Fundamenta Informaticae XXI (2004) 1–43 IOS Press

# First and Second Order Recursion on Abstract Data Types

Jian Xu \* and Jeffery Zucker †

Department of Computing and Software McMaster University 1280 Main Street West, Hamilton, Ontario, Canada L8S 4L7 xuj5@mcmaster.ca zucker@mcmaster.ca

Abstract. This paper compares two scheme-based models of computation on abstract many-sorted algebras A: Feferman's system ACP(A) of "abstract computational procedures" based on a least fixed point operator, and Tucker and Zucker's system  $\mu PR(A)$  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 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.

**Keywords:** models of computation, many-sorted algebras, recursive schemes, recursive procedures, fixed points, computation on abstract data types, abstract computability.

# 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

<sup>\*</sup>The research of the first author was funded by a postgraduate scholarship from the Natural Sciences and Engineering Research Council of Canada. Address for correspondence:Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario, Canada L8S 4L7 —- Address for correspondence:Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario, Canada L8S 4L7

<sup>&</sup>lt;sup>†</sup>The research of the second author was funded by a grant from the Natural Sciences and Engineering Research Council of Canada.

Kleene's schemes [9] 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 [12, 14, 15, 3, 4, 5, 6] 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  $\mu PR$  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* (*ACP*) for functionals of type level 2 over abstract algebras, characterized by using the LFP (*least fixed point*) scheme, is developed in [6]. A natural question is the following.

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

Since *ACP*, unlike  $\mu PR$ , deals with functionals of type level 2, in order to compare two schemes, we need some definitions.

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

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

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

S. Feferman raised this question in [6] and conjectured that the answer is "Yes".

Inspired by the denotational (or "fixed point") semantics of recursive procedures in [1, 17], we prove the following circle of inclusions in Figure 1.



Figure 1. Implication cycle

**Rec** is an imperative language employed to generate the least fixed points of second-order functionals by properly chosen recursive procedure calls. **Rec**<sup>\*</sup> is the extension of **Rec** with arrays. **Rec**<sup>\*</sup>(A) is the set of **Rec**<sup>\*</sup> computable functions on A. Similarly, **While**<sup>\*</sup>(A) is the set of **While**<sup>\*</sup> computable functions on A, where **While** is another imperative programming language characterized by the 'While' construct. (Precise definitions are given in Section 4.)

The equivalence between  $While^*(A)$  and  $\mu PR^*(A)$  was proved in [18]. We proceed by proving the following relations.

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

$$ACP^{*1}(A) \subseteq Rec^*(A) \tag{2}$$

$$Rec^*(A) \subseteq While^*(A)$$
 (3)

Of the above three inclusions, (1) is quite straightforward, and (3) can be derived from the semantic investigation of *While* programs in [20]. The really interesting new result is (2), which forms the core of the paper (Section 6).

In the proof of (2), even if we are considering functions of type level  $\leq 1$ , we nevertheless have to deal with functionals of type level 2, since functions are defined as the least fixed points of level 2 functionals. To generate these, we therefore develop a second-order version of **Rec**, namely **Rec**<sub>2</sub>, and prove that

$$ACP(A) \subseteq Rec_2(A)$$

for functionals of type level  $\leq 2$ . Then (2) follows as a corollary. Although the programming language *Rec*<sub>2</sub> is used as a device for proving the circle of implications, it is interesting in its own right. Recursion schemes for functions are certainly of great importance. They have been used in studying recursion on abstract structures [6, 7], and in various applications, such as modelling and verifying hardware [8], analog machines [13, 16], and computation on continuous data type [2].

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* [18, 20], which states that the class of functions computable by finite deterministic algorithms on A is precisely  $\mu PR^*(A)$  (or equivalently *While*\*(A)).

The paper is organized as follows. In Section 2, we introduce the basic concepts of abstract manysorted 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 Section 3, we define the two computational models based on recursive schemes discussed above, namely *ACP* and  $\mu PR$ . In Section 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 [18, 20]). Sections 5, 6 and 7 prove (1), (2) and (3) respectively. As stated above, Section 6 forms the core of the paper. It proves that any function computable by an *ACP* scheme is computable by some *Rec* procedure. Section 8 concludes this paper with a short summary and future work. In the Appendix, we give some proofs of a technical nature, omitted from previous sections.

The paper developed out of the MSc thesis of the first author [23]. The authors are grateful to three anonymous referees for some very helpful comments.

# 2. Basic concepts

In this section, we will introduce some basic concepts concerning signatures and algebras, which will be used in the following sections. In particular, we have two groups of concepts extracted from [18, 20] and [6] respectively. We will use the definitions in [18, 20] as the framework, and introduce the differences and connections between that and [6] in §2.5. We present this section to make the paper self-contained, and to simplify the presentation. Interested readers can refer to [6, 18, 20] for detailed discussions.

### 2.1. Signatures

#### **Definition 2.1.1. (Many-sorted signatures)**

A many-sorted signature  $\Sigma$  is a pair  $(Sort(\Sigma), Func(\Sigma))$  where

- (a) **Sort**( $\Sigma$ ) is a finite set of sorts.
- (b) Func (Σ) is a finite set of (primitive or basic) function symbols F : s<sub>1</sub> × ··· × s<sub>m</sub> → s (m ≥ 0). Each symbol F has a type s<sub>1</sub> × ··· × s<sub>m</sub> → s, where m ≥ 0 is the arity of F, and s<sub>1</sub>,..., s<sub>m</sub> ∈ Sort(Σ) are the domain sorts and s ∈ Sort(Σ) is the range sort of F. The case m = 0 corresponds to constant symbols, we then write F : → s.

### **Definition 2.1.2. (Product types over** $\Sigma$ **)**

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

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

#### **Definition 2.1.3. (Function types)**

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

### **Definition 2.1.4.** ( $\Sigma$ -algebras)

A  $\Sigma$ -algebra A has, for each sort s of  $\Sigma$ , a non-empty set  $A_s$ , called the *carrier of sort* s, and for each  $\Sigma$ -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  $\Sigma$ -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  $\Sigma$ -function symbol  $\mathsf{F} : u \to s$  has an interpretation  $\mathsf{F}^A : A^u \rightharpoonup A_s$ . If u is empty, *i.e.*,  $\mathsf{F}$  is a constant symbol, then  $\mathsf{F}^A$  is an element of  $A_s$ .

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

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

We present some examples which will be important for us.

**Example 2.1.5.** (a) The signature of booleans can be defined as

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

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

(b) The signature of naturals can be defined as

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

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

### **Definition 2.1.6.** ( $\Sigma$ -variables)

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

#### **Definition 2.1.7.** ( $\Sigma$ -terms)

Let  $Term(\Sigma)$  be the class of  $\Sigma$ -terms  $t, \ldots$ , and  $Term_s$  be the class of terms of sort s, defined by

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

where  $\mathsf{F} \in Func(\Sigma)_{u \to s}$  and  $u = s_1 \times \cdots \times s_m$ . We write t : s to indicate that  $t \in Term_s$ . 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 a standard signature  $\Sigma$ , defined below, the definition of  $Term(\Sigma)$  is extended to include a conditional constructor, *cf.* Definition 2.2.3.)

#### Assumption 2.1.8. (Instantiation)

For each  $s \in Sort(\Sigma)$ , there is a closed term, called the default term  $\delta^s$ , of that sort.

This guarantees the existence of a *default value*  $\delta_A^s$  for all sort *s*, and *default tuple*  $\delta_A^u$  for all product types *u* in a  $\Sigma$ -algebra *A*.

#### 2.2. Standard signatures and algebras

### **Definition 2.2.1. (Standard signatures)**

A signature  $\Sigma$  is standard if  $\Sigma(\mathcal{B}) \subseteq \Sigma$ .

#### **Definition 2.2.2. (Standard algebras)**

Given a standard signature  $\Sigma$ , a  $\Sigma$ -algebra A is a *standard algebra* if it is an expansion<sup>1</sup> of  $\mathcal{B}$ , as defined in Example 2.1.5 (*a*).

## Definition 2.2.3. ( $\Sigma$ -terms for standard signatures)

We extend **Term**( $\Sigma$ ) to include a conditional constructor as follows, where b is a term of sort **bool**.

 $t^s ::= \ldots$  |if b then  $t_1^s$  else  $t_2^s$  fi

Any many-sorted signature  $\Sigma$  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  $\mathcal{B}$  together with a conditional constructor.

## 2.3. N-standard signatures and algebras

#### **Definition 2.3.1. (N-standard signature)**

A standard signature  $\Sigma$  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:

$$\begin{array}{rrl} 0: & \to \text{nat} \\ \textbf{S}: & \text{nat} & \to \text{nat} \\ \textbf{eq}_{\text{nat}}: & \text{nat}^2 & \to \text{bool} \\ \textbf{less}_{\text{nat}}: & \text{nat}^2 & \to \text{bool}. \end{array}$$

#### **Definition 2.3.2.** (N-standard algebra)

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

Note that any standard  $\Sigma$ -algebra A can be N-standardized to a  $\Sigma^N$ -algebra  $A^N$ , by adjoining a carrier  $\mathbb{N}$  of sort nat, and the operation listed in Definition 2.3.1.

### Assumption 2.3.3. (N-Standardness)

All signatures  $\Sigma$  and  $\Sigma$ -algebras A are N-standard.

## **2.4.** Algebras $A^*$ of signature $\Sigma^*$

### **Definition 2.4.1.** (Signature $\Sigma^*$ and algebras $A^*$ )

Given a signature  $\Sigma$ , and  $\Sigma$ -algebra A, we extend  $\Sigma$  to a signature  $\Sigma^*$ , and expand A to a  $\Sigma^*$ -algebra  $A^*$ , as follows. Include, for each  $\Sigma$ -sort s, a new *starred sort*  $s^*$ , and also the function symbols described below. Define, for each sort s of  $\Sigma$ , the carrier  $A_s^*$  of sort  $s^*$ , to be the set of finite sequences (or arrays)  $a^*$  over  $A_s$ .

<sup>&</sup>lt;sup>1</sup>The concept expansion of an algebra is defined in [20]

- (i) Lgth<sub>s</sub> :  $s^* \rightarrow \text{nat}$ , where Lgth<sub>s</sub><sup>A</sup>( $a^*$ ) gives the length of the array  $a^* \in A_s^*$ ;
- (*ii*)  $\operatorname{Null}_s : \to s^*$ , where  $\operatorname{Null}_s^A$  is the array in  $A_s^*$  of zero length;
- (*iii*)  $Ap_s : s^* \times nat \rightarrow s$ , where

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

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

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

(v) Newlength<sub>s</sub> :  $s^* \times nat \rightarrow s^*$ , where Newlength<sub>s</sub><sup>A</sup> $(a^*, m)$  is the array  $b^*$  of length *m*, such that for all k < m,

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

A sort of  $\Sigma^*$  is called *simple* or *starred* according as it has the form *s* or *s*<sup>\*</sup>(respectively), for some  $s \in Sort(\Sigma)$ . Similarly, a variable is called *simple* or *starred* according as its sort is simple or starred.

**Remark 2.4.2.** The reason for introducing starred sorts is the lack of effective coding of finite sequences within abstract algebras in general. 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.

### 2.5. Second-order signatures and algebras

The algebras in [18, 20] are first order algebras, since all functional symbols are interpreted as first-order functions within the algebras. In general, however, Feferman's *ACP* deals with second-order many-sorted algebras [6]. This subsection provides the background for Feferman's *ACP* schemes in the next section. The N-Standardness Assumption (Assumption 2.3.3) holds here as elsewhere throughout this paper.

### **Definition 2.5.1. (Second-order signatures)**

A second-order signature  $\Sigma$  is a pair  $(Sort(\Sigma), Func(\Sigma))$  where

- (a) **Sort**( $\Sigma$ ) is a finite set of *sorts*, where **bool**  $\in$  **Sort**( $\Sigma$ ), *i.e.*,  $\Sigma$  is standard.
- (b) Func (Σ) is a finite set of functional symbols F: τ<sub>1</sub> × ··· × τ<sub>m</sub>×s<sub>1</sub> × ··· × s<sub>n</sub> → s. Each symbol F has a type τ<sub>1</sub> × ··· × τ<sub>m</sub>×s<sub>1</sub> × ··· × s<sub>n</sub> → s, where m ≥ 0 and n ≥ 0, s<sub>1</sub>, ..., s<sub>m</sub>, s ∈ Sort(Σ), and τ<sub>1</sub>, ..., τ<sub>m</sub> are Σ-function types (see Definition 2.1.3). When m = 0, the symbol F is first-order, *i.e.* a function symbol.

#### **Definition 2.5.2. (Second-order algebras)**

A (full) second-order  $\Sigma$ -algebra A has:

- (a) for each sort s of  $\Sigma$ , 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  $A_s$ .)

