m \geq 0$.

- The procedure call has the more general form

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

where $T \equiv T_1, \ldots, T_m$ is a tuple of function instances and $0 \leq m$. Note that each T_i $(0 \leq i \leq m)$ is either a function variable declared in the current procedure, or a primitive function symbol F_k. (For a discussion of an alternative, more complicated form of the procedure call statements, see §6.11).

Notation 4.10.1. We will use the notation $\bar{R}, \ldots$ for ***Rec***$_2$ procedures.
Following is a general form of a ***Rec***$_2$ procedure:
$\bar{R} \equiv$

functions ϕ
$P_1 \Longleftarrow R_1, \ldots, P_n \Longleftarrow R_n$
in a out b aux c
S

Remark 4.10.2. Note the differences between ***RelRec*** and ***Rec***$_2$:

(a) In ***RelRec*** the function symbols ϕ are interpreted as (oracles for) function parameters, while in ***Rec***$_2$ they are interpreted as function inputs.

(b) In ***RelRec*** the oracle declaration is global, and inner procedures have no oracle declaration, and so have type level 1; while in ***Rec***$_2$ each procedure can have its own function symbol declaration, and so may have type level 2.

The semantic functions for terms will depend on the interpretations of the function variables. We will use the notation $[\![t]\!]^A_\varphi$ for the semantic function of t when function variables ϕ in t are interpreted as φ. The definitions are similar to those in §4.4 except that we need to give the semantics of the new term constructor as follows:

$$[\![\phi(t_1, \ldots, t_m)]\!]^A_\varphi \sigma \ \simeq \ \begin{cases} \varphi([\![t_1]\!]^A\sigma, \ldots, [\![t_m]\!]^A\sigma) & \text{if } \ [\![t_i]\!]^A\sigma \downarrow \ (1 \leq i \leq m) \\ \uparrow & \text{otherwise.} \end{cases}$$

Similar as for ***RelRec***, we will have functions $\boldsymbol{Rest}_{\varphi}^{A^*}$, $\boldsymbol{Comp}_{\varphi}^{A^*}$, $\boldsymbol{CompLength}_{\varphi}^{A^*}$, and $[\![S]\!]_{\varphi}^{A^*}$ depending on the interpretation of oracles.

Therefore, the semantics of a $\boldsymbol{Rec}_2$ procedure $\bar{R}: \ \pi \times u \to v$, where $\bar{R} \equiv$

functions ϕ
$P_1 \Longleftarrow R_1, \ldots, P_n \Longleftarrow R_n$
in a out b aux c
S

(where $\phi : \pi$ and $\mathsf{a} : u$) is a functional

$$[\![\bar{R}]\!]^A : \ A^{\pi} \times A^u \stackrel{\cdot}{\longrightarrow} A^v$$

given by

$$[\![\bar{R}]\!]^A(\varphi, a) \ \simeq \ \begin{cases} \sigma'[\mathsf{b}] & \text{if } \ [\![S]\!]_{\varphi}^{A^*}\sigma \downarrow \sigma' \\ \uparrow & \text{if } \ [\![S]\!]_{\varphi}^{A^*}\sigma\uparrow. \end{cases}$$

where σ can be any state on A^* such that $\sigma[\mathsf{a}] = a$.

In this way we can define the notion of $\boldsymbol{Rec}_2$ *computability* and $\boldsymbol{Rec}_2^*$ *computability*.

We will prove (Theorem 4.10.5) a correspondence between ***RelRec*** and $\boldsymbol{Rec}_2$ computability. We need two lemmas.

Lemma 4.10.3 (***RelRec*** $\Rightarrow$ $\boldsymbol{Rec}_2$)**.** *Let R be a **RelRec** procedure of type $u \to v$ with oracle tuple ϕ of type π. We can transform R to a $\boldsymbol{Rec}_2$ procedure $\bar{R}$ of type $\pi \times u \to v$ such that for all $\varphi : \pi$ and $x : u$,*

$$[\![\bar{R}]\!]^A(\varphi, x) \ \simeq \ [\![R]\!]_{\varphi}^A(x).$$

Proof. (This is the easy direction). The transformation consists of re-interpreting the *oracle* declaration of R as a *function* declaration and adding the same function variable declaration "functions ϕ" to every inner procedure of R. Some points to be notes are:

(1) The oracle call statement $\mathtt{x} := \phi(t)$ is re-interpreted as an assignment statement.

(2) The new function variable declaration for any inner procedures has the same form as the main function variable declaration. This guarantees that ϕ_i in any inner procedures has the same interpretation as ϕ_i in the main procedure.

(3) Some new function variable declaration for inner procedures may be redundant in the sense that the function variables are not used in the body of the procedure; however, this does no harm.

$\square$

Lemma 4.10.4 (***Rec***$_2$ $\Rightarrow$ ***RelRec***)**.** *Let $\bar{R}$ be a **Rec**$_2$ procedure of type $\pi \times u \to v$. We can transform $\bar{R}$ to a **RelRec** procedure R of type $u \to v$ with oracle ϕ of type π such that for all $\varphi : \pi$ and $x : u$,*

$$[\![\bar{R}]\!]^A(\varphi, x) \;\simeq\; [\![R]\!]^A_\varphi(x).$$

Proof. The idea of this transformation is fairly simple, but it is complicated to write out in detail. We therefore illustrate the transformation by some simple examples, which we believe will make the general situation clear. There are two main points to consider.

(1) (**Interpreting assignment as oracle calls**) In $\bar{R}$ the new term constructor makes it possible that a term t has as a subterm a function application which is not allowed in R. The following example illustrate how to eliminate such a function application within a term. Consider an assignment statement

$$\mathtt{x} := \mathsf{F}_k(\phi(t')),$$

where F_k is a primitive function symbol. We replace the assignment statement by sequence of statements

$$\mathtt{z} := t';\ \mathtt{y} := \phi(\mathtt{z});\ \mathtt{x} := \mathsf{F}_k(\mathtt{y}),$$

where `y` and `z` are two newly introduced variables disjoint from the variables currently declared. This procedure is then repeated if necessary for the term tuple t', and so on. The method can also be generalized to the case that t occurs in other contexts, such as boolean tests.

(2) (**Interpreting inner function variable declarations**) Consider the following $\boldsymbol{Rec}_2$ procedure $\bar{R} \equiv$

functions $\phi_1,\ \phi_2$
$P' \Longleftarrow \bar{R}'$
in a out b aux c
$\dots$ $\mathtt{x}_1 := P'(\phi_1, t_1);$ $\dots$ $\mathtt{x}_2 := P'(\phi_2, t_2);$ $\dots$ $\mathtt{x}_3 := P'(\mathsf{F}_k, t_3);$ $\dots$

where $\bar{R}' \equiv$

functions ρ
in a$'$ out b$'$ aux c$'$
S

We can see that ρ is being interpreted as three different functions, corresponding to the interpretations of ϕ_1, ϕ_2 and the primitive function symbol F_k respectively. We can transform the above $\boldsymbol{Rec}_2$ procedure to a $\boldsymbol{RelRec}$ procedure

$R \equiv$

<table>
<tr><td>oracles $\phi_1,\ \phi_2$</td></tr>
<tr><td>$P_1 \Longleftarrow R_1, P_2 \Longleftarrow R_2, P_3 \Longleftarrow R_3$</td></tr>
<tr><td>in a out b aux c</td></tr>
<tr><td>$\dots$
$\mathtt{x}_1 := P_1(t_1);$
$\dots$
$\mathtt{x}_2 := P_2(t_2);$
$\dots$
$\mathtt{x}_3 := P_3(t_3);$
$\dots$</td></tr>
</table>

where for $i = 1, 2, 3,\ R_i \equiv$

<table>
<tr><td></td></tr>
<tr><td>in a′ out b′ aux c′</td></tr>
<tr><td>S_i</td></tr>
</table>

and S_1, S_2 and S_3 are obtained from S by replacing all occurrences of ρ by ϕ_1, ϕ_2 and F_k respectively.

This technique can be extended to cover all possible cases, because we have only finitely many function variables declared and finitely many primitive function symbols.

With this techniques, we can eliminate all inner function variable declarations by instantiating the function variables in the inner procedures either as the function variables in the main procedure (which, in turn, are re-interpreted as oracles) or as primitive function symbols.

□

From Lemmas 4.10.3 and 4.10.4 immediately follows:

Theorem 4.10.5. *Let* $F : A^\pi \times A^u \stackrel{\cdot}{\longrightarrow} A^v$ *be a second-order functional. F is computable by a* $\boldsymbol{Rec}_2$ *procedure* $\bar{R}$ *iff there exist a* ***RelRec*** *procedure* R *such that for all* $\varphi : \pi$ *and* $x : u$,

$$F(\varphi, x) \;\simeq\; [\![\bar{R}]\!]^A(\varphi, x) \;\simeq\; [\![R]\!]^A_\varphi(x).$$

4.11 *While* procedures

The syntax of the language $\boldsymbol{While}(\Sigma)$ is like $\boldsymbol{Rec}(\Sigma)$, except that $\boldsymbol{While}(\Sigma)$ contains a loop statement instead of the procedure call statement. In short, statements S in $\boldsymbol{While}(\Sigma)$ are defined by:

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

The semantics of ***While*** procedures are derived along similar lines as those for the semantics of ***Rec*** procedures. Details can be found in [TZ00].

Definition 4.11.1 (***While*** computable functions)**.** (a) A function f on A is *computable on A by a* ***While*** *procedure P* if $f = [\![P]\!]^A$. It is ***While*** *computable on A* if it is computable on A by some ***While*** procedure.

(b) $\boldsymbol{While}(A)$ is the class of functions ***While*** computable on A.

Definition 4.11.2. A $\boldsymbol{While}^*(\Sigma)$ procedure is a $\boldsymbol{While}(\Sigma^*)$ procedure in which the *input* and *output* variables are *simple*. (However the auxiliary variables may be starred.)

Definition 4.11.3 (***While**** computable functions)**.** (a) A function f on A is *computable on A by a* ***While**** *procedure P* if $f = [\![P]\!]^A$. It is ***While**** *computable on A* if it is computable on A by some ***While**** procedure.

(b) $\boldsymbol{While}^*(A)$ is the class of functions ***While**** computable on A.

We will not discuss ***While*** computability any further, since [TZ00] contains a full discussion. However, we need to mention following significant theorem proved in [TZ88].

Theorem 4.11.4. (a) $\boldsymbol{While}(A) \;=\; \mu\boldsymbol{PR}(A)$.

(b) $\boldsymbol{While}^*(A) \;=\; \mu\boldsymbol{PR}^*(A)$.

Chapter 5

From $\mu\boldsymbol{PR}$ to $\boldsymbol{ACP}$

In this chapter, we will prove that, if a function f on A is $\mu\boldsymbol{PR}$ computable, then it is $\boldsymbol{ACP}$ computable; and hence, if f is $\mu\boldsymbol{PR}^*$ computable, it is $\boldsymbol{ACP}^*$ computable. We will prove the theorem by structural induction on the the schemes of $\mu\boldsymbol{PR}$, *i.e.* associate with every $\mu\boldsymbol{PR}$ scheme an $\boldsymbol{ACP}$ scheme. Even though we gave formal definitions for $\boldsymbol{ACP}$ and $\mu\boldsymbol{PR}$ schemes in Chapter 3, we prefer to present informal proofs in this chapter, in the sense that we ignore the distinction between syntax and semantics for both $\boldsymbol{ACP}$ and $\mu\boldsymbol{PR}$. We believe that our informal approach is convincing.

Lemma 5.1. *Let f, g, h, and p be functions defined respectively by*

(a) $f(x) \simeq F_k(x)$ *where* $\mathsf{F}_k \in \boldsymbol{Func}(\Sigma)$

(b) $g(\vec{x}) = x_i$

(c) $h(x) \simeq \begin{cases} h_2(x) & \text{if } h_1(x) \downarrow \mathtt{tt} \\ h_3(x) & \text{if } h_1(x) \downarrow \mathtt{ff} \\ \uparrow & \text{if } h_1(x) \uparrow \end{cases}$,

then f, g, h, and $p \in \boldsymbol{ACP}(A)$, provided h_1, h_2, and $h_3 \in \boldsymbol{ACP}(A)$.

