Science of Computer Programming 12 (1989) 207-253
North-Holland

DENOTATIONAL ENGINEERING*

Andrzej BLIKLE

Institute of Computer Science, Polish Academy of Sciences, PKiN, P.O. Box 22, 00-901 Warsaw, Poland

Communicated by C.B. Jones
Received August 1988
Revised February 1989

Abstract. This paper is devoted to the methodology of using denotational techniques in software design. Since denotations describe the essential components comprising a system and syntax provides ways for the user to access and communicate with these components, we suggest that denotations be developed in the first place and that syntax be derived from them later. That viewpoint is opposite to the traditional (descriptive) style where denotational techniques are used in assigning a meaning to some earlier defined syntax. Our methodology is discussed on an algebraic ground where both denotations and syntax constitute many-sorted algebras and where denotational semantics is a homomorphism between them. On that ground the construction of a denotational model of a software system may be regarded as a derivation of a sequence of algebras. We discuss some mathematical techniques which may support that process especially this part where syntax is derived from denotations. The suggested methodology is illustrated on two small examples.

The authors have the peculiar idea that domains of our concepts can be quite rigorously laid out before we make the final choice of the language in which we are going to describe these concepts. (...) What we suggest is that in order to sort out your ideas, you put your domains on the table first.

Scott and Strachey [28]

1. Introduction

Denotational semantics is most frequently understood as a method of assigning meaning to syntax. It is implicit in such an understanding that syntax comes first

* Research reported in this paper contributes to project MetaSoft. Sections 1 to 7 are extensively revised versions of the corresponding sections in earlier papers [7, 8]. The approach to the representation of algebras by grammars has been significantly simplified.

into the play and that semantics is assigned to it later. In classical textbooks on denotational semantics (such as Stoy [29] or Gordon [17]) or in the monographs devoted to applications (e.g. Bjørner and Oest [4] or Bjørner and Jones [3]) the construction of a denotational model of a software system is regarded as a four-step process:

- first we describe a concrete syntax of the system,
- then we derive a corresponding abstract syntax,
- next we define the domains of denotations,
- finally we assign denotations to syntax.

The way in which that process is organized is typical to the case where denotational techniques are used in formalizing the definitions of existing programming languages. In that case concrete syntax is always given ahead and what remains to be done is to formally define the semantics.

Giving formal definitions to existing programming languages was the first practical problem tackled on the ground of denotational semantics. Since the early experiment with ALGOL-60 (Mosses [24]), many programming languages have been formalized on that ground, frequently in supporting a later compiler writing (e.g. [4]). On the other hand, the formalization of existing software is not a goal in itself. From the very beginning denotational semantics has been aimed primarily as a tool for the development of new software.

This paper is devoted to studying the methodology of using denotational techniques in software design. By "software" we mean any complex software system, which includes but is not restricted to programming languages. Since denotations describe the essential components comprising a system and syntax provides ways for the user to access and communicate with these components, we suggest that denotations be developed in the first place and that syntax be derived from them later. More precisely, we suggest that the development of a denotational model of a software system be organized in four following steps:

(1) We develop a mathematical model of the mechanisms of the future system. We define the objects which are to be manipulated by the system (numbers, strings of characters, databases, spreadsheets, etc.) and the corresponding operations. We also define facilities which are offered by a computer environment such as storing and restrieving data in computer memory, combining single operations into programs, etc.

(2) Among the mechanisms defined in the first step we select these which are to be accessible to the user.

(3) We define a prototype syntax for the part of the system defined in the second step.

(4) We modify the prototype syntax in making it more user-friendly.

Of course, each of these steps splits into several substeps.

As a mathematical framework for our discussion we have chosen an algebraic model advocated in early ADJ's papers (cf. Goguen, Thatcher, Wagner and Wright [16]). In that model a software system is described by two many-sorted algebras: **Syn** of syntax and **Den** of denotations. A homomorphism between them:

$$S : \mathbf{Syn} \to \mathbf{Den} \tag{1.1}$$

represents the denotational semantics of **Syn** in **Den**. The fact that S is a homomorphism reflects the compositionality property of our semantics. The carriers of **Syn** and **Den**, i.e. the syntactic domains and the domains of denotations, are constructed on the usual set theory (cf. Blikle and Tarlecki [10], Blikle [6, 9]), rather than as Scott's reflexive domains.

The idea of developing denotations prior to syntax is, of course, not new. It has been suggested, although never explored, by the pioneers of denotational semantics, and is implicit in all schools of algebraic semantics. Similar ideas may be found also in action semantics (Mosses and Watt [25]) which offers a support in the derivation of the algebras of denotations. Finally, in some textbooks on denotational semantics, such as [27], it is advocated that a denotational model of a programming language be organized around a set of semantic algebras. The author has spelled out his ideas for the first time in [5].

One of the major aims of this paper is to systematize the development of the denotational models of software. This is achieved by splitting the development process into several steps where in each step we develop a certain algebra. Although it is not essential for the method whether the successive algebras are described axiomatically or constructively, in this paper we concentrate on the constructive style. A special attention is also given to the problem of deriving (an algebra of) a concrete syntax from an algebra of denotations. This leads to several technical problems on the representation of the algebras of words by context-free grammars and on the possibility of supporting such a derivation process by a computer.

In Sections 2 and 3 we recall basic algebraic concepts and we introduce a notation. This should make our paper selfcontained for readers less familiar with many-sorted algebras. In Section 4 we discuss the definability of the algebras of syntax by context-free grammars. In Section 5 we briefly discuss the process of the construction of an algebra of denotations. Section 6 is devoted to the derivation of syntax from an algebra of denotations. Section 7 contains a list of open problems related to the derivation of syntax. In Section 8 we illustrate our method by showing how to develop a denotational model of a simple wordprocessor.

2. Introductory concepts

In this section we introduce a notation and we recall basic algebraic concepts. Our notation is a dialect of META-IV (the metalanguage of VDM [3]) and has been thoroughly described in [6].

For any sets A and B:

$A \mid B$	denotes the union of A and B,
$A \to B$	denotes the set of all total functions from A into B,
$A \rightleftharpoons B$	denotes the set of all partial functions from A into B,
$A \to_m B$	denotes the set of all finite-domain functions, called *mappings* from A into B,
A^{c^*}	denotes the set of all finite tuples $\langle a_1, \ldots, a_n \rangle$ over A including the empty tuple $\langle\ \rangle$,
A^{c+}	denotes A^{c^*} without the empty tuple $\langle\ \rangle$,
A-**set**	denotes the set of all subsets of A,
A-**finset**	denotes the set of all finite subsets of A.

By "$\hat{}$" we denote the operation of concatenation both for single tuples and for languages. If L is a language, i.e. if $L \subseteq A^{c^*}$ for some A, then L^* denotes the usual Kleene-iteration of L. Observe that "c*" is applicable to any set whereas "*" is applicable only to languages. Moreover $(A^{c^*})^* = A^{c^*}$ is the set of tuples of the elements of A, whereas $(A^{c^*})^{c^*}$ is a set of such tuples whose elements are the tuples of the elements of A.

From Tennent [30] we borrow a convention of writing domain equations in the form:

$$d : D = (\text{domain expression})$$

by which we mean that d possibly with indices denotes an element of domain D. For indexed families $\{A_i\}_{i \in I}$ we use alternatively the notation $\{A.i \mid i \in I\}$.

By $f : A \to B$, $f : A \rightleftharpoons B$ or $f : A \to_m B$ we denote the fact that f is respectively a total function, a partial function, or a mapping from A to B. For *curried functions* like $f : A \to (B \to (C \to D))$ we write $f : A \to B \to C \to D$. We also write $f.a$ for $f(a)$ and $f.a.b.c$ for $((f.a).b).c$. For uniformity reasons each many-argument noncurried function is regarded as a one-argument function on tuples. Consequently we write $f.\langle a_1, \ldots, a_n \rangle$ for $f(a_1, \ldots, a_n)$. Formally this should have led us to writing $f.\langle a \rangle$ rather than $f.a$, but we keep the latter notation as more natural and simpler. If $f : A \rightleftharpoons B$ and $g : B \rightleftharpoons C$, then $f \cdot g : A \rightleftharpoons C$ where

$$f \cdot g = \{(a, c) \mid (\exists b)(f.a = b \ \&\ g.b = c)\}.$$

In the definitions of functions we frequently use conditional expressions of the form $b \to c, d$ which stand for

if b **then** c **else** d.

This may be iterated in which case the expression

$$b_1 \to (a_1, (b_2 \to \cdots (b_n \to a_n, a_{n+1}) \ldots))$$

is written in a column:

$$\begin{array}{ll} b_1 & \to a_1, \\ \vdots & \\ b_n & \to a_n, \\ \text{TRUE} & \to a_{n+1}. \end{array}$$

Sometimes in conditional expressions we shall nest "local constant declarations" of the form **let** $x = exp_1$ **in** exp_2 borrowed from VDM. The scope of such a declaration is the expression exp_2.

For any partial function $f: A \xrightarrow{\sim} B$, by $f[b/a]$ where $a \in A$, $b \in B$ and $f[b/a]: A \xrightarrow{\sim} B$ we denote the following modification of f:

$$f[b/a].x \;=\; x = a \to b, f.x.$$

By $[b_1/a_1, \ldots, b_n/a_n]$ we denote a mapping which assigns b_i to a_i for $i = 1, \ldots, n$.

Now we shall briefly recall some basic concepts associated with many-sorted algebras. We also fix our notation. By a *signature* we mean a four-tuple:

$$\mathbf{Sig} = (Sn, Fn, sort, arity),$$

where Sn is a nonempty possibly infinite set of *sort names*, Fn is a nonempty possibly infinite set of *function names* and where

$$sort: Fn \to Sn,$$

$$arity: Fn \to Sn^{c*}$$

are functions which associate sorts and arities to function names. By an *algebra* over the signature **Sig**, or shortly by a **Sig**-algebra, we mean a triple $\mathbf{Alg} = (\mathbf{Sig}, car, fun)$ where *car* and *fun* are functions interpreting sort names as nonempty sets and function names as total functions on these sets. More precisely, for any $sn \in Sn$, $car.sn$ is a set called the *carrier* of sort sn, and for any $fn \in Fn$ with $sort.fn = sn$ and $arity.fn = \langle sn_1, \ldots, sn_n \rangle$, $fun.fn$ is a total function between corresponding carriers, i.e.

$$fun.fn: car.sn_1 \times \cdots \times car.sn_n \to car.sn.$$

If $arity.fn = \langle\,\rangle$, then $fun.fn$ is a zero-ary function, i.e. accepts only the empty tuple "$\langle\,\rangle$" as an argument. The fact that f is a zero-ary function with value in A is denoted by $f: \to A$ and the unique value of f is denoted by $f.\langle\,\rangle$. Zero-ary functions are also called *algebraic constants.*

In applications we frequently do not define the signature of an algebra explicitly. As long as we do not talk about the derivation of syntax, a signature usually remains implicit in the definitions of the carriers and the operations of the algebra. Consider as an example a two-sorted algebra of integers and booleans with the carriers

$Int = \{\ldots, -1, 0, 1, \ldots\}$ and $Bool = \{tt, ff\}$ and with the following operations:

$1 : \to Int$	an integer constant "one",
$+ : Int \times Int \to Int$	an integer operation of addition,
$tt : \to Bool$	a boolean constant "true",
$< : Int \times Int \to Bool$	an integer-to-boolean function "less than",
$\tilde{} : Bool \to Bool$	a boolean function "not".

The signature of this algebra is implicit (up to the choice of the names of sorts and functions) in the above description. For instance we may choose:

$$Sn = \{int, bool\}, \qquad Fn = \{one, plus, true, less, not\},$$

in which case the functions of *sort* and *arity* are defined as follows:

$$arity.one = \langle\,\rangle, \qquad sort.one = int,$$
$$arity.plus = \langle int, int\rangle, \qquad sort.plus = int,$$
etc.

Now, our algebra may be more formally defined as $\mathbf{Arith} = (\mathbf{Sig}, car, fun)$, where **Sig** has been defined above and where:

$$car.int = Int, \qquad fun.one = 1,$$
$$car.bool = Bool, \qquad fun.plus = +,$$
etc.

Two algebras with the same signature are called *similar*. If $\mathbf{Alg}_i = (\mathbf{Sig}, car_i, fun_i)$ for $i = 1, 2$ are two similar algebras, then we say that $\mathbf{Alg}_1$ is a *subalgebra* of $\mathbf{Alg}_2$ if for any $sn \in Sn$,

$$car_1.sn \subseteq car_2.sn$$

and for any $fn \in Fn$, $fun_1.fn$ coincides with $fun_2.fn$ on the appropriate carriers of $\mathbf{Alg}_1$, i.e. if the restriction of $fun_2.fn$ to the carriers of $\mathbf{Alg}_1$ is identical to $fun_1.fn$. By a *homomorphism* from $\mathbf{Alg}_1$ (a *source algebra*) into $\mathbf{Alg}_2$ (a *target algebra*) we mean a higher-order function H which with any sort $sn \in Sn$ assigns a function:

$$(1) \quad H.sn : car_1.sn \to car_2.sn \tag{2.1}$$

such that for any $fn \in Fn$ with $sort.fn = sn$:

(2) if $arity.fn = \langle\,\rangle$, then
$H.sn.(fun_1.fn.\langle\,\rangle) = fun_2.fn.\langle\,\rangle$,

(3) if $arity.fn = \langle sn_1, \ldots, sn_n\rangle$ with $n > 0$, then for any tuple of arguments $\langle a_1, \ldots, a_n\rangle \in car.sn_1 \times \cdots \times car.sn_n$ we have
$H.sn.(fun_1.fn.\langle a_1, \ldots, a_n\rangle) = fun_2.fn.\langle H.sn_1.a_1, \ldots, H.sn_n.a_n\rangle$.