We will write  $\pi, \ldots$  for function product types  $\tau_1 \times \cdots \times \tau_m$   $(m \ge 0)$ . If  $\pi = \tau_1 \times \cdots \times \tau_m$ , we write  $A^{\pi} = A_{\tau_1} \times \cdots \times A_{\tau_m}$ .

- **Remarks 2.5.3.** (a) Given a signature  $\Sigma$ , a  $\Sigma$ -function symbol  $F : \tau_1 \times \cdots \times \tau_m \times s_1 \times \cdots \times s_n \to s$  is of type level 2, 1, or 0, according as m > 0, m = 0 and n > 0, or m = n = 0.
- (b)  $\Sigma$  is said to be first-order if each  $F \in Func(\Sigma)$  is of type level  $\leq 1$ , in that it is equivalent to the standard (first-order) signature defined in §2.2.
- (c) Corresponding to each  $\mathsf{F} \in Func(\Sigma)$ ,  $\mathsf{F}^A$  is of type level 2, 1 or 0; and corresponding to  $\Sigma$ , a  $\Sigma$ -algebra A is of second or first order.

We note that a thorough investigation of higher order algebras has been undertaken in [10].

# 3. Models of computation based on recursive schemes

In this section, we will introduce two models of computation based on recursive schemes, ACP and  $\mu PR$ . The contents are taken from [6] and [18] respectively with necessary modification.

### 3.1. Feferman's ACP schemes

In general, *abstract computational procedures* (*ACP*) deal with many-sorted algebras A with objects of type level  $\leq 2$  (see Remark 2.5.3). With each signature  $\Sigma$  are associated the following formal schemes for computation procedures on  $\Sigma$ -algebras.

I.	(Initial functionals)	$F(\varphi, x) \simeq F_k(\varphi, x) \ (for \ each \ F_k \in Func \ (\Sigma));$
II.	(Identity)	F(x) = x;
III.	(Application)	$F(\varphi, x) \simeq \varphi(x);$
IV.	(Conditional)	$F(\varphi, x, b) \simeq \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ $
V.	(Structural)	$F(\varphi, x) \simeq G(\varphi_f, x_g);$
VI.	(Individual substitution)	$F(\varphi, x) \simeq G(\varphi, x, H(\varphi, x)));$
VII.	(Function substitution)	$F(\varphi, x) \simeq G(\varphi, \lambda y \cdot H(\varphi, x, y), x);$
VIII.	(Least fixed point)	$F_1(\varphi, x, y_1) \simeq \varrho_1^{\varphi, x}(y_1), \ \dots, \ F_n(\varphi, x, y_n) \simeq \varrho_n^{\varphi, x}(y_n)$
		where $(\varrho_1^{\varphi,x},\ldots,\varrho_n^{\varphi,x})=$
		LFP $((\lambda \varrho_1 \cdot \ldots \cdot \lambda \varrho_n \cdot \lambda z_1 \cdot G_1(\varphi, \varrho_1, \ldots, \varrho_n, x, z_1)),$
		,
		$(\lambda \varrho_1 \cdot \ldots \cdot \lambda \varrho_n \cdot \lambda z_n \cdot \mathbf{G}_n(\varphi, \varrho_1, \ldots, \varrho_n, x, z_n))).$

The partial equality " $\simeq$ " above is to be interpreted as meaning that either both sides of the equation converge and are equal, or both sides diverge. In scheme V,  $f : \{1, \ldots, m'\} \rightarrow \{1, \ldots, m\}$ ,  $g : \{1, \ldots, n'\} \rightarrow \{1, \ldots, n\}$  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 [6], the schemes are invariant under isomorphism.

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

- (b) For any  $\Sigma$ -algebra A, ACP(A) is the collection of all  $F^A$  for  $F \in ACP(\Sigma)$ .
- (c)  $ACP^{1}(A)$  is the collection of all functions of type level  $\leq 1$  in ACP(A).
- **Definition 3.1.2.** (a)  $ACP^*(\Sigma)$  is the collection of all  $\mathsf{F}$  in  $ACP(\Sigma^*)$ , with the restriction that the domain and range types of  $\mathsf{F}$  are simple (*i.e.*, unstarred).
- (b) For any  $\Sigma$ -algebra A,  $ACP^*(A)$  is the collection of all  $F^A$  for  $F \in ACP^*(\Sigma)$ .
- (c)  $ACP^{*1}(A)$  is the collection of all functions of type level  $\leq 1$  in  $ACP^{*}(A)$ .

Notation 3.1.3. In context of scheme VIII, we use

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

**Notation 3.1.4.** Let (a)  $\hat{G}_i^{\varphi,x}$  be the interpretation of  $\hat{G}_i^{\varphi,x}$  in A; (b)  $\hat{G}_i^x$  be the interpretation of  $\hat{G}_i^x$  in A.

### Remark 3.1.5. (Simultaneous LFP)

In the least fixed points scheme VIII, we diverge from [6] by using *simultaneous least fixed points*, in the sense that, for i = 1, ..., n,

$$\begin{array}{rcl} \varrho_i^0 &=& \bot\\ \varrho_i^{k+1} &=& \hat{G}_i^{\varphi,x}(\varrho_1^k,\dots,\varrho_n^k) \end{array}$$

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

This seems necessary to prove the equivalence of  $ACP^{1}(A)$  with  $\mu PR(A)$  which uses *simultaneous* primitive recursion [18, 20].

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 clear.

(b) Since we consider only first-order algebras, *i.e.* all primitive functions  $F_k$  are objects of type level  $\leq 1$ , by [6, Theorem 4] all  $F^A$  are trivially *continuous*, hence, *monotonic*<sup>2</sup>. This justifies the use of scheme VIII, *i.e.* the existence of the least fixed points.

**Remark 3.1.7.** Let  $ACP_0$  stand for ACP minus scheme VII. By [6, Theorem 3],  $ACP_0(A)$  is closed under scheme VII for first-order algebras A, *i.e.* if A is first-order, then  $ACP_0(A) = ACP(A)$ . Therefore, we need not distinguish ACP and  $ACP_0$ .

# 3.2. $\mu PR$ schemes

We give the definitions of  $\mu PR$  computability in this section. Most of the contents are taken from [18] with some necessary modifications. We avoid excessive formality.

From now on, we will use  $\downarrow$  and  $\uparrow$  to denote, respectively, convergence (definedness) and divergence (undefinedness) of relevant function applications.

For each  $\Sigma$ , we have the following induction schemes which specify the functions over all N-standard algebras A of signature  $\Sigma$ .

I.	(Primitive functions)	$f(x) \simeq F_k(x) \ (for \ each \ F_k \in Func \ (\Sigma));$
II.	(Projection)	$\mathbf{f}(x) = x_i;$
III.	(Definition by cases)	$f(x) \simeq \left\{ \begin{array}{ll} g_1(x) & \text{if } h(x) \downarrow t \\ g_2(x) & \text{if } h(x) \downarrow f \\ \uparrow & \text{if } h(x) \uparrow; \end{array} \right.$
IV.	(Composition)	$f(x) \simeq h(g_1(x), \dots, g_m(x));$
V.	(Simultaneous primitive	recursion)
		$f_1(x,0) \simeq g_1(x)$
		,
		$f_n(x,0) \simeq g_n(x)$
		$f_1(x,z+1) \simeq h_1(x,z,f_1(x,z),\ldots,f_n(x,z))$
		,
		$f_n(x,z+1) \simeq h_n(x,z,f_1(x,z),\ldots,f_n(x,z));$
VI.	(Least number operator)	$f(x) \simeq \mu z[g(x,z) \downarrow t].$

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 clear.

(b) The semantics of the schemes should be clear from their formal presentation. (Formal semantics can be found in [18].) We should however point out that the least number or  $\mu$  operator in scheme VI

<sup>&</sup>lt;sup>2</sup>Our definitions of monotonic and continuous functionals follow the treatment in [6] as follows:

F is monotonic if  $(F(\varphi, x) \downarrow$  and  $\varphi \subseteq \psi) \implies F(\varphi, x) = F(\psi, x)$ .

F is continuous if, whenever  $F(\varphi, x) = y$ , there exists finite  $\psi \subseteq \varphi$  such that  $F(\psi, x) = y$ .

is the *constructive*  $\mu$ -operator, with the operational semantics: "Test g(z,0), g(z,1), g(z,2),... 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  $g(z,0) \downarrow \mathbf{f}$ ,  $g(z,1) \uparrow$  and  $g(z,2) \downarrow \mathbf{t}$ , then  $f(z) \uparrow$  (*i.e.*, it does not converge to 2).

- (c)  $\mu PR(A)$  is the set of all partial functions obtained from the basic functions defined in I-III by means of the operations defined in IV-VI.
- (d) We can see, from schemes V and VI, the reason for the N-standardness assumption.

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

- (b) For any  $\Sigma$ -algebra A,  $\mu PR(A)$  is the collection of all  $f^A$  for  $f \in \mu PR(\Sigma)$ .
- **Definition 3.2.3.** (a)  $\mu PR^*(\Sigma)$  is the collection of f in  $\mu PR(\Sigma^*)$ , with the restriction that the domain and range types of f are simple.
- (b) For any  $\Sigma$ -algebra A,  $\mu PR^*(A)$  is the collection of all  $\mathfrak{f}^A$  for  $\mathfrak{f} \in \mu PR^*(\Sigma)$ .

#### Remark 3.2.4. (PR schemes)

 $PR(\Sigma)$  is the collection of all f generated by the schemes I-V for signature  $\Sigma$ , and similarly for the collections PR(A),  $PR^*(A)$ , and  $PR(A^*)$  for a  $\Sigma$ -algebra A. We say that f is *primitive recursive* on A to mean that  $f \in PR(A)$ .

# 4. Models of computation based on imperative languages

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

First, we define an imperative programming language  $Rec = Rec(\Sigma)$  on standard  $\Sigma$ -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)  $Var(\Sigma)$  is the class of  $\Sigma$ -variables x, y, ... (see Definition 2.1.6).
- (b) **ProcName**( $\Sigma$ ) is the class of procedure names  $P_1, P_2, \ldots$ . We write **ProcName**<sub> $u \to v$ </sub> for all procedure names of type  $u \to v$ .
- (c)  $Term(\Sigma)$  is the class of  $\Sigma$ -terms  $t, \ldots$  (see Definition 2.2.3).
- (d)  $Stmt(\Sigma)$  is the class of statements  $S, \ldots$ , defined by

 $S ::= \text{skip} \mid \mathbf{x}^u := t^u \mid S_1; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi} \mid \mathbf{x}^v := P(t^u)$ 

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

(e) **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

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

where  $R_i ::= \langle D_i^{\mathsf{p}} : D_i^{\mathsf{v}} : S_i \rangle$ , for i = 1, ..., m;  $D_i^{\mathsf{p}}$  and  $D_i^{\mathsf{v}}$  are defined like  $D^{\mathsf{p}}$  and  $D^{\mathsf{v}}$ .  $D^{\mathsf{v}}$  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: (i) a, b, and c are pairwise disjoint; (ii) every variable occurring in S must be declared in  $D^{v}$ ; (iii) the *input variables* must not occur on the left hand side of assignments in S.

We will sometimes write *Stmt* for *Stmt*( $\Sigma$ ), and *Stmt*<sup>\*</sup> for *Stmt*( $\Sigma$ <sup>\*</sup>), etc.

### 4.2. Closed programs

Notation 4.2.1. For a procedure declaration  $D^{p}$ , we use the 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  $D^{p} \equiv \langle P_{i} \leftarrow R_{i} \rangle_{i=1}^{m}$  and  $R_{i} \equiv \langle D_{i}^{p} : D_{i}^{v} : S_{i} \rangle$  for i = 1, ..., m. If  $D^{p} \equiv D^{p[k]}$ , we write  $D_{i}^{p[k+1]}$  for  $D_{i}^{p}$ , for i = 1, ..., 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.** *ProcSet*( $D^{p[k]}$ ) is the set of procedure variables associated with  $D^{p[k]}$  defined as follows:

(a) for  $D^{\mathsf{p}[0]} \equiv \langle P_i \iff R_i \rangle_{i=1}^m$ , **ProcSet** $(D^{\mathsf{p}[0]}) \equiv \{P_1, \dots, P_m\};$ 

(b) for 
$$D^{\mathsf{p}[\mathsf{k}]} \equiv \langle P_i \longleftrightarrow R_i \rangle_{i=1}^m$$
, where  $R_i \equiv \langle D_i^{\mathsf{p}[\mathsf{k}+1]} : D_i^{\mathsf{v}} : S_i \rangle$ , and  $D_i^{\mathsf{p}[\mathsf{k}+1]} \equiv \langle P_{ij} \hookleftarrow R_{ij} \rangle_{j=1}^n$ ,  
**ProcSet** $(D_i^{\mathsf{p}[\mathsf{k}+1]}) \equiv ProcSet(D^{\mathsf{p}[\mathsf{k}]}) \cup \{P_{i1}, \dots, P_{in}\}$ 

Note that the definition is by recursion on the depth k of the declaration, *i.e.* "top-down". **ProcSet** $(D^{p[k]})$  consists of all procedure variables currently declared in  $D^{p[k]}$ , as well as those declared in the "prior" declarations  $D^{p[0]}, \ldots, D^{p[k-1]}$ . Thus the definition depends implicitly on a main declaration  $D^{p[0]}$  as a global context.