Proof. (a) By $f(x) \simeq F_k(x)$ (scheme I in $\boldsymbol{ACP}$);

(b) By $g(x) \simeq g'(x_i)$, and $g'(x_i) = x_i$ (scheme V and II);

(c) By $h(x) \simeq h'(h_1(x), x)$, and

$$h'(b,x) \simeq \begin{cases} h_2(x) & \text{if } \ b = \mathtt{tt} \\ h_3(x) & \text{if } \ b = \mathtt{ff} \end{cases}$$

(schemes VI and IV).

$\square$

Lemma 5.2. *Let f be a function defined by $f(x) \simeq h(g_1(x), \ldots, g_m(x))$.*

If $h, g_1, \ldots, g_m \in \boldsymbol{ACP}(A)$, then so is f.

Proof. Directly follows from scheme VI in $\boldsymbol{ACP}$. $\square$

Lemma 5.3. *Let $f_1, \ldots, f_m$ be functions defined by*

$$\begin{array}{rcl} f_1(x,0) & \simeq & g_1(x) \\ & \ldots, & \\ f_m(x,0) & \simeq & g_m(x) \\ f_1(x,z+1) & \simeq & h_1(x,z,f_1(x,z),\ldots,f_m(x,z)) \\ & \ldots, & \\ f_m(x,z+1) & \simeq & h_m(x,z,f_1(x,z),\ldots,f_m(x,z)). \end{array}$$

If $g_1, \ldots, g_m$, $h_1, \ldots, h_m \in \boldsymbol{ACP}(A)$, so are $f_1, \ldots, f_m$.

Proof. Let

$$\begin{array}{rcl} \varphi_{x,1} & =_{df} & \lambda z \cdot f_1(x,z) \\ & \ldots, & \\ \varphi_{x,m} & =_{df} & \lambda z \cdot f_m(x,z); \end{array}$$

and

$$\begin{array}{rcl} F_{\varphi,x,1}(z) & \simeq & \begin{cases} g_1(x) & \text{if } \ z{=}0 \\ h_1(x, z-1, \varphi_1(z-1), \ldots, \varphi_m(z-1)) & \text{otherwise} \end{cases} \\ & \ldots, & \\ F_{\varphi,x,m}(z) & \simeq & \begin{cases} g_m(x) & \text{if } \ z{=}0 \\ h_m(x, z-1, \varphi_1(z-1), \ldots, \varphi_m(z-1)) & \text{otherwise} \end{cases} \end{array}$$

where $\varphi \equiv \varphi_1, \ldots, \varphi_m$, and

$$\begin{array}{lll} \widehat{F}_{x,1} & =_{df} & \lambda\varphi_1 \cdot \ldots \cdot \lambda\varphi_m \cdot F_{\varphi,x,1} \\ & \ldots, & \\ \widehat{F}_{x,m} & =_{df} & \lambda\varphi_1 \cdot \ldots \cdot \lambda\varphi_m \cdot F_{\varphi,x,m}. \end{array}$$

Note that $F_{x,1}, \ldots, F_{x,m}$ are ***ACPs*** by scheme IV. We will show that

$$(\varphi_{x,1}, \ldots, \varphi_{x,m}) \;=\; LFP(\widehat{F}_{x,1}, \ldots, \widehat{F}_{x,m}).$$

(i) $(\varphi_{x,1}, \ldots, \varphi_{x,m})$ are *fixed points* of $(\widehat{F}_{x,1}, \ldots, \widehat{F}_{x,m})$, since for $1 \le i \le m$,

$$\begin{array}{ll} \widehat{F}_{x,i}(\varphi_{x,1}, \ldots, \varphi_{x,m})(z) & \\ \simeq & \begin{cases} g_i(x) & \text{if } z{=}0 \\ h_i(x, z-1, \varphi_{x,1}(z-1), \ldots, \varphi_{x,m}(z-1)) & \text{otherwise} \end{cases} \qquad (5.1) \\ \simeq & \begin{cases} g_i(x) & \text{if } z{=}0 \\ h_i(x, z-1, f_1(x, z-1), \ldots, f_m(x, z-1)) & \text{otherwise} \end{cases} \\ \simeq & f_i(x, z) \\ \simeq & \varphi_{x,i}(z). \end{array}$$

(ii) $(\varphi_{x,1}, \ldots, \varphi_{x,m})$ are the *least fixed points* of $(\widehat{F}_{x,1}, \ldots, \widehat{F}_{x,m})$.

Put

$$\begin{array}{lll} \varphi^0_{x,1} & \simeq & \lambda z \cdot \bot \\ & \ldots, & \\ \varphi^0_{x,m} & \simeq & \lambda z \cdot \bot \\ \varphi^{k+1}_{x,1} & \simeq & \widehat{F}_{x,1}(\varphi^k_{x,1}, \ldots, \varphi^k_{x,m}) \\ & \ldots, & \\ \varphi^{k+1}_{x,m} & \simeq & \widehat{F}_{x,m}(\varphi^k_{x,1}, \ldots, \varphi^k_{x,m}). \end{array}$$

Suppose $(\psi_1, \ldots, \psi_m)$ is an arbitrary fixed point of $(\widehat{F}_{x,1}, \ldots, \widehat{F}_{x,m})$. We first show that for $1 \le i \le m$,

$$\bigsqcup_{k=0}^{\infty} \varphi^k_{x,i} \sqsubseteq \psi_i. \qquad (5.2)$$

It is sufficient to show that for all k,

$$\varphi^k_{x,i} \sqsubseteq \psi_i.$$

We prove this by induction on k. The base case is trivial, since $\varphi^0_{x,i}$ is the totally undefined function. Induction step:

$$\begin{aligned}
\varphi^{k+1}_{x,i} &= \widehat{F}_{x,i}(\varphi^k_{x,1}, \ldots, \varphi^k_{x,m}) \\
&\sqsubseteq \widehat{F}_{x,i}(\psi_1, \ldots, \psi_m) \quad \text{(by i.h. and the monotonicity of } \widehat{F}_{x,i}) \\
&= \psi_i \quad \text{(since } \psi_1, \ldots, \psi_m \text{ are fixed points of } \widehat{F}_{x,1}, \ldots, \widehat{F}_{x,m}).
\end{aligned}$$

Now we prove for $1 \leq i \leq m$,

$$\varphi_{x,i} = \bigsqcup_{k=0}^{\infty} \varphi^k_{x,i} \tag{5.3}$$

which make $\varphi_{x,1}, \ldots, \varphi_{x,m}$ the *least fixed points* of $\widehat{F}_{x,1}, \ldots, \widehat{F}_{x,m}$.

The inclusion $\bigsqcup_{k=0}^{\infty} \varphi^k_{x,i} \sqsubseteq \varphi_{x,i}$ follows from (5.2). In order to prove $\varphi_{x,i} \sqsubseteq \bigsqcup_{k=0}^{\infty} \varphi^k_{x,i}$, we prove for any $z \in \mathbb{N}$,

$$\varphi_{x,i}(z)\downarrow \implies \exists k : \mathbb{N} \cdot [\varphi_{x,i}(z) = \varphi^k_{x,i}(z)]. \tag{5.4}$$

We will show for any $z \in \mathbb{N}$,

$$\varphi_{x,i}(z) \simeq \varphi^{z+1}_{x,i}(z).$$

by induction on z.

Base case: $z = 0$.

$\varphi_{x,i}(0) \simeq f_i(x, 0) \simeq g_i(x)$;

$$\begin{aligned}
\varphi^1_{x,i}(0) &\simeq \widehat{F}_{x,i}(\varphi^0_{x,1}, \ldots, \varphi^0_{x,m})(0) \\
&\simeq g_i(x) \qquad \text{by (5.1).}
\end{aligned}$$

Induction step:

Assume, for $1 \leq i \leq m$

$$\varphi_{x,i}(z) \simeq \varphi^{z+1}_{x,i}(z).$$

We must show

$$\varphi_{x,i}(z+1) \simeq \varphi^{z+2}_{x,i}(z+1).$$

$$
\begin{aligned}
\varphi_{x,i}^{z+2}(z+1) &\simeq \widehat{F}_{x,i}(\varphi_{x,1}^{z+1},\ldots,\varphi_{x,m}^{z+1})(z+1)\\
&\simeq \begin{cases} g_i(x) & \text{if } \ \text{(z+1)=0}\\ h_i(x,z,\varphi_{x,1}^{z+1}(z),\ldots,\varphi_{x,m}^{z+1}(z)) & \text{otherwise}\end{cases}\\
&\simeq h_i(x,z,\varphi_{x,1}^{z+1}(z),\ldots,\varphi_{x,m}^{z+1}(z))\\
&\simeq h_i(x,z,\varphi_{x,1}(z),\ldots,\varphi_{x,m}(z)) \quad \text{by induction hypothesis.}\\
&\simeq h_i(x,z,f_1(x,z),\ldots,f_m(x,z))\\
&\simeq f_i(x,z+1)\\
&\simeq \varphi_{x,i}(z+1).
\end{aligned}
$$

This proves (5.4) and hence (5.3), as required □

Lemma 5.4. *Let f be a function defined by $f(x) \simeq \mu z[g(x,z) = \mathtt{tt}]$.*

*If $g \in$ **ACP**(A), so is f.*

Proof. Define (using informal but suggestive notation) the function

$$f'(x,z) \ \simeq \ \mu y \geq z[g(x,y) = \mathtt{tt}].$$

Note that

$$f'(x,z) \simeq \begin{cases} z & \text{if } \ g(x,z)=\mathtt{tt}\\ f'(x,z+1) & \text{if } \ g(x,z)=\mathtt{ff}\\ \uparrow & \text{otherwise.}\end{cases}$$

Clearly, $f(x) \simeq f'(x,0)$. Now, we can prove that f' is ***ACP***, provided g is.

Put

$$
\begin{aligned}
\varphi_x &= \lambda z \cdot f'(x,z)\\
F_{\varphi,x}(z) &\simeq \begin{cases} z & \text{if } \ g(x,z)=\mathtt{tt}\\ \varphi(z+1) & \text{if } \ g(x,z)=\mathtt{ff}\\ \uparrow & \text{otherwise}\end{cases}\\
\widehat{F}_x &= \lambda\varphi \cdot F_x.
\end{aligned}
$$

We claim that

$$\varphi_x \ = \ LFP(\widehat{F}_x).$$

By hypothesis g is $\boldsymbol{ACP}$, therefore, f' is $\boldsymbol{ACP}$ derived from g by means of schemes VIII, IV, and V.

Now, we prove that

$$\varphi_x \;=\; LFP(\widehat{F}_x).$$

(i) φ_x is a fixed point of $\widehat{F}_x$, since

$$\begin{aligned}
&\widehat{F}_x(\varphi_x)(z)\\
&\simeq \begin{cases} z & \text{if } \; g(x,z)=\mathtt{tt} \\ \varphi_x(z+1) & \text{if } \; g(x,z)=\mathtt{ff} \\ \uparrow & \text{otherwise} \end{cases}\\
&\simeq \begin{cases} z & \text{if } \; g(x,z)=\mathtt{tt} \\ f'(x,z+1) & \text{if } \; g(x,z)=\mathtt{ff} \\ \uparrow & \text{otherwise} \end{cases}\\
&\simeq f'(x,z)\\
&\simeq \varphi_x(z).
\end{aligned}$$

(ii) φ_x is the least fixed point of $\widehat{F}_x$.

