

## EXPRESSIVE POWER OF TYPED AND TYPE-FREE PROGRAMMING LANGUAGES

Elfriede FEHR

*Lehrstuhl für Informatik II, RWTH Aachen, D-5100 Aachen, Fed. Rep. Germany*

Communicated by R. Milner

Received September 1982

Revised February 1984

**Abstract.** The classes of typed and type-free lambda-schemes are studied with respect to their use as control structures of programming languages. The semantics of both classes are analyzed in the same semantical domains. In particular, it is shown that typed lambda-schemes are translatable into equivalent type-free lambda-schemes but not vice versa. Furthermore, we prove that the class of type-free lambda-schemes is universal in the sense that in the initial models all recursively enumerable  $\Sigma$ -trees ( $\Sigma$  is the set of operation symbols) are definable.

### Introduction

High-level programming languages (e.g., ALGOL 68) usually require type specifications of *all* identifiers occurring in a program, i.e. to each identifier a type such as **int**, **real**, **bool**, **proc (char, string) bool**, etc. is restricted according to some type conventions.

In some other programming languages (e.g., LISP 1.5 or ALGOL 60) there are no or incomplete type specifications.

On the one hand it is clear that a high-level language should have some sort of typing in order to achieve a certain programming discipline and to ease compiler construction (see, e.g., [34, pp. 92-0]); on the other hand it is known that under the restriction to typed application and abstraction, only an uninteresting class of functions is programmable (see, e.g., [53]).

In most programming languages this problem is solved by admitting the use of the fixed-point operator (which cannot itself be defined by a typed expression) either explicitly as, e.g., in PCF [62] or implicitly by introducing recursively defined procedures with fixed-point semantics as in ALGOL 68 and many others.

This paper investigates the influence of type-free programming concepts on the definability of functions and objects in comparison with the exclusive use of typed concepts plus fixed-point operators.

To illustrate this by an example, consider the following ALGOL 60 procedure  $P$  (taken from Ledgard [42]):

**integer procedure**  $P(f, y)$ ; **integer procedure**  $f$ ; **integer**  $y$ ;

$P :=$  **if**  $y = 0$  **then** 1 **else**  $y * f(f, y - 1)$ .

Although each occurring identifier ( $P, f$  and  $y$ ) is declared, the type of the parameter  $f$  is not completely specified. By looking at the procedure-body one realizes that a full type-specification for  $f$  is not possible, due to an occurrence of self-application " $f(f, y - 1)$ ". Some experts in programming language design, such as Addyman et al. [2] and Strait et al. [70] who are concerned with the further development of PASCAL, exclude the use of incompletely defined procedure parameters in their languages, without commenting on the problem of whether this is a real restriction, i.e., whether or not there is an equivalent fully-typed procedure for each procedure with incomplete specifications.

In this paper we shall answer this question negatively.

In order to investigate the influence of type-free concepts in programming independently of present base operations (system-defined functions) and predicates, we consider the control structure of a programming language on the level of program schemes, i.e., we abstract in a given program from all given operations, predicates and branchings and replace  $n$ -ary application and parameterization using 'currying' by monadic application and parameterization. This process yields typed lambda-schemes in the case of typed programs and type-free lambda-schemes otherwise.

In the above example, the lambda-scheme  $P_\lambda$  which is the underlying control structure of the procedure  $P$  reads as follows:

$$P_\lambda = (Y_\lambda \lambda p. \lambda f. \lambda y. (((\text{cond } y) 1) ((\text{mult } y) ((f f) (\text{sub } y))))),$$

where  $Y_\lambda$  is the lambda-expression corresponding to the fixed-point operator.

Now the semantics of a program can be defined via the semantics of the underlying program-scheme with respect to an appropriate interpretation of the occurring function symbols (see, e.g., [23] or [31]).

An operational semantics of lambda-schemes can be derived from the well-known reduction rules of the lambda-calculus and corresponds to an extension of the copy rule in ALGOL 60 [28].

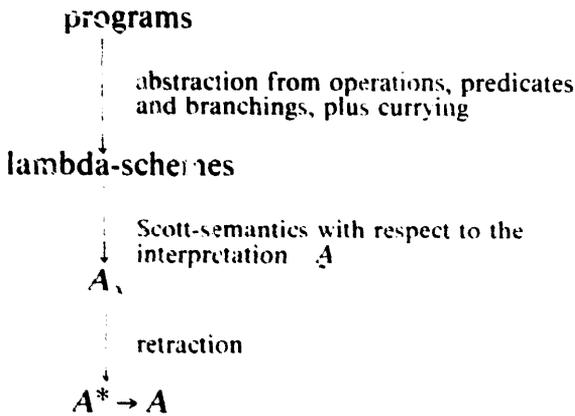
However, a denotational semantics in the sense of Scott and Strachey (e.g. [69]), which is desirable for the mathematical treatment of programs, is in the type-free case not at all straightforward.

Since the discovery of lambda-models by Scott [67] there have been of course a number of approaches to the mathematical semantics of  $\lambda$ -terms (e.g., [68, 73]); however, for the definition of the denotational semantics of lambda-schemes one would like to use 'finitely-typed' domains rather than some limit domains, i.e., even if the semantics of some procedures can be understood only in a limit domain, we would still like the meaning of the main program to be an object of finite type, such as an integer, a function from reals to integers, etc.

In the above example, this means that the scheme  $P_\lambda$  must be interpreted in a limit domain (due to self-application), whereas the semantics of the scheme  $((P_\lambda P_\lambda)x)$ , which corresponds to a procedure call  $P(P, x)$  to compute  $x$  factorial, should be an object of finite type, namely an integer.

These considerations are the motivation for defining typed denotational semantics of lambda-schemes using the type-free semantics of the syntactical parts.

The following diagram illustrates this idea:



This approach has some advantages:

- Knowledge of equivalence, undefinedness, etc. in the lambda-models is also applicable to the typed semantics.
- Semantical objects are the same as in the case of typed program-schemes, thus comparisons and meaning-preserving translations are possible.
- An operational semantics can be given which is correct with respect to the denotational one.

**Remark.** In this paper we develop the theory for a particular limit domain, namely  $A_\lambda$ . Of course there are a lot of different models for the lambda-calculus besides the original limit construction by Scott, which we refer to. However, one realizes that the results to be developed in the following chapters also hold in other models, such as solutions of domain equations

$$D = A + [D \rightarrow D] \quad \text{or} \quad D = A + [D \times D] + [D \rightarrow D].$$

The reason not to formulate results for a general class of lambda-models lies in the fact that general retractions and injections as well as the denotational semantics of lambda-schemes become technically more complicated, testing for example in which part of a domain an element is found. However, using  $A_\lambda$  is general enough as formalized in Theorem 7.14 which can be read as follows:

Given a lambda-scheme  $t$  which can be typed by  $\tau$ , then the semantics of  $t$  in  $A_\infty$  is equivalent to the typed semantics of  $t$  in  $A^\tau$ .

This paper is composed of eight sections:

In Section 1 we introduce the mathematical background and notations.

In Section 2 we develop finitely typed cpo's and limit domains (Scott-cpo's) and the relations between them.

In Section 3 the formal definitions of the syntax and semantics of lambda-schemes are given.

In Section 4 we mainly discuss semantical properties in Scott-models.

In Section 5 we prove a Mezei–Wright-like result for lambda-schemes, which enables us to apply some results which hold in the initial algebra in all other interpretations.

In Section 6 we show that the class of lambda-schemes is universal.

In Section 7 we prove that typed lambda-schemes are translatable into equivalent type-free lambda-schemes.

Finally, in Section 8 we consider the lambda-definability of formal languages, i.e., the semantics of typed and type-free lambda-schemes with respect to the special interpretation by formal languages. This investigation yields the result that type-free lambda-schemes are on the level of control structures strictly more powerful than typed lambda-schemes, whereas the classes of definable objects in arbitrary lambda-models by typed and type-free lambda-schemes are not comparable.

## 1. Mathematical background

**Definition 1.1.** A structure  $A = (A, \leq)$  is a cpo (*complete partial order*) if

- (i)  $\leq$  is a partial order on  $A$ ,
- (ii) there exists a *minimal element*  $\perp_A$  in  $A$ ,
- (iii) for each directed subset  $T$  of  $A$  there exists a *least upper bound*  $\bigsqcup T$  in  $A$ , where  $T$  is *directed*, if for each  $t_1, t_2 \in T$  there exists an upper bound of  $\{t_1, t_2\}$  in  $T$ .

**Convention.** We abbreviate  $\perp_A$  to  $\perp$  if the corresponding domain  $A$  is clear from the context. For an index set  $K$  and a directed subset  $\{a_\nu \mid \nu \in K\}$  we write  $\bigsqcup_{\nu \in K} a_\nu$  for  $\bigsqcup \{a_\nu \mid \nu \in K\}$ .

**Definition 1.2.** We associate with an arbitrary set  $A$  the *flat cpo*  $A_\perp = (A \cup \{\perp\}, \leq)$  where, for  $a_1, a_2 \in A_\perp$ ,  $a_1 \leq a_2$  iff  $a_1 = \perp$  or  $a_1 = a_2$ .

**Definition 1.3.** For cpo's  $A$  and  $B$  a function  $f: A \rightarrow B$  is called

- (i) *continuous* iff for each nonempty, directed  $T$  in  $A$ ,  $f(T)$  is directed in  $B$  and  $f(\bigsqcup T) = \bigsqcup f(T)$ .
- (ii) *strict* iff  $f(\perp_A) = \perp_B$ .
- (iii) *monotonic* iff for each  $a_1 \leq a_2$  in  $A$ ,  $f(a_1) \leq f(a_2)$  in  $B$ .
- (iv) *homeomorphism* if  $f$  is bijective and continuous in both directions.
- (v) The *set of all continuous functions from  $A$  to  $B$*  will be denoted by  $[A \rightarrow B]$ .

**Lemma 1.4.** *If  $f: A \rightarrow B$  is continuous, then  $f$  is also monotonic.*

**Proof.** The proof is immediate.  $\square$

**Theorem 1.5 (Tarski).** For a cpo  $A$  and a function  $f \in [A \rightarrow A]$  there exists the least fixed-point  $\mu x.f(x)$  in  $A$  and  $\mu x.f(x) = \bigsqcup_{v \in \mathbb{N}} f^v(\perp)$ .

**Proof.** The proof follows by standard procedures.  $\square$

**Definition 1.6.** Let  $I$  be a finite set.

The set  $\mathcal{F}(I)$  of *finite types (over  $I$ )* is defined inductively by

- (i)  $I \cup \{\mathbf{0}\} \subseteq \mathcal{F}(I)$  (*base types*),
- (ii)  $(\tau_1, \tau_2) \in \mathcal{F}(I)$  for  $\tau_1, \tau_2 \in \mathcal{F}(I)$  (*functional types*),
- (iii)  $\tau_1 \dots \tau_r \in \mathcal{F}(I)$  for  $\tau_1, \dots, \tau_r \in \mathcal{F}(I)$  (*cartesian types*).

**Remark.** The empty word, denoted by  $e$ , is among the cartesian types (set  $r = 0$  in Definition 1.6(iii)).

**Definition 1.7.** The function **level**:  $\mathcal{F}(I) \rightarrow \mathbb{N}$  determines the functional depth of a type:

$$\text{level}(\tau) := \begin{cases} 0 & \text{if } \tau \in I \cup \{\mathbf{0}\}, \\ \max\{\text{level}(\tau_1), \text{level}(\tau_2)\} + 1 & \text{for } \tau = (\tau_1, \tau_2), \\ \max\{\text{level}(\tau_\nu) \mid 1 \leq \nu \leq r\} & \text{for } \tau = \tau_1 \dots \tau_r \end{cases}$$

**Definition 1.8.** Let  $I$  be a finite set.

- (i) The set  $\mathcal{P}(I)$  of *procedural types (over  $I$ )* is inductively defined by
  - (a)  $I \subseteq \mathcal{P}(I)$ , and
  - (b)  $(\tau_1 \dots \tau_n \tau_0) \in \mathcal{P}(I)$  for  $\tau_0, \dots, \tau_r \in \mathcal{P}(I)$ .
- (ii) The set of *integer types*  $\mathcal{N} := \{n \mid n \in \mathbb{N}\}$  consists of all symmetric functional types over  $\mathbf{0}$ , i.e.,  $n + 1 := (n, n)$ .
- (iii) The set of *derived types*  $D^*(I) := \bigcup_{n \in \mathbb{N}} D^n(I)$  consists of all procedural, homogeneous types over  $I$ , i.e.,  $D^0(I) := I$  and inductively

$$D^{n+1}(I) := \{(\tau_1 \dots \tau_n \tau_0) \mid \tau_\nu \in D^n(I), 0 \leq \nu \leq n\}.$$

**Remark.** Observe that **level**( $n$ ) =  $n$  and

$$\text{level}(\tau) = n \quad \text{for all } n \in \mathbb{N}, \tau \in D^n(I).$$

**Definition 1.9.** Let  $(A^i \mid i \in I)$  be a family of cpo's. For each type  $\tau \in \mathcal{F}(I)$  the cpo  $A^\tau$  is given canonically:

- (i)  $A^{\mathbf{0}} := \bigcup \{A^i \mid i \in I\}$ ,<sup>1</sup> where  $\perp_{A^i}$  are identified,

<sup>1</sup>  $\bigcup$  stands for disjoint union.

- (ii)  $A^{(\tau_1, \tau_2)} := [A^{\tau_1} \rightarrow A^{\tau_2}]$  for  $\tau_1, \tau_2 \in \tau(I)$ , with the pointwise defined partial order relation,
- (iii)  $A^{\tau_1 \dots \tau_r} := A^{\tau_1} \times \dots \times A^{\tau_r}$  for  $\tau_1 \dots \tau_r \in \mathcal{T}(I)$ , with the componentwise defined partial order relation.

**Lemma 1.10.** *The class of continuous functions contains the identities, constant functions, and is closed under composition, substitution and least upper bounds.*

**Proof.** The proof follows by standard procedures.  $\square$

**Convention.** For a family  $(A^i | i \in I)$  of sets without cpo-structure.  $A^\tau$  denotes the sets defined analogously to Definition 1.9.

**Definition 1.11.** Given a family  $(A^i | i \in I)$  (of cpo's) and a subset  $T$  of  $\mathcal{T}(I)$ , then  $A^T := \{A^\tau | \tau \in T\}$  is called a *T-set* (*T-cpo*) and for  $T$ -sets (*T-cpo*'s)  $A^T$  and  $B^T$  a *T-mapping* is a family of (continuous functions)  $(f_\tau : A^\tau \rightarrow B^\tau | \tau \in T)$ .

**Definition 1.12.** Let  $\Sigma$  be a  $D(I)$ -set of operation symbols.

A *continuous  $\Sigma$ -algebra*  $\underline{A} = (A, \alpha)$  consists of an  $I$ -cpo  $A$  as carrier and a  $D(I)$ -mapping  $\alpha : \Sigma \rightarrow A^{D(I)}$  assigning to each operation symbol  $f \in \Sigma^{(w, i)}$  a continuous base operation  $\alpha(f) : A^w \rightarrow A^i$ .

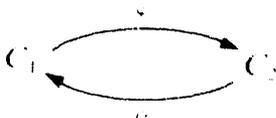
**Definition 1.13.** Let  $\Sigma$  be a  $D(I)$ -set and let  $\underline{A} = (A, \alpha)$  and  $\underline{B} = (B, \beta)$  be continuous  $\Sigma$ -algebras. An  $I$ -mapping  $h : A \rightarrow B$  is a  $\Sigma$ -homomorphism iff  $h(\alpha(f)(a_1, \dots, a_r)) = \beta(f)(h(a_1), \dots, h(a_r))$  for each  $f \in \Sigma^{(w, i)}$  and  $(a_1, \dots, a_r) \in \Sigma^w$ .

## 2. Finitely typed domains and models for the lambda-calculus

Scott presented in [67] the construction of standard-models for the lambda-calculus based on continuous lattices. We shall show that starting from an  $I$ -cpo  $A$  the Scott model  $\mathcal{A}$ , contains all procedurally typed domains  $A^{T(I)}$  as retracts. This section provides some insight into the relations between these domains. In particular we shall introduce the notion of 'idealness' which is quite essential for the translatability results in Section 7. For 'ideal' objects in a limit-domain it is possible to obtain the retract of an application by performing a typed application after retracting the function and arguments on suitable domains. This is a property which holds for integer-types but not in general for procedural types.

In this section let  $I$  be a finite set of base types and let  $A$  be an  $I$ -cpo.

**Definition 2.1.** A pair



of functions between cpo's  $C_1$  and  $C_2$  is called a *retraction pair* (from  $C_1$  to  $C_2$ ) iff  $\varphi$  and  $\psi$  are continuous and have the *retraction properties*

- (a)  $\psi \circ \varphi = \text{id}_{C_1}$ , and
- (b)  $\varphi \circ \psi \leq \text{id}_{C_2}$ .

In this case we call  $\varphi$  an *injection* (from  $C_1$  into  $C_2$ ), and the cpo  $C_1$  is a *retract* of  $C_2$  (with respect to  $(\varphi, \psi)$ ).

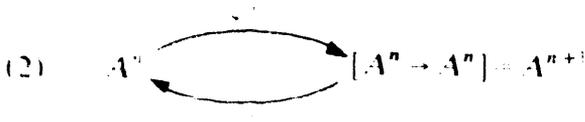
We shall now develop a number of specific retraction pairs, which can also be looked up in the table, given in Fig. 1 (see end of this section).

For the  $I$ -cpo  $A$  and corresponding integer-domains  $A^n, n \in \mathbb{N}$ , we give retraction-pairs between each two 'consecutive' cpo's.

**Definition 2.2**



where  $\iota^i(a) := a$  and  $\pi^i(a) := \begin{cases} a & \text{if } a \in A^i, \\ \perp & \text{otherwise,} \end{cases}$  for all  $i \in I$ .



where

$\varphi^{(0)}(a) := x \mapsto a, \quad \psi^{(0)}(f) := f(\perp)$

and inductively

$\varphi^{(n+1)}(a) := \varphi^{(n)} \circ a \circ \psi^{(n)}, \quad \psi^{(n+1)}(f) := \psi^{(n)} \circ f \circ \varphi^{(n)}$ .

**Lemma 2.3.**  $(\iota^i, \pi^i)$  and  $(\varphi^{(n)}, \psi^{(n)})$  are retraction-pairs for all  $i \in I, n \in \mathbb{N}$ .

**Proof.** The proof immediately follows for  $(\iota^i, \pi^i)$  and for  $(\varphi^{(n)}, \psi^{(n)})$ . Continuity is guaranteed by Lemma 1.10 and retraction properties are shown by straightforward induction on  $n$ .  $\square$

By composition of the above defined functions we obtain retraction-pairs between arbitrary integer-domains.

**Definition 2.4.** Let  $r, s \in \mathbb{N}$  and let  $s \leq r$ . The pair



is defined inductively by

$$\varphi^{(s,s)} = \text{id}_{A^s} = \psi^{(s,s)}$$

and

$$\varphi^{(s,r+1)} = \varphi^{(r)} \circ \varphi^{(s,r)}, \quad \psi^{(r+1,s)} = \psi^{(r,s)} \circ \psi^{(r)}$$

**Lemma 2.5.** For all  $r, s \in \mathbb{N}$ ,  $r \leq s$ ,  $(\varphi^{(s,r)}, \psi^{(r,s)})$  is a retraction pair.

**Proof.** The continuity of  $\varphi^{(s,r)}$  and  $\psi^{(r,s)}$  follows from Lemma 1.10 and the retraction properties can be derived from the retraction properties of the identities and the above retraction pairs.  $\square$

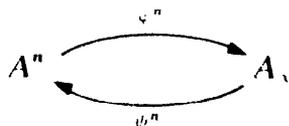
The direct and inverse limit of the integer-domains  $A^n$  is the Scott-domain  $A_\omega$ .