Let ProcVar(S) be the set of procedure names occurring in the statement S (as procedure calls).

### Definition 4.2.3. (Closed declaration)

A procedure declaration  $D^{\mathsf{p}} \equiv \langle P_i \leftarrow R_i \rangle_{i=1}^m$ , where  $R_i \equiv \langle D_i^{\mathsf{p}} : D_i^{\mathsf{v}} : S_i \rangle$ , is closed if (i)  $D_i^{\mathsf{p}}$  is closed and (ii) **ProcVar**( $S_i$ )  $\subseteq$  **ProcSet**( $D_i^{\mathsf{p}}$ ), for i = 1, ..., m.

### **Definition 4.2.4. (Closed procedure)**

A procedure  $R \equiv \langle D^{\mathsf{p}} : D^{\mathsf{v}} : S \rangle$  is closed if (i)  $D^{\mathsf{p}}$  is closed and (ii)  $ProcVar(S) \subseteq ProcSet(D^{\mathsf{p}})$ .

### Assumption 4.2.5. (Closure)

All procedure declarations and procedures are closed.

#### 4.3. States

### **Definition 4.3.1. (State)**

For each standard  $\Sigma$ -algebra A, a *state* on A is a family  $\langle \sigma_s | s \in Sort(\Sigma) \rangle$  of functions  $\sigma_s : Var_s \to A_s$ .

Let *State*(A) be the set of states on A, with elements  $\sigma, \ldots$ . For  $\mathbf{x} \in Var_s$ , we often write  $\sigma(\mathbf{x})$  for  $\sigma_s(\mathbf{x})$ . Also, for a tuple  $\mathbf{x} \equiv (\mathbf{x}_1, \ldots, \mathbf{x}_m)$ , we write  $\sigma[\mathbf{x}]$  for  $(\sigma(\mathbf{x}_1), \ldots, \sigma(\mathbf{x}_m))$ .

## **Definition 4.3.2. (Variant of a state)**

Let  $\sigma$  be a state over A,  $\mathbf{x} \equiv (\mathbf{x}_1, \dots, \mathbf{x}_n) : u$  and  $a = (a_1, \dots, a_n) \in A^u$  (for  $n \ge 1$ ). Then  $\sigma\{\mathbf{x}/a\}$  is the *variant* of  $\sigma$  defined by, for all variables y:

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

### 4.4. Semantics of terms

For  $t \in Term_s$ , we define the partial function

г пА

$$\llbracket t \rrbracket^A : State(A) \rightharpoonup A_s,$$

where  $\llbracket t \rrbracket^A \sigma$  is the value of t in A at state  $\sigma$ . The definition is by structural induction on t,

$$\begin{split} \|\mathbf{x}\|^{A}\sigma &= \sigma(\mathbf{x}) \\ \|\mathbf{F}(t_{1},\ldots,t_{m})\|^{A}\sigma &\simeq \begin{cases} \mathbf{F}^{A}([t_{1}]]^{A}\sigma,\ldots,[t_{m}]]^{A}\sigma) & \text{if } [t_{i}]]^{A}\sigma\downarrow(i=1,\ldots,m) \\ \uparrow & \text{otherwise} \end{cases} \\ \\ \|\mathbf{if}\ b\ \text{then}\ t_{1}^{s}\ \text{else}\ t_{2}^{s}\ \text{fi}]^{A}\sigma &\simeq \begin{cases} [t_{1}^{s}]]^{A}\sigma & \text{if } [b]]^{A}\downarrow\mathbf{t} \\ [t_{2}^{s}]]^{A}\sigma & \text{if } [b]]^{A}\downarrow\mathbf{t} \\ \uparrow & \text{if } [b]]^{A}\uparrow. \end{cases} \end{split}$$

For a *tuple* of terms  $t = (t_1, \ldots, t_m)$ , we use the notation  $[t]^A \sigma =_{df} ([t_1]^A \sigma, \ldots, [t_m]^A \sigma)$ .

**Definition 4.4.1.** For any  $M \subseteq Var$ , and states  $\sigma_1$  and  $\sigma_2$ ,  $\sigma_1 \underset{M}{\approx} \sigma_2$  means  $\sigma_1 \upharpoonright M = \sigma_2 \upharpoonright M$ .

#### Lemma 4.4.2. (Functionality lemma for terms)

For any term t and states  $\sigma_1$  and  $\sigma_2$ , if  $\sigma_1 \underset{M}{\approx} \sigma_2 (M = var(t))$ , then  $\llbracket t \rrbracket^A \sigma_1 \simeq \llbracket t \rrbracket^A \sigma_2$ .

### **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

 $\llbracket S \rrbracket^A : State(A) \rightharpoonup State(A).$ 

We will present an outline of this approach following [20]. Assume, *firstly*, that (for the language under consideration) there is a class  $AtSt \subset Stmt$  of *atomic statements* for which we have a (partial) meaning function

$$\langle \! \langle S \rangle \! \rangle^A : State(A) \rightarrow State(A),$$

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

First:Stmt 
$$\rightarrow$$
 AtStRest  $^A$ :Stmt  $\times$  State $(A) \rightarrow$  Stmt

where, for a statement S and state  $\sigma$ , *First*(S) is an atomic statement which gives the *first* step in the execution of S, and *Rest* <sup>A</sup>(S,  $\sigma$ ) is a statement which gives the *rest* of the execution in state  $\sigma$ .

Then, we define the "one-step computation of S at  $\sigma$ " function

$$Comp_1^A$$
:  $Stmt \times State(A) \rightharpoonup State(A)$ 

by

$$Comp_1^A(S,\sigma) \simeq \langle First(S) \rangle^A \sigma.$$

Finally, the definition of the computation step function

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

follows by a simple recursion on n:

$$Comp^{A}(S, \sigma, 0) = \sigma$$

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

Note that for n = 1, this yields  $Comp^{A}(S, \sigma, 1) \simeq Comp_{1}^{A}(S, \sigma)$ .

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

If we put  $\sigma_n = Comp^A(S, \sigma, n)$ , assuming it converges, then the sequence of states  $\sigma_0, \ldots, \sigma_n, \ldots$  is called the *computation sequence* generated by S at  $\sigma$ . There are three possibilities: (a) the sequence terminates in a final state  $\sigma_l$ , where  $Comp^A(S, \sigma, l+1)\downarrow *$ ; (b) it is infinite (global divergence); (c) it is undefined from some point on (local divergence). In case (a) the computation has an output, given by the final state; in case (b) the computation is non-terminating, and has no output; and in case (c) the computation is also non-terminating, and has no output, because a state at one of the 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  $\sigma$ , as the partial function

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

by

$$CompLength^{A}(S,\sigma) \simeq \begin{cases} \text{least } n \text{ s.t. } Comp^{A}(S,\sigma,n+1) \downarrow * & \text{if such an } n \text{ exists (which implies} \\ & Comp^{A}(S,\sigma,k) \downarrow & \text{for all } k < n+1) \\ \uparrow & \text{otherwise.} \end{cases}$$

Note that *CompLength*<sup>A</sup>(S,  $\sigma$ )  $\downarrow$  in case (a) above only. Then we define

$$\llbracket S \rrbracket^{A}(\sigma) \simeq Comp^{A}(S, \sigma, CompLength^{A}(S, \sigma)).$$

# 4.6. Operational semantics of statements

We now apply the above theory to the language  $Rec(\Sigma)$ . 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,  $\sigma \in State(A^*)$ , 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:

$$\langle \mathsf{skip} \rangle^{A^*} \sigma = \sigma \langle \mathsf{x} := t \rangle^{A^*} \sigma = \sigma \{ \mathsf{x} / \llbracket t \rrbracket^{A^*} \sigma \}.$$

Note that x and t could be of starred sort.

Next we define *First* and *Rest*  $A^*$  by structural induction on  $S \in Stmt^*$ . *Case 1.* S is atomic.

$$First(S) = S$$
  
 $Rest^{A^*}(S, \sigma) =$ skip.

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

$$First(S) = First(S_1)$$
  

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

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

$$First(S) = skip$$

$$Rest^{A^*}(S, \sigma) \simeq \begin{cases} S_1 & \text{if } \llbracket b \rrbracket^{A^*} \sigma \downarrow \mathbf{t} \\ S_2 & \text{if } \llbracket b \rrbracket^{A^*} \sigma \downarrow \mathbf{f} \\ \uparrow & \text{if } \llbracket b \rrbracket^{A^*} \sigma \uparrow. \end{cases}$$

Case 4.  $S \equiv x := P_i(t)$  (i = 1, ..., m)

$$First(S) = skip$$
  
 $Rest^{A^*}(S, \sigma) = \hat{S}_i$ 

where  $\hat{S}_i$  is the statement defined in Figure 2.

a*	:=	$Newlength_{s_a}(\mathtt{a}^*,Lgth_{s_a}(\mathtt{a}^*)+1);$
b*	:=	Newlength <sub><math>s_b</math></sub> ( $b^*$ , Lgth <sub><math>s_b</math></sub> ( $b^*$ ) + 1);
c*	:=	Newlength $_{s_c}(c^*, Lgth_{s_c}(c^*) + 1);$
$a^*$	:=	$Update_{s_a}(\mathtt{a}^*,Lgth_{s_a}(\mathtt{a}^*)-1,\mathtt{a});$
b*	:=	$Update_{s_b}(\mathtt{b}^*,Lgth_{s_b}(\mathtt{b}^*)-1,\mathtt{b});$
c*	:=	$Update_{s_c}(c^*,Lgth_{s_c}(c^*)-1,c);$
а	:=	<i>t</i> ;
b	:=	$oldsymbol{\delta}^{s_b};$
с	:=	$\delta^{s_c};$
	$S_{i}$	
	$D_{i}$	
b <sub>tmp</sub>	:=	b;
b <sub>tmp</sub> a	:= :=	b; $Ap_{s_a}(\mathbf{a}^*,Lgth_{s_a}(\mathbf{a}^*)-1);$
b <sub>tmp</sub> a b	:= := :=	b; Ap <sub>sa</sub> ( $\mathbf{a}^*$ , Lgth <sub>sa</sub> ( $\mathbf{a}^*$ ) - 1); Ap <sub>sb</sub> ( $\mathbf{b}^*$ , Lgth <sub>sb</sub> ( $\mathbf{b}^*$ ) - 1);
b <sub>tmp</sub> a b c	:= := := :=	$ \begin{split} & \texttt{b;} \\ & Ap_{s_a}(\texttt{a}^*,Lgth_{s_a}(\texttt{a}^*)-1); \\ & Ap_{s_b}(\texttt{b}^*,Lgth_{s_b}(\texttt{b}^*)-1); \\ & Ap_{s_c}(\texttt{c}^*,Lgth_{s_c}(\texttt{c}^*)-1); \end{split} $
b <sub>tmp</sub> a b c a <sup>*</sup>	:= := := := :=	$ \begin{split} & \texttt{b}; \\ & Ap_{s_a}(\texttt{a}^*,Lgth_{s_a}(\texttt{a}^*)-1); \\ & Ap_{s_b}(\texttt{b}^*,Lgth_{s_b}(\texttt{b}^*)-1); \\ & Ap_{s_c}(\texttt{c}^*,Lgth_{s_c}(\texttt{c}^*)-1); \\ & Newlength_{s_a}(\texttt{a}^*,Lgth_{s_a}(\texttt{a}^*)-1); \end{split} $
b <sub>tmp</sub> a b c a <sup>*</sup> b <sup>*</sup>	:= := := := :=	$ \begin{split} & \texttt{b}; \\ & Ap_{s_a}(\texttt{a}^*,Lgth_{s_a}(\texttt{a}^*)-1); \\ & Ap_{s_b}(\texttt{b}^*,Lgth_{s_b}(\texttt{b}^*)-1); \\ & Ap_{s_c}(\texttt{c}^*,Lgth_{s_c}(\texttt{c}^*)-1); \\ & Newlength_{s_a}(\texttt{a}^*,Lgth_{s_a}(\texttt{a}^*)-1); \\ & Newlength_{s_b}(\texttt{b}^*,Lgth_{s_b}(\texttt{b}^*)-1); \end{split} $
b <sub>tmp</sub> a b c a* b* c*	:= := := := := :=	$ \begin{split} & \texttt{b}; \\ & Ap_{s_a}(\texttt{a}^*,Lgth_{s_a}(\texttt{a}^*)-1); \\ & Ap_{s_b}(\texttt{b}^*,Lgth_{s_b}(\texttt{b}^*)-1); \\ & Ap_{s_c}(\texttt{c}^*,Lgth_{s_c}(\texttt{c}^*)-1); \\ & Newlength_{s_a}(\texttt{a}^*,Lgth_{s_a}(\texttt{a}^*)-1); \\ & Newlength_{s_b}(\texttt{b}^*,Lgth_{s_b}(\texttt{b}^*)-1); \\ & Newlength_{s_c}(\texttt{c}^*,Lgth_{s_c}(\texttt{c}^*)-1); \\ & Newlength_{s_c}(\texttt{c}^*,Lgth_{s_c}(\texttt{c}^*)-1); \end{split} $
b <sub>tmp</sub> a b c a* b* c* x	:= := := := := := :=	$ \begin{split} & \text{b;} \\ & \text{Ap}_{s_a}(\mathbf{a}^*, \text{Lgth}_{s_a}(\mathbf{a}^*) - 1); \\ & \text{Ap}_{s_b}(\mathbf{b}^*, \text{Lgth}_{s_b}(\mathbf{b}^*) - 1); \\ & \text{Ap}_{s_c}(\mathbf{c}^*, \text{Lgth}_{s_c}(\mathbf{c}^*) - 1); \\ & \text{Newlength}_{s_a}(\mathbf{a}^*, \text{Lgth}_{s_a}(\mathbf{a}^*) - 1); \\ & \text{Newlength}_{s_b}(\mathbf{b}^*, \text{Lgth}_{s_b}(\mathbf{b}^*) - 1); \\ & \text{Newlength}_{s_c}(\mathbf{c}^*, \text{Lgth}_{s_c}(\mathbf{c}^*) - 1); \\ & \text{Newlength}_{s_c}(\mathbf{c}^*) - 1); \\ & \text{Newlength}_{s_c}($