If all the component functions $H.sn$ are onto-functions, then H is called an *onto-homomorphism.* Otherwise it is called a *strictly into-homomorphism.* If each component of a onto-homomorphism H is reversible, then H is called an *isomorphism.* The componentwise reverse of H is then denoted by H^{-1}. All the elements of $\mathbf{Alg}_2$ which are the images of some elements from $\mathbf{Alg}_1$ through H constitute a subalgebra of $\mathbf{Alg}_2$ called the *image of* $\mathbf{Alg}_1$.

By $H: \mathbf{Alg}_1 \to \mathbf{Alg}_2$ we denote the fact that H is a homomorphism from $\mathbf{Alg}_1$ into $\mathbf{Alg}_2$. If

$$H_{12}: \mathbf{Alg}_1 \to \mathbf{Alg}_2 \quad \text{and} \quad H_{23}: \mathbf{Alg}_2 \to \mathbf{Alg}_3,$$

then $H_{12} \cdot H_{23} = \{H_{12}.sn \cdot H_{23}.sn \mid sn \in Sn\}$ is a homomorphism from $\mathbf{Alg}_1$ into $\mathbf{Alg}_3$.

As we already mentioned before the signature of an algebra may constitute a basis for the construction of a language (syntax) of expressions over that algebra. Starting from a signature $\mathbf{Sig} = (Sn, Fn, sort, arity)$ we construct the least family $\{car_t.sn \mid sn \in Sn\}$ of formal languages of *terms* over the alphabet $Fn \mid \{(\ ,\ ),\ ``,\ ''\}$ such that for any $sn \in Sn$ and any fn with $sort.fn = sn$:

(1) if $arity.fn = \langle\ \rangle$, then $fn \in car_t.sn$,
(2) if $arity.fn = \langle sn_1, \ldots, sn_n \rangle$, then for any terms $ter_i \in car_t.sn_i$, $fn\hat{}(\hat{}ter_1\hat{}, \hat{}\cdots\hat{}, \hat{}ter_n\hat{}) \in car_t.sn$,

where "$\hat{}$" denotes the concatenation of terms. If all the sets $car_t.sn$ are not empty, then we may define a so-called **Sig**-*algebra of terms*:

$$\mathbf{Term} = (\mathbf{Sig}, car_t, fun_t),$$

where the operations are defined as follows: for any fn with $sort.fn = sn$

(1) if $arity.fn = \langle\ \rangle$, then $fun_t.fn.\langle\ \rangle = fn$,
(2) if $arity.fn = \langle sn_1, \ldots, sn_n \rangle$ and $ter_i \in car_t.sn_i$, then,
$fun_t.fn.\langle ter_1, \ldots, ter_n \rangle = fn\hat{}(\hat{}ter_1\hat{}, \hat{}\cdots\hat{}, \hat{}ter_n\hat{})$. (2.2)

For instance, in the case of the algebra **Arith** we have:

$car_t.int = \{one, plus(one, one), plus(one, plus(one, one)), \ldots\}$,
$car_t.bool = \{true, not(true), less(one, one), \ldots\}$,
$fun_t.plus.\langle plus(one, one), one \rangle = plus(plus(one, one), one)$,
etc.

The algebra **Term** over **Sig** is a so-called *initial algebra of terms.* It constitutes a universal language of expressions for the class of all **Sig**-algebras. Formally, for any **Sig**-algebra $\mathbf{Alg} = (\mathbf{Sig}, car, fun)$ there exists exactly one homomorphism

$$T: \mathbf{Term} \to \mathbf{Alg}.$$

This homomorphism represents the semantics of **Term** in **Alg** and is called the *canonical term-homomorphism* for **Alg**. It maps terms into their corresponding values

in **Alg** and is defined in the following way:

(1) for any primitive term fn with $sort.fn = sn$ and $arity.fn = \langle\,\rangle$
$T.sn.fn = fun.fn.\langle\,\rangle$,

(2) for any compound term $fn(ter_1, \ldots, ter_n)$, where $sort.fn = sn$ and $arity.fn = \langle sn_1, \ldots, sn_n\rangle$,
$T.sn.fn(ter_1, \ldots, ter_n) = fun.fn.\langle T.sn_1.ter_1, \ldots, T.sn_n.ter_n\rangle$. (2.3)

E.g. in **Arith** we have:

$T.int.one = 1$,
$T.int.plus(one, one) = 2$,
$T.bool.less(one, one) = ff$,
etc.

Observe that the definition of T is an instance of the general definition (2.1) of a homomorphism. This implies that T is indeed a homomorphism. On the other hand, since any homomorphism from **Term** into **Alg** must satisfy (2.3) and since these equations define T unambiguously, T is the unique homomorphism between our algebras.

Let $\mathbf{Sig}_i = (Sn_i, Fn_i, sort_i, arity_i)$ for $i = 1, 2$ be two arbitrary signatures and let $\mathbf{Alg}_i = (\mathbf{Sig}_i, car_i, fun_i)$ be two algebras over these signatures. We say that:

(a) $\mathbf{Sig}_2$ is an *extension* of $\mathbf{Sig}_1$, or $\mathbf{Sig}_1$ is a *restriction* of $\mathbf{Sig}_2$, if $Sn_1 \subseteq Sn_2$, $Fn_1 \subseteq Fn_2$ and the functions $sort_2$ and $arity_2$ coincide with $sort_1$ and $arity_1$ on Fn_1.

(b) $\mathbf{Alg}_2$ is an *extension* of $\mathbf{Alg}_1$ or $\mathbf{Alg}_1$ is a *restriction* of $\mathbf{Alg}_2$ if:
(1) $\mathbf{Sig}_2$ is an extension of $\mathbf{Sig}_1$,
(2) for any $sn \in Sn_1$, $car_1.sn \subseteq car_2.sn$,
(3) for any $fn \in Fn_1$, $fun_1.fn$ coincides with $fun_2.fn$ on appropriate carriers of $\mathbf{Alg}_1$.

In other words, we extend an algebra if we add new carriers, new functions and new elements.

3. Reachability, ambiguity and initiality

In general, not every element of an algebra is a value of a term. E.g. in **Arith** terms of sort *int* assume only positive values, whereas $car_a.int$ contains all integers. The elements of an algebra which are the values of some terms are called *reachable elements*. Elements which are not reachable are referred to as the *junk* of an algebra. For each sort the set of all reachable elements of that sort is called the *reachable carrier* of that sort.

As is easy to see, an element is reachable in an algebra **Alg** if and only if it can be constructed from the constants of **Alg** in using the operations of **Alg**. This

immediately implies that reachable carriers are always closed under all the operations (functions) of **Alg** and therefore, if none are empty, they constitute the least subalgebra of **Alg**. We call this subalgebra the *reachable subalgebra* of **Alg** and denote it by $\mathbf{Alg}^R$. If **Alg** and $\mathbf{Alg}^R$ are equal, then **Alg** is called a *reachable algebra.*

Reachable algebras play a very important role in our applications. In particular the algebras of syntax **Syn** (cf. (1.1)) are always reachable since syntax is always defined in a constructive way. Below we recall some important properties of reachable algebras.

Proposition 3.1. *The following properties are equivalent*:

(1) **Alg** *is reachable,*

(2) *the (unique) evaluating homomorphism* $T: \mathbf{Term} \to \mathbf{Alg}$ *is onto,*

(3) *any homomorphism which has* **Alg** *as a target is onto.*

Proposition 3.2. *If* $\mathbf{Alg}_1$ *and* $\mathbf{Alg}_2$ *are similar and if* $\mathbf{Alg}_1$ *is reachable, then there exists at most one homomorphism*:

$$H: \mathbf{Alg}_1 \to \mathbf{Alg}_2.$$

If that homomorphism exists, then the image of $\mathbf{Alg}_1$ *in* $\mathbf{Alg}_2$ *is reachable.*

Since this proposition plays an especially significant role in our applications we show its proof.

Proof. Let **Term** be the common algebra of terms of both algebras and let T_1 and T_2 be the unique corresponding homomorphisms (Fig. 1). If H exists, then $T_1 \cdot H$ is a homomorphism from **Term** into $\mathbf{Alg}_2$ and since T_2 is a unique such homomorphism the following equation must hold:

$$T_1 \cdot H = T_2. \tag{3.1}$$

Therefore, since T_1 and T_2 are unique and $\mathbf{Alg}_1$ is reachable, H must be unique as well. The reachability of the image of $\mathbf{Alg}_1$ in $\mathbf{Alg}_2$ also follows from (3.1). □

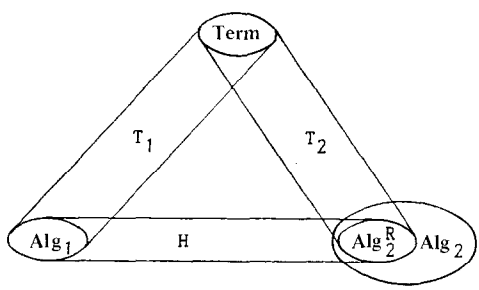

Fig. 1

In our applications the diagram of Fig. 1 usually represents a denotational model of a software system where $\mathbf{Alg}_2$ is an algebra of denotations, $\mathbf{Alg}_1$ is a corresponding algebra of (final concrete) syntax and **Term** is the algebra of prototype (abstract) syntax (cf. Section 1). The homomorphism H represents the denotational semantics of $\mathbf{Alg}_1$ in $\mathbf{Alg}_2$.

So far we have noticed that between syntax and denotations there may be at most one denotational semantics. Of course, there may also be none. Below we formulate some conditions which guarantee the existence of a homomorphism between two similar algebras.

We say that $\mathbf{Alg}_1$ is *not more ambiguous than* $\mathbf{Alg}_2$, in symbols

$$\mathbf{Alg}_1 \sqsubseteq \mathbf{Alg}_2,$$

if T_1 identifies not more elements than T_2, i.e. if for any $sn \in Sn$ and any $t_1, t_2 \in car_t.sn$:

$$T_1.sn.t_1 = T_1.sn.t_2 \quad \text{implies} \quad T_2.sn.t_1 = T_2.sn.t_2.$$

In this definition we do not assume that $\mathbf{Alg}_1$ is reachable. The ambiguity relation is defined in the class of all **Sig**-algebras and constitutes a *preordering*, i.e. is reflexive and transitive.

The defined preordering has a natural intuitive interpretation. Observe that for each of the algebras $\mathbf{Alg}_i$ each term $t \in car_t.sn$ describes a way of the construction of the (reachable) element $T_i.sn.t$. If two different terms have the same value in the target algebra, then they describe two different ways of the construction of the same element, E.g. in **Arith** the terms $plus(one, plus(one, one))$ and $plus(plus(one, one), one)$ describe two different ways of the construction of the integer 3. The more ways we have to construct one element in an algebra, the more that algebra may be called ambiguous. Moreover, each term may be regarded as a parsing tree of a reachable element. In fact, if **Alg** is an algebra of syntax generated by a CF-grammar (Section 4), then terms correspond exactly to parsing trees.

Proposition 3.3. *If* $\mathbf{Alg}_1$ *and* $\mathbf{Alg}_2$ *are similar and if* $\mathbf{Alg}_1$ *is reachable, then the (unique) homomorphism* $H : \mathbf{Alg}_1 \to \mathbf{Alg}_2$ *exists if and only if* $\mathbf{Alg}_1 \sqsubseteq \mathbf{Alg}_2$.

Proof. If H exists, then by (3.1) T_2 must identify values at least as much as T_1. (It identifies exactly as much as T_1 iff H is an isomorphism.) If $\mathbf{Alg}_1 \sqsubseteq \mathbf{Alg}_2$, then the homomorphism H may be constructed in the following way: For any $sn \in Sn$ and any $a \in car_1.sn$ take an arbitrary $t \in car_t.sn$ such that $T_1.sn.t = a$ and set

$$H.sn.a = T_2.sn.t.$$

Since $\mathbf{Alg}_1$ is reachable, t always exists and since T_2 glues at least as much as T_1, the choice of t does not matter. It is not difficult to prove that H is indeed a homomorphism. □

As is easy to see, the algebra **Term** is less ambiguous than any other algebra **Alg** with the same signature, i.e. **Term**$\sqsubseteq$**Alg**. An algebra for which the opposite relationship, i.e. **Alg**$\sqsubseteq$**Term**, is satisfied is called *unambiguous.* In an unambiguous algebra each reachable element is the value of exactly one term, i.e. may be constructed in exactly one way. If that property is not satisfied, then the algebra is called *ambiguous.*

An algebra **Alg** is called *initial* in the class of all **Sig**-algebras, or is simply called **Sig**-*initial*, if for any other **Sig**-algebra $\mathbf{Alg}_2$ there is exactly one homomorphism from **Alg** into $\mathbf{Alg}_1$.