Put

$$\begin{aligned}
\varphi_x^0 &= \lambda z \cdot \bot \\
&\ldots, \\
\varphi_x^{k+1} &= \widehat{F}_x(\varphi_x^k).
\end{aligned}$$

Suppose ψ is an arbitrary fixed point of $\widehat{F}_x$. We first show

$$\bigsqcup_{k=0}^{\infty} \varphi_x^k \sqsubseteq \psi. \tag{5.5}$$

It is sufficient to show that for all k,

$$\varphi_x^k \sqsubseteq \psi.$$

We prove this by induction on k. The base case is trivial, since φ_x^0 is the totally undefined function. Induction step:

$$\begin{aligned}
\varphi_x^{k+1} &= \widehat{F}_x(\varphi_x^k)\\
&\sqsubseteq \widehat{F}_x(\psi) \quad \text{(by i.h. and the monotonicity of } \widehat{F}_x)\\
&= \psi \quad \text{(since } \psi \text{ is the fixed point of } \widehat{F}_x).
\end{aligned}$$

Now we prove

$$\varphi_x \;=\; \bigsqcup_{k=0}^{\infty} \varphi_x^k \tag{5.6}$$

which make φ_x the *least fixed point* of $\widehat{F}_x$.

The inclusion $\bigsqcup_{k=0}^{\infty} \varphi_x^k \sqsubseteq \varphi_x$ follows from (5.5). In order to prove $\varphi_x \sqsubseteq \bigsqcup_{k=0}^{\infty} \varphi_x^k$, we prove for any $z \in \mathbb{N}$,

$$\varphi_x(z)\downarrow \implies \exists k : \mathbb{N} \cdot [\varphi_x(z) = \varphi_x^k(z)]. \tag{5.7}$$

We will show for any $z \in \mathbb{N}$

$$\text{if } \varphi_x(z) = k, \text{ then } \varphi_x^{k+1}(z) = k$$

by induction on z.

By hypothesis $\varphi_x(z) = k$. Clearly, $g(x, k) = \mathtt{tt}$ and $\forall_{\, z \leq z' < k} \cdot [g(x, z') = \mathtt{ff}]$. So

$$\begin{aligned}
\varphi_x^{k+1}(z) \;&\simeq\; \widehat{F}_x(\varphi_x^k)(z) \\
&\simeq\; \begin{cases} z & \text{if } \; g(x,z)=\mathtt{tt} \\ \varphi_x^k(z+1) & \text{if } \; g(x,z)=\mathtt{ff} \\ \uparrow & \text{otherwise} \end{cases} \\
&\simeq\; \varphi_x^k(z+1) \qquad\quad (\text{if } g(x,z) = \mathtt{ff}) \\
&\simeq\; \begin{cases} z+1 & \text{if } \; g(x,z+1)=\mathtt{tt} \\ \varphi_x^{k-1}(z+2) & \text{if } \; g(x,z+1)=\mathtt{ff} \\ \uparrow & \text{otherwise} \end{cases} \\
&\ldots \\
&\simeq\; \begin{cases} k & \text{if } \; g(x,k)=\mathtt{tt} \\ \varphi_x^z(k+1) & \text{if } \; g(x,k)=\mathtt{ff} \\ \uparrow & \text{otherwise} \end{cases} \\
&=\; k.
\end{aligned}$$

This proves (5.7) and hence (5.6), as required. $\square$

Theorem 5.5. $\mu\boldsymbol{PR}(A) \subseteq \boldsymbol{ACP}(A)$.

Proof. We associate, with each $\mu\boldsymbol{PR}$ scheme for a function f, an $\boldsymbol{ACP}$ scheme for f, by structural induction on $\mu\boldsymbol{PR}$ schemes.

The result directly follows from Lemmas 5.1, 5.2, 5.3, and 5.4. $\square$

Corollary 5.6. $\mu\boldsymbol{PR}^*(A) \subseteq \boldsymbol{ACP}^*(A)$.

Proof. From Theorem 5.5. $\square$

Chapter 6

From $\boldsymbol{ACP}$ to $\boldsymbol{Rec}$

In this chapter, we will prove

$$\boldsymbol{ACP}(A) \subseteq \boldsymbol{Rec}_2(A).$$

We will prove this by induction on the schemes of $\boldsymbol{ACP}$, *i.e.* associate to every $\boldsymbol{ACP}$ scheme a $\boldsymbol{Rec}_2$ procedure for the same functional. From this will follow:

$$\boldsymbol{ACP}^1(A) \subseteq \boldsymbol{Rec}(A),$$

and hence

$$\boldsymbol{ACP}^{*1}(A) \subseteq \boldsymbol{Rec}^{*}(A).$$

Lemma 6.1. *Let* $F \equiv \mathsf{F}^A$, $G \equiv \mathsf{G}^A$, *and* $H \equiv \mathsf{H}^A$ *be functionals defined by*

(i) $\mathsf{F}(\varphi, x) \simeq \mathsf{F}_k(\varphi, x)$,

(ii) $\mathsf{G}(x) \simeq x$, *and*

(iii) $\mathsf{H}(\varphi, x) \simeq \varphi(x)$.

Then, F, G *and* H *are* $\boldsymbol{Rec}_2$*-computable.*

Proof. We can construct $\boldsymbol{Rec}_2$ procedures R_F, R_G and R_H as follows.

$R_F \equiv$

functions ϕ
in a_F out b_F
$\mathsf{b}_F := \mathsf{F}_k(\phi, \mathsf{a}_F)$

$R_G \equiv$

in a_G out b_G
$\mathsf{b}_G := \mathsf{a}_G$

$R_H \equiv$

functions ϕ
in a_H out b_H
$\mathsf{b}_H := \phi(\mathsf{a}_H)$

Clearly, $F = [\![R_F]\!]^A$, $G = [\![R_G]\!]^A$ and $H = [\![R_H]\!]^A$. $\square$

Lemma 6.2. *Let* $F \equiv \mathsf{F}^A$, $G \equiv \mathsf{G}^A$, *and* $H \equiv \mathsf{H}^A$ *be functionals, and let* F *be defined by*

$$\mathsf{F}(\varphi, x, b) \;\simeq\; [\text{if } b = \mathtt{tt} \text{ then } \mathsf{G}(\varphi, x) \text{ else } \mathsf{H}(\varphi, x)].$$

If G *and* H *are* ***Rec***$_2$*-computable, then so is* F*.*

Proof. By assumption, we have ***Rec***$_2$ procedures R_G and R_H as follows, such that $G = [\![R_G]\!]^A$ and $H = [\![R_H]\!]^A$.

$R_G \equiv$
We can construct a $\boldsymbol{Rec}_2$ procedure R_F as follows
$[\![R_F]\!]^A(\varphi, x, b) \simeq F(\varphi, x, b)$. The proof is obvious and details are omitted. $\square$

Lemma 6.3. *Let $F \equiv \mathsf{F}^A$ and $G \equiv \mathsf{G}^A$ be functionals, and let F be defined by*

$$\mathsf{F}(\varphi, x) \simeq \mathsf{G}(\varphi_f, x_g) \qquad \text{(refer to §3.1 for the meanings of } f \text{ and } g).$$

If G is $\boldsymbol{Rec}_2$-computable, then so is F.

Proof. By assumption, we have $\boldsymbol{Rec}_2$ procedures R_G as follows, such that $G = [\![R_G]\!]^A$.
$R_G \equiv$
By modifying R_G, We can construct a $\boldsymbol{Rec}_2$ procedure R_F as follows.
S_F is the same as S_G, except that we replace all occurrences of ϕ_i by $\phi_{f(i)}$, and all occurrences of a_{G_i} by $\mathsf{a}_{F_{g(i)}}$. Essentially, R_F permutes the function symbol tuple and the input tuple, therefore $[\![R_F]\!]^A(\varphi, x) \simeq F(\varphi, x)$. The proof is obvious and details are omitted. $\square$

Lemma 6.4. *Let $F \equiv \mathsf{F}^A$, $G \equiv \mathsf{G}^A$, and $H \equiv \mathsf{H}^A$ be functionals, and let F be defined by*

$$\mathsf{F}(\varphi, x) \simeq \mathsf{G}(\varphi, x, \mathsf{H}(\varphi, x)).$$

If G and H are $\boldsymbol{Rec}_2$-computable, then so is F.

Proof. By assumption, we have $\boldsymbol{Rec}_2$ procedures R_G and R_H as follows, such that $G = [\![R_G]\!]^A$ and $H = [\![R_H]\!]^A$.
We can construct a $\boldsymbol{Rec}_2$ procedure R_F as follows
$[\![R_F]\!]^A(\varphi, x) \simeq F(\varphi, x)$. The proof is obvious and details are omitted. $\square$

Lemma 6.5. *Let* $F_1 \equiv \mathsf{F}_1^A, \dots, F_n \equiv \mathsf{F}_n^A,\ G_1 \equiv \mathsf{G}_1^A, \dots, G_n \equiv \mathsf{G}_n^A$ *be functionals, and* $\mathsf{F}_1, \dots, \mathsf{F}_n$ *are defined by*

$$\mathsf{F}_1(\varphi, x, y_1) \simeq \varrho_1^{\varphi,x}(y_1)$$
$$\dots,$$
$$\mathsf{F}_n(\varphi, x, y_n) \simeq \varrho_n^{\varphi,x}(y_n)$$

where

$$(\varrho_1^{\varphi,x}, \dots, \varrho_n^{\varphi,x}) = \mathrm{LFP}(\hat{\mathsf{G}}_1^{\varphi,x}, \dots, \hat{\mathsf{G}}_n^{\varphi,x}).$$

If $G_1, \dots, G_n$ *are* ***Rec***$_2$*-computable, then so are* $F_1, \dots, F_n$.

Refer to Notation 3.1.3, for $\hat{\mathsf{G}}_i^{\varphi,x}$ and $\hat{\mathsf{G}}_i^{x}$ used above; and Notation 3.1.4, for $\hat{G}_i^{\varphi,x}$ and $\hat{G}_i^{x}$ used in the following proof.

Proof. By assumption, we have ***Rec***$_2$ procedures $R_{G_1}, \dots, R_{G_n}$ as follows such that, for $0 \le i \le n,\ G_i = [\![R_{G_i}]\!]^A$.

$R_{G_1} \equiv$

functions $\phi,\ \rho_1, \dots, \rho_n$
$P_{G_{1,1}} \Longleftarrow R_{G_{1,1}}, \dots, P_{G_{1,m_1}} \Longleftarrow R_{G_{1,m_1}}$
in $\mathsf{a}_{G_{1,1}}$ $\mathsf{a}_{G_{1,2}}$ out b_{G_1} aux c_{G_1}
S_{G_1}

...

$R_{G_n} \equiv$

functions $\phi,\ \rho_1, \dots, \rho_n$
$P_{G_{n,1}} \Longleftarrow R_{G_{n,1}}, \dots, P_{G_{n,m_n}} \Longleftarrow R_{G_{n,m_n}}$
in $\mathsf{a}_{G_{n,1}}$ $\mathsf{a}_{G_{n,2}}$ out b_{G_n} aux c_{G_n}
S_{G_n}

We can construct $\boldsymbol{Rec}_2$ procedures $R_{F1}, \ldots, R_{Fn}$ as follows.
$R_{F_1} \equiv$

$$\begin{array}{|l|}\hline \mathsf{functions}\ \phi \\ \hline P_{G_1} \Longleftarrow R'_{G_1}, \ldots, P_{G_n} \Longleftarrow R'_{G_n} \\ \hline \mathsf{in}\ \mathsf{a}_{F_{1,1}}\ \mathsf{a}_{F_{1,2}}\ \mathsf{out}\ \mathsf{b}_{F_1} \\ \hline \mathsf{b}_{F_1} := P_{G_1}(\phi,\ \mathsf{a}_{F_{1,1}},\ \mathsf{a}_{F_{1,2}}) \\ \hline \end{array}$$

$\ldots$
$R_{F_n} \equiv$

$$\begin{array}{|l|}\hline \mathsf{functions}\ \phi \\ \hline P_{G_1} \Longleftarrow R'_{G_1}, \ldots, P_{G_n} \Longleftarrow R'_{G_n} \\ \hline \mathsf{in}\ \mathsf{a}_{F_{n,1}}\ \mathsf{a}_{F_{n,2}}\ \mathsf{out}\ \mathsf{b}_{F_n} \\ \hline \mathsf{b}_{F_n} := P_{G_n}(\phi,\ \mathsf{a}_{F_{n,1}},\ \mathsf{a}_{F_{n,2}}) \\ \hline \end{array}$$

where, for $1 \leq i \leq n$, $R'_{G_i} \equiv$

$$\begin{array}{|l|}\hline \mathsf{functions}\ \phi \\ \hline P_{G_{i,1}} \Longleftarrow R^P_{G_{i,1}}, \ldots, P_{G_{i,m_i}} \Longleftarrow R^P_{G_{i,m_i}} \\ \hline \mathsf{in}\ \mathsf{a}_{G_{i,1}}\ \mathsf{a}_{G_{i,2}}\ \mathsf{out}\ \mathsf{b}_{G_i}\ \mathsf{aux}\ \mathsf{c}_{G_i} \\ \hline S^P_{G_i} \\ \hline \end{array}$$

Here, for $1 \leq i \leq n$, $R^P_{G_{i,1}}, \ldots, R^P_{G_{i,m_i}}$, and $S^P_{G_i}$ are the same as $R_{G_{i,1}}, \ldots, R_{G_{i,m_i}}$, and S_{G_i}, except that all occurrences of function application statements of the form $\mathsf{c} := \rho_j(t)$ are replaced by procedure calls $\mathsf{c} := P_{G_j}(\phi, \mathsf{a}_{G_{i,1}}, t)$. *We are, essentially, replacing function application statements by simultaneous recursive calls.* For details, we need to eliminate all declarations for ρ and all occurrences of ρ as procedure call inputs.