Figure 2. The statement of  $\hat{S}_i$ 

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 with 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 data types. Actually, we simulate stacks by our array variables in  $\hat{S}_i$ . It is here that starred variables are introduced in the definition of **Rest**  $A^*$  (see Remark 4.6.6).
- In the construction of  $\hat{S}_i$ , we assume a, b, and c are single variables 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  $b_{tmp}$  to avoid erasing the output of  $S_i$  when restoring the value of 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  $\delta^{s_b}$  and  $\delta^{s_c}$  are the corresponding default values for b and c.
- The expressions 't + 1' and 't 1' (for a term t : nat) can easily be interpreted in the language of N-standard signatures (§2.3).

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,  $\llbracket S \rrbracket^{A^*} = \langle S \rangle^{A^*}$ , *i.e.*,

$$\{ \mathsf{skip} \}^{A^*} \sigma = \sigma \{ \mathsf{x} := t \}^{A^*} \sigma \simeq \begin{cases} \sigma \{ \mathsf{x} / \llbracket t \rrbracket^A \sigma \} & \text{if } \llbracket t \rrbracket^A \sigma \downarrow \\ \uparrow & \text{otherwise.} \end{cases}$$

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

(c)

$$[if b then S_1 else S_2 fi]^{A^*} \sigma \simeq \begin{cases} [S_1]^{A^*} \sigma & \text{if } [b]^{A^*} \downarrow \mathbf{t} \\ [S_2]^{A^*} \sigma & \text{if } [b]^{A^*} \downarrow \mathbf{f} \\ \uparrow & \text{if } [b]^{A^*} \sigma^{\uparrow}. \end{cases}$$

(d) 
$$\llbracket \mathbf{x} := P_i(t) \rrbracket^{A^*} \sigma \simeq \llbracket \hat{S}_i \rrbracket^{A^*} \sigma.$$

#### **Proof:**

The results follow from Lemmas 4.6.2 – 4.6.5 below.

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

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

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

 $\textbf{Lemma 4.6.4. } \textbf{Comp}^{A^*}(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \ \sigma, \ n+1) \ \simeq \ \begin{cases} \textbf{Comp}^{A^*}(S_1, \sigma, n) & \textbf{ if } \llbracket b \rrbracket^{A^*} \sigma \downarrow \textbf{t} \\ \textbf{Comp}^{A^*}(S_2, \sigma', n) & \textbf{ if } \llbracket b \rrbracket^{A^*} \sigma \downarrow \textbf{f} \\ \uparrow & \textbf{ if } \llbracket b \rrbracket^{A^*} \sigma \uparrow. \end{cases}$ 

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

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

For a **Rec** procedure, we 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 the array structure.

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

Lemma 4.6.7. (Functionality lemma for computation sequences) Suppose  $var(S) \subseteq M$ . If  $\sigma_1 \underset{M}{\approx} \sigma_2$ , then for all  $n \ge 0$ , either

- (i)  $Comp^{A^*}(S, \sigma_1, n) \downarrow \sigma'_1$  and  $Comp^{A^*}(S, \sigma_2, n) \downarrow \sigma'_2$  (say), where  $\sigma'_1 \approx \sigma'_2$ , or
- (*ii*)  $Comp^{A^*}(S, \sigma_1, n)\uparrow$  and  $Comp^{A^*}(S, \sigma_2, n)\uparrow$ .

#### **Proof:**

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

Lemma 4.6.8. (Functionality lemma for statements)

Suppose  $var(S) \subseteq M$ . If  $\sigma_1 \approx \sigma_2$ , 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*)  $\llbracket S \rrbracket^A \sigma_1 \uparrow$  and  $\llbracket S \rrbracket^A \sigma_2 \uparrow$ .

### 4.7. Semantics of procedures

### Assumption 4.7.1. (Initialization)

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

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

Let  $R \equiv \langle D^{p} : D^{v} : S \rangle$ , where  $D^{v} \equiv in$  a out b aux c, be a procedure of type  $u \to v$ . Then its meaning is a function

$$\llbracket R \rrbracket^A : A^u \rightharpoonup A^v$$

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

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

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

#### Lemma 4.7.3. (Procedure assignment lemma)

Consider a statement  $\mathbf{x} := P_i(t)$ , where  $P_i \longleftrightarrow R_i$ . Then  $[\![\mathbf{x} := P_i(t)]\!]^{A^*} \sigma \simeq \sigma\{\mathbf{x}/[\![R_i]\!]^A([\![t]\!]^{A^*}\sigma)\}$ .

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 x gets the required values while all other variables are left unchanged; in other words, there are *no side effects*.

#### **Proof:**

Suppose  $R_i \equiv \langle D_i^{\mathsf{p}} : D_i^{\mathsf{v}} : S_i \rangle$ . Consider  $\hat{S}_i$  (Figure 2) and let  $\sigma' = \sigma\{\mathsf{a}, \mathsf{b}, \mathsf{c}/\llbracket t \rrbracket^{A^*} \sigma, \delta^{s_b}, \delta^{s_c}\}$ . By Definition 4.7.2,

$$\llbracket R_i \rrbracket^A(\llbracket t \rrbracket^{A^*} \sigma) \simeq (\llbracket S_i \rrbracket^{A^*} \sigma') [b].$$
(4)

By Theorem 4.6.1 (*d*),

$$[\mathbf{x} := P_i(t)]^{A^*} \sigma \simeq [\hat{S}_i]^{A^*} \sigma.$$
(5)

We will show

$$[\hat{S}_i]^{A^*} \sigma \simeq \sigma \{ \mathbf{x} / ([S_i]^{A^*} \sigma') [\mathbf{b}] \}.$$
(6)

The result follows from (4), (5) and (6).

To show (6), note that  $[\hat{S}_i]^{A^*}$  is a state transformation involving only variables  $a^*$ ,  $b^*$ ,  $c^*$ , a, b, c,  $b_{tmp}$ , and x (*cf.* Figure 2). We will investigate the behavior of these variables to show that  $a^*$ ,  $b^*$ ,  $c^*$ , a, b, c are unchanged, and 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)  $a^*$ ,  $b^*$ , and  $c^*$  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  $a^*$ ,  $b^*$ , and  $c^*$ , which are trimmed, are modified. Clearly,  $a^*$ ,  $b^*$ , and  $c^*$  keep their original values.
- (b) The original values of a, b, and c are stored in the last locations in  $a^*$ ,  $b^*$ , and  $c^*$  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  $b_{tmp}$  takes the default value.
- (d) The atomic statements  $b_{tmp} := b$  and  $x := b_{tmp}$  in  $\hat{S}_i$  guarantee that x takes the desired value  $([S_i]]^{A^*} \sigma')[b]$ .

**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 the 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*<sup>\*</sup>( $\Sigma$ ) procedure is a *Rec*( $\Sigma^*$ ) procedure in which the *input* and *output* variables are *simple*. (However the auxiliary variables may be starred.)

### **Definition 4.7.7.** (*Rec*<sup>\*</sup> computable functions)

- (a) A function f on A is computable on A by a **Rec**<sup>\*</sup> procedure R if  $f = [\![R]\!]^A$ . It is **Rec**<sup>\*</sup> computable on A if it is computable on A by some **Rec**<sup>\*</sup> 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  $\varphi_i : A^{u_i} \rightharpoonup A^{v_i}$ . We define the programming language *Rec*( $\phi$ ) (or by abuse of notation, *Rec*( $\varphi$ )) 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  $\varphi_1, \ldots, \varphi_n$ .

We will use *RelRec* for the class of all  $Rec(\phi)$  procedures without specifying the oracle names.

The atomic statements of  $Rec(\phi)$  include *oracle calls* as follows, where  $t : u_i$  and  $x : v_i$ .

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

The semantics of this is given by

$$\langle\!\! \langle \mathbf{x} := \phi_i(t) \rangle\!\! \rangle^A \sigma \ \simeq \ \left\{ \begin{array}{ll} \sigma\{\mathbf{x}/b\} & \text{if } \llbracket t \rrbracket^A \sigma \downarrow a \text{ and also } \varphi_i(a) \downarrow b \\ \uparrow & \text{otherwise.} \end{array} \right.$$

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

oracles $\phi_1, \ldots, \phi_m$
$P_1 \Leftarrow R_1, \ldots, P_n \Leftarrow R_n$
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  $\operatorname{Rest}_{\varphi}^{A^*}, \operatorname{Comp}_{\varphi}^{A^*}, \operatorname{CompLength}_{\varphi}^{A^*}, \operatorname{and} [S]_{\varphi}^{A^*}$  instead of  $\operatorname{Rest}^{A^*}, \operatorname{Comp}^{A^*}, \operatorname{CompLength}^{A^*}, \operatorname{and} [S]_{\varphi}^{A^*}$ . The definitions of these functions follow along lines similar to those in §4.6.

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

Therefore, the semantics of a *RelRec* procedure  $R : u \rightarrow v$ , where  $R \equiv$ 

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

(with  $\phi : \pi$  and a : u) is a function  $\llbracket R \rrbracket_{\varphi}^A : A^u \rightharpoonup A^v$ , given by

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

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

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

The reason for introducing  $Rec(\varphi)$  computability is that 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  $\varphi$  and  $\varphi'$  of the same type,  $\varphi \sqsubseteq \varphi'$  means that for any input x,

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

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,  $\varphi$  and  $\varphi'$  are two interpretations of the oracle tuple  $\phi$ .

**Lemma 4.9.3.** Let S be a statement with oracles  $\phi$ . If  $\varphi \sqsubseteq \varphi'$ , then  $\langle First(S) \rangle_{\varphi}^{A^*} \sqsubseteq \langle First(S) \rangle_{\varphi'}^{A^*}$ .

#### **Proof:**

By definition, First(S) is an atomic statement. We have three cases: (a)  $First(S) \equiv skip$ ; (b)  $First(S) \equiv x := t$ ; (c)  $First(S) \equiv x := \phi_i(t)$ . Cases (a) and (b) are trivial, while Case (c) follows directly from condition  $\varphi \sqsubseteq \varphi'$ .

**Lemma 4.9.4.** Let S be a statement with oracles  $\phi$ . If  $\varphi \sqsubseteq \varphi'$ , then **Rest**  $A^*(S, \cdot) \sqsubseteq \text{Rest} A^*(S, \cdot)^3$ .

### **Proof:**

By induction on the complexity of S.

**Lemma 4.9.5.** Let S be a statement with oracles  $\phi$ . If  $\varphi \sqsubseteq \varphi'$ , then  $Comp_{\varphi}^{A^*}(S, \cdot, n) \sqsubseteq Comp_{\varphi'}^{A^*}(S, \cdot, n)$ .

#### **Proof:**

By induction on n.

**Corollary 4.9.6.** Let S be a statement with oracles  $\phi$ . If  $\varphi \sqsubseteq \varphi'$ , then  $[S]^{A^*}_{\varphi} \sqsubseteq [S]^{A^*}_{\varphi'}$ .