Proposition 3.4. *The following properties are equivalent*:

(1) **Alg** *is initial,*

(2) **Alg** *is reachable and unambiguous,*

(3) **Alg** *is isomorphic to* **Term**.

Proofs are immediate from previous propositions.

4. Algebras versus grammars

In our model of a software system syntax is represented by an algebra. This allows us to express the compositionality principle of denotational semantics and to formulate the rules governing the systematic derivation of syntax (Section 6). It turns out, however, that in applications it is rather inconvenient to describe the algebras of syntax in the style used for the algebras of denotations, since this leads to long definitions with a lot of superfluous technical notation.

Below we discuss a technique of defining algebras of syntax by context-free grammars. This is not only more common for the definitions of syntax, but also better fits into the style of syntax derivation through successive refinements and provides an adequate starting point for the construction of parsers.

The idea of associating algebras with grammars is not new. It has been already explored by Goguen et al. [16], where a grammar gives rise to an algebra of parsing trees referred to as *abstract syntax.* In this paper we are interested mostly in the derivation of a *concrete syntax*, the abstract syntax being developed only at an intermediate stage. We analyze therefore, the relationship between grammars and the algebras of words, rather than of parsing trees.

For technical reasons we slightly redefine the classical concept of a context-free grammar by associating it with a signature. Let $\mathbf{Sig} = (Sn, Fn, sort, arity)$ be an arbitrary signature with finite sets Sn and Fn, and let T be a finite alphabet of terminal symbols disjoint with Sn. By a **Sig**-*grammar* we mean a pair,

$$\mathbf{Gra} = (\mathbf{Sig}, pro),$$

where pro is a function which assigns productions to functional symbols:

$$pro : Fn \to (Sn \times (T \mid Sn)^{c*}),$$

$$pro.fn = \langle sn, x_0 sn_1 \ldots s_{n-1} sn_n x_n \rangle$$

and where $sn = sort.fn$, $\langle sn_1, \ldots, sn_n \rangle = arity.fn$ and each x_i is a word over the alphabet T. The elements of Sn play the role of nonterminals. For the sake of uniqueness (of T) we assume that T is the least alphabet such that all x_i are words over T.

Unlike in the traditional setting (see e.g. Harrison [18]) we do not distinguish any initial nonterminal in a grammar. We do not do so since in our case a grammar defines a class of languages rather than a single language. With every **Sig**-grammar **Gra** = (**Sig**, *pro*) over a terminal alphabet T we unambiguously associate a **Sig**-algebra of words AL.**Gra** = (**Sig**, *car*, *fun*) such that:

(1) for any $sn \in Sn$, *car.sn* is the set of all words over T derivable in the usual sense from the nonterminal *sn* by the productions of the grammar;

(2) for any $fn \in Fn$ with $sort.fn = sn$ and $arity.fn = \langle sn_1, \ldots, sn_n \rangle$ if $pro.fn = \langle sn, x_0 sn_1 \ldots x_{n-1} sn_n x_n \rangle$, then

$$fun.fn.\langle y_1, \ldots, y_n \rangle = x_0 y_1 \ldots x_{n-1} y_n x_n. \tag{4.1}$$

It is not difficult to prove that AL.**Gra** is well-defined, i.e. that its carriers are closed under its operations. One can also prove that every derivable word is constructable in the algebra, i.e. that our algebra is reachable.

We say that **Gra** *defines* AL.**Gra**. We say that a **Sig**-algebra **Alg** is a *context-free algebra* (abbreviated *CF-algebra*), if there exists a **Sig**-grammar **Gra** which defines that algebra, i.e. for which AL.**Gra** = **Alg**.

In the remaining part of this section we discuss the problem of the definability of the algebras of words by grammars. This is an important practical problem which we have to solve if we wish to organize the process of the derivation of syntax in a systematic way and if we wish to support it by a computer. At the same time, however, this is a rather technical problem and therefore we suggest that the readers who are not especially interested in the derivation of syntax skip the rest of this section in the first reading. In that case we only advise the reading of the Proposition 4.3.

By a *syntactic algebra* we mean any reachable algebra of words with finite sets of sorts and operations. Of course, every context-free algebra is a syntactic algebra. The converse implication is not true since in every context-free algebra every carrier must be a context-free language. It is also true, that the context-freeness of carriers is not sufficient for the context-freeness of an algebra. As an example consider a one-sorted algebra with a carrier $A = \{a, aa\}$ and two operations:

$$\begin{aligned} &h : \to A, \qquad f : A \to A, \\ &h.\langle\,\rangle = a, \qquad f.y = aa. \end{aligned} \tag{4.2}$$

This algebra is not context-free since f is not expressible by a production in the sense of (4.1). Of course, if we modify our algebra by replacing f by $f' : \to A$ with $f'.\langle\,\rangle = aa$, then the algebra becomes context-free.

A syntactic algebra with context-free carriers may be non-context-free for one more reason, namely if its operations "permute the arguments". If in a syntactic

algebra we have an operation g with a signature say:

$$g : A \times B \to C,$$

then for our algebra to be context-free, g must be defined by an equation of the form $g.\langle a, b\rangle = x_0 a x_1 b x_2$. If we set

$$g.\langle a, b\rangle = x_0 b x_1 a x_2,$$

then there is no grammar which defines our algebra since any production which corresponds to the signature of g must be of the form $C \to x_0 A x_1 B x_2$. This example shows that our concept of a CF-algebra may be a little too narrow for applications, since it forces the designer of syntax to obey to the order of arguments chosen by the designer of denotations. We discuss that problem in more detail at the end of the section.

Below we formulate a property of syntactic algebras which is necessary and sufficient for context-freeness. We start from the introduction of some auxiliary concepts. Let **Alg** = (**Sig**, *car*, *fun*) with **Sig** = (*Sn*, *Fn*, *sort*, *arity*) be an arbitrary syntactic algebra over some (minimal) alphabet T. If for an operation symbol *fn* with $arity.fn = \langle sn_1, \ldots, sn_n\rangle$ where $n \geqslant 0$, there exists a string of words $\langle x_0, \ldots, x_n\rangle \in (T^{c*})^{c(n+1)}$ such that for any argument $\langle y_1, \ldots, y_n\rangle$ of that function:

$$fun.fn.\langle y_1, \ldots, y_n\rangle = x_0 y_1 \ldots x_{n-1} y_n x_n$$

then $\langle x_0, \ldots, x_n\rangle$ is called a *skeleton* of *fn* (and of *fun.fn*) in **Alg**.

A function in a syntactic algebra may have from none to many skeletons. Consider the following one-sorted algebra with a carrier $\{a\}^{c+}$:

$$h_1.\langle\ \rangle = a, \qquad f_1.y = ya. \tag{4.3}$$

Function h_1 has exactly one skeleton, namely $\langle a\rangle$, and function f_1 has two skeletons: $\langle \varepsilon, a\rangle$ and $\langle a, \varepsilon\rangle$ where ε denotes the empty word. Function f from (4.2) has no skeleton at all. A function which has a skeleton is called a *skeleton function* and if this skeleton is unique then it is called a *monoskeleton function.*

If every operation of **Alg** is a skeleton function, then **Alg** is called a *skeleton algebra* and by a *skeleton* of **Alg** we mean any function:

$$sk : Fn \to (T^{c*})^{c+}$$

such that *sk.fn* is a skeleton of *fn*. Of course, similarly to functions, also algebras may have from none to many skeletons. An algebra which has exactly one skeleton is called a *monoskeleton algebra.*

With every skeleton algebra **Alg** = (**Sig**, *car*, *fun*) and a chosen skeleton *sk* of that algebra we may unambiguously associate a grammar GR.(**Alg**, *sk*) = (**Sig**, *pro*) where for any $fn \in Fn$ with $sort.fn = sn$, $arity.fn = \langle sn_1, \ldots, sn_n\rangle$ and $sk.fn = \langle x_0, \ldots, x_n\rangle$ we set $pro.fn = \langle sn, x_0 sn_1 \ldots x_{n-1} sn_n x_n\rangle$. By the definition of AL.**Gra** that grammar defines **Alg**, i.e.

$$\text{AL.(GR.}\langle \textbf{Alg}, sk\rangle) = \textbf{Alg}.$$

Consequently, **Alg** is context-free. Since every context-free algebra is a skeleton algebra we can formulate the following simple proposition:

Proposition 4.1. *A syntactic algebra is context-free if and only if it is a skeleton algebra.*

In Section 6 we discuss a systematic method of the derivation of an algebra of syntax **Syn** from a given algebra of denotations **Den**. This consists in constructing a sequence of syntactic algebras

$$\mathbf{Syn}_0, \ldots, \mathbf{Syn}_n$$

such that $\mathbf{Syn}_0 = \mathbf{Term}$, $\mathbf{Syn}_n = \mathbf{Syn}$ and each $\mathbf{Syn}_i$ is a homomorphic image of $\mathbf{Syn}_{i-1}$. Of course, **Term** is a skeleton algebra (cf. (2.2)), hence is context-free. It turns out, however, that a homomorphism may destroy the context-freeness of an algebra. Indeed, take as a source algebra the algebra (4.3) which is obviously context-free, and as a target algebra the algebra (4.2) which is not context-free. The (unique) homomorphism between them is $i_1.a^n = n = 1 \to a, aa$ for any $n \geqslant 0$.

Also an isomorphism may destroy the context-freeness of an algebra. Consider again (4.3) as a source, and as a target a one-sorted algebra with the carrier $\{a^n b^n c^n \mid n = 1, 2, \ldots\}$ and with the following operations:

$$h_2.\langle\,\rangle = abc,$$

$$f_2.a^n b^n c^n = a^{n+1} b^{n+1} c^{n+1} \quad \text{for } n = 1, 2, \ldots.$$

Of course, our new algebra is not context-free and the corresponding (unique) isomorphism between them is defined by:

$$I.a^n = a^n b^n c^n \quad \text{for } n = 1, 2, \ldots.$$

Let $\mathbf{Alg}_i = (\mathbf{Sig}, car_i, fun_i)$, $i = 1, 2$, be two syntactic algebras over a common signature $\mathbf{Sig} = (Sn, Fn, sort, arity)$ and let $H : \mathbf{Alg}_1 \to \mathbf{Alg}_2$ be a (unique) homomorphism between them. We say that H is a *skeleton homomorphism* if there exists a function

$$sk : Fn \to (T^{c^*})^{c+}$$

called *a skeleton of* H such that for every $fn \in Fn$ with $sort.fn = sn$, $arity.fn = \langle sn_1, \ldots, sn_n\rangle$ and $sk.fn = \langle x_0, \ldots, x_n\rangle$:

$$H.sn.(fun_1.fn.\langle y_1, \ldots, y_n\rangle) = x_0(H.sn_1.y_1) \ldots x_{n-1}(H.sn_n.y_n)x_n \tag{4.4}$$

for any argument $\langle y_1, \ldots, y_n\rangle$ of $fun_1.fn$. Similarly to functions, also homomorphisms may have from none to many skeletons. E.g. both our formerly defined homomorphisms which have (4.3) as a source have no skeletons.

Proposition 4.2. *For any syntactic algebra* **Alg** *the following properties are equivalent*:

(1) **Alg** *is context-free,*

(2) *every homomorphism which has* **Alg** *as a target is a skeleton homomorphism,*

(3) *there exists a skeleton homomorphism which has* **Alg** *as a target.*

Proof. Let $\mathbf{Alg}_2$ be context-free and let *sk* be one of its skeletons. Consider an arbitrary $\mathbf{Alg}_1$ with a homomorphism $H : \mathbf{Alg}_1 \to \mathbf{Alg}_2$. For every $fn \in Fn$ with $sort.fn = sn$, $arity.fn = \langle sn_1, \ldots, sn_n \rangle$ and $sk.fn = \langle x_0, \ldots, x_n \rangle$ we have

$$H.sn.(fun_1.fn.\langle y_1, \ldots, y_n \rangle) =$$

$$fun_2.fn.\langle H.sn_1.y_1, \ldots, H.sn_n.y_n \rangle =$$

$$x_0(H.sn_1.y_1) \ldots x_{n-1}(H.sn_n.y_n)x_n. \tag{4.5}$$

This proves that H is a skeleton homomorphism, i.e. that (1) implies (2). The implication from (2) to (3) is obvious because the set of homomorphisms which point to $\mathbf{Alg}_2$ is not empty since it contains the canonical term-homomorphism. Now assume that there exists $\mathbf{Alg}_1$ with $H : \mathbf{Alg}_1 \to \mathbf{Alg}_2$ which is a skeleton homomorphism. Let *sk* be the skeleton of this homomorphism and take an arbitrary $fn \in Fn$ with $sort.fn = sn$, $arity.fn = \langle sn_1, \ldots, sn_n \rangle$, $sk.fn = \langle x_0, \ldots, x_n \rangle$ and an arbitrary argument tuple $\langle w_1, \ldots, w_n \rangle$ for $fun_2.fn$. Since $\mathbf{Alg}_2$ is reachable H must be onto (Proposition 3.1) and therefore there exists a tuple $\langle y_1, \ldots, y_n \rangle$ of elements of $\mathbf{Alg}_1$ such that $H.sn_i.y_i = w_i$. Consequently

$$fun_2.fn.\langle w_1, \ldots, w_n \rangle =$$

$$fun_2.fn.\langle H.sn_1.y_1, \ldots, H.sn_n.y_n \rangle =$$

$$H.sn.(fun_1.fn.\langle y_1, \ldots, y_n \rangle) =$$

$$x_0(H.sn_1.y_1) \ldots x_{n-1}(H.sn_n.y_n)x_n.$$

This proves that *sk* is a skeleton of $\mathbf{Alg}_2$. Hence $\mathbf{Alg}_2$ is context free. □

Observe that the grammar of a target algebra is always implicit in the definition of a corresponding skeleton homomorphism. Indeed, for any equation of the form (4.4) the corresponding production is:

$$sn \to sk.fn.$$

Examples will be shown in Section 6. In the same section we shall also see that our approach to the derivation of syntax will allow, and encourage us to use ambiguous grammars. Below we define that property formally for grammars over signatures.