Remark 6.6. In this proof, we will only consider ***RelRec*** like $\boldsymbol{Rec}_2$ procedures in the sense that,

(a) a function symbol occurs either in a function application statement of form $\mathsf{c} := \phi(t)$ or in a procedure call statement as inputs;

(b) all function variable declarations for any inner procedures has the same form as the main function variable declaration, which guarantees that ϕ_i in any inner procedures has the same interpretation as ϕ_i in the main procedure. This jusifies the replacement of $\mathsf{c} := \rho_j(t)$ by $\mathsf{c} := P_{G_j}(\phi, \mathsf{a}_{G_{i,1}}, t)$ in the inner procedures.

This assumption is based on the fact that we can transform every ***Rec***$_2$ procedure to a ***RelRec*** like ***Rec***$_2$ procedure by the technique decribed in the proofs of Lemmas 4.10.4 and 4.10.3. We believe this makes this proof simpler and clearer.

We claim that, if ϕ are interpreted as φ and for $1 \le i \le n$, $\sigma[\mathsf{a}_{F_{i,1}}] = x$ and $\sigma[\mathsf{a}_{F_{i,2}}] = y_i$, then

$$[\![R_{F_i}]\!]^A(\varphi, x, y_i) \simeq F_i(\varphi, x, y_i). \tag{6.1}$$

In order to prove (6.1) we prove, for $1 \le i \le n$,

$$\lambda y_i \cdot F_i(\varphi, x, y_i) \sqsubseteq \lambda y_i \cdot [\![R_{F_i}]\!]^A(\varphi, x, y_i), \tag{6.2}$$
$$\lambda y_i \cdot [\![R_{F_i}]\!]^A(\varphi, x, y_i) \sqsubseteq \lambda y_i \cdot F_i(\varphi, x, y_i). \tag{6.3}$$

To prove (6.2): Putting

$$\begin{array}{rcl}
\varrho_1^0 & \equiv & \perp \\
\cdots & & \\
\varrho_n^0 & \equiv & \perp \\
\cdots & & \\
\varrho_1^{k+1} & \equiv & \hat{G}_1^{\varphi,x}(\varrho_1^k, \ldots, \varrho_n^k) \\
\cdots & & \\
\varrho_1^{k+1} & \equiv & \hat{G}_n^{\varphi,x}(\varrho_1^k, \ldots, \varrho_n^k)
\end{array}$$

By definition of least fixed points, it is sufficient to prove that,

$$\text{for all } k, \varrho_i^k \sqsubseteq \lambda y_i \cdot [\![R_{F_i}]\!]^A(\varphi, x, y_i), \text{for } 1 \le i \le n. \tag{6.4}$$

We will prove this by simultaneous induction on k.

Note first that by definition of procedure R_{G_i}, and interpreting ϕ, $\rho_1, \ldots, \rho_n$ as φ, $\varrho_1^k, \ldots, \varrho_n^k$, respectively, we get

$$[\![R_{G_i}]\!]^A(\varphi, \varrho_1^k, \ldots, \varrho_n^k, x, y_i) \simeq G_i(\varphi, \varrho_1^k, \ldots, \varrho_n^k, x, y_i) \simeq \varrho_i^{k+1}(y_i). \tag{6.5}$$

By induction hypothesis $\varrho_i^k \sqsubseteq \lambda y_i \cdot [\![R_{F_i}]\!]^A(\varphi, x, y_i)$, for $i = 1, \ldots, n$. Therefore by the monotonicity theorem (Thorem 4.9.7) of functions

$$\begin{aligned}&\lambda y_i \cdot [\![R_{G_i}]\!]^A(\varphi, \varrho_1^k, \ldots, \varrho_n^k, x, y_i) \;\sqsubseteq\\ &\lambda y_i \cdot [\![R_{G_i}]\!]^A(\varphi, \lambda y_1 \cdot [\![R_{F_1}]\!]^A(\varphi, x, y_1), \ldots, \lambda y_n \cdot [\![R_{F_n}]\!]^A(\varphi, x, y_n), x, y_i),\end{aligned}$$

for $i = 1, \ldots, n$.

So by (6.5) and Sublemma 6.7 below,

$$\varrho_i^{k+1} \sqsubseteq \lambda y_i \cdot [\![R_{F_i}]\!]^A(\varphi, x, y_i)$$

which proves (6.4) by induction on k, and hence (6.2).

The reverse direction (6.3) is proved by simultaneous course of values induction on $\boldsymbol{CompLength}(R,\varphi,a)$. Here, $\boldsymbol{CompLength}(R,\varphi,a)$ denotes the computation length of procedure R with inputs φ and a, defined by

$$\boldsymbol{CompLength}(R, \varphi, a) = \boldsymbol{CompLength}^A(S, \sigma)$$

where $R \equiv \langle D^{\mathsf{f}} : D^{\mathsf{p}} : D^{\mathsf{v}} : S\rangle$, $D^{\mathsf{f}} \equiv$ $\mathsf{functions}\ \phi$ where ϕ is intepreted as φ, $D^{\mathsf{v}} \equiv$ $\mathsf{in\ a\ out\ b\ aux\ c}$, and $\sigma[\mathsf{a}] = a$.

Assume that, for $1 \le i \le n$, for all inputs φ, x and y_i, if $\boldsymbol{CompLength}(R_{F_i}, \varphi, (x, y_i)) < l$, then

$$[\![R_{F_i}]\!]^A(\varphi, x, y_i)\downarrow \quad\Longrightarrow\quad [\![R_{F_i}]\!]^A(\varphi, x, y_i) = F_i(\varphi, x, y_i). \tag{6.6}$$

Suppose now that for some φ, x and y_i

$$[\![R_{F_i}]\!]^A(\varphi, x, y_i)\downarrow \text{ and } \boldsymbol{CompLength}(R_{F_i}, \varphi, (x, y_i)) = l.$$

By Sublemma 6.7 below and $[\![R_{F_i}]\!]^A(\varphi, x, y_i)\downarrow$, we have:

$$\begin{aligned}[\![R_{F_i}]\!]^A(\varphi, x, y_i) &= [\![R_{G_i}]\!]^A(\varphi, \lambda z_1 \cdot [\![R_{F_1}]\!]^A(\varphi, x, z_1), \ldots, \lambda z_n \cdot [\![R_{F_n}]\!]^A(\varphi, x, z_n), x, y_i)\\ &= G_i(\varphi, \lambda z_1 \cdot [\![R_{F_1}]\!]^A(\varphi, x, z_1), \ldots, \lambda z_n \cdot [\![R_{F_n}]\!]^A(\varphi, x, z_n), x, y_i).\end{aligned}$$

Clearly, within the computation for $[\![R_{F_i}]\!]^A(\varphi, x, y_i)$, $\lambda z_j \cdot [\![R_{F_j}]\!]^A(\varphi, x, z_j)$ (for $j = 1, \ldots, n$) will only be applied on some z (say) which is the value of some term t, however

$$\boldsymbol{CompLength}(R_{F_j}, \varphi, (x, z)) < \boldsymbol{CompLength}(R_{F_i}, \varphi, (x, y_i)) = l.$$

Therefore for all such z, by induction hypothesis

$$[\![R_{F_j}]\!]^A(\varphi, x, z) = F_j(\varphi, x, z)$$

This justifies the replacement of $\lambda z_j \cdot [\![R_{F_j}]\!]^A(\varphi, x, z_j)$ by $\lambda z_j \cdot F_j(\varphi, x, z_j)$ within the computation of $[\![R_{F_i}]\!]^A(\varphi, x, y_i)$ and hence

$$\begin{aligned} & [\![R_{F_i}]\!]^A(\varphi, x, y_i) \\ = \; & G_i(\varphi, \lambda z_1 \cdot [\![R_{F_1}]\!]^A(\varphi, x, z_1), \ldots, \lambda z_n \cdot [\![R_{F_n}]\!]^A(\varphi, x, z_n), x, y_i) \\ = \; & G_i(\varphi, \lambda z_1 \cdot F_1(\varphi, x, z_1), \ldots, \lambda z_n \cdot F_n(\varphi, x, z_n), x, y_i)) \\ = \; & F_i(\varphi, x, y_i), \end{aligned}$$

which proves (6.6) is "True" for arbitrary computation length l by simultaneous course of value inductionon on l, and hence (6.3). □

Sublemma 6.7. *Let R_{F_i} and R_{G_i}, $1 \le i \le n$, be the procedures defined in the proof of Lemma 6.5. Then for arbitrary input φ, x and y_i,*

$$[\![R_{F_i}]\!]^A(\varphi, x, y_i) \simeq [\![R_{G_i}]\!]^A(\varphi, \lambda z_1 \cdot [\![R_{F_1}]\!]^A(\varphi, x, z_1), \ldots, \lambda z_n \cdot [\![R_{F_n}]\!]^A(\varphi, x, z_n), x, y_i).$$

Proof. By definition of the semantics of procedures,

$$[\![R_{F_i}]\!]^A(\varphi, x, y_i) \;\simeq\; \begin{cases} \sigma'[\mathtt{b}_{F_i}] & \text{if } \; [\![\mathtt{b}_{F_i} := P_{G_i}(\phi,\ \mathtt{a}_{F_{i,1}},\ \mathtt{a}_{F_{i,2}})]\!]^A\sigma{\downarrow}\sigma' \\ \uparrow & \text{if } \; [\![\mathtt{b}_{F_i} := P_{G_i}(\phi,\ \mathtt{a}_{F_{i,1}},\ \mathtt{a}_{F_{i,2}})]\!]^A\sigma{\uparrow} \end{cases}$$

where $\sigma[\mathtt{a}_{F_{i,1}}] = x$, $\sigma[\mathtt{a}_{F_{i,2}}] = y_i$ and ϕ are intepreted as φ.

By the procedure assignment lemma (Lemma 4.7.3),

$$\begin{aligned} [\![\mathtt{b}_{F_i} := P_{G_i}(\phi,\ \mathtt{a}_{F_{i,1}},\ \mathtt{a}_{F_{i,2}})]\!]^A\sigma \;&\simeq\; \sigma\{\mathtt{b}_{F_i}/[\![R^P_{G_i}]\!]^A(\varphi, [\![\mathtt{a}_{F_{i,1}}]\!]^A\sigma, [\![\mathtt{a}_{F_{i,2}}]\!]^A\sigma)\} \\ &\simeq\; \sigma\{\mathtt{b}_{F_i}/[\![R^P_{G_i}]\!]^A(\varphi, x, y_i)\} \end{aligned}$$

Therefore,

$$[\![R_{F_i}]\!]^A(\varphi, x, y_i) \;\simeq\; (\sigma\{\mathtt{b}_{F_i}/[\![R^P_{G_i}]\!]^A(\varphi, x, y_i)\})[\mathtt{b}_{F_i}] \;\simeq\; [\![R^P_{G_i}]\!]^A(\varphi, x, y_i). \tag{6.7}$$

In other words, $[\![R_{F_i}]\!]^A = [\![R^P_{G_i}]\!]^A$. Now we must just show

$$[\![R^P_{G_i}]\!]^A(\varphi, x, y_i) \simeq [\![R_{G_i}]\!]^A(\varphi, \lambda z_1 \cdot [\![R_{F_1}]\!]^A(\varphi, x, z_1), \ldots, \lambda z_n \cdot [\![R_{F_n}]\!]^A(\varphi, x, z_n), x, y_i).$$

and the result will follow.