**Definition 2.6.** The Scott-domain  $A_\omega$  is defined by

$$A_\omega = \{ \langle a_i \rangle_{i \in \mathbb{N}} \mid a_i \in A^i \text{ and } \psi^{(i)}(a_{i+1}) = a_i \},$$

with componentwise defined cpo-structure.

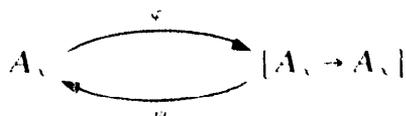
For  $n \in \mathbb{N}$ , the pair



is defined by

$$\varphi^n(a)_i = \begin{cases} \psi^{(n,i)}(a) & \text{for } n \geq i, \\ \varphi^{(n,i)}(a) & \text{for } n < i, \end{cases} \quad \text{and} \quad \psi^n(\langle a_i \rangle_{i \in \mathbb{N}}) = a_n$$

The pair



is defined by

$$\varphi(\langle a_i \rangle_{i \in \mathbb{N}}) = \bigsqcup_{n \in \mathbb{N}} \varphi^n \circ a_{n+1} \circ \psi^n \quad \text{and} \quad \psi(f) = \bigsqcup_{n \in \mathbb{N}} \varphi^{n+1}(\psi^n \circ f \circ \varphi^n).$$

**Lemma 2.7.** For all  $n \in \mathbb{N}$ ,  $(\varphi^n, \psi^n)$  is a retraction-pair.

**Proof.** The continuity of  $\varphi^n$  and  $\psi^n$  is immediate, and the retraction properties follow from those of  $\varphi^{(\nu, n)}$  and  $\psi^{(n, \nu)}$  for all  $\nu \in \mathbb{N}$ .  $\square$

The next lemma gives insight into some connections between the integer-domains and the Scott-domain and applications in them.

**Lemma 2.8.** *Let  $a, f \in A_x$ ,  $a_\nu \in A^\nu$  and  $b \in A^0$ . Then*

- (i)  $a = \bigsqcup_{\nu \in \mathbb{N}} \varphi^\nu(\psi^\nu(a)),$
- (ii)  $\psi^n\left(\bigsqcup_{\nu \in \mathbb{N}} \varphi^\nu(a_\nu)\right) = a_n,$
- (iii)  $\varphi(f)(a) = \bigsqcup_{\nu \in \mathbb{N}} \varphi^\nu(\psi^{\nu+1}(f)(\psi^\nu(a))),$
- (iv)  $\psi^0(\varphi(f)(\perp)) = \psi^0(f),$
- (v)  $\varphi(\varphi^0(b))(a) = \varphi^0(b).$

**Proof.** (i) For all  $n \in \mathbb{N}$  the following holds:

$$\begin{aligned} \psi^n\left(\bigsqcup_{\nu \in \mathbb{N}} \varphi^\nu(\psi^\nu(a))\right) &= \bigsqcup_{\nu \in \mathbb{N}} \psi^n \circ \varphi^\nu \circ \psi^\nu(a) \quad \text{by continuity of } \psi^n \\ &= \psi^n(a). \end{aligned}$$

Hence,  $a$  equals  $\bigsqcup_{\nu \in \mathbb{N}} \varphi^\nu(\psi^\nu(a))$  in all components,

(ii) immediately follows by continuity of  $\psi^n$ , Lemmas 2.5 and 2.7,

(iii) immediately follows by definition of  $\varphi$ , (i) and continuity,

$$\begin{aligned} \text{(iv)} \quad \psi^0(\varphi(f)(\perp)) &= \psi^0\left(\bigsqcup_{\nu \in \mathbb{N}} \varphi^\nu(\psi^{\nu+1}(f)(\psi^\nu(\perp)))\right) \quad \text{by (iii)} \\ &= \psi^1(f)(\psi^0(\perp)) \quad \text{by (ii) and Lemma 2.7} \\ &= \psi^1(f)(\perp) \\ &= \psi^{(0)}(\psi^1(f)) \quad \text{by Definition 2.2(2)} \\ &= \psi^0(f) \quad \text{by Definition 2.6,} \end{aligned}$$

$$\begin{aligned} \text{(v)} \quad \varphi(\varphi^0(b))(a) &= \\ &= \bigsqcup_{\nu \in \mathbb{N}} \varphi^\nu(\psi^{\nu+1}(\varphi^0(b))(\psi^\nu(a))) \quad \text{by (iii)} \\ &= \bigsqcup_{\nu \in \mathbb{N}} \varphi^\nu(\varphi^{(\nu-1)} \circ \dots \circ \varphi^{(0)} \circ \varphi^{(0)}(b) \circ \psi^{(0)} \circ \dots \circ \psi^{(\nu-1)}(\psi^\nu(a))) \end{aligned}$$

by Definition 2.4

$$\begin{aligned}
 &= \bigsqcup_{\nu \in \mathbb{N}} \varphi^\nu(\varphi^{(0, \nu)}(b)) \quad \text{by Definition 2.2} \\
 &= \bigsqcup_{\nu \in \mathbb{N}} \varphi^\nu(\psi^\nu(\varphi^0(b))) \quad \text{by Definition 2.6} \\
 &= \varphi^0(b) \quad \text{by (i).} \quad \square
 \end{aligned}$$

**Theorem 2.9 (Scott).**  $A_\infty$  is homeomorphic to  $[A_\infty \rightarrow A_\infty]$  via  $\varphi$ .

**Proof.** The continuity of  $\varphi$  and  $\psi$  follows from Lemma 1.10. We show

(a)  $\psi \circ \varphi = \text{id}_{A_\infty}$ , and

(b)  $\varphi \circ \psi = \text{id}_{[A_\infty \rightarrow A_\infty]}$ .

(a) 
$$\begin{aligned}
 \psi(\varphi(\langle a_\nu \rangle_{\nu \in \mathbb{N}})) &= \\
 &= \bigsqcup_{n \in \mathbb{N}} \varphi^{n+1} \left( \psi^n \circ \left( \bigsqcup_{\nu \in \mathbb{N}} \varphi^\nu \circ a_{\nu+1} \circ \psi^\nu \right) \circ \varphi^n \right) \\
 &= \bigsqcup_{n \in \mathbb{N}} \varphi^{n+1} (\psi^n \circ (\varphi^n \circ a_{n+1} \circ \psi^n) \circ \varphi^n)
 \end{aligned}$$

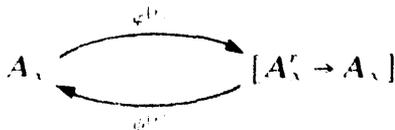
by the continuity of all functions and the retraction properties of  $\psi^n, \varphi^n$

$$\begin{aligned}
 &= \bigsqcup_{n \in \mathbb{N}} \varphi^{n+1}(a_{n+1}) \\
 &= \langle a_n \rangle \quad \text{by Lemma 2.8(i).}
 \end{aligned}$$

(b) 
$$\begin{aligned}
 \varphi(\psi(f)) &= \bigsqcup_{n \in \mathbb{N}} \varphi^n \circ \psi^n \circ f \circ \varphi^n \circ \psi^n \\
 &= \left( \bigsqcup_{n \in \mathbb{N}} \varphi^n \circ \psi^n \right) \circ f \circ \left( \bigsqcup_{n \in \mathbb{N}} \varphi^n \circ \psi^n \right) \\
 &= \text{id}_{A_\infty} \circ f \circ \text{id}_{A_\infty} = f. \quad \square
 \end{aligned}$$

The next definition serves for the treatment of Cartesian types.

**Definition 2.10.** Let  $r \in \mathbb{N}$ . The retraction pair



is defined inductively by  $\varphi^{[0]}(a) = ( ) \mapsto a$ ,  $\psi^{[0]}(f) = f( )$  and

$$\begin{aligned}
 \varphi^{[r+1]}(a)(a_1, \dots, a_{r+1}) &= \varphi(\varphi^{[r]}(a)(a_1, \dots, a_r))(a_{r+1}), \\
 \psi^{[r+1]}(f) &= \psi^{[r]}((a_1, \dots, a_r) \mapsto \psi(a_{r+1} \mapsto f(a_1, \dots, a_{r+1})))
 \end{aligned}$$

for all  $a, a_\nu \in A_\infty$ ,  $1 \leq \nu \leq r+1$  and  $f \in [A_\infty^{r+1} \rightarrow A_\infty]$ .

**Lemma 2.11.**  $A_\infty$  is homeomorphic to  $[A'_\infty \rightarrow A_\infty]$  via  $\varphi^{[r]}$ , and  $\psi^{[r]} = \varphi^{[r]}^{-1}$  for all  $r \in \mathbb{N}$ .

**Proof.** The continuity of  $\varphi^{[r]}$  and  $\psi^{[r]}$  follows again from Lemma 1.10, and for  $a \in A_\infty, f \in [A'_\infty \rightarrow A_\infty]$  one can prove

(i)  $\psi^{[r]}(\varphi^{[r]}(a)) = a$ , and

(ii)  $\varphi^{[r]}(\psi^{[r]}(f)) = f$

by induction on  $r$ .  $\square$

The next lemma describes the behaviour of elements in  $A_\infty$  under conversion between different cartesian types.

**Lemma 2.12.** For  $r, n \in \mathbb{N}, f \in [A'_x \rightarrow A_\infty], g \in [A'_x{}^{r+n} \rightarrow A_\infty]$  and  $a_\nu \in A_\infty, 1 \leq \nu \leq r+n$ , the following hold:

(i)  $\varphi^{[r+n]}(\psi^{[r]}(f))(a_1, \dots, a_{r+n}) = \varphi^{[n]}(f(a_1, \dots, a_r))(a_{r+1}, \dots, a_{r+n})$ ,

(ii)  $\varphi^{[r]}(\psi^{[r+n]}(g))(a_1, \dots, a_r) = \psi^{[n]}((a_{r+1}, \dots, a_{r+n}) \mapsto g(a_1, \dots, a_{r+n}))$ .

**Proof.** The proof follows by straightforward induction on  $n$ .  $\square$

For objects of cartesian type, properties of Lemma 2.8(iv), (v) hold appropriately.

**Lemma 2.13.** Let  $r \in \mathbb{N}, a, a_\nu \in A_\infty, 1 \leq \nu \leq r$  and  $b \in A^0$ .

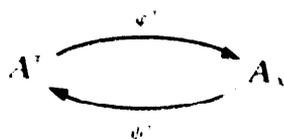
(i)  $\psi^0(\varphi^{[r]}(a))(\perp, \dots, \perp) = \psi^0(a)$ ,

(ii)  $\varphi^{[r]}(\varphi^0(b))(a_1, \dots, a_r) = \varphi^0(b)$ .

**Proof.** The proof follows by straightforward induction on  $r$ .  $\square$

Every cpo  $A^\tau$  of procedural type  $\tau$  can now be imbedded into  $A_\infty$  using the above defined functions.

**Definition 2.14.** For  $\tau \in \mathcal{F}(I)$ , the retraction pair



is defined inductively by

$$\varphi^i := \varphi^0 \circ \iota^i, \psi^i := \pi^i \circ \psi^0 \quad \text{for } i \in I.$$

$$\varphi^\tau(a) := \psi^{[r]}(\varphi^{\tau_0} \circ a \circ (\psi^{\tau_1} \times \dots \times \psi^{\tau_r})),$$

$$\psi^\tau(f) := \psi^{\tau_0} \circ \varphi^{[r]}(f) \circ (\varphi^{\tau_1} \times \dots \times \varphi^{\tau_r})$$

for  $\tau = (\tau_1 \dots \tau_n \tau_0)$  and all  $a \in A^\tau, f \in A_\infty$ ,

where the product

$$(f_1 \times \cdots \times f_r)(a_1, \dots, a_r) := (f_1(a_1), \dots, f_r(a_r))$$

degenerates for  $r=0$  to  $\mathbf{id}_{\{\}}.$

**Lemma 2.15.**  $(\varphi^\tau, \psi^\tau)$  is a retraction pair for all  $\tau \in \mathcal{F}(I).$

**Proof.** The continuity immediately follows from the definition and Lemma 1.10.

The retraction properties can be shown by induction on the structure of  $\tau.$   $\square$

In the specification of typed semantics for untyped Lambda-schemes, we perform application in  $A_x$  before projecting into the typed domain. One could expect that in correspondence with Lemma 2.8(iii) this would be equal to an application in typed cpo's after projecting function and arguments into the appropriate typed cpo's. We show that in general this is not the case.

**Conjecture 2.16.** Let  $A$  be an  $I$ -cpo,  $\tau = (\tau_1 \dots \tau_n, \tau_0) \in \mathcal{F}(I), f \in A_\tau, a_r \in A_{\tau_r}, 1 \leq r \leq n.$  Then

$$\psi^{\tau_0}(\varphi^{[r]}(f)(a_1, \dots, a_r)) = \psi^\tau(f)(\psi^{\tau_1}(a_1), \dots, \psi^{\tau_n}(a_r)).$$

**Counterexample.** Let  $i, j \in I, i \neq j, \perp \neq a^i \in A^i, f \in A_\tau$  such that  $\varphi(f) = \mathbf{id}_{A_\tau}$  and  $a = \varphi^i(a^i) \in A_\tau.$  The left-hand side of Conjecture 2.16 then reads as follows:

$$\psi^i(\varphi(f)(a)) = \psi^i(\varphi^i(a^i)) = a^i$$

and the right-hand side as

$$\psi^{(\tau_0)}(f)(\psi^i(a)) = \psi^i(\psi(f)(\varphi^i(\psi^i(a)))) = \psi^i(\perp) = \perp.$$

This is a contradiction of the assumption  $\perp \neq a^i$  and thus Conjecture 2.16 is false.

The following definition characterizes a class of elements in  $A_\tau$  having the desired property.

**Definition 2.17.** Let  $A$  be an  $I$ -cpo,  $\tau \in \mathcal{F}(I)$  and  $f \in A_\tau.$   $f$  is called  $\tau$ -ideal iff inductively:

- (i)  $f = \varphi^i(\psi^i(f))$  for  $\tau = i \in I$
- (ii) (a)  $\psi^{\tau_0}(\varphi^{[r]}(f)(a_1, \dots, a_r)) = \psi^\tau(f)(\psi^{\tau_1}(a_1), \dots, \psi^{\tau_n}(a_r)),$   
 (b)  $\varphi^{[r]}(f)(a_1, \dots, a_r)$  is  $\tau_0$ -ideal for  $\tau = (\tau_1 \dots \tau_n, \tau_0)$   
 and all  $\tau_r$ -ideal  $a_r \in A_{\tau_r}, 1 \leq r \leq n.$

First, we can prove that all embedded elements share this property.

**Lemma 2.18.** For  $\tau \in \mathcal{F}(I)$  and  $a \in A^\tau, \varphi^\tau(a)$  is  $\tau$ -ideal.

**Proof.** The proof follows by straightforward induction on the structure of  $\tau$ .  $\square$

In general, the projection  $\psi^\tau(\mu a.\varphi(f)(a))$  of the minimal fixed-point of  $f$  in  $A_\infty$  does not coincide with the minimal fixed-point of the function  $\psi^{(\tau, \tau)}(f)$  in  $A^\tau$ . But we can show this relation for  $(\tau, \tau)$ -ideal elements in  $A_\infty$ .

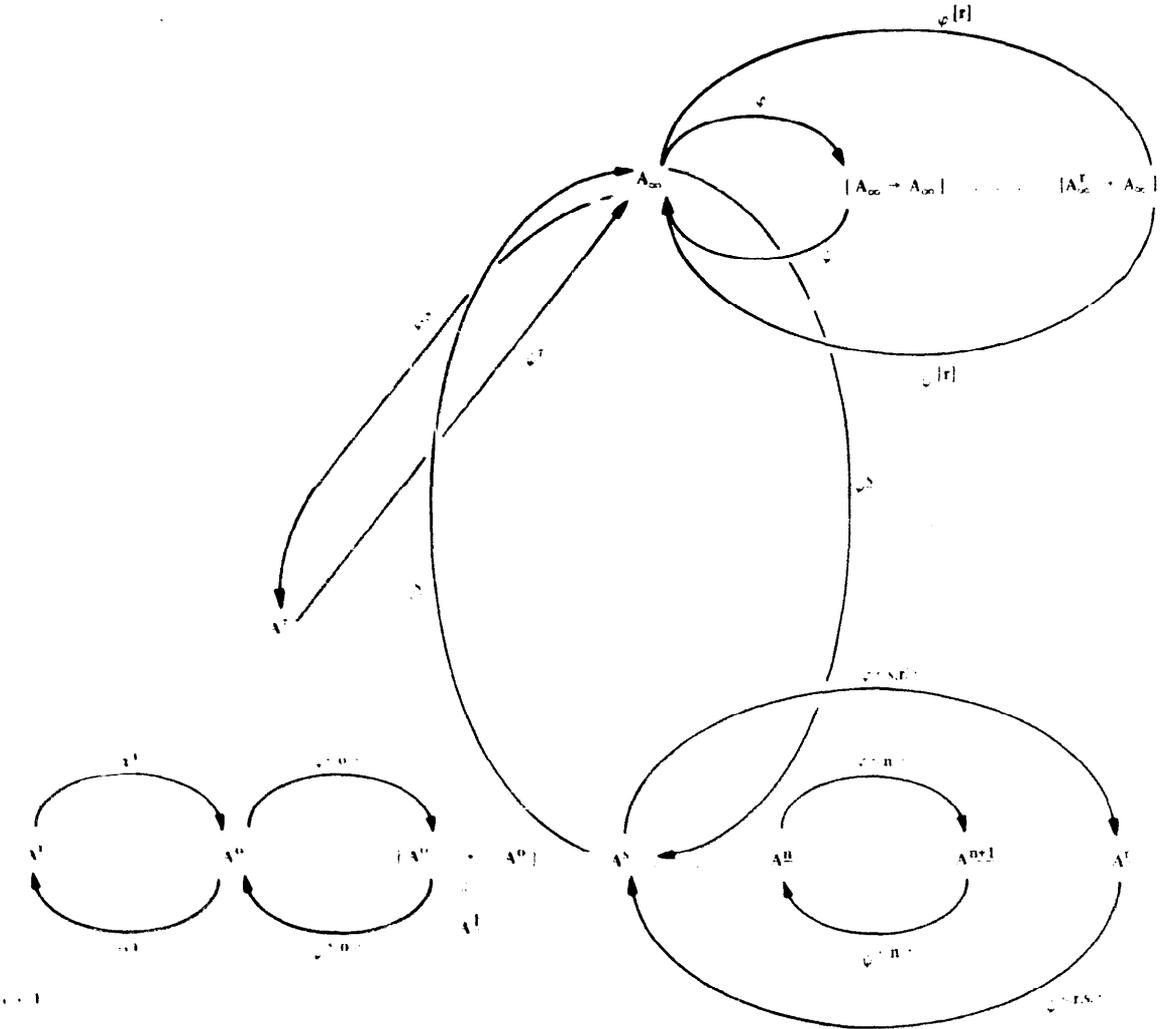


Fig. 1. Table of retraction pairs.<sup>2</sup>

**Lemma 2.19.** Let  $A$  be an  $I$ -cpo,  $\sigma = (\tau, \tau) \in \widehat{\mathcal{F}}(I)$  and  $f \in A_\infty$ . If  $f$  is  $\sigma$ -ideal, then

$$\psi^\tau(\mu a.\varphi(f)(a)) = \mu a.\psi^\tau(f)(a).$$

**Proof.** First we realize that  $\varphi(f)^n(\perp)$  is  $\sigma$ -ideal for all  $n \in \mathbb{N}$ , by  $\sigma$ -idealness of  $f$  and Lemma 2.18. Then one can show by an easy induction on  $n$  that  $\psi^\tau(\varphi(f)^n(\perp)) = (\psi^\tau(f))^n(\perp)$ .