A **Sig**-grammar **Gra** is said to be *unambiguous* if for any $sn \in Sn$ the corresponding traditional grammar with *sn* as the initial symbol, is unambigous in the usual sense (cf. [18]). In the opposite case **Gra** is said to be *ambiguous.*

Proposition 4.3. *A grammar* **Gra** *is unambiguous if and only if the algebra* AL.**Gra** *is unambiguous.*

The proof of this proposition (which we leave to the reader) is based on the fact that terms over the signature of **Gra** unambiguously represent parsing trees over

Gra. More formally, if T: **Term** → AL.**Gra** is the canonical term-homomorphism for AL.**Gra**, then for any $sn \in Sn$ and any $x \in car.sn$ the elements of the set

$$T^{-1}.sn.x = \{t \mid T.sn.t = x\}$$

represent—and in fact may be regarded as—all parsing trees of x in the grammar $\mathbf{Gra}_{sn}$.

At the end of this section one remark is in order. We have to say that for the sake of making the theory possibly simple we have assumed a simplified definition of a grammar over a signature where the order of sorts in the arity of a functional symbol determines the order of nonterminals on the right-hand side of the corresponding production. This makes an algebra with the operation:

$$while : Command \times Expression \to Command,$$

$$while.\langle com, exp\rangle = \textbf{while } exp \textbf{ do } com \textbf{ od}$$

not context-free since *while* permutes *com* and *exp* whereas any production with the signature of *while* cannot do that. For applications this means that the signature of the algebra of denotations determines the order in which syntactic components appear in syntactic compound objects. This is, of course, not very practical since it forces us to think about concrete syntax when we design denotations.

There seems to be at least two ways of repairing the described situation. One consists in redefining the concept of a grammar over a signature by allowing permutations. In that case we have to redefine also the concept of a skeleton and to introduce a few further technicalities if we want to have a unique association of a grammar with a skeleton to an algebra. Such a solution has been described in [8], although—as has been pointed out to the author by M. Ryćko—it requires further technical modifications. Another solution consists in keeping the definition of a **Sig**-grammar unchanged while allowing that the signature of the algebra of syntax does not necessarily coincide with that of the algebra of denotations. This requires the redefinition of the concept of a homomorphism by enriching it with the morphism of signatures (cf. [13]). Such a solution introduces some additional mathematical machinery, but it seems more appropriate than the former since the signature morphisms are known to be useful in the specifications on software anyway. We leave this as a little research problem to our readers (cf. Section 7).

5. Designing denotations

The method which we discuss in this paper consists of developing the denotational model (1.1) of a software system in two basic steps: first we develop the algebra of denotations **Den**, then we derive from it a corresponding algebra of syntax **Syn**. Of

course, each of these steps consists of many substeps. In this section we briefly discuss the development of **Den**. The derivation of **Syn** is discussed in Section 6.

The development of **Den** constitutes the most creative step in the mathematical process of software development. Of course, what this step looks like depends on the type of software which we design. The development of **Den** for one programming language will differ from the development of **Den** for another programming language, as much as one language may differ from another. The development of **Den** for a language will differ from the development of **Den** for an operating system and this will differ from the development of **Den** for a database system. The art, craft and science of developing **Den** is not much less than just the art, craft and science of the mathematical engineering of software and therefore goes far beyond the scope of this paper. In this section we restrict ourselves to only a few general remarks, remarks which are independent of the kind of a software system under consideration.

We start from an observation that in general the algebra **Den** represents only a small user-visible part of a corresponding software system. For instance, in a database management system we usually have (hidden) procedures which temporarily destroy some constraints imposed on databases and which, therefore, are not to be seen by the user. In an operating system or in a compiler the user has a direct access to only a very few functions of the system. It seems advisable, therefore, that the design process of a software system start from the development of an algebra **Sys** of the whole system, which we later restrict to **Den** by explicitly indicating which of the operations are to be seen by the user.

From a general viewpoint the development of **Sys** may be compared to programming in a high-level functional language. As in programming, also here, we may choose a bottom-up approach and a top-down approach.

In the bottom-up approach a software system is regarded as a computer environment which supports the manipulation of objects of certain types by means of some operations. First step towards the development of **Sys** consists, there, of defining an algebra with these objects and operations. We call it an *algebra of data* and denote by **Dat**. For instance, if we design a database management system, then the carriers of **Dat** contain such objects as numbers, logical values, character strings, records, databases, reports, etc. whereas operations are arithmetic, Boolean and string operations, the operations on records and databases, the generators of reports, etc.

When we are done with **Dat** we proceed to the next step where we describe a system which supports the use of the mechanisms of **Dat** in a computer environment. Formally, we define the algebra **Sys**. Since in a computer data are not available in isolation, but only as objects stored in a memory, we introduce a mathematical model of a memory state and we define a domain of functions which we call *evaluators* and which map states into data. Then we introduce a domain of *declarators* which are functions from states to states and which describe the mechanisms of putting data into the store. If we design a system with imperative mechanisms, then we also introduce a domain of *executors*, which are again functions from states to

states, but which have some slightly different properties than declarators. In the context of syntax the evaluators, declarators and executors play the role of the denotations of expressions, declarations and commands respectively.

The sets of evaluators, declarators and executors constitute the major carriers of **Sys**. The operations on the elements of these carriers correspond to major programming facilities of the system, e.g. in a typical programming language they correspond to the constructors of expressions, declarations and commands. We shall see this in an example at the end of the section.

Now a few words about the top-down approach. In that case we regard a computer system as a machinery which is supposed to do a certain job and we postpone till later the decision about the selection of primitive tools (represented by **Dat**), which the system may need in order to perform that job. Formally, we start from a "parameterized" description of **Sys**, where some carriers and/or operations are left unspecified. As the design process progresses we "fill the holes" in the definition of **Sys**, and in doing that we define a suitable algebra **Dat**. An example of a top-down design is shown in [7].

The bottom-up approach is convenient if we start from a fixed set of basic operations which we want to put together into a software system. For instance, if we design a system handling spreadsheets, we shall first define a suitable many-sorted algebra of numbers, texts, spreadsheets, etc. and then we shall extend it by all computer facilities such as storing data in and retrieving them from the memory, elaborating data in a programmable way, printing data, sending data to other computers, etc.

The top-down approach seems suitable whenever we want to, and can, postpone the decision about the primitives of the system till we define the major functions of the system. This may happen, for instance, when we design a computer system which is supposed to react in some expected way with some environment, e.g. with a plant-control system. This approach may be compared to the "behavioral school" in software specification, where the external behavior of the system is described in the first step of system development (cf. [26] and references there).

The process of developing **Den** is summarized on the diagram of Fig. 2. The algebra **Dat** defines a collection of basic tools which must be provided by the future system. The algebra **Sys** defines a workshop where the former tools may be used in a computer environment. This includes the mechanisms of storing data and combining simple universal tools (basic instructions) into complex problem-oriented tools (programs). The algebra **Den** contains a selection of the mechanisms of **Sys** which are to be seen by the user.

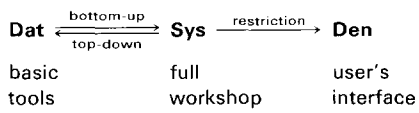

Fig. 2

Now, let us briefly discuss the mathematical techniques which may be chosen in defining our three algebras. Essentially they may be split into two classes: axiomatic (property-oriented) techniques and constructive (model-oriented) techniques.

Axiomatic techniques have been extensively studied on the ground of algebraic semantics approach and several of them were implemented (see e.g. Goguen, Meseguer and Plaisted [15] or Ehrig and Mahr [13]). Their major advantage is abstractness. An axiomatic definition lists the intended properties of the future system. The obligation of the implementation designer is to satisfy these properties, no matter how. The disadvantage of axiomatic specifications is that the completeness and the consistency of such specifications is usually far from evident and may be difficult to prove (cf. Titterington [31]). This problem is, of course, the more critical the larger and more complicated is the system. Axiomatic definitions require also more mathematical maturity from the designer and provide less hints about how to implement the system. Of course, we do not talk here about a standard implementation of axiomatic definition by rewriting rules (cf. Dershowitz [12] or Meseguer and Goguen [23] and references there), since this—at least so far—may only be regarded as a rapid-prototyping facility.

Constructive techniques lead to definitions which are, of course, less abstract. Each such a definition describes a concrete mathematical model of a system and therefore it is much easier to be checked for completeness and consistency and gives more hints about a future implementation. On the other hand, the adequacy of such a definition, i.e. the satisfaction of some expected properties of the system, is now implicit and must be proved.

Contrary to what is usually claimed, the axiomatic techniques do not leave—in the opinion of the author—more freedom of choice for an implementor than do the constructive techniques. They only lead to definitions which provide *less hints* about a future *possible implementation* than constructive definitions. An axiomatic definition of a system, i.e. of an algebra **Sys**, identifies a class of algebras which are somehow equivalent, e.g. mutually isomorphic. It identifies this class by a set of axioms, which tell very little about what the elements and the operations of these algebras can be, but only what properties do they have. A constructive definition identifies also a class of algebras, just that in this case it explicitly points to one element of this class, the others being (implicitly) all appropriately equivalent algebras to the chosen one, e.g. isomorphic with it. The obligation of an implementor consist in each case of finding an implementation algebra **Imp** which is appropriately related, e.g. by a homomorphism, to *any* algebra in the former class. Of course, given a constructive definition of an algebra, the implementor may use it in the construction of the definition of **Imp**. Moreover, he may be tempted to use it which, of course, somehow "spoils" his imagination. But from the mathematical viewpoint he has in each case the same freedom of choice.

Of our algebras of Fig. 2 only **Dat** seems to be an obvious candidate for an axiomatic definition. Of course, this always depends on the system under design, but in general **Dat** is of an order of magnitude simpler than **Sys**. It should be

emphasized at this point that an axiomatic definition of **Dat** does not imply any (mathematical) obligation to define **Sys** axiomatically as well. If we have a sufficiently rich definitional metalanguage, then we may define **Dat** axiomatically and later refer to this definition in a constructive definition of **Sys**. Since **Den** is a restriction (reduct) of **Sys** its definition is always of the same style as that of **Sys**. In applications, the definition of **Den** consists of just a list of selected operations and carriers of **Sys**.

Example 5.1. In this example we discuss the development of an algebra of denotations of a simple software system. The development of a corresponding (algebra of) syntax is described in the next section (Example 6.1).

Assume that we want to design a computer system which communicates with the external world by receiving and emitting messages and where we can elaborate each received message in a programmable way. The communication with the external world is to be automatic, i.e. not programmable. For the simplicity of example we assume that each message consists of only two values, each value being either a natural number or a boolean value or an error element.