By definition,

$$[\![R^P_{G_i}]\!]^A(\varphi, x, y_i) \;\simeq\; \begin{cases} \sigma_1'[\mathtt{b}_{G_i}] & \text{if } \; [\![S^P_{G_i}]\!]^A\sigma_1{\downarrow}\sigma_1' \\ \uparrow & \text{if } \; [\![S^P_{G_i}]\!]^A\sigma_1{\uparrow} \end{cases}$$

where $\sigma_1[\mathtt{a}_{G_{i,1}}] = x$, $\sigma_1[\mathtt{a}_{G_{i,2}}] = y_i$ and ϕ are intepreted as φ.

$$[\![R_{G_i}]\!]^A(\varphi, \lambda z_1 \cdot [\![R_{F_1}]\!]^A(\varphi, x, z_1), \ldots, \lambda z_n \cdot [\![R_{F_n}]\!]^A(\varphi, x, z_n), x, y_i) \simeq \begin{cases} \sigma_2'[\mathsf{b}_{G_i}] & \text{if } [\![S_{G_i}]\!]^A\sigma_2{\downarrow}\sigma_2' \\ \uparrow & \text{if } [\![S_{G_i}]\!]^A\sigma{\uparrow} \end{cases}$$

where $\sigma_2[\mathsf{a}_{G_{i,1}}] = x$, $\sigma_2[\mathsf{a}_{G_{i,2}}] = y_i$ and ϕ, $\rho_1, \ldots, \rho_n$ are intepreted as $\varphi, \lambda z_1 \cdot [\![R_{F_1}]\!]^A(\varphi, x, z_1), \ldots, \lambda z_n \cdot [\![R_{F_n}]\!]^A(\varphi, x, z_n)$ respectively.

Now $S^P_{G_i}$ and $R^P_{G_{i,1}}, \ldots, R^P_{G_{i,m_i}}$ are the same as S_{G_i} and $R_{G_{i,1}}, \ldots, R_{G_{i,m_i}}$, except that all occurrences of procedure calls $\mathsf{c} := P_{G_j}(\phi, \mathsf{a}_{G_{i,1}}, t)$ in $S^P_{G_i}$ are replaced by function application statements $\mathsf{c} := \rho_j(t)$ in S_{G_i}. Thus, it is sufficient to prove

$$[\![\mathsf{c} := P_{G_j}(\phi, \mathsf{a}_{G_{i,1}}, t)]\!]^A \simeq [\![\mathsf{c} := \rho_j(t)]\!]^A.$$

By the procedure assignment lemma, and since $[\![\mathsf{a}_{G_{i,1}}]\!]^A\sigma = x$ and ϕ are intepreted as φ,

$$[\![\mathsf{c} := P_{G_j}(\phi, \mathsf{a}_{G_{i,1}}, t)]\!]^A\sigma \simeq \sigma\{\mathsf{c}/[\![R^P_{G_j}]\!]^A(\varphi, x, [\![t]\!]^A\sigma)\}.$$

By the semantics of term and assignment statements, and since ρ_j is the oracle for $\lambda z \cdot [\![R_{F_j}]\!]^A(\varphi, x, z)$,

$$[\![\mathsf{c} := \rho_j(t)]\!]^A\sigma \simeq \sigma\{\mathsf{c}/[\![R_{F_j}]\!]^A(\varphi, x, [\![t]\!]^A\sigma)\}.$$

By (6.7), $[\![R^P_{G_j}]\!]^A = [\![R_{F_j}]\!]^A$, and hence, $[\![\mathsf{c} := P_{G_j}(\phi, \mathsf{a}_{G_{i,1}}, t)]\!]^A = [\![\mathsf{c} := \rho_j(t)]\!]^A$, which ends the proof. □

Theorem 6.8. ***ACP***$(A) \subseteq$ ***Rec***$_2(A)$.

Proof. We prove this by induction on schemes for ***ACPs***. Precisely, we will associate, with each ***ACP*** scheme, a ***Rec***$_2$ procedure.

For schemes I-III, use Lemma 6.1. For schemes IV-VI, use Lemmas 6.2, 6.3, 6.4, respectively. For scheme VIII, use Lemma 6.5. Recall Remark 3.1.8 that we can ignore scheme VII for first-order algebras. □

Corollary 6.9. ***ACP***$^1(A) \subseteq$ ***Rec***(A).

Proof. For any function $f \in$ ***ACP***$^1(A)$, it follows directly from Theorem 6.8 that there is a ***Rec***$_2$ procedure R without function variable in the main procedure, such that

$$f = [\![R]\!]^A.$$

By Theorem 4.10.5, there exist a ***RelRec*** procedure R' with no oracles, such that for all $x : u$,

$$f(x) \simeq [\![R]\!]^A(x) \simeq [\![R']\!]^A(x).$$

However, R' turns out to be a ***Rec*** procedure which ends the proof. □

Corollary 6.10. $\boldsymbol{ACP}^{*1}(A) \subseteq \boldsymbol{Rec}^{*}(A)$.

Proof. From Theorem 6.9. □

Remark 6.11. We prove Lemma 6.4 in order to show that functionals defined by $\boldsymbol{Rec}_2$ are closed under the individual substitution scheme VI. In the proof we use the procedure call statement to simulate scheme VI.

To show that functionals defined by $\boldsymbol{Rec}_2$ are closed under the function substitution scheme VII *directly*, we will need the following lemma.

Let $F \equiv \mathsf{F}^A$, $G \equiv \mathsf{G}^A$, and $H \equiv \mathsf{H}^A$ be functionals, and let F be defined by

$$\mathsf{F}(\varphi, x) \simeq \mathsf{G}(\varphi, \lambda y \cdot \mathsf{H}(\varphi, x, y), x).$$

If G and H are $\boldsymbol{Rec}_2$-computable, then so is F.

Note the form of the procedure call of $\boldsymbol{Rec}_2$, there is obvious way to simulate function abstraction as input using this simple form procedure call. We need more general form of procedure call which contains λ abstraction. Let us extend $\boldsymbol{Rec}_2$ to another second-order programming language $\lambda\boldsymbol{Rec}_2$ which contains procedure call statement like

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

where $T \equiv T_1, \ldots, T_m$ $(m \geq 0)$. For $1 \leq i \leq m$, T_i can be one of following forms.

(1) A primitive function symbol F_k.

(2) A function variable ϕ declared in current procedure.

(3) A term abstraction $\lambda\mathtt{x}{\cdot}t$ obtained by λ abstraction from a term. If $\lambda\mathtt{x}{\cdot}t$ instantiate a function symbol ρ, a term $\rho(t')$ is instantiated by $(\lambda\mathtt{x} \cdot t)(t') \simeq t[\mathtt{x}/t']$. $t[\mathtt{x}/t']$ is a term obtained from t by replacing all occurrences of $\mathtt{x}$ by t'.

(4) A procedure abstraction $\lambda\mathtt{y} \cdot P(\phi, \mathtt{x}, \mathtt{y})$. If $\lambda\mathtt{y} \cdot P(\phi, \mathtt{x}, \mathtt{y})$ instantiate a function symbol ρ, a term $\rho(t)$ is instantiated by $(\lambda\mathtt{y} \cdot P(\phi, \mathtt{x}, \mathtt{y}))(t)$, which is $P(\phi, \mathtt{x}, t)$. Note that $P(\phi, \mathtt{x}, t)$ is not term. Therefore, Similar as in Remark 4.10.4 part (1), an assignment statement $\mathtt{z} := \mathsf{F}_k(\rho(t))$ is instantiated by $\mathtt{z}' := P(\phi, \mathtt{x}, t); \mathtt{z} := \mathsf{F}_k(\mathtt{z}')$, where $\mathtt{z}'$ is a newly introduced variables disjoint from the variables currently declared. Again it is not hard to generalize this method.

Then we can prove that functionals defined by $\lambda\boldsymbol{Rec}_2$ are closed under the function substitution scheme VII by using the procedure call of the new form to simulate scheme VII as follows.

Assump that we have $\boldsymbol{Rec}_2$ procedures R_G and R_H as follows, such that $G = [\![R_G]\!]^A$ and $H = [\![R_H]\!]^A$.

$R_G \equiv$

functions ϕ ρ
$P_{G_1} \Longleftarrow R_{G_1}, \ldots, P_{G_m} \Longleftarrow R_{G_m}$
in a_G out b_G aux c_G
S_G

$R_H \equiv$

functions ϕ
$P_{H_1} \Longleftarrow R_{H_1}, \ldots, P_{H_m} \Longleftarrow R_{H_m}$
in a_{H_x} a_{H_y} out b_H aux c_H
S_H

We can construct a ***Rec***$_2$ procedure R_F as follows

functions ϕ
$P_G \Longleftarrow R_G,\ P_H \Longleftarrow R_H$
in a_F out b_F aux c_F
$\mathsf{b}_F := P_G(\phi, \lambda \mathsf{c}_F \cdot P_H(\phi, \mathsf{a}_F, \mathsf{c}_F), \mathsf{a}_F)$

$[\![R_F]\!]^A(\varphi, x) \simeq F(\varphi, x)$. Again, the proof is obvious and details are omitted.

This also gives the correspondence between ***ACP***(A) and λ***Rec***$_2(A)$ for second-order algebra A, *i.e.* we may be able to prove

$$\boldsymbol{ACP}(A) \subseteq \lambda \boldsymbol{Rec}_2(A)$$

which cannot avoid scheme VII. However this is out of our range.

In short we have to work over λ***Rec***$_2$ if we are considering ***ACP***(A) for second-order algerbra A. As a special case, when A is first-order, ***Rec***$_2$ is *good enough* for our purpose.

Chapter 7

From *Rec* to μ*PR*

In this chapter, we want to prove that, if a function f over A is ***Rec****-computable, then it is $\mu\boldsymbol{PR}^*$ computable. We will first prove that ***Rec**** computability implies ***While**** computability, and the result then follows from Theorem 4.11.4.

We begin by giving a Gödel numbering of the syntax of ***Rec*** procedures and representations of states. In this way, we can define representation functions for $\boldsymbol{Comp}^A$ and ***CompLength***, which we prove to be ***While**** computable.

The proof is parallel to the argument in [TZ00, §4], which this chapter follows closely, except that we are considering ***Rec*** procedures, while [TZ00] considers ***While*** procedures. We just present the differences between them, and interested readers can refer to [TZ00] for details.

7.1 Gödel numbering of syntax

We assume given a family of numerical codings, or Gödel numberings, of the classes of syntactic expressions of Σ and Σ^*, *i.e.*, a family ***gn*** of effective mappings from expressions E to natural numbers $\ulcorner E \urcorner = \boldsymbol{gn}(E)$, which satisfy certain basic properties:

- $\ulcorner E \urcorner$ increases strictly with $\boldsymbol{compl}(E)$, and in particular, the code of an expression is larger than those of its subexpressions;
- sets of codes of the various syntactic classes, and of their respective subclasses, such as $\{\ulcorner t \urcorner \mid t \in \boldsymbol{Term}\}$, $\{\ulcorner t \urcorner \mid t \in \boldsymbol{Term}_s\}$, etc., are primitive recursive;
- we can go primitive recursively from codes of expressions to codes of their immediate subexpressions, and vice versa.

In short, *we can primitive recursively simulate all operations involved in processing the syntax of the programming language.*

We will use the notation

$$\ulcorner \boldsymbol{Term} \urcorner =_{df} \ \{\ulcorner t \urcorner \mid t \in \boldsymbol{Term}\},$$

etc., for sets of Gödel numbers of syntactic expressions.