Now the assertion can be derived using the limit representation of fixed-points and the continuity of  $\psi^\tau$ .  $\square$

<sup>2</sup> Underlined characters in Fig. 1 appear as boldface characters on text.

Another important property is the fact that idealness is preserved under least upper bounds.

**Lemma 2.20.** *For an I-cpo  $A$  and a directed set  $\{a_\nu \mid \nu \in \mathbb{N}\}$  of  $\tau$ -ideal elements in  $A_\infty$ ,  $\bigsqcup_{\nu \in \mathbb{N}} a_\nu$  is  $\tau$ -ideal.*

**Proof.** The proof follows by induction on the structure of  $\tau$ .  $\square$

### 3. Lambda-schemes

In this section we introduce the class of  $\lambda_\Sigma$ -schemes as the set of closed  $\lambda_\Sigma$ -terms over the set  $X$  of variables and the set  $\Sigma$  of operation symbols. Syntactically, this is exactly the pure  $\lambda$ - $k$ -calculus developed by Church in [15] with atoms taken from  $X$  and  $\Sigma$ .

An interpretation of the operation symbols in a Scott-domain  $A_\infty$  can be extended to an interpretation of lambda-schemes in  $A_\infty$ , the standard semantics. This semantic specification is treated in detail by Milner [49], Wadsworth [73] and others. Here we shall repeat it in our notations.

However, as the semantics of programs and procedures is in general an object of procedural type, we wish to define for each  $\lambda_\Sigma$ -scheme a family of typed semantics, namely for each procedural type  $\tau$  a  $\tau$ -semantics in  $A^\tau$  as retract of the standard semantics. Thus we can define typed objects using untyped schemes and define the notion of equivalence between  $\lambda$ -schemes and other known classes of program-schemes.

For an introduction to the theory of program-schemes we would like to refer to Engelfriet [23] and Greibach [31].

In this section we always assume  $X$  to be an infinite, denumerable set of variables,  $\Sigma$  a set of operation symbols and  $A = (A, \alpha)$  a continuous  $\Sigma$ -algebra.

**Definition 3.1.** The set  $\lambda(\Sigma)$  of  $\lambda_\Sigma$ -terms (over  $X$ ) is the least set of words over  $X \cup \Sigma \cup \{\lambda, \cdot, \cdot, (\cdot)\}$  such that

- (i)  $X \cup \Sigma \subseteq \lambda(\Sigma)$  (atoms),
- (ii)  $(t_1 t_2) \in \lambda(\Sigma)$  for  $t_1, t_2 \in \lambda(\Sigma)$  (application), and
- (iii)  $\lambda x.t \in \lambda(\Sigma)$  for  $x \in X$  and  $t \in \lambda(\Sigma)$  (abstraction).

**Lemma 3.2.** *The partition of a  $\lambda_\Sigma$ -term  $t$  into subterms is unique.*

**Proof.** The proof follows by standard procedures.  $\square$

**Convention 3.3.** We abbreviate  $\lambda x_1 \dots \lambda x_r.t$  to  $\lambda x_1 \dots x_r.t$ , which stands for  $t$  in the case of  $r = 0$ . To avoid too many brackets, we let the application associate to the left, i.e.,  $t t_1 \dots t_r$  stands for  $(\dots (t t_1) \dots t_r)$ , in particular, for  $r = 0$ ,  $t t_1 \dots t_r$  is equal to  $t$ .

**Definition 3.4.** For a  $\lambda_\Sigma$ -term  $t \in \lambda(\Sigma)$ , the sets  $\text{Fr}(t)$  of *free variables in  $t$*  and  $\text{Bnd}(t)$  of *bound variables in  $t$*  are defined inductively on the structure of  $t$ :

- (i)  $\text{Fr}(x) = \{x\}$ ,  $\text{Bnd}(x) = \emptyset$  for  $x \in X$ ,  
 $\text{Fr}(f) = \text{Bnd}(f) = \emptyset$  for  $f \in \Sigma$ ,
- (ii)  $\text{Fr}(t_1 t_2) = \text{Fr}(t_1) \cup \text{Fr}(t_2)$ ,  $\text{Bnd}(t_1 t_2) = \text{Bnd}(t_1) \cup \text{Bnd}(t_2)$ ,
- (iii)  $\text{Fr}(\lambda x.t) = \text{Fr}(t) \setminus \{x\}$ ,  $\text{Bnd}(\lambda x.t) = \text{Bnd}(t) \cup \{x\}$ .

The set  $\text{Var}(t) := \text{Fr}(t) \cup \text{Bnd}(t)$  is the *set of variables in  $t$* .

Now we define the set of  $\lambda_\Sigma$ -schemes as the set of all closed  $\lambda_\Sigma$ -terms.

**Definition 3.5.** The set  $\lambda^c(\Sigma)$  of  $\lambda$ -schemes (over  $\Sigma$ ) is defined by

$$\lambda^c(\Sigma) = \{t \mid t \in \lambda(\Sigma) \text{ and } \text{Fr}(t) = \emptyset\}.$$

Now we want to define the semantics of  $\lambda_\Sigma$ -terms in  $A_x$ . To this end we first introduce the notion of environment in order to associate meanings to free variables.

**Definition 3.6.** A mapping  $\rho: X \rightarrow A_x$  is called an *environment* (with respect to  $A$ ).  $\mathcal{U}_A$  denotes the set of all environments.  $\rho[x/a]$  is the environment which derives from  $\rho$  by changing the value of  $x$  to  $a$ , i.e.,

$$\rho[x/a](y) = \begin{cases} a & \text{for } x = y, \\ \rho(y) & \text{otherwise.} \end{cases}$$

**Definition 3.7.** The *semantics* (in  $A_x$ ) of  $\lambda_\Sigma$ -terms is defined inductively by the function  $\llbracket \cdot, \mathcal{A} \rrbracket: \mathcal{U}_A \rightarrow \lambda(\Sigma) \rightarrow A_x$  with

- (i)  $\llbracket x, \mathcal{A} \rrbracket \rho = \rho(x)$  for  $x \in X$ ,  
 $\llbracket f, \mathcal{A} \rrbracket \rho = \varphi^r(\alpha(f))$  for  $f \in \Sigma^r$ ,
- (ii)  $\llbracket t_1 t_2, \mathcal{A} \rrbracket \rho = \varphi(\llbracket t_1, \mathcal{A} \rrbracket \rho)(\llbracket t_2, \mathcal{A} \rrbracket \rho)$ ,
- (iii)  $\llbracket \lambda x.t, \mathcal{A} \rrbracket \rho = \psi(a \mapsto \llbracket t, \mathcal{A} \rrbracket \rho[x/a])$ .

**Convention 3.8.** We shall abbreviate  $\llbracket \cdot, \mathcal{A} \rrbracket$  to  $\llbracket \cdot \rrbracket$  whenever the interpretation is clear from the context.

**Lemma 3.9.** The semantics of a  $\lambda_\Sigma$ -term is already determined by the value of the environment on the free variables, i.e., for all environments  $\rho_1, \rho_2 \in \mathcal{U}_A$  and every  $\lambda_\Sigma$ -term  $t \in \lambda(\Sigma)$  the following holds:

$$[\forall x \in \text{Fr}(t), \rho_1(x) = \rho_2(x)] \Rightarrow \llbracket t \rrbracket \rho_1 = \llbracket t \rrbracket \rho_2.$$

In particular, for all  $\lambda_\Sigma$ -schemes  $S \in \lambda^c(\Sigma)$ ,  $\llbracket S \rrbracket \rho_1 = \llbracket S \rrbracket \rho_2$  for all environments  $\rho_1, \rho_2 \in \mathcal{U}_A$ .

**Proof.** The proof follows by straightforward induction on the structure of  $t$ .  $\square$

According to Lemma 3.9 the semantics of  $\lambda_{\Sigma}$ -schemes is independent of environments.

**Definition 3.10.** The *standard semantics* (with respect to  $\mathcal{A}$ )  $\llbracket \cdot, \mathcal{A} \rrbracket : \lambda^c(\Sigma) \rightarrow A_{\Sigma}$  of  $\lambda_{\Sigma}$ -schemes is defined by  $\llbracket S, \mathcal{A} \rrbracket = \llbracket S, \mathcal{A} \rrbracket \rho$  with arbitrary  $\rho \in \mathcal{U}_{\mathcal{A}}$  for all  $S \in \lambda^c(\Sigma)$ .

The next definition gives a typed semantics of any procedural type to an untyped  $\lambda_{\Sigma}$ -scheme using the retraction pairs defined in Section 2.

**Definition 3.11.** Let  $S \in \lambda^c(\Sigma)$  and  $\tau \in \mathcal{F}(I)$ . The  $\tau$ -*semantics* of  $S$  (over  $\mathcal{A}$ ) is defined by  $\llbracket S, \mathcal{A} \rrbracket^{\tau} := \psi^{\tau}(\llbracket S, \mathcal{A} \rrbracket)$ .

We can now characterize the class of  $\lambda_{\Sigma}$ -definable elements in  $A_{\Sigma}$  and in each  $A^{\tau}$ ,  $\tau \in \mathcal{F}(I)$ , as follows.

**Definition 3.12.** An element  $a$  in  $A_{\Sigma}$  is  $\lambda_{\Sigma}$ -definable iff there exists a  $\lambda_{\Sigma}$ -scheme  $S$  with  $\llbracket S, \mathcal{A} \rrbracket = a$ . The set  $\lambda^c(\Sigma)_{A_{\Sigma}}$  is the *set of all  $\lambda_{\Sigma}$ -definable elements in  $A_{\Sigma}$* .

Analogously,  $a \in A^{\tau}$  is  $\lambda_{\Sigma}$ -definable iff there exists an  $S \in \lambda^c(\Sigma)$  with  $\llbracket S, \mathcal{A} \rrbracket^{\tau} = a$  and  $\lambda^c(\Sigma)_{A^{\tau}}$  denotes the *set of  $\lambda_{\Sigma}$ -definable elements in  $A^{\tau}$* .

**Convention.** Instead of “ $\lambda_0$ –” we shall write “ $\lambda$ –” and for “ $\lambda(0)$ ” and “ $\lambda^c(0)$ ” we shall write “ $\lambda(\cdot)$ ” and “ $\lambda^c(\cdot)$ ” respectively.

It is now desirable to give an abstract characterization of the classes of  $\lambda_{\Sigma}$ -definable elements in  $A_{\Sigma}$  and  $A^{\tau}$ , for all  $\tau \in \mathcal{F}(I)$ . It is easy to show that the class of  $\lambda$ -definable elements of  $A^0$  contains only the minimal element.

Plotkin introduced in [61] a superset of  $\lambda$ -definable elements in  $A_{\Sigma}$ , namely all objects lying in any ‘logical relation’.

It is not known to us whether this is a strict superset or not.

**Definition 3.13.** Let  $S_1, S_2 \in \lambda^c(\Sigma)$  and  $\tau \in \mathcal{F}(I)$ .

$S_1$  is *equivalent* to  $S_2$  ( $S_1 =_{\Sigma} S_2$ ) iff  $\llbracket S_1, \mathcal{A} \rrbracket = \llbracket S_2, \mathcal{A} \rrbracket$  for all interpretations  $\mathcal{A}$ .

$S_1$  is  $\tau$ -*equivalent* to  $S_2$  ( $S_1 =_{\tau} S_2$ ) iff  $\llbracket S_1, \mathcal{A} \rrbracket^{\tau} = \llbracket S_2, \mathcal{A} \rrbracket^{\tau}$  for all interpretations  $\mathcal{A}$ .

This equivalence relation can be extended to classes of schemata with semantics defined with respect to continuous algebras.

We shall make this more clear in Definitions 7.9 and 7.10 for the case of typed lambda-schemes.

**Example 3.14.** Consider the following ALGOL-60 procedure “double”, which applies a function twice to the double value of an integer:

```
integer procedure double (f, n) integer procedure f; integer n;
double := f(f(2 * n)).
```

The expected type of “double” with respect to  $I = \{\text{int}, \dots\}$  is  $((\text{int}, \text{int})\text{int}, \text{int}) \in \mathcal{F}(I)$ .

The  $\lambda_{\Sigma}$ -scheme  $S$  with  $\Sigma = \{\mathbf{2}^{(\text{int}, \text{int})}, \mathbf{mult}^{(\text{int int}, \text{int})}, \dots\}$  was abstracted from “double”:

$$S = \lambda fx.(f(f(\mathbf{mult} \ 2x))).$$

Let  $\mathbb{Z} = (\mathbb{Z}_1, \{\mathbf{2} \mapsto \{(\ ) \mapsto 2\}, \mathbf{mult} \mapsto \{(z_1, z_2) \mapsto z_1 \cdot z_2\}, \dots\})$  be a continuous  $\Sigma$ -algebra for the integers, then

$$\llbracket S, \mathbb{Z} \rrbracket \in \mathbb{Z}_1, \quad \text{and} \quad \llbracket S, \mathbb{Z} \rrbracket^{((\text{int}, \text{int})\text{int}, \text{int})} \in [[\mathbb{Z}_1 \rightarrow \mathbb{Z}_1] \times \mathbb{Z}_1 \rightarrow \mathbb{Z}_1].$$

Now let  $(p, z) \in [\mathbb{Z}_1 \rightarrow \mathbb{Z}_1] \times \mathbb{Z}_1$ ; then the typed semantics of  $S$  applied to  $(p, z)$  is equal to

$$\llbracket S, \mathbb{Z} \rrbracket^{(\text{int}, \text{int}, \text{int}, \text{int})}(p, z) = p(p(2 \cdot z)),$$

as can be obtained by application of the semantic function given by Definition 3.11 and the retraction properties.

We omit a more interesting example, because in general for the treatment of programs and procedures the corresponding schemes have to be interpreted over more complicated algebras as, e.g., the algebra of state-transformations, where the states are themselves functions from locations to values.

A formal treatment of the connection between ALGOL-68 and lambda-schemes will be given in a forthcoming paper by Damm and Fehr.

A treatment of the relation between programming languages and the  $\lambda$ -calculus can be found, e.g., in [140].

#### 4. Analysis of the standard semantics

The correctness of the reduction semantics of  $\lambda_{\Sigma}$ -schemes is based on the fact that the domains  $A_{\Sigma}$  are models for the  $\lambda$ -calculus, as was shown by Scott [67], i.e., in other terms, the semantics of  $\lambda$ -equivalent  $\lambda_{\Sigma}$ -terms are equal.

The reduction semantics permits an approximation of the standard-semantics by a possibly infinite sequence of terms of finite length. This result is formalized in the approximation theorem by Wadsworth [74].

A characterization of those  $\lambda$ -schemes with a standard semantics different from  $\perp$  is given in the theorem of Böhm and Wadsworth [72].

We recall the  $\lambda$ -definability of the fixed-point operators, the identities and the minimal element and then we prove that the retractions from  $A_\infty$  onto the integer domains  $A^n (n \in \mathbb{N})$  are not  $\lambda$ -definable.

The sets  $I, \Sigma, X$  and  $\underline{A} = (A, \alpha)$  are considered to agree with the preceding sections.

The first mathematical model for the  $\lambda$ -calculus was the  $D_\infty$ -model developed by Scott, the construction of which we described in Section 2, starting from cpo's rather than continuous lattices. The model property of the domain  $A_\infty$  is the main theorem in the semantical analysis of  $\lambda_\Sigma$ -schemes.

**Theorem 4.1.**  $A_\infty$  is a model for the  $\lambda$ -calculus, i.e.,

$$\forall s, t \in \lambda(\Sigma), \forall \rho \in \mathcal{U}_A [s =_\lambda t \Rightarrow \llbracket s \rrbracket \rho = \llbracket t \rrbracket \rho],$$

where  $=_\lambda$  denotes the equivalence relation generated by the conversion rules of the  $\lambda$ -calculus.

**Proof.** As the semantics is defined inductively on the structure of  $\lambda_\Sigma$ -terms it is sufficient to prove the assertion for  $\alpha$ -,  $\beta$ - and  $\eta$ -redexes:

$$(\alpha) \quad \llbracket \lambda x. t \rrbracket \rho = \llbracket \lambda y. \$ \int y t \rrbracket \rho \quad \text{for } y \notin \text{Var}(t),$$

$$(\beta) \quad \llbracket (\lambda x. ts) \rrbracket \rho = \llbracket \$ \int x t \rrbracket \rho \quad \text{if } \text{Fr}(s) \cap \text{Bnd}(t) = \emptyset,$$

$$(\eta) \quad \llbracket \lambda x. (tx) \rrbracket \rho = \llbracket t \rrbracket \rho \quad \text{for } x \notin \text{Fr}(t).$$

( $\alpha$ ) and ( $\beta$ ) can be shown by induction on the structure of  $t$ , whereas ( $\eta$ ) is proved directly.  $\square$

From Lemma 2.8(i) it is clear that the semantics of a  $\lambda_\Sigma$ -term can be obtained as the limit of objects of finite type. Wadsworth [74] gave a method of how to approximate the semantics of  $\lambda_\Sigma$ -terms in normal form. The idea is to take all  $\lambda_\Sigma$ -terms which can be reduced from  $t$ , substitute the symbol  $\perp$  for all  $\beta$ -redexes occurring in these terms and then interpret them.

**Definition 4.2.** The  $\perp$ -substitution  $\phi : \lambda(\Sigma) \rightarrow \lambda(\Sigma_\perp)$  is given inductively by

$$\begin{aligned} & \phi(\lambda x_1 \dots x_r. h t_1 \dots t_k) \\ &= \begin{cases} \perp & \text{if } h \text{ is a } \beta\text{-redex,} \\ \lambda x_1 \dots x_r. (h \phi(t_1) \dots \phi(t_k)) & \text{otherwise.} \end{cases} \end{aligned}$$

**Definition 4.3.** The set  $A(t) \subseteq \lambda(\Sigma_\perp)$  of approximations of  $t \in \lambda(\Sigma)$  is defined by

$$A(t) = \{s \mid \exists r \in \lambda(\Sigma), t =_\lambda r \text{ and } s = \phi(r)\}.$$

**Theorem 4.4** (Wadsworth). Let  $t \in \lambda(\Sigma)$  and  $\rho \in \mathcal{U}_A$ ; then  $\llbracket t \rrbracket \rho = \bigsqcup \{\llbracket s \rrbracket \rho \mid s \in A(t)\}$ .

**Proof.** The proof follows from Wadsworth [74].

**Remark.** The set  $A(t)$  as given by Definition 4.3 is only a subset of the set of approximations as defined by Wadsworth. However, it is easy to prove that the semantics of each element in Wadsworth's set is less than the semantic of some element in  $A(t)$ .