We shall design our system in a bottom-up style thus starting from the algebra of data. First we define the carriers of this algebra:

$$n : Nat = \{1, \ldots, N, err\},$$

$$b : Bool = \{tt, ff, ee\}.$$

In each of these carriers we have included a so-called *abstract error* (cf. Goguen [14]) which represents an error signal. We assume to have in **Dat** the following operations:

$$one : \rightarrow Nat,$$
$$plus : Nat \times Nat \rightarrow Nat,$$
$$times : Nat \times Nat \rightarrow Nat,$$
$$less : Nat \times Nat \rightarrow Bool,$$
$$and : Bool \times Bool \rightarrow Bool.$$

For the sake of brevity we explicitly define only three of these operations. As is easy to see our algebra is reachable, no matter how *times* and *and* are defined.

$$one.\langle\,\rangle = 1,$$

$$\begin{aligned} plus.\langle n_1, n_2\rangle = & \\ & n_1 = err \rightarrow err, \\ & n_2 = err \rightarrow err, \\ & n_1 + n_2 > N \rightarrow err, \\ & \text{TRUE} \rightarrow n_1 + n_2, \end{aligned} \qquad \begin{aligned} less.\langle n_1, n_2\rangle = & \\ & n_1 = err \rightarrow ee, \\ & n_2 = err \rightarrow ee, \\ & n_1 < n_2 \rightarrow tt, \\ & \text{TRUE} \rightarrow ff. \end{aligned}$$

When we have defined **Den** we can proceed to the construction of a computer system

over that algebra. For simplicity we assume that our system contains only two registers—call them x and y—where to store values in computer memory. We define four following domains:

$$ide : Identifier = \{x, y\},$$
$$val : Value = Nat \mid Bool,$$
$$mes : Message = Value \times Value,$$
$$sta : State = Identifier \to Value.$$

Now, we have to think about the type of actions which our system should be able to perform. Again, for the sake of simplicity we assume only four types of such actions: *readings* and *writings* which correspond to input and output operations, *evaluators* which are used to retrieve and elaborate data stored in the registers x and y, and *executors* which are used to transform states. We introduce therefore four further domains:

$$rea : Reading = Message \to State,$$
$$wri : Writing = State \to Message,$$
$$eva : Evaluator = State \to Value \quad \text{(expression denotations)},$$
$$exe : Executor = State \rightleftharpoons State \quad \text{(command denotations)}.$$

Here we have assumed that readings, writings and the evaluations of expressions always terminate. Therefore the first three domains contain only total functions. The last domain contains also partial functions since we intend to allow loops in the execution of commands. This, of course, does not stand in a contradiction with our earlier assumption (Section 2) that all operations in our algebras must be total. As we shall see below, executors are to be the elements of **Sys** rather than its operations.

The next step consists of defining the constructors of readings, writings, evaluators and executors. Here we describe the programming facilities of the system. Let us start from the readings and writings. We assume to have only one action of each of these types. We introduce, therefore, two zero-ary constructors:

$$read : \to Reading, \qquad write : \to Writing,$$

which we define as follows:

$$read.\langle\,\rangle.\langle val_1, val_2\rangle = [val_1/x, val_2/y],$$
$$write.\langle\,\rangle.[val_1/x, val_2/y] = \langle val_1, val_2\rangle.$$

Note that *read* is a zero-ary function, whose value $read.\langle\,\rangle$ is a function from messages into states. The function *write* is similar. In an algebraic approach zero-ary functions represent constants. Instead of saying that an algebra consists of carriers, constants and functions, we have only carriers and functions, the latter possibly of a zero-arity. From the viewpoint of applications, constants are primitive actions of

the system, i.e. such actions which do not need to be constructed from anything "smaller".

Now we define the constructors of evaluators. First, with every identifier we assign an evaluator that given a state returns the value stored under this identifier in that state:

$$\begin{aligned} &\mathit{evaluate} : \mathit{Identifier} \to \mathit{Evaluator}, \\ &\mathit{evaluate.ide.ste} = \mathit{sta.ide}. \end{aligned}$$

In the context of the future syntax (Section 6) this means that each identifier constitutes an expression. The definition of *evaluate* describes the way in which such an expression is evaluated.

Next, we want to have such evaluators in the system which allow for the applications of data-type operations (the operators of **Dat**) to data retrieved from the store. To this end for each operation on data we define a corresponding constructor of evaluators:

$$\begin{aligned} &\mathit{e\_one} : \to \mathit{Evaluator}, \\ &\mathit{e\_plus} : \mathit{Evaluator} \times \mathit{Evaluator} \to \mathit{Evaluator}, \\ &\mathit{e\_times} : \mathit{Evaluator} \times \mathit{Evaluator} \to \mathit{Evaluator}, \\ &\mathit{e\_less} : \mathit{Evaluator} \times \mathit{Evaluator} \to \mathit{Evaluator}, \\ &\mathit{e\_and} : \mathit{Evaluator} \times \mathit{Evaluator} \to \mathit{Evaluator}. \end{aligned} \tag{5.1}$$

All these constructors are defined according to the same scheme. We give one such a definition as an example:

$$\begin{aligned} &\mathit{e\_plus}.\langle \mathit{eva}_1, \mathit{eva}_2 \rangle.\mathit{sta} = \\ &\quad \textbf{let } \mathit{val}_1 = \mathit{eva}_1.\mathit{sta} \textbf{ in} \\ &\quad \textbf{let } \mathit{val}_2 = \mathit{eva}_2.\mathit{sta} \textbf{ in} \\ &\quad \mathit{val}_1 \notin \mathit{Nat} \to \mathit{err}, \\ &\quad \mathit{val}_2 \notin \mathit{Nat} \to \mathit{err}, \\ &\quad \mathrm{TRUE} \to \mathit{plus}.\langle \mathit{val}_1, \mathit{val}_2 \rangle. \end{aligned}$$

The evaluator $\mathit{e\_plus}.\langle \mathit{eva}_1, \mathit{eva}_2 \rangle$ evaluates its argument evaluators eva_1 and eva_2 in the current state and then, if the computed values are of numeric type, applies the data-type operation *plus* to these values.

Since identifiers have been used in the construction of evaluators, we must be able to construct identifiers as well. To this end we introduce two zero-ary constructors, one for each identifier:

$$\begin{aligned} &\mathit{create\_x} : \to \mathit{Identifier}, \qquad \mathit{create\_y} : \to \mathit{Identifier}, \\ &\mathit{create\_x}.\langle\,\rangle = x, \qquad\qquad \mathit{create\_y}.\langle\,\rangle = y. \end{aligned}$$

For the implementation this means that the user of the system can somehow "generate identifiers from nothing", e.g. may tape them in from the keyboard.

In the last step we define four typical constructors of executors which correspond to four typical program connectives in a programming language:

$$\begin{aligned}
&\mathit{assign} : \mathit{Identifier} \times \mathit{Evaluator} \to \mathit{Executor},\\
&\mathit{continue} : \mathit{Executor} \times \mathit{Executor} \to \mathit{Executor},\\
&\mathit{while} : \mathit{Evaluator} \times \mathit{Executor} \to \mathit{Executor},\\
&\mathit{if} : \mathit{Evaluator} \times \mathit{Executor} \times \mathit{Executor} \to \mathit{Executor}.
\end{aligned}$$

Their definitions are quite routine. We give first two of them as an example:

$$\begin{aligned}
&\mathit{assign}.\langle \mathit{ide}, \mathit{eva}\rangle.\mathit{sta} = \mathit{sta}[\mathit{eva}.\mathit{sta}/\mathit{ide}],\\
&\mathit{continue}.\langle \mathit{exe}_1, \mathit{exe}_2\rangle = \mathit{exe}_2 \cdot \mathit{exe}_1,
\end{aligned}$$

where "$\cdot$" denotes the composition operation of partial functions (Section 2).

Now we are ready to define the algebras **Sys** and **Den**. The former algebra is supposed to describe our future system and therefore it should contain all and only these operations which we want to implement in that system. The latter algebra is a restriction (reduct) of the former and contains only these operations of **Sys** which are to be accessible by the user. In both cases our choice is pragmatic, rather than mathematical, and depends on what we want to have in the system.

In our example we assume the following signature of **Sys**:

$$\begin{aligned}
&\mathit{read} : \to \mathit{Reading},\\
&\mathit{write} : \to \mathit{Writing};\\
&\\
&\mathit{create_x} : \to \mathit{Identifier},\\
&\mathit{create_y} : \to \mathit{Identifier};\\
&\\
&\mathit{e_one} : \to \mathit{Evaluator},\\
&\mathit{evaluate} : \mathit{Identifier} \to \mathit{Evaluator},\\
&\mathit{e_plus} : \mathit{Evaluator} \times \mathit{Evaluator} \to \mathit{Evaluator},\\
&\mathit{e_times} : \mathit{Evaluator} \times \mathit{Evaluator} \to \mathit{Evaluator},\\
&\mathit{e_less} : \mathit{Evaluator} \times \mathit{Evaluator} \to \mathit{Evaluator},\\
&\mathit{e_and} : \mathit{Evaluator} \times \mathit{Evaluator} \to \mathit{Evaluator};\\
&\\
&\mathit{assign} : \mathit{Identifier} \times \mathit{Evaluator} \to \mathit{Executor},\\
&\mathit{continue} : \mathit{Executor} \times \mathit{Executor} \to \mathit{Executor},\\
&\mathit{while} : \mathit{Evaluator} \times \mathit{Executor} \to \mathit{Executor},\\
&\mathit{if} : \mathit{Evaluator} \times \mathit{Executor} \times \mathit{Executor} \to \mathit{Executor}.
\end{aligned} \tag{5.2}$$

All other operations such as e.g. the operations of **Dat** are regarded as auxiliary functions introduced only for the sake of the definitions of our constructors. They do not need to be implemented.

When we are done with **Sys** we proceed to the definition of **Den**, i.e. to deciding which operations of the system are to be visible by the user. In our case we shall assume that the only nonvisible operations will be *read* and *write*. These operations

are to be performed by the hardware. The signature of **Den** results from the signature of **Sys** by removing these two operations with the corresponding carriers.

Although the choice of the operations, and therefore also of the carriers of our two algebras, is essentially only a pragmatic issue, we should make sure that all carriers in **Den** have nonempty reachable parts. A carrier with an empty reachable part corresponds to an empty syntactic category (i.e. to an empty carrier of that sort in the algebra of syntax) and therefore it is pointless to have such a carrier in the algebra. For instance, if we remove from **Den** the constructors *create_x*, *create_y* and *e_one*, then the reachable subalgebra of **Den** becomes empty and therefore also the corresponding syntactic algebra becomes empty.

6. Designing syntax

In this section we discuss the process of designing a customized syntax for a given algebra of denotations. Given an algebra **Den** our task consists of constructing an algebra **Syn** with four following properties:

(1) **Syn** is a syntactic algebra,
(2) there is a homomorphism from **Syn** into **Den**,
(3) **Syn** is context-free,
(4) the notation offered by **Syn** is sufficiently convenient.

Property (2) is called the correctness of **Syn** with respect to **Den**. It guarantees that **Syn** may be used as a syntax for **Den**. The (unique) homomorphism between **Syn** and **Den** is the corresponding denotational semantics.

Of course, in order to be correct **Syn** must have the same signature as **Den**. This implies that these algebras must fit into the diagram of Fig. 3 (cf. also Fig. 1), where **Term** is a common algebra of terms and *Ts* (term-to-syntax), *Td* (term-to-denotation) and *Sd* (syntax-to-denotation) are the corresponding unique homomorphisms (this model will become slightly more complicated in the sequel).

As our diagram shows, **Term** is correct and any correct syntax must be a homomorphic image of **Term**. Since **Term** is obviously context-free, we suggest that **Syn**

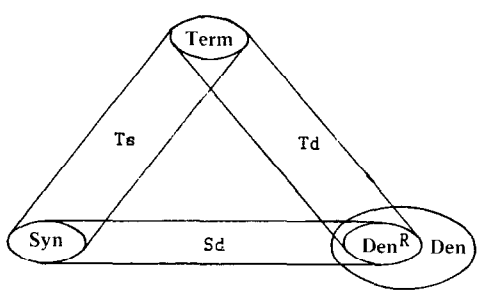

Fig. 3

be derived from **Term** through a sequence of stepwise homomorphic refinements preserving both the correctness and the context-freeness. This should lead to a sequence of algebras $\mathbf{Syn}_0, \ldots, \mathbf{Syn}_n$ such that:

(1) $\mathbf{Syn}_0 = \mathbf{Term}$,
(2) each $\mathbf{Syn}_i$ is a homomorphic refinement of $\mathbf{Syn}_{i-1}$, i.e. there exists a homomorphism $Ss_i : \mathbf{Syn}_{i-1} \to \mathbf{Syn}_i$,
(3) each $\mathbf{Syn}_i$ is correct, i.e. there exists a homomorphism $Sd_i : \mathbf{Syn}_i \to \mathbf{Den}$,
(4) each $\mathbf{Syn}_i$ is context-free,
(5) $\mathbf{Syn}_n = \mathbf{Syn}$.

The first step in this derivation process, i.e. the construction of **Term**, is quite routine since the algebra of terms is unambiguously determined by the signature of **Den**. Further steps are, of course, much less routine, since they correspond to design decisions, such as passing from a prefix to an infix notation, introducing keywords, omitting superfluous parentheses, etc.

In our approach, the algebra **Term** is called a *prototype syntax* and the algebra **Syn** is called a *final syntax*. This roughly corresponds to the traditional classification into an *abstract syntax* and a *concrete syntax* (see McCarthy [22], Goguen, Thatcher, Wagner and Wright [16], Bjørner and Jones [3]). We use different terms here in order to emphasize that our derivation of **Syn** from **Term** proceeds within one abstraction level, i.e. that we derive our concrete final syntax from an equally concrete prototype syntax.