<sup>&</sup>lt;sup>3</sup>Here we use the notation  $f(x, \cdot)$  for  $\lambda y \cdot f(x, y)$ .

#### Theorem 4.9.7. (Monotonicity Theorem for *RelRec* procedures)

Let *R* be a *RelRec* procedure with oracles  $\phi$ . If  $\varphi \sqsubseteq \varphi'$ , then  $[\![R]\!]^A_{\omega}(x) \sqsubseteq [\![R]\!]^A_{\omega'}(x)$ .

### **Proof:**

Suppose  $R \equiv \langle D^{p} : D^{v} : S \rangle$ . Let  $\sigma$  be any state such that  $\sigma[a] = x$ . By definition of the semantics of procedures

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

and

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

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

# 4.10. *Rec*<sup>2</sup> computability

We will extend Rec to a second-order programming language  $Rec_2$  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 as follows, where  $\phi : u \to s, t^u : u$  and  $t^s : s$ .

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

• A function variables declaration, where  $\phi \equiv \phi_1, \ldots, \phi_m$  is a tuple of function symbols and  $m \ge 0$ .

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

• A more general form for the procedure call

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

where  $T \equiv T_1, \ldots, T_m$  is a tuple of function instances and  $0 \leq m$ . Note that each  $T_i$   $(i = 1, \ldots, m)$  is either a function variable declared in the current or "higher" procedure, or a primitive function symbol  $F_k$ . (For a discussion of an alternative, more complicated form of the procedure call statements, see §6.2).

Notation 4.10.1. We will use the notation  $\overline{R}$ , ..., and the general form as follows for  $Rec_2$  procedures.



22

#### **Remark 4.10.2.** Note the differences between *RelRec* and *Rec*<sub>2</sub>:

- (a) In *RelRec* the function symbols  $\phi$  are interpreted as (oracles for) function parameters, while in *Rec*<sub>2</sub> 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**<sub>2</sub> 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  $[t]]_{\varphi}^{A}$  for the semantic function of t when function variables  $\phi$  in t are interpreted as  $\varphi$ . The definitions are similar to those in §4.4 except that we need to give the semantics of the new term constructor as follows:

$$\llbracket \phi_i(t_1, \dots, t_m) \rrbracket_{\varphi_i}^A \sigma \simeq \begin{cases} \varphi_i(\llbracket t_1 \rrbracket^A \sigma, \dots, \llbracket t_m \rrbracket^A \sigma) & \text{if } \llbracket t_j \rrbracket^A \sigma \downarrow \ (j = 1, \dots, m) \\ \uparrow & \text{otherwise.} \end{cases}$$

Similarly as for *RelRec*, we will have functions  $Rest_{\varphi}^{A^*}$ ,  $Comp_{\varphi}^{A^*}$ ,  $CompLength_{\varphi}^{A^*}$ , and  $[S]_{\varphi}^{A^*}$ , depending on the interpretation of oracles.

Therefore, the semantics of a *Rec*<sub>2</sub> procedure  $\bar{R}$ :  $\pi \times u \rightarrow v$ , where  $\bar{R} \equiv$ 

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

(with  $\phi : \pi$  and a : u) is a functional  $[\bar{R}]^A : A^{\pi} \times A^u \rightharpoonup A^v$  given by

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

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

In this way we define the notion of  $Rec_2$  computability and  $Rec_2^*$  computability.

We will prove (Theorem 4.10.5) a correspondence between *RelRec* and *Rec*<sub>2</sub> computability. We need two lemmas.

#### Lemma 4.10.3. (*RelRec* $\Rightarrow$ *Rec*<sub>2</sub>)

Let *R* be a *RelRec* procedure of type  $u \to v$  with oracle tuple  $\phi$  of type  $\pi$ . We can transform *R* to a *Rec*<sub>2</sub> procedure  $\overline{R}$  of type  $\pi \times u \to v$  such that for all  $\varphi : \pi$  and x : u,

$$\llbracket R \rrbracket^A(\varphi, x) \simeq \llbracket R \rrbracket^A_{\varphi}(x).$$

#### **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  $\phi$ " to every inner procedure of *R*. Some points to be notes are:

- (1) The oracle call statement  $\mathbf{x} := \phi(t)$  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  $\phi_i$  in any inner procedures has the same interpretation as  $\phi_i$  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.

### Lemma 4.10.4. ( $Rec_2 \Rightarrow RelRec$ )

Let  $\overline{R}$  be a  $Rec_2$  procedure of type  $\pi \times u \to v$ . We can transform  $\overline{R}$  to a *RelRec* procedure R of type  $u \to v$  with oracle  $\phi$  of type  $\pi$  such that for all  $\varphi : \pi$  and x : u,

$$\llbracket \overline{R} \rrbracket^A(\varphi, x) \simeq \llbracket R \rrbracket^A_{\varphi}(x).$$

### **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 assignments as oracle calls) In R the new term constructor makes it possible that a term t, in an assignment x := 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

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

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

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

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  $Rec_2$  procedure  $R \equiv$ 

$$\begin{array}{c} \mbox{functions } \phi_1 \\ \hline P' & \longleftarrow \overline{R}' \\ \hline \mbox{in a out b aux c} \\ \hline \hdots \\ \mbox{x}_1 := P'(\phi_1, t_1); \\ \hline \hdots \\ \mbox{x}_2 := P'(\mathsf{F}_k, t_2); \\ \hline \hdots \\ \hline \hdots \\ \end{array}$$

where  $\bar{R}' \equiv$ 

functions $\rho$
in a' out b' aux c'
S

We can see that  $\rho$  is being interpreted as two different functions, corresponding to the interpretations of  $\phi_1$  and the primitive function symbol  $F_k$  respectively. We can transform the above  $Rec_2$ procedure to a *RelRec* procedure  $R \equiv$ 

oracles $\phi_1$
$P_1 \Longleftarrow R_1, P_2 \Longleftarrow R_2$
in a out b aux c
$\mathbf{x}_1 := P_1(t_1);$
$\mathbf{x}_2 := P_2(t_2);$

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

in a' out b' aux c'
$S_i$

 $S_1$  and  $S_2$  are obtained from S by replacing all occurrences of  $\rho$  by  $\phi_1$  and  $\mathsf{F}_k$  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 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  $F : A^{\pi} \times A^{u} \rightarrow A^{v}$  be a second-order functional. *F* is computable by a *Rec*<sub>2</sub> procedure  $\overline{R}$  iff there exist a *RelRec* procedure *R* such that for all  $\varphi : \pi$  and x : u,

$$F(\varphi, x) \simeq \llbracket \overline{R} \rrbracket^A(\varphi, x) \simeq \llbracket R \rrbracket^A_{\varphi}(x).$$

# 4.11. While procedures

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

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

The semantics of *While* procedures are derived along similar lines as those for the semantics of *Rec* procedures. Details can be found in [20].

#### **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) While(A) is the class of functions *While* computable on A.

**Definition 4.11.2.** A *While*<sup>\*</sup>( $\Sigma$ ) procedure is a *While*( $\Sigma$ <sup>\*</sup>) procedure in which the *input* and *output* variables are *simple*. (However the auxiliary variables may be starred.)

### **Definition 4.11.3.** (*While*<sup>\*</sup> computable functions)

- (a) A function f on A is computable on A by a While<sup>\*</sup> procedure P if  $f = [\![P]\!]^A$ . It is While<sup>\*</sup> computable on A if it is computable on A by some While<sup>\*</sup> procedure.
- (b)  $While^*(A)$  is the class of functions  $While^*$  computable on A.

We will not discuss *While* computability any further, since [20] contains a full discussion. However, we need to mention following significant theorem proved in [18].

**Theorem 4.11.4.** (a) **While** $(A) = \mu PR(A)$ .

(b) **While**<sup>\*</sup>(A) =  $\mu PR^{*}(A)$ .

# 5. From $\mu PR$ to ACP

In this section, we will prove that, if a function f on A is  $\mu PR$  computable, then it is ACP computable; and hence, if f is  $\mu PR^*$  computable, it is  $ACP^*$  computable. Even though we gave formal definitions for ACP and  $\mu PR$  schemes in Section 3, we prefer to present informal proofs in this section, in the sense that we ignore the distinction between syntax and semantics for both ACP and  $\mu PR$ . We believe that our informal approach is convincing.

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

(a) 
$$f(x) \simeq F_k(x)$$
 where  $\mathsf{F}_k \in Func(\Sigma)$   
(b)  $g(\vec{x}) = x_i$   
(c)  $h(x) \simeq \begin{cases} h_2(x) & \text{if } h_1(x) \downarrow \mathsf{t} \\ h_3(x) & \text{if } h_1(x) \downarrow \mathsf{f} \\ \uparrow & \text{if } h_1(x) \uparrow \end{cases}$ 

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

**Lemma 5.2.** Let f be a function defined by  $f(x) \simeq h(g_1(x), \ldots, g_m(x))$ . If  $h, g_1, \ldots, g_m \in ACP(A)$ , then so is f.

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

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

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

**Lemma 5.4.** Let f be a function defined by  $f(x) \simeq \mu z[g(x, z) \downarrow \mathbf{t}]$ . If  $g \in ACP(A)$ , then so is f.

The proofs of Lemma 5.3 and 5.4 are given in Appendices A.1 and A.2.

**Theorem 5.5.**  $\mu PR(A) \subseteq ACP(A)$ .

#### **Proof:**

Following Lemmas 5.1–5.4, we can associate, with each  $\mu PR$  scheme for a function f, an ACP scheme for f, by structural induction on  $\mu PR$  schemes.

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

# 6. From ACP to Rec

In this section, we will prove

$$ACP(A) \subseteq Rec_2(A).$$

We will prove this by induction on the schemes of ACP, *i.e.* associate to every ACP scheme a  $Rec_2$  procedure for the same functional. From this will follow

$$ACP^{1}(A) \subseteq Rec(A)$$
 (cf. Definition 3.1.1 for  $ACP^{1}$ )

and hence

$$ACP^{*1}(A) \subseteq Rec^*(A)$$
 (cf. Definition 3.1.2 for  $ACP^{*1}$ )

### 6.1. Translating ACP to Rec<sub>2</sub>

In this subsection, we describe the translation of *ACP* to *Rec*<sub>2</sub> (both second order systems). As a Corollary (6.1.7), we derive the translation of *ACP*<sup>1</sup> to *Rec* (both first order systems). For the latter translation, we need to replace *Rec*<sub>2</sub> by *RelRec* (using Theorem 4.10.5; see Remark 6.1.8). This replacement of *Rec*<sub>2</sub> by *RelRec* is also convenient to one of the steps of this (second order) translation (Lemma 6.1.5).

**Lemma 6.1.1.** 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 **Rec**<sub>2</sub>-computable.

**Lemma 6.1.2.** Let  $F \equiv F^A$ ,  $G \equiv G^A$ , and  $H \equiv H^A$  be functionals, and let F be defined by

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

If G and H are  $Rec_2$ -computable, then so is F.

**Lemma 6.1.3.** Let  $F \equiv F^A$  and  $G \equiv G^A$  be functionals, and let F be defined by

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

If G is  $Rec_2$ -computable, then so is F.

**Lemma 6.1.4.** Let  $F \equiv F^A$ ,  $G \equiv G^A$ , and  $H \equiv H^A$  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  $Rec_2$ -computable, then so is F.

The LFP scheme, treated in the following lemma, is the most interesting case for the proof of Theorem 6.1.6 below.

**Lemma 6.1.5.** Let  $F_1 \equiv \mathsf{F}_1^A, \ldots, F_n \equiv \mathsf{F}_n^A, G_1 \equiv \mathsf{G}_1^A, \ldots, G_n \equiv \mathsf{G}_n^A$  be functionals, where  $\mathsf{F}_1, \ldots, \mathsf{F}_n$  are defined by

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

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

If  $G_1, \ldots, G_n$  are *Rec*<sub>2</sub>-computable, then so are  $F_1, \ldots, F_n$ .

where

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

### **Proof:**

We can prove this lemma by constructing directly the  $Rec_2$  procedures for  $F_1, \ldots, F_n$  analogous to the proofs of Lemmas 6.1.1–6.1.4. However, this would require repeating many of the technical details in the proofs of Lemmas 4.10.3 and 4.10.4. We therefore construct corresponding *RelRec* procedures.

By assumption, we have  $Rec_2$  procedures  $\bar{R}_{G_1}, \ldots, \bar{R}_{G_n}$ , such that for  $i = 1, \ldots, n$ ,  $G_i = [\![\bar{R}_{G_i}]\!]^A$ .