**Theorem 4.5 (Böhm and Wadsworth).** For  $t \in \lambda^c(\ )$  the following three assertions are equivalent:

- (i) there exist  $t_1, \dots, t_k \in \lambda(\ )$  such that  $(t t_1 \dots t_k) =_\lambda \lambda x.x$ .
- (ii)  $\llbracket t \rrbracket \neq \perp$ .
- (iii)  $t$  has a head-normal form (HNF), i.e., a sequence of abstractions followed by an atom applied to a number of arbitrary terms (see [74] for a formal definition).

**Proof.** (i)  $\Rightarrow$  (ii): Let  $t_1, \dots, t_k \in \lambda(\ )$  such that  $(t t_1 \dots t_k) =_\lambda \lambda x.x$ ; Then  $\llbracket t t_1 \dots t_k \rrbracket = \llbracket \lambda x.x \rrbracket$  by Theorem 4.1.

Assume  $\llbracket t \rrbracket = \perp$ . Then  $\llbracket t t_1 \dots t_k \rrbracket = \perp$ , but  $\perp \neq \llbracket \lambda x.x \rrbracket$ . Hence  $\llbracket t \rrbracket \neq \perp$ .

(ii)  $\Rightarrow$  (iii): Let  $\llbracket t \rrbracket \neq \perp$ . Assume  $t$  has no HNF. Then  $\llbracket t \rrbracket = \perp$  by Theorem 4.4. Hence (iii) holds.

(iii)  $\Rightarrow$  (i): Let  $\lambda x_1. \dots \lambda x_r.(x_i t_1 \dots t_n)$  be a HNF of  $t$ . Since  $t$  is closed, it is necessary that  $i \leq r \leq n$ . Choose arbitrary  $s_1, \dots, s_r$  with  $s_i = \lambda y_1 \dots y_n. \lambda x.x$ . Then  $t s_1 \dots s_r =_\lambda \lambda x.x$  follows.  $\square$

The fixed-point operator, the identity and the minimal element are all in the class of  $\lambda$ -definable elements. In order to make use of this fact we shall now give a representative for each of them.

**Definition 4.6.** Let  $x, y \in X$ .

$$Y_\lambda := \lambda y.(\lambda x.(y(x x))\lambda x.(y(x x))), \quad I := \lambda x.x,$$

$$\Omega := (\lambda x.(x x)\lambda x.(x x)).$$

**Theorem 4.7 (Park).** The semantics of  $Y_\lambda$  applied to a function  $f$  is equal to the minimal fixed-point of  $f$ , i.e.,

$$\varphi(\llbracket Y_\lambda \rrbracket)(f) = \mu x.\varphi(f)(x) \quad \text{for all } f \in A_\lambda.$$

**Proof.** For the proof, see Park [59].  $\square$

**Lemma 4.8.**  $I$  defines the identity on  $A_\lambda$  and  $\Omega$  defines the minimal element, i.e.,

$$\varphi(\llbracket I \rrbracket) = \text{id}_{A_\lambda} \quad \text{and} \quad \llbracket \Omega \rrbracket = \perp.$$

**Proof.** The proof is straightforward, using the definitions and Definition 4.3.  $\square$

Now we can show [64] that the family of retraction pairs between  $A_x$  and  $A^n$ ,  $n \in \mathbb{N}$ , is not  $\lambda$ -definable, i.e., no  $\lambda$ -scheme computes the injection into  $A_x$  after retraction onto  $A^n$ .

**Theorem 4.9.** *For all  $n \in \mathbb{N}$ ,  $\psi(\varphi^n \circ \psi^n)$  is not  $\lambda$ -definable.*

**Proof.** The proof follows by induction on  $n$ :

*Basis ( $n = 0$ ):* Assume there exists an  $S \in \lambda^c(\ )$  such that  $\varphi(\llbracket S \rrbracket)(a) = \varphi^0(\psi^0(a))$ , for all Scott domains  $A_x$  and  $a \in A_x$ .

Choose  $\perp \neq a_0 \in A^0$  and let  $a = \varphi^0(a_0)$ .

*Case 1:*  $S$  has no HNF. Then  $\llbracket S \rrbracket = \perp$  by Theorem 4.5. But then  $\varphi(\llbracket S \rrbracket)(a) = \perp \neq a = \varphi^0(a_0) = \varphi^0(\psi^0(\varphi^0(a_0))) = \varphi^0(\psi^0(a))$ .

*Case 2:*  $S$  has an HNF. By Theorem 4.5 there exist  $t_1 \dots t_k \in \lambda(\ )$ , such that  $S t_1 \dots t_k =_\lambda \lambda x.x$ . Let  $a \in A_x$  be such that  $\varphi^0(\psi^0(\llbracket t_1 \rrbracket)) \neq a$  (note that  $k \neq 0$ , because  $\text{id}_{A_x} \neq \varphi^0 \circ \psi^0$ ). Then we realize

$$\begin{aligned} \varphi(\llbracket S t_1 \dots t_k \rrbracket)(a) &= \\ &= \varphi^{1k}(\varphi(\llbracket S \rrbracket)(\llbracket t_1 \rrbracket))(\llbracket t_2 \rrbracket, \dots, \llbracket t_k \rrbracket, a) \\ &= \varphi^{1k}(\varphi^0(\psi^0(\llbracket t_1 \rrbracket)))(\llbracket t_2 \rrbracket, \dots, \llbracket t_k \rrbracket, a) \quad \text{by the assumption} \\ &= \varphi^0(\psi^0(\llbracket t_1 \rrbracket)) \quad \text{by Lemma 2.13(ii)} \\ &\neq a \\ &= \varphi(\llbracket I \rrbracket)(a) \\ &= \varphi(\llbracket S t_1 \dots t_k \rrbracket)(a). \end{aligned}$$

Hence the theorem is true for  $r = 0$ .

*Induction Step ( $n + 1$ ):* Assume there exists  $S \in \lambda^c(\ )$  with  $\varphi(\llbracket S \rrbracket)(a) = \varphi^{n+1}(\psi^{n+1}(a))$  for all  $a \in A_x$ . It can be shown that the retractions  $\psi^{(n)}$  are  $\lambda$ -definable, i.e., there exist  $R_n \in \lambda(\ )$  with

$$\varphi(\llbracket R_n \rrbracket)(\varphi^{n+1}(a)) = \varphi^n(\psi^{(n)}(a)),$$

(Define  $R_0 = \lambda x.(x\Omega)$ ,  $I_0 = I$  and inductively

$$R_{n+1} = \lambda xy.(R_n(x(I_n y))), \quad I_{n+1} = \lambda xy.(I_n(x(R_n y))).)$$

It follows that

$$\begin{aligned} \varphi^n(\psi^n(a)) &= \varphi^n(\psi^{(n)}(\psi^{n+1}(a))) \\ &= \varphi(\llbracket R_n \rrbracket)(\varphi^{n+1}(\psi^{n+1}(a))) \\ &= \varphi(\llbracket R_n \rrbracket)(\varphi(\llbracket S \rrbracket)(a)) \quad \text{by assumption} \\ &= \varphi(\llbracket \lambda x.(R_n(Sx)) \rrbracket)(a). \end{aligned}$$

This contradicts the induction hypothesis and thus the assertion holds for  $n + 1$  as well.  $\square$

### 5. Interpretation in the initial algebra

In this section we first consider the algebra  $\underline{CT}_\Sigma$  of finite and infinite  $\Sigma$ -trees, which is initial in the class of interpretations [29]. The interpretation of an untyped  $\lambda_\Sigma$ -term  $t$  in  $\underline{CT}_\Sigma$  enforces in a certain sense a type-structure on all subterms of  $t$  (Lemmas 5.7 and 5.8). If  $t$  is typed in such a way, then it can be interpreted in an arbitrary continuous  $\Sigma$ -algebra  $A$  using the unique homomorphism  $h_A$ . It turns out that this procedure exactly determines the  $\mathbf{0}$ -semantics of  $t$  in  $A$  as defined in Section 3.

This result corresponds to the Mezei–Wright theorem for recursive schemes. It allows us to derive properties of programs such as equivalence, termination, etc., which were proved in the initial algebra, for all other interpretations.

From now on let the maximal arity of operation symbols in  $\Sigma$  be  $n \in N$ , and let  $\Delta = \{1, \dots, n\}^*$  denote the set of all words over “1”, “2”, ..., “ $n$ ”.

**Definition 5.1.** The  $I$ -cpo  $CT_\Sigma$  of finite and infinite  $\Sigma$ -trees is the least  $I$ -set of partial functions  $t: \Delta \rightarrow \Sigma$  with

- (i)  $\text{dom}(t) = \emptyset \Rightarrow \perp = t \in CT_\Sigma^i$  for all  $i \in I$ ,
- (ii)  $\delta\nu \in \text{dom}(t)$  with  $t(\delta\nu) \in \Sigma^{(w, \nu)} \Rightarrow$   
 $t(\delta) \in \Sigma^{(i_0, \dots, i_{\nu-1}, \nu, i_{\nu+1}, \dots, i_r)}$   
for some  $1 \leq \nu \leq r \in N$  and  $i_k \in I$  for  $0 \leq k \leq r$ ,
- (iii)  $t(e) \in \Sigma^{(w, \nu)} \Rightarrow t \in CT_\Sigma^i$ .

The order relation on  $CT_\Sigma$  is the set-theoretical inclusion of the corresponding graphs, i.e.,

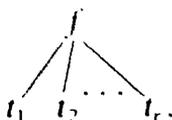
$$t_1 \leq t_2 \text{ iff } \text{graph}(t_1) \subseteq \text{graph}(t_2).$$

**Definition 5.2.** The  $\Sigma$ -algebra  $\underline{CT}_\Sigma = (CT_\Sigma, \sigma)$  is defined by  $\sigma: \Sigma \rightarrow CT_\Sigma^{D(I)}$  with

$$\sigma(f)(t_1, \dots, t_r)(\delta) = \begin{cases} f & \text{for } \delta = e, \\ t_\nu(\eta) & \text{for } \delta = \nu\eta \text{ and } 1 \leq \nu \leq r, \\ \perp & \text{otherwise} \end{cases}$$

for all  $f \in \Sigma^{(i_0, \dots, i_{r-1})}$ ,  $t_\nu \in CT_\Sigma^{i_\nu}$ ,  $1 \leq \nu \leq r$  and  $\delta \in \Delta$ .

**Notation.** Instead of  $\sigma(f)(t_1, \dots, t_r)$  we also write



**Definition 5.3.** Let  $t$  be a  $\Sigma$ -tree,  $\delta \in \Delta$  and  $k \in \mathbb{N}$ ; then  $t_\delta$  denotes the *subtree of  $t$  at node  $\delta$* , i.e.,  $t_\delta(\eta) = t(\delta\eta)$  for all  $\eta \in \Delta$ . In particular,  $t_e = t$ .

The *initial tree  $t^k$  of  $t$  with height  $k$*  is obtained from  $t$  by restricting the domain of  $t$  on words with length less or equal to  $k$ , i.e.,

$$t^k(\eta) = \begin{cases} t(\eta) & \text{for } |\eta| \leq k, \\ \perp & \text{otherwise,} \end{cases} \quad \text{for all } \eta \in \Delta.$$

**Lemma 5.4**

$$t = \bigsqcup_{k \in \mathbb{N}} t^k \quad \text{for all } t \in \text{CT}_\Sigma.$$

**Proof.** For the proof, see [29, Proposition 4.2].  $\square$

**Theorem 5.5 (ADJ).**  $\text{CT}_\Sigma$  is initial in the class of interpretations, i.e., there exists a unique, strict continuous homomorphism  $h_A: \text{CT}_\Sigma \rightarrow \underline{A}$  for each continuous  $\Sigma$ -algebra  $\underline{A}$ .

**Proof.** For the proof, see ADJ [29, Corollary 4.10].  $\square$

We now need the property whereby the semantics of  $\lambda_\Sigma$ -schemes is compatible with the Curry isomorphisms, as there is only monadic application in  $\lambda_\Sigma$ -schemes, whereas in  $\text{CT}_\Sigma$  arities are taken into consideration.

**Lemma 5.6.** Let  $\underline{A} = (A, \alpha)$  be an interpretation,  $\rho \in \mathcal{U}_A$ ,  $t, t_r \in \lambda(\Sigma)$ ,  $x_r \in X$ ,  $1 \leq r \leq r$ .

- (a)  $\llbracket t t_1 \dots t_r \rrbracket \rho = \varphi^{lr}(\llbracket t \rrbracket \rho)(\llbracket t_1 \rrbracket \rho, \dots, \llbracket t_r \rrbracket \rho)$ ,
- (b)  $\llbracket \lambda x_1 \dots x_r. t \rrbracket \rho = \psi^{lr}((a_1, \dots, a_r) \mapsto \llbracket t \rrbracket \rho[x_1/a_1] \dots [x_r/a_r])$ .

**Proof.** The proof follows by straightforward induction on  $r$ , using Definition 2.10.  $\square$

The next two lemmas give insight into the way in which  $\lambda_\Sigma$ -schemes with untyped abstraction and application are forced into a type-structure by interpretation in the carrier of a continuous  $\Sigma$ -algebra.

**Lemma 5.7.** Let  $\underline{A} = (A, \alpha)$  be an interpretation,  $i \in I$ ,  $\lambda x_1 \dots x_r. S \in \lambda^c(\Sigma)$  and  $x_r \in X$ ,  $1 \leq r \leq r$ . Then

$$\llbracket \lambda x_1 \dots x_r. S, \underline{A} \rrbracket' = \psi^i(\llbracket S, \underline{A} \rrbracket \rho[x_1/\perp] \dots [x_r/\perp]) \quad \text{for arbitrary } \rho \in \mathcal{U}_A.$$

**Proof.** The proof follows by induction on  $r$ .

*Basis* ( $r = 0$ ):  $\llbracket S, \underline{A} \rrbracket' = \psi^i(\llbracket S, \underline{A} \rrbracket)$  by definition.

**Induction Step ( $r+1$ ):**

$$\begin{aligned}
 \llbracket \lambda x_1 \dots x_{r+1}. S, \mathcal{A} \rrbracket^i &= \\
 &= \psi^i(\llbracket \lambda x_{r+1}. S, \mathcal{A} \rrbracket^i \rho[x_1/\perp] \dots [x_r/\perp]) \quad \text{by induction hypothesis} \\
 &= \psi^i(\varphi(\llbracket \lambda x_{r+1}. S, \mathcal{A} \rrbracket^i \rho[x_1/\perp] \dots [x_r/\perp])(\perp)) \quad \text{by Lemma 2.8(iv)} \\
 &= \psi^i(\llbracket S, \mathcal{A} \rrbracket^i \rho[x_1/\perp] \dots [x_{r+1}/\perp]). \quad \square
 \end{aligned}$$

**Lemma 5.8.** *Let  $\mathcal{A}$  be an interpretation,  $f \in \Sigma^{(i_1 \dots i_k)}$ ,  $r \in \mathbb{N}$  and  $S_\nu \in \lambda^c(\Sigma)$ ,  $1 \leq \nu \leq r$ . Then*

$$\llbracket f S_1 \dots S_r, \mathcal{A} \rrbracket^i = \begin{cases} \alpha(f)(\llbracket S_1, \mathcal{A} \rrbracket^{i_1}, \dots, \llbracket S_k, \mathcal{A} \rrbracket^{i_k}) & \text{for } k \leq r \\ \alpha(f)(\llbracket S_1, \mathcal{A} \rrbracket^{i_1}, \dots, \llbracket S_r, \mathcal{A} \rrbracket^{i_r}, \perp, \dots, \perp) & \text{for } k > r. \end{cases}$$

**Proof.** The proof is divided in two cases:

*Case  $k \leq r$ :*

$$\begin{aligned}
 \llbracket f S_1 \dots S_r \rrbracket^i &= \\
 &= \psi^i(\varphi^{i_1}(\llbracket f \rrbracket)(\llbracket S_1 \rrbracket, \dots, \llbracket S_r \rrbracket)) \quad \text{by Lemma 5.6} \\
 &= \psi^i(\varphi^{i_1}(\psi^{i_1}(\varphi^{i_1}(\alpha(f) \circ (\psi^{i_1} \times \dots \times \psi^{i_k}))))(\llbracket S_1 \rrbracket, \dots, \llbracket S_r \rrbracket)) \\
 & \quad \text{by Definition 2.14} \\
 &= \psi^i(\varphi^{i_1}(\varphi^{i_1}(\alpha(f)(\llbracket S_1 \rrbracket^{i_1}, \dots, \llbracket S_k \rrbracket^{i_k}))))(\llbracket S_{k+1} \rrbracket, \dots, \llbracket S_r \rrbracket)) \\
 & \quad \text{by Lemma 2.12(i)} \\
 &= \psi^i(\varphi^i(\alpha(f)(\llbracket S_1 \rrbracket^{i_1}, \dots, \llbracket S_k \rrbracket^{i_k}))) \quad \text{by Lemma 2.13(ii)} \\
 &= \alpha(f)(\llbracket S_1 \rrbracket^{i_1}, \dots, \llbracket S_k \rrbracket^{i_k}) \quad \text{by Lemma 2.15.}
 \end{aligned}$$

*Case  $k > r$ :*

$$\begin{aligned}
 \llbracket f S_1 \dots S_r \rrbracket^i &= \\
 &= \psi^i(\psi^{i_k}(\varphi^{i_k}((a_{r+1}, \dots, a_k) \mapsto \varphi^i(\alpha(f)(\llbracket S_1 \rrbracket^{i_1}, \dots, \llbracket S_r \rrbracket^{i_r}, a_{r+1}, \dots, a_k)))))) \\
 & \quad \text{by Lemma 2.12(ii)} \\
 &= \psi^i(\varphi^i(\alpha(f)(\llbracket S_1 \rrbracket^{i_1}, \dots, \llbracket S_r \rrbracket^{i_r}, \perp, \dots, \perp))) \\
 & \quad \text{by Lemmas 2.13(i) and 2.11} \\
 &= \alpha(f)(\llbracket S_1 \rrbracket^{i_1}, \dots, \llbracket S_r \rrbracket^{i_r}, \perp, \dots, \perp) \quad \text{by Lemma 2.15.} \quad \square
 \end{aligned}$$

Lemmas 5.7 and 5.8 can be combined for terms in HNF with head-atoms out of  $\Sigma$  as follows.

**Corollary 5.9.** Let  $\underline{A}$ ,  $x_\nu$ ,  $1 \leq \nu \leq s$ ,  $f$  and  $S_\mu$ ,  $1 \leq \mu \leq r$  be defined as in Lemmas 5.7 and 5.8:

$$\llbracket \lambda x_1 \dots x_s (f S_1 \dots S_r) \rrbracket^i = \begin{cases} \alpha(f)(\llbracket \lambda x_1 \dots x_s S_1 \rrbracket^i, \dots, \llbracket \lambda x_1 \dots x_s S_k \rrbracket^{i_k}) & \text{for } k \leq r, \\ \alpha(f)(\llbracket \lambda x_1 \dots x_s S_1 \rrbracket^i, \dots, \llbracket \lambda x_1 \dots x_s S_r \rrbracket^i, \perp, \dots, \perp) & \text{for } k > r. \end{cases}$$

The following lemma is a ‘Mezei–Wright-like’ result for  $\lambda_\Sigma$ -schemes in normal form (NF).