7.2 Representation of states

We are interested in the representation of various semantic functions on syntactic classes by functions on A or A^*, and in the computability of these representing functions. These semantic functions have states as arguments, so we must first define a representation of states.

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

The *state representing function*

$$\boldsymbol{Rep}_{\mathtt{x}}^{A} : \ \boldsymbol{State}(A) \ \to \ A^u$$

is defined by

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

The *modified state representing function*

$$\boldsymbol{Rep}_{\mathtt{x}*}^{A} : \ \boldsymbol{State}(A) \cup \{*\} \ \to \ \mathbb{B} \times A^u$$

is defined by

$$\begin{aligned} \boldsymbol{Rep}_{\mathtt{x}*}^{A}(\sigma) \quad &= \ (\mathtt{tt},\, \sigma[\mathtt{x}]) \\ \boldsymbol{Rep}_{\mathtt{x}*}^{A}(*) \quad &= \ (\mathtt{ff},\, \boldsymbol{\delta}_A^u) \end{aligned}$$

where $\boldsymbol{\delta}_A^u$ is the default tuple of type u in A.

7.3 Representation of term evaluation

Let $\mathtt{x}$ be a u-tuple of variables. Let $\boldsymbol{Term}_{\mathtt{x}}$ be the class of all $\boldsymbol{Rec}(\Sigma)$ program terms (see §4.1 for definition) with variables among $\mathtt{x}$ only, and for all sorts s of Σ, let $\boldsymbol{Term}_{\mathtt{x},s} = \boldsymbol{Term}_{\mathtt{x},s}(\Sigma)$ be the class of such terms of sort s. Similarly we write $\boldsymbol{TermTup}_{\mathtt{x}}$ for the class of all term tuples with variables among $\mathtt{x}$ only, and $\boldsymbol{TermTup}_{\mathtt{x},v}$ for the class of all v-tuples of such terms.

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

$$\boldsymbol{TE}^{A}_{\mathtt{x},s} :\ \boldsymbol{Term}_{\mathtt{x},s} \times \boldsymbol{State}(A) \ \stackrel{\cdot}{\longrightarrow}\ A_s,$$

defined by

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

is *represented* by the function

$$\boldsymbol{te}^{A}_{\mathtt{x},s} :\ \ulcorner\boldsymbol{Term}_{\mathtt{x},s}\urcorner \times A^u \ \stackrel{\cdot}{\longrightarrow}\ A_s$$

defined by

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

where σ is any state on A such that $\sigma[\mathtt{x}] = a$. (This is well defined, by the functionality lemma for terms.) We can see that a term t is represented by its Gödel number, and a state by a tuple of values. In other words, the following diagram commutes:

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

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

$$\boldsymbol{te}^{A}_{\mathtt{x},v} :\ \ulcorner\boldsymbol{TermTup}_{\mathtt{x},v}\urcorner \times A^u \ \stackrel{\cdot}{\longrightarrow}\ A^v$$

similarly, by

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

7.4 Representation of computation step function

Let $\boldsymbol{AtSt}_{\,\mathtt{x}}$ be the class of $\boldsymbol{Rec}(\Sigma)$ atomic statements (see §4.5 for definition) with variables among $\mathtt{x}$ only. The *atomic statement evaluation function on A relative to* $\mathtt{x}$,

$$\boldsymbol{AE}^{\,A}_{\,\mathtt{x}} : \ \boldsymbol{AtSt}_{\,\mathtt{x}} \times \boldsymbol{State}(A) \ \overset{\cdot}{\longrightarrow} \ \boldsymbol{State}(A),$$

defined by

$$\boldsymbol{AE}^{\,A}_{\,\mathtt{x}}(S, \sigma) = \langle\!| S |\!\rangle^{A} \sigma$$

is *represented* by the function

$$\boldsymbol{ae}^{\,A}_{\,\mathtt{x}} : \ \ulcorner \boldsymbol{AtSt}_{\,\mathtt{x}} \urcorner \times A^{u} \ \overset{\cdot}{\longrightarrow} \ A^{u},$$

defined by

$$\boldsymbol{ae}^{\,A}_{\,\mathtt{x}}(\ulcorner S \urcorner, a) \ = \ (\langle\!| S |\!\rangle^{A} \sigma)[\mathrm{x}],$$

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

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

Next, let $\boldsymbol{Stmt}_{\,\mathtt{x}}$ be the class of $\boldsymbol{Rec}(\Sigma)$ statements (see §4.1 for definition) with variables among $\mathtt{x}$ only, and define

$$\boldsymbol{Rest}^{\,A}_{\,\mathtt{x}} =_{df} \boldsymbol{Rest}^{\,A} \restriction (\boldsymbol{Stmt}_{\,\mathtt{x}} \times \boldsymbol{State}(A)) :$$

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

$$\begin{aligned}
&\boldsymbol{first} && : \ulcorner \boldsymbol{Stmt} \urcorner \ \to \ \ulcorner \boldsymbol{AtSt} \urcorner \\
&\boldsymbol{rest}^{\,A}_{\,\mathtt{x}} && : \ulcorner \boldsymbol{Stmt}_{\,\mathtt{x}} \urcorner \times A^{u} \ \overset{\cdot}{\longrightarrow} \ \ulcorner \boldsymbol{Stmt}_{\,\mathtt{x}} \urcorner
\end{aligned}$$

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

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

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

Note that $\boldsymbol{first}$ is a function from $\mathbb{N}$ to $\mathbb{N}$, and (unlike $\boldsymbol{rest}^{A}_{\mathtt{x}}$ and most of the other representing functions here) does not depend on A or $\mathtt{x}$.

Next, the computation step function (relative to $\mathtt{x}$)

$$
\begin{aligned}
&\boldsymbol{Comp}^{A}_{\mathtt{x}} = \boldsymbol{Comp}^{A} \restriction (\boldsymbol{Stmt}_{\mathtt{x}} \times \boldsymbol{State}(A) \times \mathbb{N}) : \\
&\quad \boldsymbol{Stmt}_{\mathtt{x}} \times \boldsymbol{State}(A) \times \mathbb{N} \;\stackrel{\cdot}{\longrightarrow}\; \boldsymbol{State}(A) \cup \{*\}
\end{aligned}
$$

is *represented* by the function

$$
\boldsymbol{comp}^{A}_{\mathtt{x}} : \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \times A^u \times \mathbb{N} \;\stackrel{\cdot}{\longrightarrow}\; \mathbb{B} \times A^u
$$

which is defined so as to make the following diagram commute:

$$
\begin{tikzcd}
\boldsymbol{Stmt}_{\mathtt{x}} \times \boldsymbol{State}(A) \times \mathbb{N} \arrow[r, "\boldsymbol{Comp}^{A}_{\mathtt{x}}"] \arrow[d, "\langle \boldsymbol{gn}, \boldsymbol{Rep}^{A}_{\mathtt{x}}, \boldsymbol{id}_{\mathbb{N}} \rangle"'] & \boldsymbol{State}(A) \cup \{*\} \arrow[d, "\boldsymbol{Rep}^{A}_{\mathtt{x}}"] \\
\ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \times A^u \times \mathbb{N} \arrow[r, "\boldsymbol{comp}^{A}_{\mathtt{x}}"'] & \mathbb{B} \times A^u
\end{tikzcd}
$$

We put

$$
\boldsymbol{comp}^{A}_{\mathtt{x}}(\ulcorner S \urcorner, a, n) \;=\; (\boldsymbol{notover}^{A}_{\mathtt{x}}(\ulcorner S \urcorner, a, n),\; \boldsymbol{state}^{A}_{\mathtt{x}}(\ulcorner S \urcorner, a, n))
$$

with the two "component functions"

$$\begin{aligned} \boldsymbol{notover}^{A}_{\mathtt{x}} &: \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \times A^u \times \mathbb{N} \ \overset{\cdot}{\longrightarrow}\ \mathbb{B} \\ \boldsymbol{state}^{A}_{\mathtt{x}} &: \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \times A^u \times \mathbb{N} \ \overset{\cdot}{\longrightarrow}\ A^u \end{aligned}$$

where $\boldsymbol{notover}^{A}_{\mathtt{x}}(\ulcorner S \urcorner, a, n)$ tests whether the computation of $\ulcorner S \urcorner$ at a is over by step n, and $\boldsymbol{state}^{A}_{\mathtt{x}}(\ulcorner S \urcorner, a, n)$ gives the value of the state (representative) at step n.

7.5 Representation of statement evaluation

The *statement evaluation function on A relative to* $\mathtt{x}$,

$$\boldsymbol{SE}^{A}_{\mathtt{x}} : \ \boldsymbol{Stmt}_{\mathtt{x}} \times \boldsymbol{State}(A) \ \overset{\cdot}{\longrightarrow}\ \boldsymbol{State}(A),$$

defined by

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

is *represented* by the (partial) function

$$\boldsymbol{se}^{A}_{\mathtt{x}} : \ \ulcorner \boldsymbol{Stmt}_{\mathtt{x}} \urcorner \times A^u \ \overset{\cdot}{\longrightarrow}\ A^u,$$

defined by

$$\boldsymbol{se}^{A}_{\mathtt{x}}(\ulcorner S \urcorner, a) \ = \ ([\![S]\!]^A \sigma)[\mathtt{x}]$$

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

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

7.6 Representation of procedure evaluation

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

$$\boldsymbol{PE}^{\,A}_{\,\mathsf{a},\,\mathsf{b},\,\mathsf{c}} : \ \boldsymbol{Proc}_{\,\mathsf{a},\,\mathsf{b},\,\mathsf{c}} \times A^u \ \stackrel{\cdot}{\longrightarrow} \ A^v$$

defined by

$$\boldsymbol{PE}^{\,A}_{\,\mathsf{a},\,\mathsf{b},\,\mathsf{c}}(R, a) \ = \ [\![R]\!]^A(a)$$

is *represented* by the function

$$\boldsymbol{pe}^{\,A}_{\,\mathsf{a},\,\mathsf{b},\,\mathsf{c}} : \ \ulcorner \boldsymbol{Proc}_{\,\mathsf{a},\,\mathsf{b},\,\mathsf{c}} \urcorner \times A^u \ \stackrel{\cdot}{\longrightarrow} \ A^v$$

defined by

$$\boldsymbol{pe}^{\,A}_{\,\mathsf{a},\,\mathsf{b},\,\mathsf{c}}(\ulcorner R \urcorner,\, a) \ = \ [\![R]\!]^A(a).$$

In other words, the following diagram commutes:

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

7.7 Computability of semantic representing functions

By examining the definitions of the various semantic functions in Section 4, we can infer the relative computability of the corresponding representing functions, as follows. Note that by Remark 4.6.6, we need to work over A^*.

Lemma 7.7.1. *The function* $\boldsymbol{first} : \mathbb{N} \to \mathbb{N}$ *is primitive recursive, and hence* ***While*** *computable on A^N, for any standard Σ-algebra A.*

Lemma 7.7.2. *Let* $\mathtt{x}$ *be a tuple of program variables and* A^* *a standard* Σ^**-algebra.*

(*a*) $\boldsymbol{ae}_{\mathtt{x}}^{A^*}$ *and* $\boldsymbol{rest}_{\mathtt{x}}^{A^*}$ *are* ***While*** *computable in* $\langle \boldsymbol{te}_{\mathtt{a},s}^{A^*} \mid s \in \boldsymbol{Sort}(\Sigma^*)\rangle$ *on* A^*.

(*b*) $\boldsymbol{comp}_{\mathtt{x}}^{A^*}$, *and its two component functions* $\boldsymbol{notover}_{\mathtt{x}}^{A^*}$ *and* $\boldsymbol{state}_{\mathtt{x}}^{A^*}$, *are* ***While*** *computable in* $\boldsymbol{ae}_{\mathtt{x}}^{A^*}$ *and* $\boldsymbol{rest}_{\mathtt{x}}^{A^*}$ *on* A^*.

(*c*) $\boldsymbol{se}_{\mathtt{x}}^{A^*}$ *is* ***While*** *computable in* $\boldsymbol{comp}_{\mathtt{x}}^{A^*}$ *on* A^*.