By Theorem 4.10.5, we have, for i = 1, ..., n, *RelRec* procedures  $R_{G_i} \equiv$ 

oracles $\phi, \rho_1, \dots, \rho_n$
$P_{G_{i,1}} \longleftarrow R_{G_{i,1}}, \dots, P_{G_{i,m_i}} \longleftarrow R_{G_{i,m_i}}$
in a $_{G_{i,1}}$ a $_{G_{i,2}}$ out b $_{G_i}$ aux c $_{G_i}$
$S_{G_i}$

such that for all  $\varphi$ ,  $\varrho_1, \ldots, \varrho_n, x$  and  $y_i, [\![\bar{R}_{G_i}]\!]^A(\varphi, \varrho_1, \ldots, \varrho_n, x, y_i) \simeq [\![R_{G_i}]\!]^A_{\varphi, \varrho_1, \ldots, \varrho_n}(x, y_i)$ . We can then construct, for  $i = 1, \ldots, n$ , *RelRec* procedures  $R_{F_i} \equiv$ 

oracles $\phi$
$P_{G_1} \longleftrightarrow R^P_{G_1}, \ldots, P_{G_n} \Longleftarrow R^P_{G_n}$
in $\mathtt{a}_{F_{i,1}}$ $\mathtt{a}_{F_{i,2}}$ out $\mathtt{b}_{F_i}$
$b_{F_i} := P_{G_i}(a_{F_{i,1}},  a_{F_{i,2}})$

where, for  $i = 1, \ldots, n, R_{G_i}^p \equiv$ 

$P_{G_{i,1}} \longleftarrow R^P_{G_{i,1}}, \dots, P_{G_{i,m_i}} \longleftarrow R^P_{G_{i,m_i}}$
in a $_{G_{i,1}}$ a $_{G_{i,2}}$ out b $_{G_i}$ aux c $_{G_i}$
$S^P_{G_i}$

Here, for i = 1, ..., n,  $R_{G_{i,1}}^P, ..., R_{G_{i,m_i}}^P$ , and  $S_{G_i}^P$  are the same as  $R_{G_{i,1}}, ..., R_{G_{i,m_i}}$ , and  $S_{G_i}$ , except that all occurrences of oracle call statements of the form  $c := \rho_j(t)$  are replaced by procedure calls  $c := P_{G_j}(a_{G_{i,1}}, t)$ . We are, essentially, replacing oracle call statements by simultaneous recursive calls.

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

$$\llbracket R_{F_i} \rrbracket_{\varphi}^A(x, y_i) \simeq F_i(\varphi, x, y_i).$$
<sup>(7)</sup>

Then by Theorem 4.10.5, there exist *Rec*<sub>2</sub> procedures  $\bar{R}_{F_1}, \ldots, \bar{R}_{F_n}$  such that, for all  $\varphi$ , x and  $y_i$ 

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

Hence  $F_1, \ldots, F_n$  are *Rec*<sub>2</sub>-computable. The proof of (7) is given in Appendix A.3.

**Theorem 6.1.6.**  $ACP(A) \subseteq Rec_2(A)$ .

#### **Proof:**

By induction on schemes for *ACPs*. Precisely, we will associate, with each *ACP* scheme, a *Rec*<sub>2</sub> procedure. For schemes I-III, use Lemma 6.1.1. For schemes IV-VI, use Lemmas 6.1.2–6.1.4. For scheme VIII, use Lemma 6.1.5. Recall Remark 3.1.7 that we can ignore scheme VII for first-order algebras.  $\Box$ 

Corollary 6.1.7.  $ACP^{1}(A) \subseteq Rec(A)$ .

### **Proof:**

For any function  $f \in ACP^{1}(A)$ , it follows directly from Theorem 6.1.6 and Theorem 4.10.5 that there are a *Rec*<sub>2</sub> procedure  $\overline{R}$  without function variables in the main procedure and a *RelRec* procedure *R*, such that for all x : u,

$$f(x) \simeq \llbracket \overline{R} \rrbracket^A(x) \simeq \llbracket R \rrbracket^A(x).$$

Since  $f \in ACP^{1}(A)$ , it follows that  $\overline{R}$  has no first order arguments, and so R has no oracles, and is therefore a *Rec* procedure, which ends the proof.

**Remark 6.1.8.** Note that (unlike the proof of Lemma 6.1.5) *RelRec* is necessary in above proof, in the sense that we need to eliminate all inner function calls in the *Rec*<sub>2</sub> procedure, which essentially translates the *Rec*<sub>2</sub> procedure to a *RelRec* procedure — in fact a *Rec* procedure in this case.

Corollary 6.1.9.  $ACP^{*1}(A) \subseteq Rec^*(A)$ .

### 6.2. Digression: The programming language $\lambda Rec_2$

Our translation of ACP into  $Rec_2$  (Lemma 6.1.1–6.1.5) made use of the fact that in ACP, the function substitution scheme

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

is redundant [6, Theorem 3] for first-order algebras.

The simple individual substitution scheme

VII: 
$$F(\varphi, x) \simeq G(\varphi, x, H(\varphi, x)))$$

can easily be interpreted by a *Rec*<sub>2</sub> functional (see Lemma 6.1.4) or a *RelRec* functional, for that matter.

Suppose we were unaware that scheme VII was redundant in *ACP*. We should then have had to interpret it by a recursive functional as follows.

**Lemma 6.2.1.** Assume that we have procedures  $R_G$  and  $R_H$  such that  $G = [\![R_G]\!]^A$  and  $H = [\![R_H]\!]^A$ :  $R_G \equiv$ 

functions $\phi$ , $\rho$
$P_{G_1} \longleftarrow R_{G_1}, \ldots, P_{G_m} \longleftarrow R_{G_m}$
in $\mathtt{a}_G$ out $\mathtt{b}_G$ aux $\mathtt{c}_G$
$S_G$

$$R_H \equiv$$

functions $\phi$
$P_{H_1} \Longleftarrow R_{H_1}, \ldots, P_{H_m} \Longleftarrow R_{H_m}$
in $\mathtt{a}_{H_x} \mathtt{a}_{H_y}$ out $\mathtt{b}_H$ aux $\mathtt{c}_H$
$S_H$

We can construct a procedure  $R_F$  as follows

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

Then  $\llbracket R_F \rrbracket^A(\varphi, x) \simeq F(\varphi, x)$ .

Note now, however, that  $R_F$  is *not* a  $Rec_2$  procedure, as we have defined. It is in an extended language, which we call  $\lambda Rec_2$ , in which procedure calls have a more complicated form

$$\mathbf{x} := P(T, t) \tag{8}$$

where  $T \equiv T_1, \ldots, T_m$  ( $m \ge 0$ ) and each  $T_i$  can have any one of the following forms:

- (1) A primitive function symbol  $F_k$ .
- (2) A function variable  $\phi$  declared in the current procedure.
- (3) A term abstraction λx · t obtained by λ abstraction from a term. If λx · t instantiates a function symbol ρ, then the term ρ(t') is instantiated by (λx · t)(t'), which is rewritten as t[x/t'] (i.e. automatic β-conversion).
- (4) A procedure abstraction λy · P(φ, x, y). If λy · P(φ, x, y) instantiates a function symbol ρ, then the term ρ(t) is instantiated by (λy · P(φ, x, y))(t), which is rewritten as P(φ, x, t).

Note that in the definition of  $Rec_2$  we only have cases (1) and (2). Note also that we are assuming automatic  $\beta$ -conversion in the operational semantics of  $\lambda Rec_2$ .

Next, in order to proceed with our translation of *Rec* into *While* (Section 7), we would have to prove:

$$\lambda Rec_2$$
 is *reducible* to  $Rec_2$  (and hence to  $RelRec$ ). (9)

The proof of (9) involves showing that all terms  $T_i$  defined by  $\lambda$ -abstraction occurring as parameters in procedure calls (8) eventually occur in the context of an application to an individual term t, and so disappear by  $\beta$ -conversion, so that their call was redundant (cf. the proof of Theorem 4 in [19, §7.6]).

This proof actually *parallels* the proof of the redundancy of scheme VII in *ACP*. So in fact, the proof of the redundancy of scheme VII in *ACP* in [6] saved us the trouble of having to prove (9).

# 7. From $Rec^*$ to $\mu PR^*$

In this section, we want to prove that, if a function f over A is **Rec**<sup>\*</sup>-computable, then it is  $\mu PR^*$  computable. We will first prove that **Rec**<sup>\*</sup> computability implies **While**<sup>\*</sup> 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 *Comp*<sup>A</sup> and *CompLength*, which we prove to be *While*<sup>\*</sup> computable.

The proof is parallel to the argument in [20,  $\S4$ ], which this section follows closely, except that we are considering *Rec* procedures, while [20] considers *While* procedures. We just present the differences between them, and interested readers can refer to [20] 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  $\Sigma$  and  $\Sigma^*$ , *i.e.*, a family gn of effective mappings from expressions E to natural numbers  $\lceil E \rceil = gn(E)$ , which satisfy certain basic properties: (i)  $\lceil E \rceil$  increases strictly with the complexity of E, and in particular, the code of an expression is larger than those of its subexpressions; (ii) sets of codes of the various syntactic classes, and of their respective subclasses, such as  $\{\lceil t \rceil \mid t \in Term\}$ ,  $\{\lceil t \rceil \mid t \in Term_s\}$ , etc., are primitive recursive; (iii) we can go primitive recursively from codes of expressions to codes of their immediate subexpressions, and vice versa.

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

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

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

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

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

The modified state representing function,  $\operatorname{Rep}_{\mathbf{x}*}^A$ :  $\operatorname{State}(A) \cup \{*\} \rightarrow \mathbb{B} \times A^u$ , is defined by

$$\begin{array}{lll} \operatorname{\textit{Rep}}_{\mathtt{x}*}^{A}(\sigma) & = & (\mathtt{t}, \, \sigma[\mathtt{x}]) \\ \operatorname{\textit{Rep}}_{\mathtt{x}*}^{A}(*) & = & (\mathtt{f}, \, \boldsymbol{\delta}_{A}^{u}) \end{array}$$

where  $\delta_A^u$  is the default tuple of type u in A.

#### 7.3. Representation of term evaluation

Let x be a *u*-tuple of variables. Let  $Term_x$  be the class of all  $Rec(\Sigma)$  program terms (see §4.1 for definition) with variables among x only, and for all sorts s of  $\Sigma$ , let  $Term_{x,s}$  be the class of such terms of sort s. Similarly we write  $TermTup_x$  for the class of all term tuples with variables among x only, and  $TermTup_{x,v}$  for the class of all v-tuples of such terms.

The term evaluation function on A relative to x,  $TE_{x,s}^A$ :  $Term_{x,s} \times State(A) \rightarrow A_s$ , defined by

$$TE_{\mathbf{x},s}^{A}(t,\sigma) \simeq [t]^{A}\sigma,$$

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

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

where  $\sigma$  is any state on A such that  $\sigma[\mathbf{x}] = a$ . (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:



Further, an evaluation function for tuples of terms can easily be defined in a similar fashion.

# 7.4. Representation of computation step function

Let  $AtSt_x$  be the class of  $Rec(\Sigma)$  atomic statements (see §4.5 for definition) with variables among x only. The *atomic statement evaluation function on A relative to* x,  $AE_x^A : AtSt_x \times State(A) \rightarrow State(A)$ , defined by

$$AE_{\mathbf{x}}^{A}(S,\sigma) \simeq \langle \! |S \rangle \! \rangle^{A} \sigma$$

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

$$ae_{\mathbf{x}}^{A}(\lceil S \rceil, a) \simeq (\langle \! \langle S \! \rangle \rangle^{A} \sigma)[\mathbf{x}],$$

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



Next, let  $Stmt_x$  be the class of  $Rec(\Sigma)$  statements (see §4.1 for definition) with variables among x only, and define

$$\operatorname{Rest}_{\mathtt{x}}^{A} =_{df} \operatorname{Rest}^{A} \upharpoonright (\operatorname{Stmt}_{\mathtt{x}} \times \operatorname{State}(A)) :$$

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

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

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



Note that *first* is a function from  $\mathbb{N}$  to  $\mathbb{N}$ , and (unlike *rest*<sup>A</sup><sub>x</sub> and most of the other representing functions here) does not depend on A or x.