**Lemma 5.10.** Let  $N \in \lambda^c(\Sigma)$  in NF,  $i \in I$  and  $\underline{A} = (A, \alpha)$  be an interpretation; then

$$h_{\underline{A}}(\llbracket N, \underline{CT}_\Sigma \rrbracket^i) = \llbracket N, \underline{A} \rrbracket^i.$$

**Proof.** The proof follows by induction on the structure of  $N$ .

*Basis* ( $N = \lambda x_1 \dots x_r . x$ ):

$$\begin{aligned} h_{\underline{A}}(\llbracket N, \underline{CT}_\Sigma \rrbracket^i) &= \\ &= h_{\underline{A}}(\psi^i(\psi^{1^r}((a_1, \dots, a_r) \mapsto \llbracket x \rrbracket \rho[x_1/a_1] \dots [x_r/a_r]))) \\ &= h_{\underline{A}}(\psi^i(\llbracket x \rrbracket \rho[x_1/\perp] \dots [x_r/\perp])) \quad \text{by Lemmas 2.13(i) and 2.11.} \end{aligned}$$

*Case*  $x \in \{x_1, \dots, x_r\}$ :

$$\begin{aligned} &= h_{\underline{A}}(\perp_{CT_\Sigma^i}) = \perp_{A^i} \quad \text{since } h_{\underline{A}} \text{ is strict} \\ &= \llbracket N, \underline{A} \rrbracket^i. \end{aligned}$$

*Case*  $x \in \Sigma^{(i_1 \dots i_n)}$ :

$$\begin{aligned} &= h_{\underline{A}}(\psi^i(\psi^{1^n}(\varphi^{i_0} \circ \sigma(x) \circ (\psi^{i_1} \times \dots \times \psi^{i_n}))) \quad \text{by Definition 2.14} \\ &= h_{\underline{A}}(\psi^i(\varphi^{i_0}(\sigma(x)(\perp_{CT_\Sigma^{i_1}}, \dots, \perp_{CT_\Sigma^{i_n}})))) \quad \text{by Lemmas 2.13(i) and 2.11} \\ &= \begin{cases} \perp_{A^i} = \llbracket N, \underline{A} \rrbracket^i & \text{for } i \neq i_0, \\ \alpha(x)(\perp_{A^{i_1}}, \dots, \perp_{A^{i_n}}) = \llbracket N, \underline{A} \rrbracket^i & \text{for } i = i_0. \end{cases} \end{aligned}$$

*Induction Step* ( $N = \lambda x_1 \dots x_r . (x t_1 \dots t_k)$ ):

$$\begin{aligned} h_{\underline{A}}(\llbracket N, \underline{CT}_\Sigma \rrbracket^i) &= \\ &= h_{\underline{A}}(\psi^i(\psi^{1^r}((a_1, \dots, a_r) \mapsto \llbracket x t_1 \dots t_k, \underline{CT}_\Sigma \rrbracket \rho[x_1/a_1] \dots [x_r/a_r]))) \\ &= h_{\underline{A}}(\psi^i(\llbracket x t_1 \dots t_k, \underline{CT}_\Sigma \rrbracket \rho')) \quad \text{with } \rho' = \rho[x_1/\perp] \dots [x_r/\perp] \\ & \hspace{15em} \text{by Lemmas 2.13(i) and 2.11} \\ &= h_{\underline{A}}(\psi^i(\varphi^{1^k}(\llbracket x, \underline{CT}_\Sigma \rrbracket \rho')(\llbracket t_1, \underline{CT}_\Sigma \rrbracket \rho', \dots, \llbracket t_k, \underline{CT}_\Sigma \rrbracket \rho')))) \\ & \hspace{15em} \text{by Lemma 5.6(a).} \end{aligned}$$

Case  $x \in \{x_1, \dots, x_r\}$ :

$$= h_A(\perp_{\mathcal{C}T_S}) = \perp_{A'} = \llbracket N, A \rrbracket' \quad \text{since } h_A \text{ is strict.}$$

Case  $x \in \Sigma^{(i_1, \dots, i_s, i_0)}$ :

$$= h_A(\psi^i(\varphi^{[k]}(\psi^{[s]}(\varphi^{i_0} \circ \sigma(x) \circ (\psi^{i_1} \times \dots \times \psi^{i_s}))) (\llbracket t_1, \underline{\mathcal{C}T_S} \rrbracket \rho', \dots, \llbracket t_k, \underline{\mathcal{C}T_S} \rrbracket \rho'))).$$

Case  $s \leq k$ :

$$\begin{aligned} &= h_A(\psi^i(\varphi^{[k-s]}(\varphi^{i_0}(\sigma(x)(\psi^{i_1}(\llbracket t_1, \underline{\mathcal{C}T_S} \rrbracket \rho'), \dots, \psi^{i_s}(\llbracket t_s, \underline{\mathcal{C}T_S} \rrbracket \rho')))) \\ &\quad (\llbracket t_{s+1}, \underline{\mathcal{C}T_S} \rrbracket \rho', \dots, \llbracket t_k, \underline{\mathcal{C}T_S} \rrbracket \rho')) \quad \text{by Lemma 2.12(i)} \\ &= h_A(\psi^i(\varphi^{i_0}(\sigma(x)(\psi^{i_1}(\llbracket t_1, \underline{\mathcal{C}T_S} \rrbracket \rho'), \dots, \psi^{i_s}(\llbracket t_s, \underline{\mathcal{C}T_S} \rrbracket \rho'))))) \\ &\quad \text{by Lemma 2.13(ii).} \end{aligned}$$

Case  $i \neq i_0$ :

$$= h_A(\perp_{\mathcal{C}T_S}) = \perp_{A'} = \llbracket N, A \rrbracket'.$$

Case  $i = i_0$ :

$$\begin{aligned} &= h_A(\sigma(x)(\psi^{i_1}(\llbracket t_1, \underline{\mathcal{C}T_S} \rrbracket \rho'), \dots, \psi^{i_s}(\llbracket t_s, \underline{\mathcal{C}T_S} \rrbracket \rho')))) \quad \text{by Definition 2.14} \\ &= \alpha(x)(\llbracket \lambda x_1 \dots x_r. t_1, \underline{\mathcal{C}T_S} \rrbracket^{i_1}, \dots, \llbracket \lambda x_1 \dots x_r. t_s, \underline{\mathcal{C}T_S} \rrbracket^{i_s}) \\ &\quad \text{by Lemmas 2.13(i) and 2.11} \\ &= \llbracket N, A \rrbracket' \quad \text{by induction hypothesis.} \end{aligned}$$

Case  $s > k$ :

$$\begin{aligned} &= h_A(\psi^i(\varphi^{[s-k]}(a_{k+1}, \dots, a_s) \mapsto \varphi^{i_0}(\sigma(x)(\psi^{i_1}(\llbracket t_1, \underline{\mathcal{C}T_S} \rrbracket \rho'), \dots, \\ &\quad \psi^{i_k}(\llbracket t_k, \underline{\mathcal{C}T_S} \rrbracket \rho'), a_{k+1}, \dots, a_s)))) \quad \text{by Lemma 2.12(ii)} \\ &= h_A(\psi^i(\varphi^{i_0}(\sigma(x)(\psi^{i_1}(\llbracket t_1, \underline{\mathcal{C}T_S} \rrbracket \rho'), \dots, \psi^{i_k}(\llbracket t_k, \underline{\mathcal{C}T_S} \rrbracket \rho'), \\ &\quad \perp, \dots, \perp)))) \quad \text{by Lemmas 2.13(i) and 2.11.} \end{aligned}$$

Case  $i \neq i_0$ :

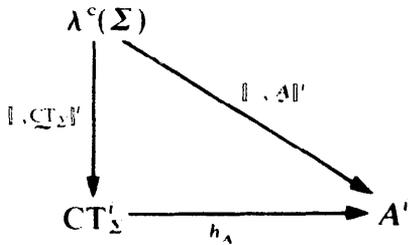
$$= h_A(\perp_{\mathcal{C}T_S}) = \perp_{A'} = \llbracket N, A \rrbracket' \quad \text{since } h_A \text{ is strict.}$$

Case  $i = i_0$ :

$$\begin{aligned} &= h_A(\sigma(x)(\llbracket \lambda x_1 \dots x_r. t_1, \underline{\mathcal{C}T_S} \rrbracket^{i_1}, \dots, \llbracket \lambda x_1 \dots x_r. t_k, \underline{\mathcal{C}T_S} \rrbracket^{i_k}, \\ &\quad \perp, \dots, \perp)) \quad \text{by Definition 2.14, Lemmas 2.13(i) and 2.11} \\ &= \alpha(x)(\llbracket \lambda x_1 \dots x_r. t_1, A \rrbracket^{i_1}, \dots, \llbracket \lambda x_1 \dots x_r. t_k, A \rrbracket^{i_k}, \perp, \dots, \perp) \\ &\quad \text{by induction hypothesis.} \\ &= \llbracket N, A \rrbracket'. \quad \square \end{aligned}$$

We can now formulate an important theorem for  $\lambda_\Sigma$ -schemes which allows us to transfer results which hold in  $\underline{CT}_\Sigma$  to all other interpretations.

**Theorem 5.11.** *The diagram*



*commutes for all interpretations  $\mathcal{A}$  and all  $i \in I$ .*

**Proof.** Let  $S \in \lambda^c(\Sigma)$ .

$$\begin{aligned}
 h_A(\llbracket S, \underline{CT}_\Sigma \rrbracket') &= \\
 &= h_A(\psi'(\bigsqcup\{\llbracket N, \underline{CT}_\Sigma \rrbracket' \mid N \in \mathcal{A}(S)\})) && \text{by Theorem 4.4} \\
 &= \bigsqcup\{h_A(\llbracket N, \underline{CT}_\Sigma \rrbracket') \mid N \in \mathcal{A}(S)\} && \text{by continuity of } h_A \text{ and } \psi' \\
 &= \bigsqcup\{\llbracket N, \mathcal{A}' \rrbracket' \mid N \in \mathcal{A}(S)\} && \text{by Lemma 5.10} \\
 &= \llbracket S, \mathcal{A}' \rrbracket' && \text{by Theorem 4.4. } \square
 \end{aligned}$$

### 6. Abstract characterization of the power of $\lambda_\Sigma$ -schemes

In this section we obtain the result that an element of the carrier of a continuous  $\Sigma$ -algebra is  $\lambda_\Sigma$ -definable iff it is the homomorphic image of a partial recursive  $\Sigma$ -tree.

Basic notions and results like partial recursive word functions, Church's thesis, etc., from recursion theory are assumed to be known (take, e.g., Rogers [66] as reference).

In the following we shall make use of a special coding of words over a finite alphabet in closed lambda-terms. Let from now on  $\Sigma = \{f_1, \dots, f_r\}$  and  $I := \Sigma \cup \{1, \dots, n\}$ , where  $n$  is the maximal arity of the symbols in  $\Sigma$ .

**Definition 6.1.** *The coding  $\bar{\cdot} : I^* \rightarrow \lambda^c(\ )$  is defined by  $\bar{w} := \lambda x_1 \dots x_r y_1 \dots y_n y. \hat{w}$ , where  $\hat{\cdot} : I^* \rightarrow \lambda(\ )$  is given inductively by*

$$\hat{e} := y$$

and

$$\hat{w}a := \begin{cases} (x_\nu, \hat{w}) & \text{for } a = f_\nu \in \Sigma, \\ (y_i, \hat{w}) & \text{for } a = i \in \{1, \dots, n\}. \end{cases}$$

**Remark.** This coding is a generalization of Church's representation of integers to words over  $\Gamma$ , where  $\Gamma^*$  is considered to be the freely generated algebra over the empty word  $e$  and a unary operation for each  $a \in \Gamma$ .

**Theorem 6.2 (Church).** A function  $f: \Gamma^* \rightarrow \Gamma^*$  is partial recursive iff  $f$  is  $\lambda$ -definable, i.e.,

$$\exists f_\lambda \in \lambda^c(\quad), \forall w \in \Gamma^*, \overline{f(w)} =_\lambda (f_\lambda \bar{w}).$$

**Proof.** A detailed proof for the integer case can be found in [32a]. The  $\lambda$ -definability of functions over Peano-algebras was shown by Mitschke [51] with respect to a coding different from the one used in this paper. For a full proof of the above theorem, see [26a].  $\square$

We shall now define a  $\lambda_\Sigma$ -scheme  $P$ , which applied to an element out of  $\Sigma$  yields this element as result:

$$P := \lambda x. (\dots ((\dots (x \lambda z. f_1) \dots \lambda z. f_r) \Omega) \dots \Omega).$$

**Lemma 6.3.** Let  $f \in \Sigma$ . Then

$$(P \bar{f}) =_\lambda f.$$

**Proof.** Let  $f = f_i \in \Sigma$ ,  $1 \leq i \leq r$ . Then

$$\begin{aligned} (P \bar{f}) &=_\lambda (\dots ((\dots (\bar{f} \lambda z. f_1) \dots \lambda z. f_r) \Omega) \dots \Omega) \\ &=_\lambda (\lambda z. f_i \Omega) \\ &=_\lambda f_i. \quad \square \end{aligned}$$

Let  $\text{PR}(\text{CT}_\Sigma)$  denote the set of partial recursive  $\Sigma$ -trees, where  $t: \Delta \rightarrow \Sigma \in \text{CT}_\Sigma$  is partial recursive iff there exists  $f_i: \Gamma^* \rightarrow \Gamma^*$  partial recursive such that  $f_{i|\Delta} = t$ . (For a more precise definition of partial recursive word functions, see [32].)

**Lemma 6.4.** Let  $t \in \text{CT}_\Sigma$  be partial recursive via  $f_i: \Gamma^* \rightarrow \Gamma^*$ ; then there exists a  $\lambda_\Sigma$ -scheme  $t_\lambda$  which 'approximates'  $t$ , i.e., for all  $\delta \in \Delta$  with  $t_\delta \in \text{CT}_\Sigma^i$  the following holds:

$$\llbracket t_\lambda^i \Omega \bar{\delta}, \text{CT}_\Sigma \rrbracket^i = t_\delta^{i-1} \quad \text{for all } 1 \leq i \in \mathbb{N}.$$

**Proof.** By the theorem of Church (Theorem 6.2) there exists a  $\lambda$ -scheme  $f_{i\lambda}$  with  $(f_{i\lambda} \bar{\delta}) =_\lambda \overline{f_i(\delta)}$ , and for each  $1 \leq i \leq n$  there exists a  $\lambda$ -scheme  $S_\nu$  with  $(S_\nu \bar{\delta}) = \overline{\delta \nu}$ .

Define

$$t_\lambda := \lambda r x. (\dots (P(f_{i_\lambda} x))(r(S_1 x))) \dots (r(S_n x)).$$

We show the assertion by induction on  $\nu$ :

*Basis* ( $\nu = 1$ ):

$$\begin{aligned} \llbracket t_\lambda \Omega \bar{\delta} \rrbracket^i &= \llbracket (\dots (P(f_{i_\lambda} \bar{\delta}))(\Omega(S_1 \bar{\delta})) \dots (\Omega(S_n \bar{\delta}))) \rrbracket^i \quad \text{by Theorem 4.1} \\ &= \llbracket (\dots (\overline{P t(\delta)})(\Omega(S_1 \bar{\delta})) \dots (\Omega(S_n \bar{\delta}))) \rrbracket^i \\ &\quad \text{by choice of } f_{i_\lambda} \text{ and Theorem 4.1} \\ &= \llbracket (\dots (t(\delta))(\Omega(S_1 \bar{\delta})) \dots (\Omega(S_n \bar{\delta}))) \rrbracket^i \\ &\quad \text{by Lemma 6.3 and Theorem 4.1} \\ &= \sigma(t(\delta)) \llbracket \Omega(S_1 \bar{\delta}) \rrbracket^i \dots \llbracket \Omega(S_n \bar{\delta}) \rrbracket^i \\ &\quad \text{by Lemma 5.8 with } t(\delta) \in \Sigma^{(i_1, \dots, i_n)} \\ &= \sigma(t(\delta))(\perp, \dots, \perp) \quad \text{by Lemma 4.8} \\ &= t_\delta^0 \quad \text{refer to Definitions 5.2 and 5.3.} \end{aligned}$$

*Induction Step* ( $\nu + 1$ ):

$$\begin{aligned} \llbracket t_\lambda^{\nu+1} \Omega \bar{\delta} \rrbracket^i &= \\ &= \llbracket t_\lambda (t_\lambda^\nu \Omega) \bar{\delta} \rrbracket^i = \\ &= \llbracket (\dots (P(f_{i_\lambda} \bar{\delta}))(t_\lambda^\nu \Omega)(S_1 \bar{\delta})) \dots ((t_\lambda^\nu \Omega)(S_n \bar{\delta})) \rrbracket^i \quad \text{by Theorem 4.1} \\ &= \llbracket (\dots (\overline{P t(\delta)})((t_\lambda^\nu \Omega) \bar{\delta} 1) \dots ((t_\lambda^\nu \Omega) \bar{\delta} n)) \rrbracket^i \\ &\quad \text{by choice of } f_{i_\lambda} \text{ and } S_\nu, \nu \in \{1, \dots, n\} \text{ and Theorem 4.1} \\ &= \llbracket t(\delta)((t_\lambda^\nu \Omega) \bar{\delta} 1) \dots ((t_\lambda^\nu \Omega) \bar{\delta} n) \rrbracket^i \\ &\quad \text{by Lemma 6.3 and Theorem 4.1} \\ &= \sigma(t(\delta))(\llbracket t_\lambda^\nu \Omega \bar{\delta} 1 \rrbracket^i, \dots, \llbracket t_\lambda^\nu \Omega \bar{\delta} n \rrbracket^i) \\ &\quad \text{by Lemma 5.8 with } t(\delta) \in \Sigma^{(i_1, \dots, i_n)} \\ &= \sigma(t(\delta))(t_{\delta_1}^{\nu-1}, \dots, t_{\delta_n}^{\nu-1}) \quad \text{by induction hypothesis} \\ &= t_\delta^\nu \quad \text{refer to Definitions 5.2 and 5.3.} \quad \square \end{aligned}$$

We can now show that each partial recursive  $\Sigma$ -tree is  $\lambda_\Sigma$ -definable.

**Lemma 6.5.** *Let  $t \in \text{CT}_\Sigma^1$  be partial recursive via  $f_i: I^* \rightarrow I^*$ ; then there exists a  $\lambda_\Sigma$ -scheme  $T_\lambda$  such that  $\llbracket T_\lambda, \text{CT}_\Sigma \rrbracket^i = t$ .*

**Proof.** Let  $t_\lambda$  be the  $\lambda_\Sigma$ -scheme which approximates  $t$ . Define  $T_\lambda := ((Y_\lambda t_\lambda)\bar{e})$ .

$$\begin{aligned}
 \llbracket T_\lambda \rrbracket' &= \\
 &= \psi'(\varphi(\varphi(\llbracket Y_\lambda \rrbracket)(\llbracket t_\lambda \rrbracket))(\llbracket \bar{e} \rrbracket)) \\
 &= \psi'(\varphi(\mu a.\varphi(\llbracket t_\lambda \rrbracket)(a))(\llbracket \bar{e} \rrbracket)) \quad \text{by Theorem 4.7} \\
 &= \psi'\left(\varphi\left(\bigsqcup_{i \in \mathbb{N}} \varphi(\llbracket t_\lambda \rrbracket)'(\perp)\right)(\llbracket \bar{e} \rrbracket)\right) \quad \text{by Theorem 1.5} \\
 &= \bigsqcup_{i \in \mathbb{N}} \psi'(\varphi(\varphi(\llbracket t_\lambda \rrbracket)'(\perp))(\llbracket \bar{e} \rrbracket)) \quad \text{by continuity of } \varphi \text{ and } \psi' \\
 &= \bigsqcup_{i \in \mathbb{N}} \llbracket t_\lambda^i \Omega \bar{e} \rrbracket' \quad \text{by Lemma 4.8} \\
 &= \bigsqcup_{i \in \mathbb{N}} t_i' \quad \text{by Lemma 6.4} \\
 &= t \quad \text{with Definition 5.3 and Lemma 5.4.} \quad \square
 \end{aligned}$$

The next lemma contains the inversion.