(*d*) $\boldsymbol{pe}_{\mathtt{a,b,c}}^{A^*}$ *is* ***While*** *computable in* $\boldsymbol{se}_{\mathtt{x}}^{A^*}$ *on* A^*, *where* $\mathtt{x} \equiv \mathsf{a}, \mathsf{b}, \mathsf{c}$.

(*e*) $\boldsymbol{te}_{\mathtt{x},s}^{A^*}$ *is* ***While*** *computable in* $\boldsymbol{pe}_{\mathtt{x,y},\langle\rangle}^{A}$ *on* A^*, *where* $\mathtt{y}$ *is a variable of sort* s, *not in* $\mathtt{x}$.

Proof. Note first that if a semantic function is defined from others by *structural recursion* on a syntactic class of expressions, then a representing function for the former is definable from representing functions for the latter by *course of values recursion* [TZ88] on the set of Gödel numbers of expressions of this class [TZ00].

The proofs are analogous to those for [TZ00, Lemma 4.2]. Note, for part (*b*)-(*e*), the proofs in [TZ00] are based on the general algebraic operational semantics, without any assumption about the language, whether it is ***While*** or ***Rec***. Thus the results can be used directly, with the only difference that we are working over Σ^* algebras.

For part (*a*), clearly, the function $\boldsymbol{ae}_{\mathtt{x}}^{A^*}$ is *primitive recursive* on A^*, since we only have two kinds of atomic statements, skip and *concurrent assignment*. The function $\boldsymbol{rest}_{\mathtt{x}}^{A^*}$ is *course of value recursive* on nat with *range sort* nat, which is reducible to *primitive recursive* on nat (see proof for [TZ00, Lemma 4.2]). Hence, they are ***While*** computable on A^*. Note that a procedure call statement is not an atomic statement, recall the definition of functions ***First*** and $\boldsymbol{Rest}^{A^*}$ in §4.6 that $\boldsymbol{First}(S) = \mathsf{skip}$ and $\boldsymbol{Rest}^{A^*}(S,\sigma) = \hat{S}_i$. □

Lemma 7.7.3. *The following are equivalent.*

(*i*) *For all* $\mathtt{x}$ *and* s, *the term evaluation representing function* $\boldsymbol{te}_{\mathtt{x},s}^{A^*}$ *is* ***While*** *computable on* A^*.

(*ii*) *For all* $\mathtt{x}$, *the atomic statement evaluation representing function* $\boldsymbol{ae}_{\mathtt{x}}^{A^*}$, *and the representing function* $\boldsymbol{rest}_{\mathtt{x}}^{A^*}$, *are* ***While*** *computable on* A^*.

(*iii*) *For all* $\mathtt{x}$, *the computation step representing function* $\boldsymbol{comp}_{\mathtt{x}}^{A^*}$, *and its two component functions* $\boldsymbol{notover}_{\mathtt{x}}^{A^*}$ *and* $\boldsymbol{state}_{\mathtt{x}}^{A^*}$, *are* ***While*** *computable on* A^*.

(*iv*) *For all* $\mathtt{x}$, *the statement evaluation representing function* $\boldsymbol{se}_{\mathtt{x}}^{A^*}$ *is* ***While*** *computable on* A^*.

(v) *For all* $\mathsf{a}, \mathsf{b}, \mathsf{c}$, *the procedure evaluation representing function* $\boldsymbol{pe}^{A^*}_{\mathsf{a},\mathsf{b},\mathsf{c}}$ *is* ***While*** *computable on* A^*.

Proof. From the transitivity lemma of relative computability (cf. [TZ00, Lemma 3.32]), and Lemma 7.7.2. □

7.8 *Rec** computability $\Longrightarrow$ μ*PR** computability

Lemma 7.8.1. *The term evaluation representing function on* A^* *is* ***While*** *computable, and hence,* μ***PR*** *definable on* A^*.

Proof. See [TZ88, TZ00]. □

Theorem 7.8.2. (a) ***Rec***$(A) \subseteq$ ***While****(A),

(b) ***Rec****$(A) \subseteq$ ***While****(A).

Proof. (a) Suppose f is ***Rec*** computable on A. Then there is a ***Rec*** procedure R such that $f \simeq [\![R]\!]^A$. Suppose that

$$R ::= \langle D^{\mathsf{p}} : D^{\mathsf{v}} : S\rangle \text{ and } D^{\mathsf{v}} ::= \mathsf{in\ a\ out\ b\ aux\ c}.$$

It follows from Lemmas 7.7.3 and 7.8.1 that there exist a function $\boldsymbol{pe}^{A^*}_{\mathsf{a},\mathsf{b},\mathsf{c}}$ which is ***While*** computable on A^*, actually ***While**** computable on A, since the input and output variables are simple. Substituting the variable for the Gödel number in the ***While**** procedure for $\boldsymbol{pe}^{A^*}_{\mathsf{a},\mathsf{b},\mathsf{c}}$ by the numeral for the Gödel number of R, we obtain the ***While**** procedure for $[\![R]\!]^A$, *i.e.* f.

(b) By part (a), ***Rec****$(A) \subseteq$ ***While*****$(A) =$ ***While****(A).

Since we can effectively code a double starred object (*i.e.* two-dimensional array) of a given sort as a single starred (or one-dimensional array) of the same sort [TZ00, Remark 2.31].

□

Corollary 7.8.3. (a) ***Rec***$(A) \subseteq \mu$***PR***$^*(A)$,

(b) ***Rec****$(A) \subseteq \mu$***PR***$^*(A)$,

Proof. From Theorem 7.8.2 and 4.11.4. □

Chapter 8

Conclusion and future work

We have proved that

$$\boldsymbol{ACP}^{*1}(A) = \mu\boldsymbol{PR}^{*}(A)$$

via the following circle of inclusions for *N-standard many-sorted* algebras A.

$$\begin{tikzcd}
\boldsymbol{ACP}^{*1}(A) \arrow[r] & \boldsymbol{Rec}^{*}(A) \arrow[d] \\
\mu\boldsymbol{PR}^{*}(A) \arrow[u] & \boldsymbol{While}^{*}(A) \arrow[l]
\end{tikzcd}$$

Some questions which arise from our work are:

8.1 Simultaneous vs. simple LFP scheme

The $\boldsymbol{ACP}$ schemes introduced in §3.1 differ from those in [Fef96] by using simultaneous least fixed points scheme instead of simple least fixed point scheme (*cf.* Remark 3.1.5). An interesting question is:

> *In the absence of product types, can our $\boldsymbol{ACP}^{*}$ schemes be reduced to Feferman's version, i.e. with simple (not simultaneous) least fixed points?*

8.2 Necessity of auxiliary array sorts

Another question is : Can we prove that

$$\boldsymbol{ACP}^1(A) = \mu\boldsymbol{PR}(A)$$

for N-standard many-sorted algebras A *without* arrays?

In connection with this, we have shown that $\mu\boldsymbol{PR}(A) \subseteq \boldsymbol{ACP}^1(A)$ and $\boldsymbol{ACP}^1(A) \subseteq \boldsymbol{Rec}$ (Theorems 5.5 and 6.9). The remaining question is, whether $\boldsymbol{Rec} \subseteq \mu\boldsymbol{PR}(A)$. In Remark 4.6.6, we discuss the difficulty in avoiding the use of arrays when defining the semantics of $\boldsymbol{Rec}$ procedures. Therefore, $\boldsymbol{Rec} \subseteq \mu\boldsymbol{PR}(A)$ is unlikely to be true; however, we lack a proof.

8.3 Second-order version of equivalence results

Since $\boldsymbol{ACP}^*$ is a second-order system, and $\mu\boldsymbol{PR}^*$ is first-order, in order to prove equivalence we have to modify one or the other. We chose to work with a first-order version $\boldsymbol{ACP}^{*1}$ of $\boldsymbol{ACP}^*$. An alternative (and perhaps better) way would be to work with second-order versions of $\mu\boldsymbol{PR}^*$ and $\boldsymbol{While}^*$, *i.e.* $\boldsymbol{Rel}\mu\boldsymbol{PR}^*$ and $\boldsymbol{RelWhile}^*$ containing function parameters (*cf.* the system $\boldsymbol{RelRec}$ in Chapter 4) and then prove the complete circle of inclusions in Figure 1.1 for second-order systems. Our results for the first-order systems would then follow easily.

Appendix A

Denotational semantics of statements

In this chapter, we develop the denotational semantics of statement of ***Rec*** procedures. This chapter is independent of the rest of the thesis. It is, actually, a side issue of our research.

A.1 Complete partially ordered sets and least fixed points

In this section we will define a series of mathematic concepts and their properties, which provide the basis for defining the denotational semantics of statements, as well as the justification for ***ACP*** schemes. All definitions and theorems in this section can be found in [dB80]. We omit most proofs. Interested people can refer to [dB80].

We begin from the basic concept of *partially ordered set*.

Definition A.1.1 (Partially ordered set)**.** Let C be an arbitrary set. A *partial order* $\sqsubseteq$ on C is a subset of $C \times C$ (we write $x \sqsubseteq y$ instead of $(x,y) \in\ \sqsubseteq$) which satisfies

(a) $x \sqsubseteq x$ (reflexivity).

(b) If $x \sqsubseteq y$ and $y \sqsubseteq x$ then $x = y$ (antisymmetry).

(c) If $x \sqsubseteq y$ and $y \sqsubseteq z$ then $x \sqsubseteq y$ (transitivity).

Definition A.1.2 (Least upper bound)**.** Let C be arbitrary set and $X \subseteq C$.

(a) $z \in C$ is called the *least upper bound* (lub) of X if

> (i) $x \sqsubseteq z$ for all $x \in X$,
>
> (ii) for all $y \in C$, if $x \sqsubseteq y$ for all $x \in X$, then $z \sqsubseteq y$.
>
> The lub of a set X will be denoted by $\sqcup X$.

(b) The lub of a sequence $x_0, x_1, \ldots$ is denoted by $\bigsqcup_{i=0}^{\infty} x_i$.

Definition A.1.3 (Chains)**.** A *chain* on $(C,\sqsubseteq)$ is a sequence $x_0, x_1, \ldots$ such that $x_i \sqsubseteq x_{i+1}$, for $i = 0, 1, \ldots$.

Definition A.1.4 (Complete partially ordered sets)**.** A *complete partially ordered set* (cpo) is a set C together with a partial order $\sqsubseteq$ which satisfies the following two requirements.

(a) There is a least element with respect to $\sqsubseteq$, *i.e.* an element $\perp$ such that $\perp \sqsubseteq x$ for all $x \in C$.

(b) Each chain $(x_i)_{i=0}^{\infty}$ has a lub $\bigsqcup_{i=0}^{\infty} x_i$.

Definition A.1.5 (Monotonic functions)**.** Let $(C_1, \sqsubseteq_1)$ and $(C_2, \sqsubseteq_2)$ be partially ordered sets. A function $f\colon C_1 \rightarrow C_2$ is called *monotonic* if, for each $x, y \in C_1$, if $x \sqsubseteq_1 y$, then $f(x) \sqsubseteq_2 f(y)$.

Definition A.1.6 (Fixed point)**.** Let C be a cpo, $f : C \rightarrow C$, and $x \in C$.

(a) x is called a *fixed point* of f if $f(x) = x$.

(b) x is called the *least fixed point* of f, written LFP(f), if x is a fixed point of f, and moreover, for each fixed point y of f, $x \sqsubseteq y$.