In the derivation of **Syn** each $\mathbf{Syn}_i$ is an algebra of syntax since homomorphic transformations obviously preserve this property (cf. Proposition 3.2). In each step we should check, however, whether $\mathbf{Syn}_i$ is context-free and correct. The former property is checked by inspecting whether our homomorphism has a skeleton (Proposition 4.2). In checking the correctness of $\mathbf{Syn}_i$ two cases are possible:

(1) If Ss_i is an isomorphism, then the correctness of $\mathbf{Syn}_i$ follows from the correctness of $\mathbf{Syn}_{i-1}$ since in that case:

$$Sd_i = Ss_i^{-1} \cdot Sd_{i-1}.$$

(2) If Ss_i is not an isomorphism, which means that Ss_i glues some elements of $\mathbf{Syn}_{i-1}$ together, then $\mathbf{Syn}_i$ may be incorrect. In order to prove that $\mathbf{Syn}_i$ is correct we have to prove that it is not more ambiguous than **Den**. This amounts to proving that Ts_i does not glue more than Td (cf. Fig. 4), or—which is equivalent, but may be easier to prove—that Ss_i does not glue more than Sd_{i-1}. Observe that due to the reachability of syntax all the diagrams of Fig. 4 commute.

As the reader has probably noticed already, we do not assume that **Syn** must be unambiguous, i.e. that the corresponding grammar must be unambiguous (Proposition 4.3). Apparently this may lead to an ambiguous (or nondeterministic) parsing algorithm since for a given element x of **Syn** the corresponding set of parsing trees

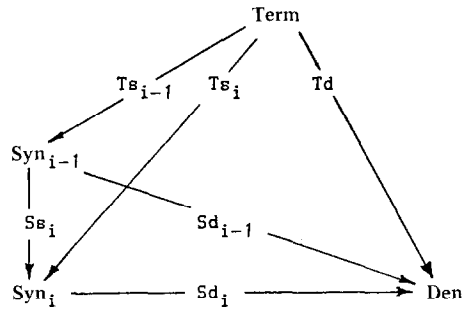

Fig. 4

(cf. Section 4)

$$Ts^{-1}.sn.x = \{t \mid Ts.sn.t = x\}$$

may contain more than one element. Observe, however, that the ambiguity of **Syn** is allowed only if all the elements of $\{t \mid Ts.sn.t = x\}$ are mapped into the same element of **Den**. This implies that we can transform our ambiguous parsing algorithm into a unambiguous one by adding to it an arbitrary procedure which chooses one element from each set $Ts^{-1}.sn.x$. Which parsing tree we choose does not matter since they are all interpreted (or compiled) into the same element of **Den**.

A practical advantage of using ambiguous grammars in the description of syntax is twofold: First, such grammars are usually simpler and more intuitive (easier to read) than the equivalent unambiguous ones. Second, parsers constructed from them in the described way are usually faster than parsers derived from equivalent unambiguous grammars. Both these facts were already discussed in [1] although without any semantic considerations. A practical implementation of that idea may also be found in an algebraic-specification language OBJ (Goguen, Meseguer and Plaisted [15]) under the name of *coercions*.

Two more remarks are in order to complete our discussion of the derivation of syntax. First we should emphasize that the transformations of syntax by skeleton isomorphisms are not as innocent as this may appear at the first moment. Although they always lead to a correct and context-free syntax, they may considerably change the parsing complexity of the corresponding grammar. The grammar of the prototype syntax **Term** is always of type LL(1), hence leads to very low-cost parsing algorithms. This is, of course, due to the prefix-notation style of **Term**. If, however, we change a prefix notation to an infix notation—which is a typical isomorphic transformation—then we may raise the complexity of our grammar to a LL(k) with $k > 1$, or we may even go beyond the LL(k)-ness (see Example 6.1). This problem seems to require more research and therefore we shall not discuss it further in this paper.

Another remark concerns some limitations of our denotational model described by Fig. 3. As it turns out this model usually does not cover the step where we introduce so called "notational conventions", e.g. where we allow for an optional

omission of parentheses and establish some priorities between operators. In such and similar cases our new syntax, call it *post-final* and denote by $\mathbf{Syn}^{pf}$, is usually still a CF-algebra, but there is no homomorphism neither from **Term** nor from **Syn** into it. Instead, there exists a many-sorted function from $\mathbf{Syn}^{pf}$ onto **Syn** (Fig. 5). This function is in general not a homomorphism and describes a preprocessing of syntax performed prior to the stage of compilation. An example of such a function is described in Example 6.1 which follows later.

The mathematical semantics of $\mathbf{Syn}^{pf}$ is a combination of the nondenotational preprocessing *Pf* and the denotational semantics *Sd*. In general this is not a denotational semantics. In this situation whether we agree to say that the semantics of $\mathbf{Syn}^{pf}$ is "sufficiently denotational" is a pragmatic issue. On one hand it is quite obvious that a bad *Pf* may completely "spoil" the denotational effect of *Sd*. On the other hand, if the differences between **Syn** and $\mathbf{Syn}^{pf}$ are minor—such as e.g. the optional omission of parentheses—then the denotational advantages of **Syn** such as the clarity of the definition of semantics, the feasibility of structured programming, the ease in developing proof rules, are all inherited by $\mathbf{Syn}^{pf}$. The user of the system does not even need to see the grammar of $\mathbf{Syn}^{pf}$ nor the formal definition of *Pf*. In the manual of the system we give a full grammar of **Syn** along with the corresponding denotational semantics. The post-final syntax and *Pf* may be described informally. Their formal definitions are of interest only for the implementor of the system.

Example 6.1. Here we derive a syntax for the algebra of denotations defined in Example 5.1. This requires a formal establishment of the signature of that algebra together with the interpretation functions car and fun. What has been informally called a signature of **Den**, formally is only a metaexpression which defines the sorts and the arities of the operations of **Den**. When we want to formally talk about syntax, we have to precisely distinguish between the carriers and the operations of an algebra on one had and their names—i.e. the elements of the signature—on the

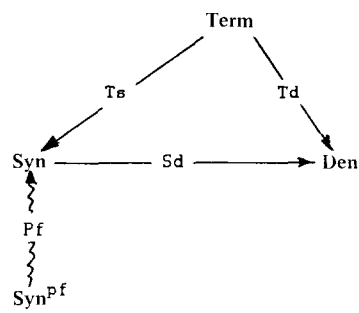

Fig. 5

other. Let then **Den** = (**Sig**, *car*, *fun*) and **Sig** = (*Sn*, *Fn*, *sort*, *arity*) where:

$$Sn = = \{\langle id\rangle, \langle ex\rangle, \langle co\rangle\},$$

$$Fn = \{\$create\_x, \$create\_y, \$evaluate, \ldots, \$while\};$$

$$car.\langle id\rangle = Identifier,$$
$$car.\langle ex\rangle = Evaluator,$$
$$car.\langle co\rangle = Executor;$$

$$fun.\$create\_x = create\_x,$$
$$\vdots$$
$$fun.\$while = while. \tag{6.1}$$

Functions *sort* and *arity* are implicit in (5.2). After having defined the signature of **Den** we have a unique corresponding algebra of terms **Term**. We describe it by a grammar $\mathbf{Gra}_1$ which can be effectively derived (in an obvious way) from the signature of **Den**:

$\langle id\rangle ::=$
  $\$create\_x$
  $|\$create\_y$

$\langle ex\rangle ::=$
  $\$evaluate(\langle id\rangle)$
  $|\$e\_one$
  $|\$e\_plus(\langle ex\rangle, \langle ex\rangle)$
  $\vdots$

$\langle co\rangle ::=$
  $\$assign(\langle id\rangle, \langle ex\rangle)$
  $|\$continue(\langle co\rangle, \langle co\rangle)$
  $|\$if(\langle ex\rangle, \langle co\rangle, \langle co\rangle)$
  $|\$while(\langle ex\rangle, \langle co\rangle).$

The prototype syntax which is defined by $\mathbf{Gra}_1$ is, of course, rather inconvenient. For instance, it forces us to write:

$$\$while(\$e\_less(\$create\_x, \$create\_y),$$
$$\quad \$assign(\$create\_x, \$e\_plus(\$create\_y, \$e\_one)))$$

where we would rather prefer to write something like:

$$\text{WHILE } x < y \text{ DO } x := y + 1 \text{ OD}. \tag{6.2}$$

In the next step we modify our prototype syntax by introducing an infix notation and simplifying keywords. This corresponds to a homomorphism

$$Ss_1 : \mathbf{Syn}_0 \to \mathbf{Syn}_1$$

which is defined below. We assume that *ide*, *exp* and *com* are metavariables ranging

over the corresponding carriers of **Term**. For a better readability of equations everywhere in the sequel we close the syntactic arguments of homomorphisms in square brackets. Notice that square brackets belong to metanotation whereas the parentheses "(" and ")" belong to the concrete syntax of the language which is being defined.

$$
\begin{aligned}
&Ss_1.\langle id\rangle.[\$create\_x] = x,\\
&Ss_1.\langle id\rangle.[\$create\_y] = y;\\
&\\
&Ss_1.\langle ex\rangle.[\$evaluate(ide)] = ide,\\
&Ss_1.\langle ex\rangle.[\$e\_one] = 1,\\
&Ss_1.\langle ex\rangle.[\$plus(exp_1, exp_2)] = (Ss_1.\langle ex\rangle.[exp_1] + Ss_1.\langle ex\rangle.[exp_2]);\\
&\vdots\\
&\\
&Ss_1.\langle co\rangle.[\$assign(ide, exp)] =\\
&\quad Ss_1.\langle id\rangle.[ide] := Ss_1.\langle ex\rangle.[exp],\\
&Ss_1.\langle co\rangle.[\$continue(com_1, com_2)] =\\
&\quad (Ss_1.\langle co\rangle.[com_1];\ Ss_1.\langle co\rangle.[com_2]),\\
&Ss_1.\langle co\rangle.[\$if(exp, com_1, com_2)] =\\
&\quad \text{IF } Ss_1.\langle co\rangle.[exp] \text{ THEN } Ss_1.\langle co\rangle.[com_1] \text{ ELSE } Ss_1.\langle co\rangle.[com_2] \text{ FI},\\
&Ss_1.\langle co\rangle.[\$while(exp, com)] =\\
&\quad \text{WHILE } Ss_1.\langle ex\rangle.[exp] \text{ DO } Ss_1.\langle co\rangle.[com] \text{ OD}.
\end{aligned}
$$

The algebra $\mathbf{Syn}_1$ is implicit in the definition of that homomorphism. Since our homomorphism is a skeleton isomorphism, the new syntax is correct and context-free. A grammar of that algebra may be derived directly (and mechanically) from the definition of Ss_1 and is the following:

$$
\begin{aligned}
&\langle id\rangle ::= x \mid y,\\
&\langle ex\rangle ::= \langle id\rangle \mid 1 \mid (\langle ex\rangle + \langle ex\rangle) \mid \ldots,\\
&\langle co\rangle ::= \langle id\rangle := \langle ex\rangle \mid (\langle co\rangle;\ \langle co\rangle)\\
&\qquad \mid \text{IF } \langle ex\rangle \text{ THEN } \langle co\rangle \text{ ELSE } \langle co\rangle \text{ FI}\\
&\qquad \mid \text{WHILE } \langle ex\rangle \text{ DO } \langle co\rangle \text{ OD}.
\end{aligned}
\tag{6.3}
$$

Observe that in the new syntax x corresponds to the former $\$create\_x$ or to $\$evaluate(x)$ depending whether it stands for an identifier or for an expression. The same concerns y. This may lead to a false conclusion that Ss_1 is not an isomorphism. In fact, however, it is an isomorphism since its "gluing effect" is split between two different sorts:

$$
\begin{aligned}
&Ss_1.\langle id\rangle.[\$create\_x] = x,\\
&Ss_1.\langle ex\rangle.[\$evaluate(x)] = x.
\end{aligned}
$$

In the future definition of semantics the meaning of x is always identified by the context where x appears.

It should also be noticed that although the grammar of $\mathbf{Syn}_0$ was of the type LL(1), the new one is not an LL(k) for any k, which is due to the infix notation of

expressions. This only shows that a skeleton-isomorphic transformation of syntax may substantially change the parsing category of a language.