**Lemma 6.6.** *Let  $T \in \lambda^c(\Sigma)$ ; then  $\llbracket T, \underline{CT}_\Sigma \rrbracket'$  is partial recursive for all  $i \in I$ .*

**Proof.** We first prove the assertion inductively for all  $T$  in normal form:

**Basis:**  $T = \lambda x_1 \dots x_r. a$  with  $a \in \Sigma \cup \{x_1, \dots, x_r\}$ .

$$\begin{aligned}
 \llbracket T, \underline{CT}_\Sigma \rrbracket' &= \psi'(\llbracket a \rrbracket \rho[x_1/\perp] \dots [x_r'/\perp]) \quad \text{by Lemma 5.7} \\
 &= \begin{cases} \perp & \text{for } a \in \{x_1, \dots, x_r\} \text{ by Lemma 5.8,} \\ \sigma(a)(\perp, \dots, \perp) & \text{for } a \in \Sigma. \end{cases}
 \end{aligned}$$

Both  $\perp$  and  $\begin{array}{c} a \\ \diagdown \quad \diagup \\ \perp \quad \perp \quad \perp \end{array}$  are partial recursive.

**Induction Step:**  $T = \lambda x_1 \dots x_r.(a t_1 \dots t_k)$ .

$$\begin{aligned}
 \llbracket T, \underline{CT}_\Sigma \rrbracket' &= \\
 &= \psi'(\llbracket a t_1 \dots t_k \rrbracket \bar{\rho}) \quad \text{with } \bar{\rho} = \rho[x_1/\perp] \dots [x_r/\perp] \quad \text{by Lemma 5.7} \\
 &= \begin{cases} \perp & \text{for } a \in \{x_1, \dots, x_r\}, \\ \sigma(a)(\psi^1(\llbracket t_1 \rrbracket \bar{\rho}), \dots, \psi^k(\llbracket t_k \rrbracket \bar{\rho})) & \\ & \text{for } a \in \Sigma^{(1) \dots (k)} \text{ and } s \leq k, \\ \sigma(a)(\psi^1(\llbracket t_1 \rrbracket \bar{\rho}), \dots, \psi^k(\llbracket t_k \rrbracket \bar{\rho}), \perp, \dots, \perp) & \\ & \text{for } a \in \Sigma^{(1) \dots (k)} \text{ and } s > k \text{ by Lemma 5.8.} \end{cases}
 \end{aligned}$$

All cases are by induction hypothesis partial recursive. We can now enumerate all approximations of  $T$ , i.e.,  $\mathcal{A}(T) = \{t_\nu \mid \nu \in \mathbb{N}\}$  (see Definition 4.3), and determine the value of

$$\llbracket T, \underline{\text{CT}}_\Sigma \rrbracket^i(\delta) = \left( \bigsqcup_{\nu \in \mathbb{N}} \llbracket t_\nu, \underline{\text{CT}}_\Sigma \rrbracket^i \right)(\delta)$$

by calculating

$$\bigsqcup_{\nu \in \mathbb{N}} (\llbracket t_\nu, \underline{\text{CT}}_\Sigma \rrbracket^i(\delta)) \quad (\text{see Lemma 1.10}).$$

Thus  $\llbracket T, \underline{\text{CT}}_\Sigma \rrbracket^i$  is partial recursive by the thesis of Church.  $\square$

From a combination of Lemmas 6.5 and 6.6 we obtain the result that the set of  $\lambda_\Sigma$ -definable  $\Sigma$ -trees is exactly the set of all partial recursive  $\Sigma$ -trees.

**Theorem 6.7.**  $\lambda^c(\Sigma)_{\underline{\text{CT}}_\Sigma} = \text{PR}(\underline{\text{CT}}_\Sigma)$ .

**Proof.** The proof is immediate by Lemmas 6.5 and 6.6.  $\square$

Together with the ‘Mezei–Wright-like’ result (Theorem 5.11), we obtain the desired characterization of the power of  $\lambda_\Sigma$ -schemes: an element out of the carrier of a continuous  $\Sigma$ -algebra is  $\lambda_\Sigma$ -definable iff it is the homomorphic image of a partial recursive  $\Sigma$ -tree.

**Theorem 6.8.**  $\lambda^c(\Sigma)_A = h_A(\text{PR}(\underline{\text{CT}}_\Sigma))$  for all continuous  $\Sigma$ -algebras  $A$ .

**Proof.** The proof is immediate by Theorems 5.11 and 6.7.  $\square$

## 7. Typed lambda-schemes and their translation

In this section we want to inspect, on the level of program schemes, the influence of type restrictions in programming languages on the definability of semantical objects.

One can find in the literature a variety of different classes of program schemes with type restrictions, i.e., recursive equation schemes [57], flowchart schemes [16], combinator schemes [38] and typed lambda-schemes [62, 3, 22, 19].

For the purposes of our investigation we shall concentrate on a most general class of typed schemes, the class of typed lambda-schemes, without giving a translatability result explicitly for the other classes of schemes.

Typed lambda-schemes are built up from typed variables and operation symbols under typed application and abstraction and fixed-point operation on symmetrical types of arbitrary functional depth. This class of program schemes is dealt with in detail by Damm [19].

Here we shall show that there exists an effective translation of typed  $\lambda_{\Sigma}$ -schemes into equivalent type-free  $\lambda_{\Sigma}$ -schemes.

The translation function, as well as the following equivalence proof, have immediate counterparts in [21], where  $n$ -rational schemes are translated into equivalent lambda-schemes.

Making use of this translatability result we can also derive a ‘Mezei–Wright-like’ result for typed lambda-schemes.

Finally, we consider the definability of objects in Scott-domains and obtain the result that on this level typed  $\lambda_{\Sigma}$ -schemes are not translatable into equivalent type-free  $\lambda_{\Sigma}$ -schemes.

In this section,  $I$  denotes again a finite set of base types,  $\Sigma$  a  $D(I)$ -set of operations symbols, but  $X$  an  $\mathcal{F}(I)$ -set of variables such that  $X^{\tau}$  is infinite and denumerable for each  $\tau \in \mathcal{F}(I)$ .

**Definition 7.1.** The  $\mathcal{F}(I)$ -set  $\mathcal{F}\text{-}\lambda(\Sigma)$  of typed  $\lambda_{\Sigma}$ -terms (over  $X$ ) is the least set  $K$  of words over  $\Sigma \cup X \cup \{(\ , \ ), [\ , \ ], \mathbf{Y}\}$  satisfying

- (i)  $\Sigma \subseteq K$  type preserving,
- (ii)  $X \subseteq K$  type preserving,
- (iii)  $t(t_1, \dots, t_r) \in K^{\tau}$  for all  $t \in K^{(\tau_1, \dots, \tau_r, \tau)}$ ,  $t_{\nu} \in K^{\tau_{\nu}}$ ,  $1 \leq \nu \leq r$ ,
- (iv)  $\lambda x_1 \dots x_r [t] \in K^{(\tau_1, \dots, \tau_r, \tau)}$  for all  $t \in K^{\tau}$ ,  $x_{\nu} \in X^{\tau_{\nu}}$ ,  $1 \leq \nu \leq r$ ,
- (v)  $\mathbf{Y}(t) \in K^{\tau}$  for all  $t \in K^{(\tau, \tau)}$ ,  $\tau \in \mathcal{F}(I)$ .

**Remark.** For  $t \in K^{\tau}$ ,  $\lambda [t] \in K^{(e, \tau)}$  holds by (iv) with  $r=0$ , and, for  $t \in K^{(e, \tau)}$ ,  $t(\ ) \in K^{\tau}$  holds by (iii) with  $r=0$ .

**Lemma 7.2.** The partition of  $t$  into subterms is unambiguous.

**Proof.** The proof follows by standard procedures.  $\square$

**Definition 7.3.** Let  $t \in \mathcal{F}\text{-}\lambda(\Sigma)$ . We define the sets of variables  $\text{Fr}(t)$  of free variables in  $t$ ,  $\text{Bnd}(t)$  of bound variables in  $t$ , and  $\text{Var}(t)$  of variables in  $t$  analogously to Definition 3.4 inductively on the structure of  $t$ :

- (i)  $\text{Fr}(f) = \text{Bnd}(f) = \emptyset$  for  $f \in \Sigma$ ,
- (ii)  $\text{Fr}(x) = \{x\}$ ,  $\text{Bnd}(x) = \emptyset$  for  $x \in X$ ,
- (iii)  $\text{Fr}(t_0(t_1, \dots, t_r)) = \bigcup_{0 \leq \nu \leq r} \text{Fr}(t_{\nu})$ ,  
 $\text{Bnd}(t_0(t_1, \dots, t_r)) = \bigcup_{0 \leq \nu \leq r} \text{Bnd}(t_{\nu})$ ,
- (iv)  $\text{Fr}(\lambda x_1 \dots x_r [t]) = \text{Fr}(t) \setminus \{x_1, \dots, x_r\}$ ,  
 $\text{Bnd}(\lambda x_1 \dots x_r [t]) = \text{Bnd}(\lambda x_1 \dots x_r [t]) = \text{Bnd}(t) \cup \{x_1, \dots, x_r\}$ ,
- (v)  $\text{Fr}(\mathbf{Y}(t)) = \text{Fr}(\cdot)$ ,  $\text{Bnd}(\mathbf{Y}(t)) = \text{Bnd}(t)$ ,  $\text{Var}(t) = \text{FR}(t) \cup \text{Bnd}(t)$ .

**Definition 7.4.** Let  $A$  be an  $I$ -set. An  $\mathcal{F}(I)$ -mapping  $\rho: X \rightarrow A^{*(I)}$  is called *typed environment* (with respect to  $A$ ). The set of all environments (with respect to  $A$ ) will be denoted by  $\mathcal{TU}_A$ .

**Definition 7.5.** The semantic function  $\llbracket \cdot, \underline{A} \rrbracket: \mathcal{F}\text{-}\lambda(\Sigma) \rightarrow \mathcal{TU}_A \rightarrow A^{*(I)}$  of typed  $\lambda_\Sigma$ -terms (with respect to the interpretation  $\underline{A} = (A, \alpha)$ ) is defined inductively on the structure of typed  $\lambda_\Sigma$ -terms:

- (i)  $\llbracket f, \underline{A} \rrbracket \rho = \alpha(f)$  for  $f \in \Sigma$ ,
- (ii)  $\llbracket x, \underline{A} \rrbracket \rho = \rho(x)$  for  $x \in X$ ,
- (iii)  $\llbracket t(t_1, \dots, t_r), \underline{A} \rrbracket \rho = \llbracket t, \underline{A} \rrbracket \rho(\llbracket t_1, \underline{A} \rrbracket \rho, \dots, \llbracket t_r, \underline{A} \rrbracket \rho)$ ,
- (iv)  $\llbracket \lambda x_1 \dots x_r [t], \underline{A} \rrbracket \rho = (a_1, \dots, a_r) \mapsto \llbracket t, \underline{A} \rrbracket \rho[x_1/a_1] \dots [x_r/a_r]$ ,
- (v)  $\llbracket Y(t), \underline{A} \rrbracket = \mu a. \llbracket t, \underline{A} \rrbracket \rho(a)$ .

**Convention.** Just as in the untyped case,  $\llbracket \cdot, \underline{A} \rrbracket$  abbreviates to  $\llbracket \cdot \rrbracket$  whenever the interpretation is clear from the context.

**Definition 7.6.** The  $\mathcal{F}(I)$ -set  $\mathcal{F}\text{-}\lambda^c(\Sigma)$  of typed  $\lambda_\Sigma$ -schemes is the set of all closed  $\lambda_\Sigma$ -terms, i.e.,

$$\mathcal{F}\text{-}\lambda^c(\Sigma) = \{t \in \mathcal{F}\text{-}\lambda(\Sigma) \mid \text{Fr}(t) = \emptyset\}.$$

The semantics of typed  $\lambda_\Sigma$ -terms over an interpretation  $\underline{A}$  depends only on the values of the environment on the free variables, in particular the semantics of typed  $\lambda_\Sigma$ -schemes is independent of the environment (cf. Lemma 3.9 for the type-free case).

**Lemma 7.7.** Let  $t \in \mathcal{F}\text{-}\lambda(\Sigma)$ ,  $\underline{A} = (A, \alpha)$  and  $\rho_1, \rho_2 \in \mathcal{TU}_A$ ; then

$$[\forall x \in \text{Fr}(t), \rho_1(x) = \rho_2(x)] \Rightarrow \llbracket t, \underline{A} \rrbracket \rho_1 = \llbracket t, \underline{A} \rrbracket \rho_2 = \llbracket t, \underline{A} \rrbracket \rho_3.$$

In particular, for  $t \in \lambda^c(\Sigma)$ ,  $\llbracket t, \underline{A} \rrbracket \rho_1 = \llbracket t, \underline{A} \rrbracket \rho_2$  holds for all environments  $\rho_1, \rho_2$ .

**Proof.** The proof follows by straightforward induction on the structure of  $t$ .  $\square$

Lemma 7.7 allows us to define the semantics of  $\lambda_\Sigma$ -schemes independently of environments.

**Definition 7.8.** Let  $\underline{A} = (A, \alpha)$  be a continuous  $\Sigma$ -algebra.

The semantic function  $\llbracket \cdot, \underline{A} \rrbracket: \mathcal{F}\text{-}\lambda^c(\Sigma) \rightarrow A^{*(I)}$  of typed  $\lambda_\Sigma$ -schemes (with respect to  $\underline{A}$ ) is defined by  $\llbracket S, \underline{A} \rrbracket := \llbracket S, \underline{A} \rrbracket \rho$  for all  $S \in \mathcal{F}\text{-}\lambda^c(\Sigma)$  with arbitrary environment  $\rho \in \mathcal{TU}_A$ .

Again  $\llbracket \cdot, \underline{A} \rrbracket$  abbreviates to  $\llbracket \cdot \rrbracket$  whenever the interpretation is clear from the context.

Analogously to Definitions 3.12 and 3.13 we define the typed  $\lambda_{\Sigma}$ -definable elements in  $A^{\tau}$  ( $\tau \in \mathcal{F}(I)$ ) and  $A_{\infty}$  and formulate the options of equivalence for typed and type-free  $\lambda_{\Sigma}$ -schemes.

**Definition 7.9.** An element  $a$  in  $A^{\tau}$ ,  $\tau \in \mathcal{F}(I)$ , is *typed  $\lambda_{\Sigma}$ -definable* iff there exists a  $\lambda_{\Sigma}$ -scheme  $S$  of type  $\tau$  with  $\llbracket S, \underline{A} \rrbracket = a$ .

An element  $a$  in  $A_{\infty}$  is *typed ( $\tau$ -)  $\lambda_{\Sigma}$ -definable* iff there exist  $\tau \in \mathcal{F}(I)$  and  $S \in \mathcal{T}\text{-}\lambda^{\circ}(\Sigma)^{\tau}$  with  $\varphi^{\tau}(\llbracket S, \underline{A} \rrbracket) = a$ . The sets of all typed  $\lambda_{\Sigma}$ -definable elements in  $A^{\tau}$  and  $A_{\infty}$  are denoted by  $\mathcal{T}\text{-}\lambda^{\circ}(\Sigma)_{\underline{A}}^{\tau}$  and  $\mathcal{T}\text{-}\lambda^{\circ}(\Sigma)_{A_{\infty}}$  respectively.

**Definition 7.10.** Let  $\tau \in \mathcal{F}(I)$ ,  $S \in \mathcal{T}\text{-}\lambda^{\circ}(\Sigma)^{\tau}$  and  $R \in \lambda^{\circ}(\Sigma)$ .

- $S$  is *equivalent* to  $R$  ( $S =_{\tau} R$ ) iff  $\llbracket S, \underline{A} \rrbracket = \llbracket R, \underline{A} \rrbracket^{\tau}$  for all interpretations  $\underline{A}$ .
- $S$  is  *$\infty$ -equivalent* to  $R$  ( $S =_{\infty} R$ ) iff  $\varphi^{\tau}(\llbracket S, \underline{A} \rrbracket) = \llbracket R, \underline{A} \rrbracket$  for all interpretations  $\underline{A}$ .

The notations for the equivalence relations in Definition 7.10 are not distinguished from those in Definition 3.13. However, it should be clear from the context which of them is meant.

We shall now define the translation function  $\Delta$  which associates in a standard way to each typed  $\lambda_{\Sigma}$ -term a type-free  $\lambda_{\Sigma}$ -term using the Curry isomorphisms and simulation of the fixed-point operator by self-application.

**Definition 7.11.** The translation function  $\Delta : \mathcal{T}\text{-}\lambda(\Sigma) \rightarrow \lambda(\Sigma)$  is defined inductively by

- (i)  $\Delta(f) = f$  for  $f \in \Sigma$ ,
- (ii)  $\Delta(x) = x$  for  $x \in X$ ,
- (iii)  $\Delta(t(t_1, \dots, t_r)) = \Delta(t)\Delta(t_1) \dots \Delta(t_r)$ ,
- (iv)  $\Delta(\lambda x_1 \dots x_r [t]) = \lambda x_1 \dots x_r \Delta(t)$ ,
- (v)  $\Delta(Y(t)) = Y_{\lambda} \Delta(t)$ .

It is clear that in general the semantics of a typed  $\lambda_{\Sigma}$ -term cannot be equal to the semantics of its translation, since different sets of environments are considered. Furthermore, in the definition of the semantics environments are inductively transformed. We therefore have to show the equivalence with respect to all environments  $\bar{\rho}$  which coincide on the retracted cpo's with the typed environment  $\rho$  and which map all variables on ideal values because the proof depends heavily on the idealness of the translated terms (for the notion of idealness, see Definition 2.17).

**Lemma 7.12.** *The translation of a typed  $\lambda_{\Sigma}$ -scheme is a  $\lambda_{\Sigma}$ -scheme, i.e.,  $\Delta(S) \in \lambda^{\circ}_{\delta}(\Sigma)$  for all  $S \in \mathcal{T}\text{-}\lambda^{\circ}(\Sigma)$ .*

**Proof.** The proof is immediate by definition of  $\Delta$ .  $\square$

**Lemma 7.13.** Let  $A = (\underline{A}, \alpha)$  be an interpretation,  $\tau \in \mathcal{F}(I)$ ,  $t \in \mathcal{T}\text{-}\lambda(\Sigma)^\tau$ ,  $\rho \in \mathcal{F}\mathcal{U}_A$  and  $\bar{\rho} \in \mathcal{U}_A$ .

If  $\bar{\rho}(x)$  is  $\sigma$ -ideal and  $\psi^\sigma(\bar{\rho}(x)) = \rho(x)$  for all  $x \in X^\sigma$ ,  $\sigma \in \mathcal{F}(I)$ , then

- (a)  $\llbracket t, \underline{A} \rrbracket \rho = \psi^\tau(\llbracket \Delta(t), \underline{A} \rrbracket \bar{\rho})$ , and  
 (b)  $\llbracket \Delta(t), \underline{A} \rrbracket \bar{\rho}$  is  $\tau$ -ideal.