Definition A.1.7 (Continuous functions)**.** Let $(C_1, \sqsubseteq_1)$ and $(C_2, \sqsubseteq_2)$ be cpo's. A monotonic function $f\colon C_1 \rightarrow C_2$ is called *continuous* if, for each chain $(x_i)_{i=0}^{\infty}$ of elements in C_1,

$$f(\bigsqcup_{i=0}^{\infty} x_i) \quad \sqsubseteq \quad \bigsqcup_{i=0}^{\infty} f(x_i).$$

Lemma A.1.8. *Let C be a cpo. Let $f : C \to C$ be monotonic. Then f is continuous iff, for each chain $(x_i)_{i=0}^{\infty}$ in C,*

$$f(\bigsqcup_{i=0}^{\infty} x_i) = \bigsqcup_{i=0}^{\infty} f(x_i).$$

Theorem A.1.9. *Let C be a cpo. Let $f : C \to C$ be continuous. f has a least fixed point, such that*

$$\mathrm{LFP}(f) = \bigsqcup_{i=0}^{\infty} f^i(\perp).$$

Theorem A.1.10. *Let $C_1, \ldots, C_n$ be cpo's. Let $f_i : C_1 \times \cdots \times C_n \to C_i$ be continuous, for $i = 1, \ldots, n$. Then the vector of functions $[f_1, \ldots, f_n]$ has a simultaneous least fixed point, such that*

$$\mathrm{LFP}(f_1, \ldots, f_n) = \bigsqcup_{k=0}^{\infty} \langle x_1^k, \ldots, x_n^k \rangle$$

where, for $i = 1, \ldots, n$,

$$\begin{array}{rcl} x_i^0 & = & \perp_{C_i} \\ x_i^{k+1} & = & f_i(x_1^k, \ldots, x_n^k). \end{array}$$

A.2 Denotational semantics of statements

In this section we will define the denotational semantics of a statement S, and then, we will justify this definition by proving its equivalence with the operational semantics. The definitions and proofs are analogous to those in [TZ88, §3.1.8-3.1.10], which consider recursive calls without parameters.

First we define a partial order on the set of partial state transformations on A.

Definition A.2.1. (a) ***StateTrans***(A) is the set of all partial state transformations on A.

(b) For $\varphi_1, \varphi_2 \in$ ***StateTrans***(A), $\varphi_1 \sqsubseteq_A \varphi_2$ iff $\forall \sigma \in$ ***State***(A),

$$\varphi_1(\sigma) \downarrow \Rightarrow \varphi_2(\sigma) \downarrow \text{ and } \varphi_2(\sigma) = \varphi_1(\sigma).$$

Lemma A.2.2. *$\sqsubseteq_A$ is a partial order on* ***StateTrans***(A).

Lemma A.2.3. *The structure (**StateTrans**$(A), \sqsubseteq_A$) is a cpo.*

Proof. We need to show that $(\boldsymbol{StateTrans}(A), \sqsubseteq_A)$ has a least element and that each chain in $(\boldsymbol{StateTrans}(A), \sqsubseteq_A)$ has a lub.

(*a*) The totally undefined transformation, denoted by $\varphi_\perp$, where $\varphi_\perp(\sigma)\uparrow$ for all σ, is clearly the $\sqsubseteq_A$-least element.

(*b*) Let $\varphi_0 \sqsubseteq_A \varphi_1 \sqsubseteq_A \ldots$ be any $\sqsubseteq_A$ chain. We define $\varphi = \bigcup_{n=0}^{\infty} \varphi_n$, *i.e.* for all σ and σ',

$$\varphi(\sigma)\downarrow\sigma' \;\Leftrightarrow\; \text{for some } n,\; \varphi_n(\sigma)\downarrow\sigma'$$

It is easy to check that φ is the desired lub of the chain.

□

Now we define the semantics of statement as the lub of a sequence of partial state transformation $[\![S]\!]_k^{A^*}$, where, for each $k \geq 0$, $[\![S]\!]_k^{A^*}$ is the approximate meaning of S given by interpreting procedure calls of depth k or more simply as diverging.

Definition A.2.4. $[\![S]\!]_k^{A^*}$, for $k = 0, 1, \ldots$, is defined by induction on $(k, \boldsymbol{compl}(S))$, where $\boldsymbol{compl}(S)$ is the complexity of S, *e.g.* the length of S.
Base case $(k = 0)$.

(*a*) For S atomic,

$$\begin{array}{rcl} [\![\mathsf{skip}]\!]_0^{A^*}\sigma & = & \sigma \\ [\![\mathsf{x} := t]\!]_0^{A^*}\sigma & = & \sigma\{\mathtt{x}/[\![t]\!]^{A}\sigma\}. \end{array}$$

(*b*)

$$[\![S_1; S_2]\!]_0^{A^*}\sigma \;\simeq\; [\![S_2]\!]_0^{A^*}([\![S_1]\!]_0^{A^*}\sigma).$$

(*c*)

$$[\![\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}]\!]_0^{A^*}\sigma \;\simeq\; \begin{cases} [\![S_1]\!]_0^{A^*}\sigma & \text{if } [\![b]\!]_0^{A^*}\sigma = \mathsf{tt} \\ [\![S_2]\!]_0^{A^*}\sigma & \text{if } [\![b]\!]_0^{A^*}\sigma = \mathsf{ff} \end{cases}$$

(*d*)

$$[\![\mathsf{x} := P_i(t)]\!]_0^{A^*}\sigma\uparrow$$

Induction step. For cases (a), (b) and (c), $[\![S]\!]_k^{A^*}$ is just like $[\![S]\!]_0^{A^*}$. For the last case,

$$[\![\mathsf{x} := P_i(t)]\!]_{k+1}^{A^*}\sigma \;\simeq\; [\![\hat{S}_i]\!]_k^{A^*}\sigma \quad (\textit{cf.}\ \text{Figure 4.1}).$$

Lemma A.2.5. *The sequence $[\![S]\!]_0^{A^*}$, $[\![S]\!]_1^{A^*}$, ... is $\sqsubseteq_A$-increasing.*

Proof. Show that $[\![S]\!]_k^{A^*} \sqsubseteq_A [\![S]\!]_{k+1}^{A^*}$ by induction on $(k, \boldsymbol{compl}(S))$. □

This lemma justifies the following definition.

Definition A.2.6 (Denotational semantics)**.** We define the denotational semantics as a partial state transformation

$$[\![S]\!]_{\mathsf{den}}^{A^*} = \bigsqcup_{k=0}^{\infty} [\![S]\!]_k^{A^*} = \bigcup_{k=0}^{\infty} [\![S]\!]_k^{A^*}.$$

The following shows that denotational semantics also satisfy the usual desirable i/o properties.

Theorem A.2.7. (a) *For S atomic, $[\![S]\!]_{\mathsf{den}}^{A^*} = \langle\!| S |\!\rangle^{A^*}$, i.e.,*

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

(b)

$$[\![S_1; S_2]\!]_{\mathsf{den}}^{A^*}\sigma \;\simeq\; [\![S_2]\!]_{\mathsf{den}}^{A^*}([\![S_1]\!]_{\mathsf{den}}^{A^*}\sigma).$$

(c)

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

(d)

$$[\![\mathsf{x} := P_i(t)]\!]_{\mathsf{den}}^{A^*}\sigma \;\simeq\; [\![\hat{S}_i]\!]_{\mathsf{den}}^{A^*}\sigma.$$

Proof. For part (a), (b) and(c), the equation hold for $[\![S]\!]_k^{A^*}$ $(k = 0, 1, \ldots)$ by definition. So they hold for $[\![S]\!]_{\mathsf{den}}^{A^*}$, by taking suprema.

For part(d),

$$\begin{aligned}
[\![\mathtt{x} := P_i(t)]\!]_{\mathsf{den}}^{A} &= \bigcup_{k=0}^{\infty}[\![\mathtt{x} := P_i(t)]\!]_k^{A^*} && \text{by Definition A.2.6} \\
&= \bigcup_{k=0}^{\infty}[\![\mathtt{x} := P_i(t)]\!]_{k+1}^{A^*} && \text{by Lemma A.2.5} \\
&= \bigcup_{k=0}^{\infty}[\![\hat{S}_i]\!]_k^{A^*} && \text{by Definition A.2.4} \\
&= [\![\hat{S}_i]\!]_{\mathsf{den}}^{A}.
\end{aligned}$$

□

Now we prove the equivalence of the operational and denotational semantics of statements.

Theorem A.2.8. $[\![S]\!]^{A^*} = [\![S]\!]_{\mathsf{den}}^{A^*}$

Proof. We need to prove two directions.

(a) $[\![S]\!]^{A^*} \sqsubseteq_A [\![S]\!]_{\mathsf{den}}^{A^*}$

If $[\![S]\!]^{A^*}\sigma$ diverges, there is nothing to prove. If $[\![S]\!]^{A^*}\sigma$ converges, we prove, by induction on $\boldsymbol{CompLength}^{A^*}(S,\, \sigma)$, that $[\![S]\!]_{\mathsf{den}}^{A^*}$ converges and $[\![S]\!]^{A^*}\sigma = [\![S]\!]_{\mathsf{den}}^{A^*}\sigma$.

(b) $[\![S]\!]_{\mathsf{den}}^{A^*} \sqsubseteq_A [\![S]\!]^{A^*}$

By Definition A.2.6, it is sufficient to prove that for all S and for all k, $[\![S]\!]_k^{A^*} \sqsubseteq_A [\![S]\!]^{A^*}$.

We show it by induction on $(k,\, \boldsymbol{compl}(S))$. We consider the following cases:

(i) S atomic:

It follows the Definition A.2.4 and Theorem 4.6.1 that

$$[\![S]\!]_k^{A^*} = \langle\!|S|\!\rangle^{A^*} = [\![S]\!]^{A^*}$$

(ii) $S \equiv S_1; S_2$

Since $\boldsymbol{compl}(S_1) < \boldsymbol{compl}(S)$ and $\boldsymbol{compl}(S_2) < \boldsymbol{compl}(S)$, by induction hypothesis, we have $[\![S_1]\!]_k^{A^*} \sqsubseteq_A [\![S_1]\!]^{A^*}$ and $[\![S_2]\!]_k^{A^*} \sqsubseteq_A [\![S_2]\!]^{A}$. It follows that $[\![S_1; S_2]\!]_k^{A^*} \sqsubseteq_A [\![S_1; S_2]\!]^{A^*}$, *i.e.* $[\![S]\!]_k^{A^*} \sqsubseteq_A [\![S]\!]^{A}$.

(iii) $S \equiv \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}$

Similar to (ii).

(iv) $S \;\equiv\; \mathtt{x} := P_i(t)$

$$
\begin{aligned}
[\![\mathtt{x} := P_i(t)]\!]_k^{A^*} &= [\![\hat{S}_i]\!]_{k-1}^{A^*} && \text{(by Definition A.2.4)} \\
&\sqsubseteq_A [\![\hat{S}_i]\!]^{A^*} && \text{(by induction hypothesis)} \\
&= [\![\mathtt{x} := P_i(t)]\!]^{A^*} && \text{(by Theorem 4.6.1)}
\end{aligned}
$$

□

Bibliography

[dB80] Jaco de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall International, 1980.

[Fef77] Solomon Feferman. Inductive schemata and recursively continuous functionals. In R. O. Gandy and M. Hyland, editors, *Logic Colloquium '76*, pages 373–392. North-Holland, Amsterdam, 1977.

[Fef92a] Solomon Feferman. A new approach to abstract data types, i: Informal development. *Mathematical Structures in Computer Science*, 2:193–229, 1992.

[Fef92b] Solomon Feferman. A new approach to abstract data types, ii: Computability on adts as ordinary computation. In *Computer Science Logic*, pages 79–95. Springer-Verlag, 1992.

[Fef96] Solomon Feferman. Computation on abstract data types. the extensional approach, with an application to streams. *Annals of Pure and Applied Logic*, 81:75–113, 1996.

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

[Mos84] Y. N. Moschovakis. Abstract recursion as a foundation for the theory of recursive algorithms. In *Computation and Proof Theory*, pages 289–364. Springer-Verlag, 1984.

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

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

[MT92] K. Meinke and J.V. Tucker. Universal algebra. In *Handbook of Logic in Computer Science*, volume 1. Oxford University Press, 1992.

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

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

[TZ93] J.V. Tucker and J.I. Zucker. Computable functions on stream algebras. In H. Schwichtenberg, editor, *NATO Advanced Study Institute, International Summer School on Proof and Computation, Marktoberdorf, Germany*, pages 341–382. Springer-Verlag, 1993.

[TZ00] J.V. Tucker and J.I. Zucker. Computable functions and semicomputable sets on many-sorted algebras. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 5, pages 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.

[Win93] Glynn Winskel. *Th Formal Semantics of Programming Languages: An Introduction.* The MIT Press, 1993.