The new syntax is much more readable than the former but it is still rather awkward since it forces us to write semantically superfluous parentheses such as e.g. in $((x := 1; y := 1); x := (x+1))$. We make therefore another modification of syntax and define the second homomorphism $Ss_2 : \mathbf{Syn}_1 \to \mathbf{Syn}_2$. This homomorphism should lead to the replacement of the production $\langle co\rangle \to (\langle co\rangle; \langle co\rangle)$ by the production $\langle co\rangle \to \langle co\rangle; \langle co\rangle$, leaving all the other productions unchanged. An explicit definition of that homomorphism is the following, where $(*)$ marks the critical equation:

$$
\begin{aligned}
& Ss_2.\langle id\rangle.[ide] = ide \quad (Ss_2.\langle id\rangle \text{ is an identity}),\\
& Ss_2.\langle ex\rangle.[exp] = exp \quad (Ss_2.\langle ex\rangle \text{ is an identity}),\\
& Ss_2.\langle co\rangle.[ide := exp] = Ss_2.\langle id\rangle.[ide] := Ss_2.\langle ex\rangle.[exp],\\
(*)\quad & Ss_2.\langle co\rangle.[(com_1; com_2)] = Ss_2.\langle co\rangle.[com_1]; Ss_2.\langle co\rangle.[com_2],\\
& Ss_2.\langle co\rangle.[\text{IF } exp \text{ THEN } com_1 \text{ ELSE } com_2 \text{ FI}] =\\
& \quad \text{IF } Ss_2.\langle co\rangle.[exp] \text{ THEN } Ss_2.\langle co\rangle.[com_1]\\
& \quad \text{ELSE } Ss_2.\langle co\rangle.[com_2] \text{ FI}\\
& Ss_2.\langle co\rangle.[\text{WHILE } exp \text{ DO } com \text{ OD}] =\\
& \quad \text{WHILE } Ss_2.\langle ex\rangle.[exp] \text{ DO } Ss_2.\langle co\rangle.[com] \text{ OD}.
\end{aligned}
$$

In applications it may be advisable that homomorphisms between context-free algebras are described by the corresponding transformations on grammars. This requires, of course, some care since not every transformation of a grammar corresponds to a homomorphism. We shall not tackle this issue here leaving it as a little research problem for the reader (cf. Section 7).

Of course, our second homomorphism is not an isomorphism and therefore we have to prove that $\mathbf{Syn}_2$ is not more ambiguous than **Den**. Since $\mathbf{Syn}_1$ is correct, this amounts to proving that Ss_2 is gluing not more than Sd_1 (cf. Fig. 4), i.e. that for any sort $sn \in \{\langle id\rangle, \langle ex\rangle, \langle co\rangle\}$ and any two elements syn_1 and syn_2 from the sn-carrier of $\mathbf{Syn}_2$:

$$Ss_2.sn.[syn_1] = Ss_2.sn.[syn_2] \quad \text{implies} \quad Sd_1.sn.[syn_1] = Sd_1.sn.[syn_2]. \tag{6.4}$$

The proof of this fact is rather long and therefore we postpone it till the end of this section.

The resulting syntax $\mathbf{Syn}_2$ is not yet quite acceptable since instead of writing (6.2) we have to write:

$$\text{WHILE } (x < y) \text{ DO } x := (y+1) \text{ OD}$$

with two pairs of "superfluous" parentheses. In this case, however, we cannot simply allow for the omission of parentheses in expressions, since this would lead to a too

ambiguous syntax. For instance, the expressions $((x+y)*x)$ and $(x+(y*x))$—which would reduce to the same expression $x+y*x$—have different denotations.

The described problem is quite typical and is usually resolved by assuming that parentheses in expressions are optional and that usual priorities between operators hold. In our case this leads to a next syntax $\mathbf{Syn}^{pf}$, which we call a *post-final syntax* and which is described by the following grammar:

$\langle id\rangle ::= x|y$
$\langle ex\rangle ::= \langle id\rangle|1|(\langle ex\rangle+\langle ex\rangle)|\langle ex\rangle+\langle ex\rangle|\ldots$
$\langle co\rangle ::=$(as before).

The relationship between the former and the new syntax is described by a many-sorted function:

$$Pf: \mathbf{Syn}^{pf} \to \mathbf{Syn}_2$$

which adds all missing parentheses to expressions in following the established priorities among operators. For instance:

$$\begin{aligned}
&Pf.\langle id\rangle.[x] = x,\\
&Pf.\langle ex\rangle.[x<y] = (x<y),\\
&Pf.\langle ex\rangle.[(x+y)*x+y] = (((x+y)*x)+y),\\
&Pf.\langle co\rangle.[\text{WHILE } x<y \text{ DO } x := (x+y)*x+y \text{ OD}] = \qquad (6.5)\\
&\quad \text{WHILE } Pf.\langle ex\rangle.[x<y] \text{ DO } Pf.\langle id\rangle.[x] := Pf.\langle ex\rangle.[(x+y)*x+y] \text{ OD} =\\
&\quad \text{WHILE } (x<y) \text{ DO } x := (((x+y)*x)+y) \text{ OD}\\
&\text{etc.}
\end{aligned}$$

Contrary to all former transformations of syntax, the many-sorted function Pf is not a homomorphism and therefore it cannot be given an inductive definition. First of all, $\mathbf{Syn}^{pf}$ has a different signature than $\mathbf{Syn}_2$. This follows from the fact that in the post-final syntax each data-type operator gives rise to two syntactic operations. E.g. the operator "+" allows for the construction of exp_1+exp_2 and of (exp_1+exp_2). For that reason Pf cannot be a homomorphism in the sense defined in Section 2. We could have thought, however, about a homomorphism in a more general sense which allows a so-called *morphism of signatures* (see e.g. [13]). In that case we are allowed to glue not only the elements of an algebra, but also its operations. A generalized homomorphism, similarly to the usual one, may be given an inductive definition and retains practically all the advantages of a denotational semantics in the old sense. In particular, the equivalence relation, which it defines in the source algebra by $x \equiv y$ iff (definition) $H.x = H.y$, is a congruence relation. It turns out, however, that Pf is not a homomorphism even in the generalized sense. Indeed, although

$$Pf.\langle ex\rangle.[x+y] = Pf.\langle ex\rangle.[(x+y)],$$

if we substitute $(x+y)$ for $x+y$ in a larger context, then the resulting expressions

need not be equivalent:

$$\begin{aligned} &Pf.\langle ex\rangle.[x+y*x]= \\ &\quad (x+(y*x)) \neq \\ &\quad ((x+y)*x)= \\ &\quad Pf.\langle ex\rangle.[(x+y)*x]. \end{aligned}$$

By the same argument we may show that the combination of *Pf* with *Sd* is not a homomorphism.

Since *Pf* is not a homomorphism it cannot be given an inductive definition. Observe, however, that this concerns only $Pf.\langle ex\rangle$. The other components retain the compositionality property as can be seen e.g. from the equation (6.5). The full definition of *Pf* splits, therefore, into two parts: a noncompositional definition of $Pf.\langle ex\rangle$ (e.g. by means of an automaton) and a structural-inductive definition of $Pf.\langle id\rangle$ and $Pf.\langle co\rangle$. We leave the details of this definition to the reader.

In applications, a formal definition of *Pf* is of interest only for compiler designers. For the user of the system the grammar of the final syntax $\mathbf{Syn}_2$ along with an informal description of *Pf* (regarded as notational conventions) will usually do.

On the other hand when we address the definition of a system to the user, then it may be appropriate to "unfold" in the definition of Sd_2 the definitions of the corresponding functions from **Den**. For instance, instead of writing:

$$\begin{aligned} &Sd_2.\langle ex\rangle.[(exp_1+exp_2)]= \\ &\quad e\_plus.\langle Sd_2.\langle ex\rangle.[exp_1],\ Sd_2.\langle ex\rangle.[exp_2]\rangle \end{aligned}$$

we write in such a case (cf. Example 5.1):

$$\begin{aligned} &E.[(exp_1+exp_2)].sta= \\ &\quad \textbf{let } val_1 = E.\langle exp_1\rangle.sta \textbf{ in} \\ &\quad \textbf{let } val_2 = E.\langle exp_2\rangle.sta \textbf{ in} \\ &\quad val_1 \notin Nat \rightarrow err, \\ &\quad val_2 \notin Nat \rightarrow err, \\ &\quad \mathrm{TRUE} \rightarrow plus.\langle val_1, val_2\rangle, \end{aligned}$$

where *E* stands for $Sd_2.\langle ex\rangle$. This is, of course, just a usual traditional form of a denotational definition.

With this remark we have completed the derivation of syntax and now we can proceed to the postponed proof of (6.4). We have to warn the reader that despite the simplicity of our example the proof is rather long since it indicates a certain general method of proving the correctness of a transformation of syntax where we allow for the omission of parentheses.

In order to prove (6.4) we have to explicitly define the homomorphism Sd_1. We recall that Sd_1 is the unique homomorphism which satisfies the equation:

$$Td = Ss_1 \cdot Sd_1.$$

Its definition is, therefore, implicit in the definitions of Ss_1 and *Td*. The former has been given explicitly earlier and the latter is implicit in equations (6.1) and (2.3).

Leaving to the reader all calculations—a job which normally should be done by a computer—we come out with the following explicit definition of Sd_1:

$$
\begin{aligned}
&Sd_1.\langle id\rangle.[x] = create\_x,\\
&Sd_1.\langle id\rangle.[y] = create\_y,\\
&Sd_1.\langle ex\rangle.[ide] = evaluate.ide,\\
&Sd_1.\langle ex\rangle.[1] = e\_one,\\
&Sd_1.\langle ex\rangle.[(exp_1 + exp_2)] =\\
&\quad e\_plus.\langle Sd_1.\langle ex\rangle.[exp_1], Sd_1.\langle ex\rangle.[exp_2]\rangle,\\
&\vdots\\
&Sd_1.\langle co\rangle.[ide := exp] = assign.\langle Sd_1.\langle id\rangle.[ide], Sd_1.\langle ex\rangle.[exp]\rangle,\\
&Sd_1.\langle co\rangle.[(com_1; com_2)] =\\
&\quad continue.\langle Sd_1.\langle co\rangle.[com_1], Sd_1.\langle co\rangle.[com_2]\rangle,\\
&\vdots
\end{aligned}
$$

Now observe that since the components $Ss_1.\langle id\rangle$ and $Ss_1.\langle ex\rangle$ are identities, we only have to prove (6.4) for $sn = \langle co\rangle$. In that proof we shall use some concepts and facts from the theory of term-rewriting systems. Since a full formal introduction of all these concepts would lead us beyond the scope of this paper, we decided to assume that the reader is familiar with the idea of term-rewriting systems and we refer him/her for details to two survey papers of Huet [19] and Klop [20].

Let *Com* denote the carrier of commands of $\mathbf{Syn}_1$ and let *com* possibly with indices denote an element of *Com*. Given a term-rewriting system (TRS) we say that com^r is a *reduct* of *com* in TRS, which we denote by

$$com \Rightarrow com^r$$

if com^r results from *com* by an application of one rule of TRS. We say that *com* is in a *normal form* if it has no reduct. By $\Rightarrow^*$ we denote the transitive and reflexive closure of $\Rightarrow$.

Now, the general idea of our proof consists in the construction of such a TRS which has the following properties:

(1) *Com* is closed under $\Rightarrow$,

(2) $\Rightarrow$ preserves the denotations of commands, i.e. if $com_1 \Rightarrow com_2$, then $Sd_1.\langle co\rangle.[com_1] = Sd_1.\langle co\rangle.[com_2]$,

(3) $\Rightarrow$ preserves the Ss_2-forms of commands, i.e. of $com_1 \Rightarrow com_2$, then $Ss_2.\langle co\rangle.[com_1] = Ss_2.\langle co\rangle.[com_2]$,

(4) each command has a unique normal form,

(5) if two commands in a normal form have the same Ss_2-form, then they are identical.

If a TRS with the properties (1)–(5) exists, then (6.4) is true. Indeed, take two commands com_1 and com_2 with the same Ss_2-form. By (4) and (1) there exist their unique normal forms com_1^N and com_2^N which by (2) have the same denotations as com_1 and com_2 and by (3) have the same Ss_2-forms respectively. By (5) our

normal-form commands are identical, hence they have the same denotations, hence also com_1 and com_2 have the same denotations.

The TRS which we shall use in our proof consists of only one rule:

$$((c_1; c_2); c_3) \to (c_1; (c_2; c_3)), \tag{6.6}$$

where the c_i are variables ranging over *Com*. Informally speaking this rule allows for the "application" of the associativity of the meaning of semicolon to commands. Now we shall show that (6.6) satisfies (1)-(5). The proofs of (1)-(3) are routine by structural induction and therefore omitted. We only mention that in the proof of (2) we use the associativity of the operation *continue* from **Den**.

Proof of (4). First observe that our TRS has a so-called *termination property*, i.e. that there are no infinite reduction sequences of the form $com_1 \Rightarrow com_2 \Rightarrow \cdots$. In our case the proof of this fact is quite simple (again by structural induction), but in the general case it may be far from trivial. There are, however, several standard techniques of proving the termination property of a TRS (see e.g. Klop [20]).

Due to the termination property the proof of (4) reduces to proving that each so-called *critical pair* of commands has a common normal form. In a general case a critical pair in a term-rewriting system is a pair $\langle p, q\rangle$ of terms constructed for a triple:

$$\langle (a_1 \to b_1), (a_2 \to b_2), u\rangle,$$

where $(a_1 \to b_1)$ and $(a_2 \to b_2)$ are rewriting rules, and u is a subterm of a_1 which is not a single variable and which is unifiable with a_2. The latter means that there exist two substitutions S and S' (mappings from variables to terms) such that $S(u) = S'(a_2)$. If this is the case, then we take the least common unification w of u and a_2 which has no common variables with a_1 and two corresponding substitutions S_1 and S_2:

$$S_1(u) = w = S_2(a_2).$$

Given this we can reduce $S_1(a_1)$ into two terms which constitute the critical pair $\langle p, q\rangle$ where p is the effect of the application of $(a_1 \to b_1)$ and therefore equals $S_1(b_1)$, and q is the effect of the application of $(a_2 \to b_2)$ and therefore results from $S_1(a_1)$ by the substitution of $S_2(b_2)$ for u.