Next, the computation step function (relative to x)

$$Comp_{x}^{A} = Comp^{A} \upharpoonright (Stmt_{x} \times State(A) \times \mathbb{N}) : Stmt_{x} \times State(A) \times \mathbb{N} \rightarrow State(A) \cup \{*\}$$

is *represented* by the function *comp*  $_{\mathbf{x}}^{A}$ :  $\lceil Stmt_{\mathbf{x}} \rceil \times A^{u} \times \mathbb{N} \rightarrow \mathbb{B} \times A^{u}$ , which is defined so as to make the following diagram commute:

We put

$$comp_{\mathbf{x}}^{A}(\ulcornerS\urcorner, a, n) = (notover_{\mathbf{x}}^{A}(\ulcornerS\urcorner, a, n), state_{\mathbf{x}}^{A}(\ulcornerS\urcorner, a, n))$$

with the two "component functions"

$$notover_{\mathbf{x}}^{A} : \lceil Stmt_{\mathbf{x}} \rceil \times A^{u} \times \mathbb{N} \rightarrow \mathbb{B}$$
  
state  $_{\mathbf{x}}^{A} : \lceil Stmt_{\mathbf{x}} \rceil \times A^{u} \times \mathbb{N} \rightarrow A^{u}$ 

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

## 7.5. Representation of statement evaluation

The statement evaluation function on A relative to x,  $SE_x^A : State(A) \rightarrow State(A)$ , defined by

$$SE_{\mathbf{x}}^{A}(S,\sigma) \simeq [S]^{A}\sigma,$$

is *represented* by the (partial) function  $se_x^A$ :  $\lceil Stmt_x \rceil \times A^u \rightarrow A^u$ , defined by

$$se_{\mathbf{x}}^{A}(\ulcorner S\urcorner, a) \simeq (\llbracket S \rrbracket^{A}\sigma)[\mathbf{x}]$$

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



### 7.6. Representation of procedure evaluation

Let a, b, c be pairwise disjoint lists of variables, with types a : u, b : v and c : w. Let **Proc**<sub>a,b,c</sub> be the class of **Rec** procedures of type  $u \to v$ , with declaration in a out b aux c. The procedure evaluation function on A relative to a, b, c, **PE**<sup>A</sup><sub>a,b,c</sub>: **Proc**<sub>a,b,c</sub>  $\times A^u \to A^v$ , defined by

$$\boldsymbol{P}\boldsymbol{E}_{a,b,c}^{A}(R,a) \simeq [\![R]\!]^{A}(a)$$

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

$$pe^{A}_{\mathbf{a},\mathbf{b},\mathbf{c}}(\lceil R \rceil, a) \simeq \llbracket R \rrbracket^{A}(a).$$

In other words, the following diagram commutes:



### 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 *first* :  $\mathbb{N} \to \mathbb{N}$  is primitive recursive, and hence *While* computable on *A*, for any N-standard  $\Sigma$ -algebra *A*.

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

- (a)  $ae_{\mathbf{x}}^{A^*}$  and *rest*  $_{\mathbf{x}}^{A^*}$  are *While* computable in  $\langle te_{\mathbf{a},s}^{A^*} | s \in Sort(\Sigma^*) \rangle$  on  $A^*$ .
- (b)  $comp_{x}^{A^{*}}$ , and its two component functions *notover*  $_{x}^{A^{*}}$  and *state*  $_{x}^{A^{*}}$ , are *While* computable in *ae*  $_{x}^{A^{*}}$  and *rest*  $_{x}^{A^{*}}$  on  $A^{*}$ .
- (c) se  $_{x}^{A^{*}}$  is While computable in comp  $_{x}^{A^{*}}$  on  $A^{*}$ .
- (d)  $pe_{a,b,c}^{A^*}$  is **While** computable in  $se_x^{A^*}$  on  $A^*$ , where  $x \equiv a, b, c$ .
- (e)  $te_{x,s}^{A^*}$  is While computable in  $pe_{x,y,\langle\rangle}^A$  on  $A^*$ , where y is a variable of sort s, not in 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* [18] on the set of Gödel numbers of expressions of this class [20].

The proofs are analogous to those for [20, Lemma 4.2]. Note, for part (b)–(e), the proofs in [20] 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  $\Sigma^*$  algebras.

For part (a), clearly, the function  $ae_x^{A^*}$  is primitive recursive on  $A^*$ , since we only have two kinds of atomic statements, skip and concurrent assignment. The function rest  $_x^{A^*}$  is course of value recursive on nat with range sort nat, which is reducible to primitive recursive on nat (see proof for [20, Lemma 4.2]). Hence, they are While computable on  $A^*$ . Note that a procedure call statement S is not an atomic statement.

Lemma 7.7.3. The following are equivalent.

- (a) For all x and s, the term evaluation representing function  $te_{x,s}^{A^*}$  is While computable on  $A^*$ .
- (b) For all x, the atomic statement evaluation representing function  $ae_x^{A^*}$ , and the representing function *rest*<sub>x</sub><sup>A^\*</sup>, are *While* computable on  $A^*$ .
- (c) For all x, the computation step representing function  $comp_{x}^{A^{*}}$ , and its two component functions *notover*  $_{x}^{A^{*}}$  and *state*  $_{x}^{A^{*}}$ , are *While* computable on  $A^{*}$ .
- (d) For all x, the statement evaluation representing function se  $a_{x}^{A^{*}}$  is While computable on  $A^{*}$ .
- (e) For all a, b, c, the procedure evaluation representing function  $pe_{a, b, c}^{A^*}$  is While computable on  $A^*$ .

### **Proof:**

From the transitivity of relative computability [20, Lemma 3.32], and Lemma 7.7.2.

# 7.8. *Rec*<sup>\*</sup> computability $\implies \mu PR^*$ computability

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

For a proof of Lemma 7.8.1, see [18, 20].

### **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  $f = [R]^A$ . Let

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

It follows from Lemmas 7.7.3 and 7.8.1 that there exist a function  $pe_{a,b,c}^{A^*}$  which is *While* computable on  $A^*$ , actually *While*<sup>\*</sup> computable on A, since the input and output variables are simple. Substituting the variable for the Gödel number in the *While*<sup>\*</sup> procedure for  $pe_{a,b,c}^{A^*}$  by the numeral for the Gödel number of R, we obtain the *While*<sup>\*</sup> 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 [20, Remark 2.31].

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

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

#### **Proof:**

From Theorems 7.8.2 and 4.11.4.

## 8. Conclusion

We have proved that

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

via the circle of inclusions in Figure 1 for *N-standard many-sorted* algebras *A*. Some questions which arise from our work are:

## 8.1. Simultaneous vs. simple LFP scheme

The *ACP* schemes introduced in §3.1 differ from those in [6] by using a *simultaneous* instead of *simple* least fixed point scheme (*cf.* Remark 3.1.5). An interesting question is:

In the absence of product types, can our *ACP*<sup>\*</sup> 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

$$ACP^1(A) = \mu PR(A)$$

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

In connection with this, we have shown that  $\mu PR(A) \subseteq ACP^1(A)$  and  $ACP^1(A) \subseteq Rec$  (Theorems 5.5 and 6.1.7). The remaining question is, whether  $Rec \subseteq \mu PR(A)$ . In Remark 4.6.6, we discuss the difficulty in avoiding the use of arrays when defining the semantics of *Rec* procedures. We therefore conjecture that  $Rec \subseteq \mu PR(A)$  is not true in general; however, we lack a proof.

### 8.3. Second-order version of equivalence results

Since  $ACP^*$  is a second-order system, and  $\mu PR^*$  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  $ACP^{*1}$  of  $ACP^*$ . An alternative, and perhaps better, approach would be to work with second-order versions of  $\mu PR^*$  and *While*\* and then prove the complete circle of inclusions in Figure 1 for second-order systems. Our results for the first-order systems would then follow easily.

# A. Proofs omitted from previous sections

#### A.1. Proof of Lemma 5.3.

Let

$$\varphi_{x,1} =_{df} \lambda z \cdot f_1(x,z)$$

$$\dots,$$

$$\varphi_{x,m} =_{df} \lambda z \cdot f_m(x,z);$$

and

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

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

$$\widehat{F}_{x,1} =_{df} \lambda \varphi_1 \cdot \ldots \cdot \lambda \varphi_m \cdot F_{\varphi,x,1}$$

$$\ldots,$$

$$\widehat{F}_{x,m} =_{df} \lambda \varphi_1 \cdot \ldots \cdot \lambda \varphi_m \cdot F_{\varphi,x,m}.$$

Note that  $F_{x,1}, \ldots, F_{x,m}$  are **ACPs** by scheme IV. It is easy to verify that

(i)  $(\varphi_{x,1}, \ldots, \varphi_{x,m})$  is a *fixed point* of the tuple  $(\widehat{F}_{x,1}, \ldots, \widehat{F}_{x,m})$ . Moreover, if we define, for  $1 \leq i \leq m$ ,

$$\varphi_{x,i}^{0} = \lambda z \cdot \bot,$$
  
 
$$\varphi_{x,i}^{k+1} = \widehat{F}_{x,i}(\varphi_{x,1}^{k}, \dots, \varphi_{x,i}^{k}),$$

then:

(*ii*) for any fixed point  $(\psi_1, \dots, \psi_m)$  of  $(\widehat{F}_{x,1}, \dots, \widehat{F}_{x,m})$ ,  $\varphi_{x,i}^k \sqsubseteq \psi_i \quad \text{ for } 1 \le i \le m$ 

by induction on k, and hence

$$\mathop{\sqcup}\limits_{k=0}^{\infty}\varphi_{x,i}^k\ \sqsubseteq\ \psi_i\ ,$$

(*iii*) for all  $z \in \mathbb{N}$ ,

$$\varphi_{x,i}(z) \simeq \varphi_{x,i}^{z+1}(z) \quad \text{for } 1 \le i \le m$$

by induction on z, and hence

$$\varphi_{x,i} \sqsubseteq \bigsqcup_{k=0}^{\infty} \varphi_{x,i}^k$$

From (i), (ii) and (iii) follows

$$\varphi_{x,i} \;=\; \underset{k=0}{\overset{\infty}{\sqcup}} \varphi_{x,i}^k$$

and hence

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

from which the lemma follows.

# A.2. Proof of Lemma 5.4.

Define (using informal but suggestive notation) the function

$$f'(x,z) \simeq \mu y \ge z[g(x,y) \downarrow \mathbf{t}].$$

Note that

$$f'(x,z) \simeq \begin{cases} z & \text{if } g(x,z) \downarrow \mathbf{t} \\ f'(x,z+1) & \text{if } g(x,z) \downarrow \mathbf{f} \\ \uparrow & \text{otherwise.} \end{cases}$$

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

$$\begin{split} \varphi_x &= \lambda z \cdot f'(x,z) \\ F_{\varphi,x}(z) &\simeq \begin{cases} z & \text{if } g(x,z) \downarrow \mathbf{t} \\ \varphi(z+1) & \text{if } g(x,z) \downarrow \mathbf{f} \\ \uparrow & \text{otherwise} \end{cases} \\ \widehat{F}_x &= \lambda \varphi \cdot F_{\varphi,x}. \end{split}$$

It is easy to show that  $\varphi_x = LFP(\hat{F}_x)$  by a method like that used in A.1, from which the lemma follows.

A.3. Proof of  $[\![\bar{\mathbf{R}}_{F_i}]\!]^A(\varphi, x, y_i) \simeq [\![\mathbf{R}_{F_i}]\!]^A_{\varphi}(x, y_i) \simeq \mathbf{F}_i(\varphi, x, y_i)$  [in Lemma 6.1.5].

In order to prove (7) we prove, for  $1 \le i \le n$ ,

$$\lambda y_i \cdot F_i(\varphi, x, y_i) \sqsubseteq \lambda y_i \cdot [\![R_{F_i}]\!]^A_{\varphi}(x, y_i), \tag{10}$$

$$\lambda y_i \cdot \llbracket R_{F_i} \rrbracket_{\varphi}^A(x, y_i) \sqsubseteq \lambda y_i \cdot F_i(\varphi, x, y_i).$$
<sup>(11)</sup>

To prove (10), put, for  $1 \le i \le n$ ,

$$\begin{array}{rcl} \varrho_i^0 & = & \bot \\ \varrho_i^{k+1} & = & \hat{G}_i^{\varphi,x}(\varrho_1^k,\dots,\varrho_n^k) \end{array}$$

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

for all 
$$k, \ \varrho_i^k \sqsubseteq \lambda y_i \cdot [\![R_{F_i}]\!]^A_{\varphi}(x, y_i), \text{ for } 1 \le i \le n.$$
 (12)

We will prove this by simultaneous induction on k.

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

$$[\mathbf{R}_{G_i}]]^A_{\varphi,\varrho_1^k,\ldots,\varrho_n^k}(x,y_i) \simeq G_i(\varphi,\varrho_1^k,\ldots,\varrho_n^k,x,y_i) \simeq \varrho_i^{k+1}(y_i).$$
(13)

By induction hypothesis  $\varrho_i^k \sqsubseteq \lambda y_i \cdot [\![R_{F_i}]\!]^A_{\varphi}(x, y_i)$ , for i = 1, ..., n. Therefore by the monotonicity theorem for *RelRec* procedures (Theorem 4.9.7), for i = 1, ..., n

$$\lambda y_i \cdot \llbracket \mathbf{R}_{G_i} \rrbracket^A_{\varphi, \varrho_1^k, \dots, \varrho_n^k}(x, y_i) \sqsubseteq \lambda y_i \cdot \llbracket \mathbf{R}_{G_i} \rrbracket^A_{\varphi, \lambda y_1 \cdot \llbracket \mathbf{R}_{F_1} \rrbracket^A_{\varphi}(x, y_1), \dots, \lambda y_n \cdot \llbracket \mathbf{R}_{F_n} \rrbracket^A_{\varphi}(x, y_n)}(x, y_i).$$

So by (13) and Sublemma A.3.1 below,

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

which proves (12) by induction on k, and hence (10).