Part (a) is illustrated by the following diagram:

$$\begin{array}{ccc}
 \mathcal{T}\text{-}\lambda(\Sigma)^\tau & \xrightarrow{\Delta} & \lambda(\Sigma) \\
 \downarrow \llbracket \cdot, \underline{A} \rrbracket \rho & \cong & \downarrow \llbracket \cdot, \underline{A} \rrbracket \rho \\
 A^\tau & \xleftarrow{\psi^\tau} & A
 \end{array}$$

**Proof.** The proof follows by induction on the structure of  $t$ .

Case (i):  $t = f \in \Sigma^\tau$

(a)  $\llbracket f \rrbracket \rho = \alpha(f) = \psi^\tau(\varphi^\tau(\alpha(f)))$  by Lemma 2.15  
 $= \psi^\tau(\llbracket \Delta(f) \rrbracket \bar{\rho})$ .

(b)  $\llbracket \Delta(f) \rrbracket \bar{\rho} = \varphi^\tau(\alpha(f))$   $\tau$ -ideal by Lemma 2.18.

Case (ii):  $t = x \in X^\tau$ . (a) and (b) follow by hypothesis.

Case (iii):  $t = t_0(t_1, \dots, t_r)$

(a)  $\llbracket t \rrbracket \rho = \llbracket t_0 \rrbracket \rho(\llbracket t_1 \rrbracket \rho, \dots, \llbracket t_r \rrbracket \rho)$   
 $= \psi^{(\tau_1, \dots, \tau_r, \tau)}(\llbracket \Delta(t_0) \rrbracket \bar{\rho})(\psi^{\tau_1}(\llbracket \Delta(t_1) \rrbracket \bar{\rho}), \dots, \psi^{\tau_r}(\llbracket \Delta(t_r) \rrbracket \bar{\rho}))$   
 by induction hypothesis  
 $= \psi^\tau(\varphi^{(\tau_1, \dots, \tau_r)}(\llbracket \Delta(t_0) \rrbracket \bar{\rho})(\llbracket \Delta(t_1) \rrbracket \bar{\rho}, \dots, \llbracket \Delta(t_r) \rrbracket \bar{\rho}))$   
 due to the idealness of  $t_\nu$ ,  $0 \leq \nu \leq r$   
 $= \psi^\tau(\llbracket \Delta(t) \rrbracket \bar{\rho})$  by Lemma 5.6.

(b) The  $\tau$ -idealness of  $\llbracket \Delta(t) \rrbracket \bar{\rho}$  immediately follows by idealness of  $t_\nu$ ,  $0 \leq \nu \leq r$ .

Case (iv):  $t = \lambda x_1 \dots x_r [t_0]$  ( $\tau = (\tau_1, \dots, \tau_r, \tau_0)$ ).

Let  $\bar{a} = (a_1, \dots, a_r) \in A^{\tau_1, \dots, \tau_r}$ :

(a)  $\llbracket t \rrbracket \rho(\bar{a}) = \llbracket t_0 \rrbracket \rho[x_1/a_1] \dots [x_r/a_r]$   
 $= \psi^{\tau_0}(\llbracket \Delta(t_0) \rrbracket \bar{\rho}[x_1/\varphi^{\tau_1}(a_1)] \dots [x_r/\varphi^{\tau_r}(a_r)])$   
 by induction hypothesis

$$= \psi^{\tau_0}(\varphi^{[r]}(\llbracket \lambda x_1 \dots x_r \Delta(t_0) \rrbracket \bar{\rho}))(\varphi^{\tau_1}(a_1), \dots, \varphi^{\tau_r}(a_r))$$

by Lemmas 5.6 and 2.11

$$= \psi^{\tau}(\llbracket \Delta(t) \rrbracket \bar{\rho})(a_1, \dots, a_r) \text{ by definition of } \psi^{\tau} \text{ and Lemma 2.15.}$$

(b) It remains to show, for  $\tau_\nu$ -ideal  $a_\nu$ , ( $1 \leq \nu \leq r$ ),

$$(1) \quad \psi^{\tau_0}(\varphi^{[r]}(\llbracket \Delta(t) \rrbracket \bar{\rho})(a_1, \dots, a_r))f = \psi^{\tau}(\llbracket \Delta(t) \rrbracket \bar{\rho})(\psi^{\tau_1}(a_1), \dots, \psi^{\tau_r}(a_r)),$$

$$(2) \quad \varphi^{[r]}(\llbracket \Delta(t) \rrbracket \bar{\rho})(a_1, \dots, a_r) \text{ is } \tau_0\text{-ideal.}$$

$$\begin{aligned} (1) \quad \psi^{\tau_0}(\varphi^{[r]}(\llbracket \Delta(t) \rrbracket \bar{\rho})(a_1, \dots, a_r)) &= \\ &= \psi^{\tau_0}(\llbracket \Delta(t_0) \rrbracket \bar{\rho}[x_1/a_1] \dots [x_r/a_r]) \\ &= \llbracket t_0 \rrbracket \rho[x_1/\psi^{\tau_1}(a_1)] \dots [x_r/\psi^{\tau_r}(a_r)] \text{ by induction hypothesis} \\ &= \psi^{\tau}(\llbracket \Delta(t) \rrbracket \bar{\rho})(\psi^{\tau_1}(a_1), \dots, \psi^{\tau_r}(a_r)) \text{ using similar} \\ &\hspace{15em} \text{conversion as in (a).} \end{aligned}$$

$$(2) \quad \varphi^{[r]}(\llbracket \Delta(t) \rrbracket \bar{\rho})(a_1, \dots, a_r) = \llbracket \Delta(t_0) \rrbracket \bar{\rho}[x_1/a_1] \dots [x_r/a_r],$$

this is  $\tau_0$ -ideal by induction hypothesis.

Case (v):  $t = Y(S)$

$$\begin{aligned} (a) \quad \llbracket t \rrbracket \rho &= \mu a. \llbracket S \rrbracket \rho(a) \\ &= \mu a. \psi^{[\tau, \tau]}(\llbracket \Delta(S) \rrbracket \bar{\rho})(a) \text{ by induction hypothesis} \\ &= \psi^{\tau}(\mu a. \varphi(\llbracket \Delta(S) \rrbracket \bar{\rho})(a)) \text{ by Lemma 2.19} \\ &= \psi^{\tau}(\llbracket Y_\lambda \Delta(S) \rrbracket \bar{\rho}) \text{ by Theorem 4.7} \\ &= \psi^{\tau}(\llbracket \Delta(t) \rrbracket \bar{\rho}). \end{aligned}$$

$$\begin{aligned} (b) \quad \llbracket \Delta(t) \rrbracket \bar{\rho} &= \llbracket Y_\lambda \Delta(S) \rrbracket \bar{\rho} = \mu a. \varphi(\llbracket \Delta(S) \rrbracket \bar{\rho})(a) \text{ by Theorem 4.7} \\ &= \bigsqcup_{n \in \mathbb{N}} \varphi(\llbracket \Delta(S) \rrbracket \bar{\rho})^n(\perp) \text{ by Theorem 1.5.} \end{aligned}$$

This is  $\tau$ -ideal by Lemma 2.20, since  $\llbracket \Delta(S) \rrbracket \bar{\rho}$  is  $(\tau, \tau)$ -ideal by induction hypothesis and  $\perp = \varphi^{\tau}(\perp)$  is  $\tau$ -ideal by Lemma 2.18.  $\square$

**Remark** (on the proof of the above lemma). The relation  $\llbracket t, A \rrbracket \rho \leq \psi^{\tau}(\llbracket \Delta(t), A \rrbracket \bar{\rho})$  is straightforward, but for the opposite relation a careful examination of translated terms is necessary because the semantics of the translation of a typed term is in general greater than the semantics of this typed term. An extension of the type-free  $\lambda$ -calculus by introduction of typed variables which will obtain injected meanings in all environments simplifies the proof of the corresponding, yet weaker, result (see [22]).

Now we can show the translatability of typed  $\lambda_{\Sigma}$ -schemes in equivalent type-free  $\lambda_{\Sigma}$ -schemes.

**Theorem 7.14.**  $\mathcal{T}\text{-}\lambda^c(\Sigma)_{\mathcal{A}}^{\tau} \subseteq \lambda^c(\Sigma)_{\mathcal{A}}^{\tau}$  for all types  $\tau \in \mathcal{F}(I)$  and all interpretations  $\mathcal{A}$ .

**Proof.** Take  $a \in \mathcal{T}\text{-}\lambda^c(\Sigma)_{\mathcal{A}}^{\tau}$ . Then there exists an  $S \in \mathcal{T}\text{-}\lambda^c(\Sigma)$  such that  $a = \llbracket S, \mathcal{A} \rrbracket$ . Choose an arbitrary environment  $\rho \in \mathcal{T}\mathcal{U}_{\mathcal{A}}$ . Then  $a = \llbracket S, \mathcal{A} \rrbracket \rho$  by Definition 7.6. Define  $\bar{\rho} \in \mathcal{U}_{\mathcal{A}}$  by  $x \mapsto \varphi^{\sigma}(\rho(x))$  for all  $x \in X^{\sigma}$ ,  $\sigma \in \mathcal{F}(I)$ .

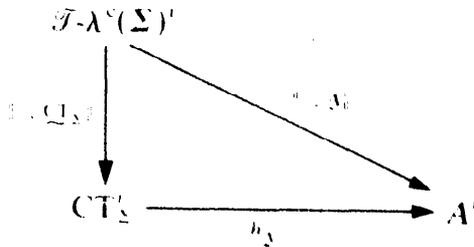
Now

$$\begin{aligned} a &= \psi^{\tau}(\llbracket \Delta(S), \mathcal{A} \rrbracket \bar{\rho}) \quad \text{by Lemma 7.13} \\ &= \llbracket \Delta(S), \mathcal{A} \rrbracket^{\tau} \quad \text{by Lemma 7.12,} \end{aligned}$$

but then  $a \in \lambda^c(\Sigma)_{\mathcal{A}}^{\tau}$  holds.  $\square$

From Theorems 7.14 and 5.11 we obtain the ‘Mezei–Wright-like’ result also for typed  $\lambda_{\Sigma}$ -schemes, a theorem which was shown directly by Damm [19].

**Theorem 7.15.** *The diagram*



*commutes for all interpretations  $\mathcal{A}$  and all  $i \in I$ .*

**Proof.** Let  $S \in \mathcal{T}\text{-}\lambda^c(\Sigma)^i$ :

$$\begin{aligned} \text{h}_{\mathcal{A}}(\llbracket S, \text{CT}_{\Sigma} \rrbracket) &= \text{h}_{\mathcal{A}}(\llbracket \Delta(S), \text{CT}_{\Sigma} \rrbracket^i) \quad \text{by Theorem 7.14} \\ &= \llbracket \Delta(S), \mathcal{A} \rrbracket^i \quad \text{by Theorem 5.11} \\ &= \llbracket S, \mathcal{A} \rrbracket \quad \text{by Theorem 7.14.} \quad \square \end{aligned}$$

The following example is an application of the ‘Mezei–Wright-like’ result, where the equivalence of an applicative (type-free) procedure and a recursive (typed) procedure, both for computing the factorial function, is demonstrated.

**Example 7.16.** Consider the ‘applicative’ ALGOL-60 Procedure  $P$  (from the

Introduction):

**integer procedure**  $P(f, y)$ ; **integer procedure**  $f$ ; **integer**  $y$ ;  
 $P :=$  **if**  $y = 0$  **then**  $1$  **else**  $y * f(f, y - 1)$ .

A procedure call of the form  $P(P, k)$  will deliver the value  $k!$  as result.  
 The  $\lambda_{\Sigma}$ -scheme which was abstracted from  $P$  reads as follows:

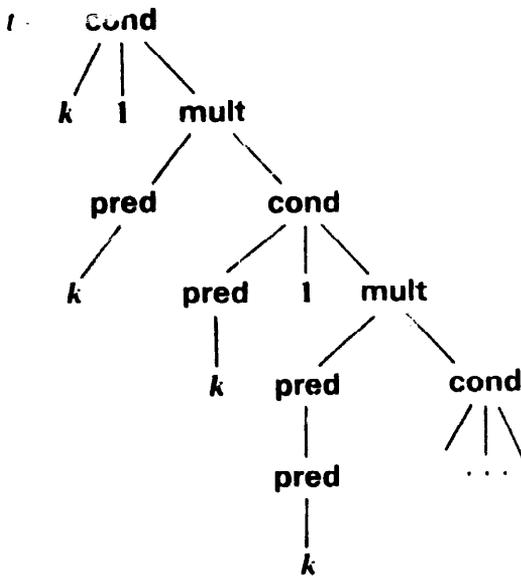
$$P_{\lambda} = (\Upsilon_{\lambda} \lambda f y.(((\mathbf{cond} \ y) \mathbf{1})(\mathbf{mult} \ y)((f \ f)(\mathbf{pred} \ y))))),$$

where

$$\mathbf{cond} \in \Sigma^{(m, 1)}, \quad \mathbf{mult} \in \Sigma^{(n, 1)}, \quad \mathbf{pred} \in \Sigma^{(1, 1)},$$

$$\mathbf{1}, k \in \Sigma^{(0, 1)} \quad \text{and} \quad f, y \in X.$$

The infinite  $\Sigma$ -tree  $t = \llbracket (P_{\lambda} P_{\lambda})k, \underline{\text{CT}}_{\Sigma} \rrbracket^t$ , which corresponds to the above procedure call, can now be evaluated using Theorem 4.1 and Lemma 5.8:



For the typed case we consider the ALGOL-68 procedure  $R$ :

**proc**  $R = (\text{int } y)\text{int}$ ;  
**begin**  
     **if**  $y = 0$  **then**  $1$  **else**  $y * R(y - 1)$   
**end.**

From  $R$  we abstract the following typed  $\lambda_{\Sigma}$ -scheme:

$$R_{\lambda} = Y(\lambda r y [\mathbf{cond}(y, \mathbf{1}, \mathbf{mult}(y, r(\mathbf{pred}(y))))]).$$

Making use of the fixed-point property we can evaluate the infinite tree  $\llbracket R_{\lambda}(k), \underline{\mathbf{CT}}_{\Sigma} \rrbracket$ , which corresponds to a procedure call of  $R$  with actual parameter  $k$ , and realize that the result is equal to  $t$ .

The 'Mezei-Wright-like' result, proved above, now guarantees that, for each interpretation  $\mathcal{A}$ ,

$$\llbracket (P_{\lambda} P_{\lambda})k, \mathcal{A} \rrbracket^i = \llbracket R_{\lambda}(k), \mathcal{A} \rrbracket.$$

In particular, we derive that in an ALGOL-60 program with procedure declaration  $P$  the call  $P(P, k)$  is equivalent to the call  $R(k)$  in an ALGOL-68 program with procedure declaration  $R$ .

The next theorem states the non-translatability of typed  $\lambda_{\Sigma}$ -schemes into  $\infty$ -equivalent type-free  $\lambda_{\Sigma}$ -schemes.

**Theorem 7.17.** *There exists a typed  $\lambda$ -scheme  $S$  such that, for all  $\lambda$ -schemes  $R$ ,  $S \neq_{\infty} R$  holds, i.e.,*

$$\mathcal{T}\text{-}\lambda^c(\Sigma)_{\mathcal{A}} \not\subseteq \lambda^c(\Sigma)_{\mathcal{A}}.$$

(cf. Definitions 3.12, 3.13, 7.9 and 7.11).

**Proof.** Let  $S = \lambda x^0 . x^0 \in \mathcal{T}\text{-}\lambda^c(\Sigma)^1$ :

$$\varphi^1(\llbracket \lambda x^0 . x^0, \mathcal{A} \rrbracket) = \psi(\varphi^0 \circ \psi^0) \quad \text{by retraction properties.}$$

This is not  $\lambda$ -definable by Theorem 4.9.  $\square$

This theorem formalizes the difficulties which arise in the proof of Lemma 7.13.

## 8. Lambda-definability of formal languages

In this section we analyse the typed and type-free lambda-definable formal languages with respect to the interpretation  $\mathcal{P}(\Sigma^*)$  with the constant  $\epsilon$  for the empty word, monadic prefix-operation for each  $a \in \Sigma$ , and a binary operation  $+$ , for the union.

We shall show that in the class of typed lambda-definable languages the emptiness problem is decidable, whereas in the type-free case all recursively enumerable languages are definable.

Hence, on the level of control structure of programming languages, type-free application and abstraction is strictly more powerful than typed application and abstraction plus recursion.

For the definability in the limit domains  $A_\infty$  we obtain the result that on this level the two classes of program schemes, typed and type-free are incomparable.

In this section, let  $\Sigma = \Sigma^{(i,i)}$  denote a finite set of monadic symbols for some  $i \in I$  and define  $\Sigma_L := \Sigma \cup \{+^{(i,i)}, e^{(e,i)}\}$ .

**Definition 8.1.** The  $\Sigma_L$ -algebra  $\underline{\mathcal{P}(\Sigma^*)}$  of formal languages over  $\Sigma$  is defined by  $\underline{\mathcal{P}(\Sigma^*)} = (\mathcal{P}(\Sigma^*), \gamma)$  where

- (i)  $\gamma(a)(L) = \{aw \mid w \in L\}$  for all  $a \in \Sigma, L \in \mathcal{P}(\Sigma^*),$
- (ii)  $\gamma(e)(L) = \{e\},$
- (iii)  $\gamma(+)(L_1, L_2) = L_1 \cup L_2$  for all  $L_1, L_2 \in \mathcal{P}(\Sigma^*).$

**Lemma 8.2.**  $\underline{\mathcal{P}(\Sigma^*)}$  is a continuous  $\Sigma_L$ -algebra.

**Proof.** The proof follows by standard procedures.  $\square$

The next theorem states the decidability of the emptiness problem for typed  $\lambda_{\Sigma_L}$ -definable languages.

**Theorem 8.3.** Let  $S \in \mathcal{F}\text{-}\lambda^c(\Sigma_L)^i$ . It is decidable whether  $\llbracket S, \underline{\mathcal{P}(\Sigma^*)} \rrbracket = \emptyset$ .

**Proof.** Let  $\underline{\mathcal{B}} = (\{t, f\}, \beta)$  be the  $\Sigma_L$ -algebra defined by

- (i)  $\beta(a) = \text{id}$  for each  $a \in \Sigma,$
- (ii)  $\beta(e)(L) = f,$
- (iii)  $\beta(+)(b_1, b_2) = \sqcup \{b_1, b_2\}.$

$\underline{\mathcal{B}}$  is continuous with respect to the ordering  $\leq$  given by

$$b_1 \leq b_2 \Leftrightarrow (b_1 = b_2) \vee (b_1 = t \text{ and } b_2 = f),$$

i.e.,



Let **empty**:  $\underline{\mathcal{P}(\Sigma^*)} \rightarrow \underline{\mathcal{B}}$  be defined by

$$L \mapsto \begin{cases} t & \text{if } L = \emptyset, \\ f & \text{if } L \neq \emptyset. \end{cases}$$

It is easy to verify that **empty** is a strict continuous  $\Sigma_L$ -homomorphism.

Observe that for each  $\tau \in \mathcal{F}(\{i\})$  the set  $\{t, f\}^\tau$  is finite. Let  $S \in \mathcal{F}\text{-}\lambda(\Sigma_L)$  be a typed term and an effective  $\rho \in \mathcal{F}\mathcal{U}_{\{t, f\}}$ , i.e.,  $\rho(x)$  is computable for all  $x \in X$ .

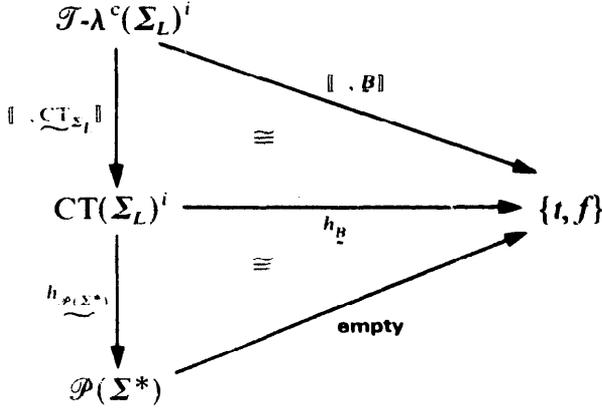
(\*)  $\llbracket S, \underline{\mathcal{B}} \rrbracket \rho$  is recursive. We prove (\*) by induction on the structure of typed  $\lambda_{\Sigma_L}$ -terms.

- (i)  $S = a \in \Sigma$ :  $\llbracket a, \underline{\mathcal{B}} \rrbracket \rho = \text{id}$  is recursive.
- (ii)  $S = x$ :  $\llbracket x, \underline{\mathcal{B}} \rrbracket \rho = \rho(x)$  is recursive by hypothesis.
- (iii)  $S = t_0(t_1, \dots, t_r)$ :  $\llbracket S \rrbracket \rho = \llbracket t_0 \rrbracket \rho(\llbracket t_1 \rrbracket \rho, \dots, \llbracket t_r \rrbracket \rho)$  is recursive.

(iv)  $S = \lambda x_1 \dots x_n . t$ :  $\llbracket S \rrbracket \rho$  is recursive.

(v)  $S = Y(t)$ :  $\llbracket S \rrbracket \rho = \mu a . \llbracket t \rrbracket \rho(a)$  is recursive by induction hypothesis.

By the ‘Mezei–Wright-like’ result (Theorem 7.15) the assertion follows:



$L = \llbracket S, \mathcal{P}(\Sigma^*) \rrbracket$  is empty iff  $\mathbf{empty}(L) = t$  iff  $\llbracket S, \{t, f\} \rrbracket = t$  and this is decidable by (\*).  $\square$

For the characterization of type-free  $\lambda_{\Sigma_L}$ -definable languages we first realize that the recursively enumerable languages over  $\Sigma$  ( $\mathbf{RE}(\Sigma)$ ) are exactly the homomorphic image of the class of partial recursive  $\Sigma_L$ -trees.

**Lemma 8.4.**  $\mathbf{RE}(\Sigma) = h_{\rho(\Sigma^*)}(\mathbf{PR}(\mathbf{CT}_{\Sigma_L}))$ .

**Proof.** For the proof, see Fehr [26a].

**Theorem 8.5.** *The  $\lambda_{\Sigma_L}$ -definable languages are exactly the recursively enumerable languages over  $\Sigma$ , i.e.,*

$$\lambda^c(\Sigma_L)_{\rho(\Sigma^*)} = \mathbf{RE}(\Sigma).$$

**Proof**

$$\begin{aligned} \lambda^c(\Sigma_L)_{\rho(\Sigma^*)} &= h_{\rho(\Sigma^*)}(\mathbf{PR}(\mathbf{CT}_{\Sigma_L})) && \text{by Theorem 6.8} \\ &= \mathbf{RE}(\Sigma) && \text{by Lemma 8.4. } \square \end{aligned}$$

The next theorem states that the class of the semantics of  $\lambda_{\Sigma_L}$ -schemes is not contained in the class of the semantics of typed  $\lambda_{\Sigma_L}$ -schemes, i.e., the converse of Theorem 7.14 does not hold.

**Theorem 8.6.**  $\lambda^c(\Sigma)_{\mathcal{A}} \not\subseteq \mathcal{T}\text{-}\lambda^c(\Sigma)_{\mathcal{A}}^{\tau}$  for some set of operation symbols  $\Sigma$ , a continuous  $\Sigma$ -algebra  $\mathcal{A}$  and a type  $\tau \in \mathcal{F}(I)$ .

**Proof.** Choose  $\mathcal{A} = \mathcal{P}(\Sigma^*)$  as above and  $\tau = i \in I$ . Let  $L \in \text{RE}(\Sigma)$  be an arbitrary recursively enumerable language. By Theorem 8.5 there exists a  $\lambda_{\Sigma_L}$ -scheme  $S$  with  $\llbracket S, \mathcal{P}(\Sigma^*) \rrbracket^i = L$ . Since the emptiness problem is unsolvable for r.e. languages, we can deduce from Theorem 8.3 that there cannot in general exist a typed  $\lambda_{\Sigma_L}$ -scheme which is equivalent to  $S$ .  $\square$

Now we derive that type-free and typed  $\lambda_{\Sigma_L}$ -definability in  $A_{\infty}$  is not comparable.

**Theorem 8.7.** *There exists an interpretation  $\mathcal{A}$  such that*

$$\mathcal{T}\text{-}\lambda^c(\Sigma)_{\mathcal{A}_1} \not\subseteq \lambda^c(\Sigma)_{\mathcal{A}_2} \quad \text{and} \quad \lambda^c(\Sigma)_{\mathcal{A}_2} \not\subseteq \mathcal{T}\text{-}\lambda^c(\Sigma)_{\mathcal{A}_1}$$

**Proof.** For  $\mathcal{P}(\Sigma^*)$  as interpretation of  $\lambda_{\Sigma_L}$ -schemes,

$$\mathcal{T}\text{-}\lambda^c(\Sigma_L)_{\mathcal{P}(\Sigma^*)_1} \not\subseteq \lambda^c(\Sigma_L)_{\mathcal{P}(\Sigma^*)_2} \quad \text{by Theorem 7.17,}$$

and by Theorem 8.6 the converse also holds.  $\square$

However, on the level of control structures of programming languages, the class of type-free lambda-schemes is strictly more powerful than the class of typed lambda-schemes.

**Theorem 8.8.**  $\mathcal{T}\text{-}\lambda^c(\Sigma)_{\mathcal{A}}^{\tau} \subseteq \lambda^c(\Sigma)_{\mathcal{A}}$  for some interpretation  $\mathcal{A}$  and some type  $\tau \in \mathcal{F}(I)$ .

**Proof.** The proof is immediate by Theorems 7.14 and 8.6.  $\square$

### Acknowledgment

I would like to thank Prof. Indermark for many fruitful ideas and his support in writing this paper. Special thanks are due to my colleague, Werner Damm, for his enthusiastic participation in discussions on the subject and for his many constructive comments.

### References

- [1] S.K. Abdali, A lambda-calculus model of programming languages I, II, *J. Comput. Languages* 1 (1976) 287-320.
- [2] A.M. Addyman et al., A draft description of PASCAL, *Software-Practice and Experience* 9 (1979).
- [3] J.W. de Bakker, Least fixed points revisited, in:  *$\lambda$ -Calculus and Computer Science Theory*, Lecture Notes in Computer Science 37 (Springer, Berlin, 1975).

- [4] J.W. de Bakker and W.P. de Roever, A calculus for recursive program schemes, in: M. Nivat, ed., *Automata, Languages and Programming* (North-Holland, Amsterdam, 1972).
- [5] H.P. Barendregt, Some extensional term models for combinatory logics and  $\lambda$ -calculi, Ph.D. Thesis, Amsterdam, 1971.
- [6] H.P. Barendregt, A global representation of the recursive functions in the  $\lambda$ -calculus, *Theoret. Comput. Sci.* **3** (1976) 225–242.
- [7] H.P. Barendregt, The type free lambda calculus, in: J. Barwise, ed., *Handbook of Mathematical Logic* (North-Holland, Amsterdam, 1977).
- [8] G. Berry, Stable models of typed  $\lambda$ -calculi, *5th ICALP*, Lecture Notes in Computer Science **62** (Springer, Berlin, 1978).
- [9] G. Berry, Séquentialité de l'évaluation formelle des  $\lambda$ -expressions, in: Robinet, ed., *Program Transformations, Proc. 3rd Internat. Coll. on Programming* (Dunod, Paris, 1978).
- [10] C. Böhm, The CUCH as a formal and description language, in: L. Steel, ed., *Formal Language Description Language for Computer Programming* (North-Holland, Amsterdam, 1966).
- [11] C. Böhm, Alcune proprietà della forma  $\beta$ - $\eta$  normali del  $\lambda$ -calcolo, *Publ. IAC-CNR* **696** (1968).
- [12] C. Böhm and M. Dezani-Ciancaglini, Combinatorial problems, combinator equations and normal forms, in: *Automata Languages and Programming*, Lecture Notes in Computer Science **14** (Springer, Berlin, 1974).
- [13] C. Böhm and M. Dezani-Ciancaglini,  $\lambda$ -terms as total or partial functions on normal forms, in:  *$\lambda$ -Calculus and Computer Science Theory*, Lecture Notes in Computer Science **37** (Springer, Berlin, 1975).
- [14] R.M. Burstall and P.J. Landin, Programs and their proofs: An algebraic approach, in: Meltzer and Michie, eds., *Machine Intelligence 4* (Edinburgh University Press, 1969).
- [15] A. Church, *The Calculi of Lambda-Conversion* (Princeton University Press, 1941).
- [16] R.L. Constable and D. Gries, On classes of program schemata, *SIAM J. Comput.* **1** (1) (1972).
- [17] H.B. Curry and R. Feys, *Combinatory Logic, Vol. I* (North-Holland, Amsterdam, 1958).
- [18] H.B. Curry, R. Hindley and J. Seldin, *Combinatory Logic, Vol. II* (North-Holland, Amsterdam, 1972).
- [19] W. Damm, Die OI-Hierarchie formaler Sprachen als Semantik rekursiver Prozeduren höheren Typs, Ph. D. Thesis, RWTH Aachen, 1980.
- [20] W. Damm and E. Fehr, On the power of self-application and higher type recursion, *5th ICALP*, Lecture Notes in Computer Science **62** (Springer, Berlin, 1978).
- [21] W. Damm, E. Fehr and K. Indermark, Higher type recursion and self-application as control structures, in: E. Neuhold, ed., *IFIP Working Conf. on Formal Description of Programming Concepts* (North-Holland, Amsterdam, 1977).
- [22] H. Egli, Typed meaning in Scott's  $\lambda$ -calculus models, *Proc. 1975 Symp. on  $\lambda$ -Calculus*, Lecture Notes in Computer Science **37** (Springer, Berlin 1975) pp. 220–239.
- [23] J. Engelfriet, *Simple Program Schemes and Formal Languages*, Lecture Notes in Computer Science **20** (Springer, Berlin, 1974).
- [24] J. Engelfriet and E.M. Schmidt, IO and OI, Datalogisk Afdelning Report, DAIMI PB-47, Aarhus University, 1975.
- [25] E. Fehr, Eine universelle Lambda-Kalkül-Programmiersprache und ihr Interpreter, *Informatik-Berichte Universität Bonn* **6** (1975).
- [26] E. Fehr, On typed and untyped  $\lambda$ -schemes, *Schriften zur Informatik und Angewandten Mathematik RWTH Aachen* **44** (1978).
- [26a] E. Fehr, Lambda-Kalkül als Kontrollstruktur von Programmiersprachen, Ph. D. Thesis, RWTH Aachen, 1979.
- [27] M.J. Fischer, Grammars with macro-like productions, in: *Proc. 9th IEEE Conf. on Switching and Automata Theory* (1968) pp. 131–142.
- [28] M.J. Fischer, Lambda calculus schemata, in: *ACM Conf. on Proving Assertions About Programs* (1972) pp. 104–109.
- [29] J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright, Initial algebra semantics and continuous algebras, *J. ACM* **24** (1) (1977) 68–95.
- [30] M. Gordon, Operational reasoning and denotational semantics, Memo AIM-264, Computer Science Dept., Stanford Univ., 1975.
- [31] S. Greibach, *Theory of Program Structures: Schemes, Semantics, Verification*, Lecture Notes in Computer Science **36** (Springer, Berlin, 1975).

- [32] K. Heidler, H. Hermes and F.-K. Mahn, *Rekursive Funktionen* (Bibliographisches Institut, Mannheim, 1977).
- [32a] H. Hermes, *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit* (Springer, Berlin, 1978).
- [33] J.R. Hindley, B. Lercher and J.P. Seldin, *Introduction to Combinatory Logic* (Cambridge University Press, 1972).
- [34] C.A.R. Hoare, Notes on data structuring, in: O.-J. Dahl, F.W. Dijkstra and C.A.R. Hoare, eds., *Structured Programming* (Academic Press, New York, 1972).
- [35] J.E. Hopcroft and J.D. Ullman, *Formal Languages and Their Relation to Automata* (Addison-Wesley, Reading, MA, 1969).
- [36] J.M.E. Hylands, A survey of some useful partial order relations on terms of the lambda-calculus, in:  *$\lambda$ -Calculus and Computer Science Theory*, Lecture Notes in Computer Science 37 (Springer, Berlin, 1975).
- [37] K. Indermark, Control structures and monadic languages, in: *Proc. Automata Theory and Formal Languages*, Lecture Notes in Computer Science 33 (Springer, Berlin, 1975).
- [38] K. Indermark, Schemes with recursion on higher types, *Proc. 5th Conf. on Math. Foundations of Computer Science*, Lecture Notes in Computer Science 45 (Springer, Berlin 1976) pp. 352-358.
- [39] D. Kfoury, Comparing algebraic structures up to algorithmic equivalence, in: M. Nivat, ed., *Automata, Languages and Programming* (North-Holland, Amsterdam, 1972).
- [40] P.J. Landin, A correspondence between ALGOL 60 and Church's lambda notation, *Comm. ACM* 8 (1965).
- [41] H. Langmaack, On procedures as open subroutines, I, II, *Acta Informatica* 2 (1973); 3 (1974).
- [42] H. Ledgard, Ten mini languages: A study of topical issues in programming languages, *Computing Surveys* 3 (3) (1971).
- [43] J.-J. Levy, An algebraic interpretation of the  $\lambda\beta K$ -calculus,  *$\lambda$ -Calculus and Computer Science Theory*, Lecture Notes in Computer Science 37 (Springer Berlin, 1975) pp. 147-165.
- [44] J.-J. Levy, Réductions correctes et optimales dans le  $\lambda$ -calcul, Thèse de Doctorat d'État, Université Paris VII, 1977.
- [45] C.H. Lindsey and S.G. van der Meulen, *An Informal Introduction to ALGOL 68* (North-Holland, Amsterdam, 1977).
- [46] Z. Manna and J. Vuillemin, Fixpoint approach to the theory of computation, in: M. Nivat, ed., *Automata, Languages and Programming* (North-Holland, Amsterdam, 1972).
- [47] J. Mezei and J.B. Wright, Algebra automata and context-free sets, *Information and Control* 11 (1967) 3-29.
- [48] R. Milne and C. Strachey, *A Theory of Programming Languages Semantics, Parts A and B* (Chapman & Hall, London, 1976).
- [49] R. Milner, Models of LCF, Memo AIM-186, Stanford University, 1973.
- [50] R. Milner, Fully abstract models of typed  $\lambda$ -calculus, *Theoret. Comput. Sci.* 4 (1) (1977) 1-22.
- [51] G. Mitschke,  $\lambda$ -definierbare Funktionen auf Peano-Algebren, *Archiv Math. Logik* 15 (1972) 31-35.
- [52] G. Mitschke, The standardization theorem for  $\lambda$ -calculus, *Z. Math. Logik und Grundlagen der Mathematik* 25 (1) (1979).
- [53] J.M. Morris, Lambda calculus models of programming languages, Ph. D. Thesis, M.I.T., 1968.
- [54] R. Nakajima, Infinite normal forms for the lambda-calculus and semantics of programming languages, Ph. D. Thesis, Univ of California, Berkeley, 1975.
- [55] R. Nakajima, Infinite normal forms for the  $\lambda$ -calculus,  *$\lambda$ -Calculus and Computer Science Theory*, Lecture Notes in Computer Science 37 (Springer, Berlin, 1975).
- [56] M. Nivat, On the interpretation of recursive polyadic program schemes, *Symposia Mathematica* 15 (1975).
- [57] M. Nivat, On the interpretation of recursive program schemes, *Symposia Matematica, Atti del Convegno d'Informatica Teorica*, Rome, 1972.
- [58] M. Nivat, Languages algébriques sur le magma libre et sémantique des schemas de programme, in: M. Nivat, ed., *Automata, Languages and Programming* (North-Holland, Amsterdam, 1972) pp. 293-307.
- [59] D. Park, The Y-combinator in Scott's  $\lambda$ -calculus models, *Symp. on Theory of Programming*, University of Warwick, 1970.

- [60] M.S. Paterson and C.E. Hewitt, Comparative schematology, *Conf. Record of Project MAC Conf. on Concurrent Systems and Parallel Computations*, ACM, 1970.
- [61] G.D. Plotkin, Lambda-definability and logical relations, Memorandum SAI-RM-4, Univ. of Edinburgh, 1973.
- [62] G.D. Plotkin, LCF considered as a programming language, *Theoret. Comput. Sci.* **5** (3) (1977) 223–255.
- [63] G.D. Plotkin,  $H^\omega$  as universal domain, *J. CSS* **17** (1978).
- [64] G.D. Plotkin, Personal correspondence, 1979.
- [65] P. Raulefs, Standard models of the otyped lambda-calculus, Interner Bericht Nr. 3, Universität Karlsruhe, 1975.
- [66] H. Rogers, *Theory of Recursive Functions and Effective Computability* (McGraw-Hill, New York, 1967).
- [67] D. Scott, Continuous lattices, in: *Proc. Dalhousie Conf.*, Lecture Notes in Mathematics **274** (Springer, Berlin, 1972) pp. 97–134.
- [68] D. Scott, Data types as lattices, *SIAM J. Comput.* **5** (3) (1976).
- [69] D. Scott and C. Strachey, Towards a mathematical semantics for computer languages, *Proc. Symp. on Computer and Automata*, New York, 1971.
- [70] J.P. Strait, A.B. Mickel and J.T. Easton, PASCAL 6000, Release 3, Univ. of Minnesota, 1979.
- [71] J. Vuillemin, Proof techniques for recursive programs, Ph. D. Thesis, Comput. Sci. Dept., Stanford Univ., 1973.
- [72] C. Wadsworth, Semantics and pragmatics of the lambda-calculus, Ph. D. Thesis, Oxford Univ., 1971.
- [73] C. Wadsworth, The relation between computational and denotational properties for Scott's  $D_\perp$ -models of the lambda-calculus, *SIAM J. Comput.* **5** (3) (1976) 488–520.
- [74] C. Wadsworth, Approximate reduction and lambda calculus models, *SIAM J. Comput.* **7** (3) (1978) 337–356.
- [75] P.H. Welch, Continuous semantics and inside-out-reductions,  *$\lambda$ -Calculus and Computer Science Theory*, Lecture Notes in Computer Science **37** (Springer, Berlin, 1975).
- [76] N. Wirth, The programming language PASCAL, *Acta Informatica* **1** (1) (1971).