The reverse direction (11) is proved by simultaneous course of values induction on *CompLength*( $R,\varphi,a$ ). Here, *CompLength*( $R,\varphi,a$ ) denotes the computation length of procedure R with inputs  $\varphi$  and a, defined by

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

where  $R \equiv \langle D^{p} : D^{v} : S \rangle$ , with oracles  $\phi$  integreted as  $\varphi$ ,  $D^{v} \equiv$  in a out b aux c, and  $\sigma[a] = a$ . Assume that, for  $1 \leq i \leq n$ , for all inputs  $\varphi$ , x and  $y_i$ , if **CompLength**( $R_{F_i}, \varphi, (x, y_i)$ ) < l, then

$$\llbracket R_{F_i} \rrbracket^A_{\varphi}(x, y_i) \downarrow \implies \llbracket R_{F_i} \rrbracket^A_{\varphi}(x, y_i) = F_i(\varphi, x, y_i).$$
(14)

Suppose now that for some  $\varphi$ , x and  $y_i$ 

$$\llbracket R_{F_i} \rrbracket_{\varphi}^A(x, y_i) \downarrow$$
 and *CompLength* $(R_{F_i}, \varphi, (x, y_i)) = l$ .

By Sublemma A.3.1 below and  $\llbracket R_{F_i} \rrbracket^A_{\varphi}(x, y_i) \downarrow$ , we have:

$$\llbracket R_{F_i} \rrbracket^A_{\varphi}(x, y_i) = \llbracket R_{G_i} \rrbracket^A_{\varphi, \varrho}(x, y_i)$$
  
=  $G_i(\varphi, \varrho, x, y_i).$ 

where  $\varrho \equiv \varrho_1, \ldots, \varrho_n$  and

$$\varrho_i = \lambda z_i \cdot [\![\mathbf{R}_{F_i}]\!]^A_{\varphi}(x, z_i) \quad i = 1, \dots, n.$$

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, ..., n) will only be applied to certain z (say) which are the values of some terms t, such that

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

Therefore for all such z, by induction hypothesis

$$\llbracket R_{F_j} \rrbracket_{\varphi}^A(x,z) = F_j(\varphi, x, z)$$

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

$$\begin{split} & \llbracket R_{F_i} \rrbracket_{\varphi}^A(x, y_i) \\ &= G_i(\varphi, \lambda z_1 \cdot \llbracket R_{F_1} \rrbracket_{\varphi}^A(x, z_1), \dots, \lambda z_n \cdot \llbracket R_{F_n} \rrbracket_{\varphi}^A(x, z_n), x, y_i) \\ &= G_i(\varphi, \lambda z_1 \cdot F_1(\varphi, x, z_1), \dots, \lambda z_n \cdot F_n(\varphi, x, z_n), x, y_i)) \\ &= F_i(\varphi, x, y_i), \end{split}$$

which proves (14) for arbitrary computation lengths, and hence (11).

**Sublemma A.3.1.** Let  $R_{F_i}$  and  $R_{G_i}$ ,  $1 \le i \le n$ , be the procedures defined in the proof of Lemma 6.1.5. Then for arbitrary  $\varphi$ , *x* and  $y_i$ ,

$$\llbracket R_{F_i} \rrbracket^A_{\varphi}(x, y_i) \simeq \llbracket R_{G_i} \rrbracket^A_{\varphi, \varrho}(x, y_i),$$

where

$$\varrho_i = \lambda z_i \cdot [\![\mathbf{R}_{F_i}]\!]^A_{\varphi}(x, z_i) \quad i = 1, \dots, n.$$

### **Proof:**

By definition of the semantics of procedures,

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

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

By the procedure assignment lemma (Lemma 4.7.3),

$$\begin{split} \llbracket \mathtt{b}_{F_i} &:= P_{G_i}(\mathtt{a}_{F_{i,1}}, \ \mathtt{a}_{F_{i,2}}) \rrbracket_{\varphi}^A \sigma \quad \simeq \quad \sigma\{ \mathtt{b}_{F_i} / \llbracket R_{G_i}^P \rrbracket_{\varphi}^A(\llbracket \mathtt{a}_{F_{i,1}} \rrbracket^A \sigma, \llbracket \mathtt{a}_{F_{i,2}} \rrbracket^A \sigma) \} \\ &\simeq \quad \sigma\{ \mathtt{b}_{F_i} / \llbracket R_{G_i}^P \rrbracket_{\varphi}^A(x, y_i) \} \end{split}$$

Therefore,

$$[\![R_{F_i}]\!]^A_{\varphi}(x, y_i) \simeq (\sigma\{\mathsf{b}_{F_i} / [\![R^P_{G_i}]\!]^A_{\varphi}(x, y_i)\})[\mathsf{b}_{F_i}] \simeq [\![R^P_{G_i}]\!]^A_{\varphi}(x, y_i).$$
(15)

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

$$\llbracket \mathbf{R}_{G_i}^P \rrbracket_{\varphi}^A(x, y_i) \simeq \llbracket \mathbf{R}_{G_i} \rrbracket_{\varphi, \varrho}^A(x, y_i).$$
(16)

and the result will follow.

By definition,

$$\llbracket R_{G_i}^P \rrbracket_{\varphi}^A(x, y_i) \simeq \begin{cases} \sigma_1' [\mathbf{b}_{G_i}] & \text{if } \llbracket S_{G_i}^P \rrbracket_{\varphi}^A \sigma_1 \downarrow \sigma_1' \\ \uparrow & \text{if } \llbracket S_{G_i}^P \rrbracket_{\varphi}^A \sigma_1 \uparrow \end{cases}$$
(17)

where  $\sigma_1[\mathbf{a}_{G_{i,1}}] = x$ ,  $\sigma_1[\mathbf{a}_{G_{i,2}}] = y_i$  and  $\phi$  are interpreted as  $\varphi$ . Also

$$\llbracket \mathbf{R}_{G_i} \rrbracket_{\varphi,\varrho}^A(x, y_i) \simeq \begin{cases} \sigma_2'[\mathbf{b}_{G_i}] & \text{if } \llbracket \mathbf{S}_{G_i} \rrbracket_{\varphi,\varrho}^A \sigma_2 \downarrow \sigma_2' \\ \uparrow & \text{if } \llbracket \mathbf{S}_{G_i} \rrbracket_{\varphi,\varrho}^A \sigma_2 \uparrow \end{cases}$$
(18)

where  $\sigma_2[\mathbf{a}_{G_{i,1}}] = x$ ,  $\sigma_2[\mathbf{a}_{G_{i,2}}] = y_i$  and  $\phi$ ,  $\rho$  are interpreted as  $\varphi$ ,  $\varrho$  respectively.

So to prove(16), we must just show that (17) and (18) define the same function. Now  $S_{G_i}$  and  $R_{G_{i,1}}, \ldots, R_{G_{i,m_i}}$  are the same as  $S_{G_i}^P$  and  $R_{G_{i,1}}^P, \ldots, R_{G_{i,m_i}}^P$ , except that all occurrences of oracle call statements  $c := \rho_j(t)$  in  $S_{G_i}$  are replaced by procedure calls  $c := P_{G_j}(a_{G_{i,1}}, t)$  in  $S_{G_i}^P$ . Thus, comparing (17) and (18), it is sufficient to prove

$$\llbracket \mathbf{c} := P_{G_j}(\mathbf{a}_{G_{i,1}}, t) \rrbracket_{\varphi}^A \simeq \llbracket \mathbf{c} := \rho_j(t) \rrbracket_{\varphi, \varrho}^A.$$
<sup>(19)</sup>

By the procedure assignment lemma, and since  $[\![\mathbf{a}_{G_i}]\!]^A \sigma = x$  and  $\phi$  are interpreted as  $\varphi$ ,

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

By the semantics of term and assignment statements, and since  $\rho_i$  is the oracle for  $\varrho_i = \lambda z$ .  $\llbracket R_{F_i} \rrbracket^A(\varphi, x, z),$ 

$$\llbracket \mathbf{c} := \rho_j(t) \rrbracket_{\varphi,\varrho}^A \sigma \simeq \sigma \{ \mathbf{c}/\varrho_j(\llbracket t \rrbracket^A \sigma) \} \simeq \sigma \{ \mathbf{c}/\llbracket R_{F_j} \rrbracket_{\varphi}^A(x, \llbracket t \rrbracket^A \sigma) \}.$$

By (15),  $[\![R^P_{G_i}]\!]^A_{\varphi} = [\![R_{F_i}]\!]^A_{\varphi}$ , from which (19) follows, ending the proof.

# References

- [1] de Bakker, J.: Mathematical Theory of Program Correctness, Prentice-Hall International, 1980.
- [2] Brattka, V.: Recursive characterisation of computable real-valued functions and relations, Theoretical Computer Science, 162, 1996, 45-77.
- [3] Feferman, S.: Inductive schemata and recursively continuous functionals, in: Logic Colloquium '76 (R. O. Gandy, M. Hyland, Eds.), North-Holland, Amsterdam, 1977, 373–392.
- [4] Feferman, S.: A new approach to abstract data types, I: Informal development, Mathematical Structures in Computer Science, 2, 1992, 193–229.
- [5] Feferman, S.: A new approach to abstract data types, II: Computability on ADTs as ordinary computation, in: Computer Science Logic, Springer-Verlag, 1992, 79-95.
- [6] Feferman, S.: Computation on abstract data types. The extensional approach, with an application to streams, Annals of Pure and Applied Logic, 81, 1996, 75–113.

- [7] Hinman, P.: Recursion on abstract structures, in: *Handbook of Computability Theory* (E. Griffor, Ed.), North-Holland, Amsterdam, 1999, 317–359.
- [8] Hobley, K., Thompson, B., Tucker, J.: Specification and verification of synchronous concurrent algorithms: a case study of a convoluted algorithm, in: *The Fusion of Hardware Design and Verification (Proceedings of IFIP Working Group 10.2 Working Conference)* (G. Milne, Ed.), North-Holland, Amsterdam, 1988, 347–374.
- [9] Kleene, S. C.: Introduction to Metamathematics, North-Holland, Amsterdam, 1952.
- [10] Meinke, K.: Universal algebra in higher types, *Theoretical Computer Science*, 100, 1992, 385–417.
- [11] Meinke, K., Tucker, J.: Universal algebra, in: *Handbook of Logic in Computer Science*, vol. 1, Oxford University Press, 1992.
- [12] Moldestad, J., Stoltenberg-Hansen, V., Tucker, J. V.: Finite algorithmic procedures and inductive definibility, *Mathematica Scandinavica*, 46, 1980, 62–76.
- [13] Moore, C.: Recursion theory on reals and continuous time computation, *Theoretical Computer Science*, **162**, 1996, 23–44.
- [14] Moschovakis, Y. N.: Abstract recursion as a foundation for the theory of recursive algorithms, in: *Computation and Proof Theory*, Springer-Verlag, 1984, 289–364.
- [15] Moschovakis, Y. N.: The formal language of recursion, *Journal of Symbolic Logic*, 54, 1989, 1216–1252.
- [16] Mycka, J., Costa, J.: Real recursive functions and their hierarchy, Journal of Complexity, 2005, To appear.
- [17] Stoy, J. E.: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, The MIT Press, 1977.
- [18] Tucker, J., Zucker, J.: Program Correctness over Abstract Data Types, with Error-State Semantics, North-Holland, 1988.
- [19] Tucker, J., Zucker, J.: Computable functions on stream algebras, in: NATO Advanced Study Institute, International Summer School at Marktoberdorf, 1993, on Proof and Computation (H. Schwichtenberg, Ed.), Springer-Verlag, 1994, 341–382.
- [20] Tucker, J., Zucker, J.: Computable Functions and Semicomputable Sets on Many-sorted Algebras, in: *Handbook of Logic in Computer Science* (S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, Eds.), vol. 5, Oxford University Press, 2000, 317–523.
- [21] Tucker, J., Zucker, J.: Abstract Computability and Algebraic Specification, ACM Transactions on Computational Logic, 3, 2002, 279–333.
- [22] Winskel, G.: The Formal Semantics of Programming Languages: An Introduction, The MIT Press, 1993.
- [23] Xu, J.: Models of computation on abstract data types based on recursive schemes, Master Thesis, Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada, August 2003.