If a critical pair exists for a given triple, then it is unique up to a permutation. In our case we have only one rule and therefore we construct a critical pair for that rule with itself. We have two instances of nontrivial subterms of the left-hand side of the rule. The first is the whole left-hand side and leads to a trivial critical pair where $p = q$. The second is $u = (c_1; c_2)$ in which case we have:

$$\begin{aligned} w &= ((c_{11}; c_{12}); c_2),\\ S_1(a_1) &= (((c_{11}; c_{12}); c_2); c_3),\\ p &= ((c_{11}; c_{12}); (c_2; c_3)),\\ q &= ((c_{11}; (c_{12}; c_2)); c_3). \end{aligned}$$

We leave it to the reader that p and q reduce to the common normal form $(c_{11}; (c_{12}; (c_2; c_3)))$. This completes the proof of (4). □

In our case the proof of (4) was rather simple due to the simplicity of the TRS. In more complicated cases we can use, however, a superposition algorithm for the generation of critical pairs and a Knuth-Bendix algorithm [21] for the completion of an underlying TRS, i.e. making it satisfy property (4).

We should mention in this place that in the general theory of term-rewriting systems one usually assumes that the underlying set of terms is a set of unambiguously parsable prefix terms. In our case commands are infix rather than prefix terms, but since the corresponding grammar is unambiguous we can apply the general theory without major modifications. It is not known to the author whether the theory extends to the case of an ambiguous syntax of terms. So far, therefore, we may only advise that the nonisomorphic modifications of syntax be performed in one step, at least if one wants to use our method of proving (6.4).

Proof of (5) (by structural induction). First we introduce an auxiliary concept. A command is said to be *open* (open for an exchange of its subcommands with a context) if it is of the form $(com_1; com_2)$, and is said to be *closed* otherwise. Let com_1 and com_2 be in a normal form and have the same Ss_2-form, i.e.

$$Ss_2.\langle co\rangle.[com_1] = Ss_2.\langle co\rangle.[com_2].$$

This implies that com_1 and com_2 are of the same grammatical category (assignment, if, ";" or while), hence either both are closed or both are open. We have three cases now:

Case 1. com_1 is an assignment, in which case we are done.

Case 2. com_1 is not an assignment but is closed, in which case we apply the inductive assumption.

Case 3. com_1 is open and therefore also com_2 is open and

$$com_1 = (com_{11}; com_{12}),$$
$$com_2 = (com_{21}; com_{22}).$$

In that case the following facts can be shown:

(i) com_{11} and com_{21} are closed and in a normal form (obvious),

(ii) com_{12} and com_{22} are in a normal form (obvious),

(iii) com_{11} and com_{21} have the same Ss_2-form (since they are closed and com_1 with com_2 have the same Ss_2-forms) and therefore by inductive assumption they are equal.

(iv) com_{12} and com_{22} have the same Ss_2-form (by (iii)) and therefore by the inductive assumption they are equal.

This completes the proof of (5) and therefore also of (6.4). □

7. Problems related to the derivation of syntax

Our approach to the derivation of a custom-made syntax (Sections 4 and 6) leaves open many theoretical and practical problems. In this section we try to identify some of them. We split our list of problems into two groups: general problems and problems related to the development of a computer-support system.

7.1. General problems

Problem 7.1.1. Investigate if it may be of a practical or theoretical interests to allow non-context-free algebras in the definitions of syntax. For instance, in the Example 6.1 we may define syntax $\mathbf{Syn}_3$ by such a modification of $\mathbf{Syn}_2$ where all operations of sort $\langle id \rangle$ and $\langle ex \rangle$ remain unchanged and where the operations of sort $\langle co \rangle$ are defined as follows:

$$\textit{fun-s.\$assign.}\langle \textit{ide}, \textit{exp} \rangle = \textit{ide} := \textit{rop.exp},$$
$$\textit{fun-s.\$if.}\langle \textit{exp}, \textit{com}_1, \textit{com}_2 \rangle =$$
$$\quad \text{IF } \textit{rop.exp} \text{ THEN } \textit{com}_1 \text{ ELSE } \textit{com}_2 \text{ FI},$$
$$\textit{fun-s.\$while.}\langle \textit{exp}, \textit{com} \rangle = \text{WHILE } \textit{rop.exp} \text{ DO } \textit{com} \text{ OD}.$$

The function *rop* (remove outer parentheses) is a function which removes outer parentheses from expressions, and *fun-s* is a function which assigns operations to operations' symbols in $\mathbf{Syn}_3$. As is easy to check, the new syntax is an isomorphic image of the former (hence is correct!), but since the corresponding isomorphism has no skeleton, our final algebra is not context-free. Formally speaking it cannot be described by a context-free grammar, but of course a part of it (the carriers of $\langle id \rangle$ and $\langle ex \rangle$ has a context-free grammar and the remaining part may be described by something like a "CF-grammar with functions". Observe also that given a parser (compiler) for $\mathbf{Syn}_2$ it is not difficult to construct one for $\mathbf{Syn}_3$.

Problem 7.1.2. Since in the applications we may wish to use ambiguous grammars, for which we still want to develop efficient parsers (cf. our remarks in Section 6), define and investigate parsing-complexity categories, such as e.g. LR(k)-ness or LL(k)-ness, for ambiguous grammars.

Problem 7.1.3. Identify a class of typical (in applications) transformations of algebras and grammars which correspond to homomorphisms (isomorphisms). This may include the renaming of selectors, passing from prefix to infix notation, the permutation of nonterminals in the right-hand sides of productions, etc. Classify these transformations with respect to their effect on:

- context-freeness,
- ambiguity,
- parsing category,
- other properties (?)

of target syntax.

Problem 7.1.4. Characterize the class of monoskeleton algebras. If there are two grammars for the same algebra, can they belong to different parsing categories?

Problem 7.1.5. Our definition of a context-free algebra does not allow for a permutation of arguments by an operation. This may be repaired by generalizing the concept of a homomorphism as suggested at the end of Section 4. Investigate this solution.

Problem 7.1.6. In the life-cycle of a software system we usually modify the system by adding and/or removing some operations. On the ground of our model this corresponds to the extensions and the restrictions of the algebra of denotations. Of course, when we change the algebra **Den** we also have to change the algebra **Syn**, and we want to do this in such a way that as much as possible of the old syntax remains legal and means the same as before. Investigate this problem and try to characterize such transformations of **Den** which lead to as much as possible "conservative" transformations of **Syn**.

7.2. Problems related to the development of a computer support

The process of syntax derivation consists of several steps where we transform algebras and/or grammars. In each step we have to prove that our target syntax has several properties. This leads to the necessity of developing appropriate algorithms and specification techniques.

Problem 7.2.1. How should we specify (represent) homomorphisms between algebras in the process of syntax design? There seems to be two generally different ways of doing this. We can either specify a homomorphism by equations, in which case the computer has to generate the target algebra or grammar, or we can specify it implicitly by modifying the source algebra, in which case the computer has to check whether our modification indeed corresponds to a homomorphism.

Problem 7.2.2. Develop algorithms which support the following logical and transformational steps in the derivation of $\mathbf{Syn}_i$ from $\mathbf{Syn}_{i-1}$ (cf. Fig. 4):

(1) Check the following properties of Ss_i:
- is Ss_i a skeleton homomorphism?
- is Ss_i an isomorphism?
- is Ss_i not gluing too much (is $\mathbf{Syn}_i$ correct)?

(2) Check the parsing category of $\mathbf{Syn}_i$ given the parsing category of $\mathbf{Syn}_{i-1}$.

(3) Construct the homomorphism $Ts_i : \mathbf{Term} \to \mathbf{Syn}_i$ or the inverse of this homomorphism, i.e. the parsing transformation. This may be constructed either only once in the last step, or by modifying Ts_{i-1} to Ts_i in each step. Observe that if the definition of **Den** has been written in an implemented metalanguage, than the inverse of Ts_i may be regarded (used as an interpreter, since with every syntactic object it

assigns an executable expression. At the same time, however, it may also be interpreted as a parser.

8. A little case study: The development of a word processor

In this section we discuss an example which is a little more realistic than that discussed in Sections 5 and 6: the development of a simple wordprocessor. For the sake of brevity we describe only a few typical functions of such a processor and we omit any formal consideration of the problem of the screen representation of a document. We start from the development of the algebra of data, where the major concept is that of a document.

A *document* appears to the user as a pair of strings of characters separated by a *cursor*. The string which precedes the cursor is called the *prefix* and the string which follows the cursor is called the *postfix*. If the postfix is nonempty, then its first character is called the *current character* and is pointed to by the cursor.

A document may be elaborated in one of two *regimes*: development (DEV) or marking (MARK). In the former regime we have available such functions as e.g. typing a character into the document, shifting the cursor, pasting a previously stored block into a document, saving the document in the memory, etc. In the latter, we may mark a part of a document and then either delete it or store it for the future copying or moving. The marked part is called a *block*. In the MARK regime a block appears on the screen as a highlighted postfix of the prefix (i.e. the marking command works only forwards). In the DEV regime the block is not displayed on the screen but is stored in the memory.

In the regime of development the user may choose between two different working *modes*: inserting (INS) and overtyping (OVE). When the system is switched from DEV to MARK the information about the actual mode is stored for the future use.

We assume that the system may communicate some *messages* to the user, especially in the case of errors.

We start our formal definition from defining the domain of documents:

$$
\begin{aligned}
&doc : Document = Prefix \times Block \times Postfix \times Mode \times Regime \times Message,\\
&pre : Prefix = Text,\\
&post : Postfix = Text,\\
&block : Block = Text,\\
&text : Text = Character^{c^*},\\
&char : Character = \{a, b, \ldots, A, B, \ldots, 1, 2, \ldots, 0, !, @, \#, \ldots\},\\
&mode : Mode = \{\text{INS}, \text{OVE}\},\\
&reg : Regime = \{\text{DEV}, \text{MARK}\},\\
&mes : Message = \{\text{OK}\} \mid Error,\\
&err : Error = \ldots .
\end{aligned}
$$

The elements of the domain Error will be established later in the course of defining the actions of our system. Now, let us start from basic operations on documents that we want to have in the system. First we choose their names, sorts and arities, i.e. we define the signature of the algebra **Dat**:

$$
\begin{aligned}
&\textit{create-ed} : \rightarrow \textit{Document}\\
&\quad \text{(create empty document)},\\
&\textit{type-a} : \textit{Document} \rightarrow \textit{Document}\\
&\quad \text{(type character ``}a\text{'' (}\textit{type-char}\text{ for any }\textit{char} \in \text{Character))},\\
&\textit{in-toggle} : \textit{Document} \rightarrow \textit{Document}\\
&\quad \text{(toggle between INS and OVE)},\\
&\textit{set-mark} : \textit{Document} \rightarrow \textit{Document}\\
&\quad \text{(set the MARK regime)},\\
&\textit{f-shift} : \textit{Document} \rightarrow \textit{Document}\\
&\quad \text{(shift cursor forwards one character)},\\
&\textit{b-shift} : \textit{Document} \rightarrow \textit{Document}\\
&\quad \text{(shift cursor backwards one character)},\\
&\textit{copy} : \textit{Document} \rightarrow \textit{Document}\\
&\quad \text{(store block for future copying)},\\
&\textit{move} : \textit{Document} \rightarrow \textit{Document}\\
&\quad \text{(store block for future moving)},\\
&\textit{del} : \textit{Document} \rightarrow \textit{Document}\\
&\quad \text{(delete block)},\\
&\textit{paste} : \textit{Document} \rightarrow \textit{Document}\\
&\quad \text{(insert block into the text)}.
\end{aligned}
\tag{8.1}
$$

Below we define our operations. Let "$\hat{}$" denote the concatenation of strings and let *head* and *tail* denote the usual functions on strings with $\textit{head}.\langle\,\rangle = \langle\,\rangle$ and $\textit{tail}.\langle\,\rangle = \langle\,\rangle$, where $\langle\,\rangle$ denotes the empty string.

$$\textit{create-ed}.\langle\,\rangle = \langle\langle\,\rangle, \langle\,\rangle, \langle\,\rangle, \text{INS}, \text{DEV}, \text{OK}\rangle.$$

This operation creates a document with an empty prefix, empty block and empty postfix and with the default mode INS and default regime DEV. The initial message is, of course, "OK".

$$
\begin{aligned}
&\textit{type-a}.\langle \textit{pre}, \textit{block}, \textit{post}, \textit{mode}, \textit{reg}, \textit{mes}\rangle =\\
&\quad \textit{reg} = \text{MARK} \rightarrow \langle \textit{pre}, \textit{block}, \textit{post}, \textit{mode}, \textit{reg}, \text{INVALID OPERATION}\rangle,\\
&\quad \textit{mode} = \text{INS} \rightarrow \langle \textit{pre}\hat{}\langle a\rangle, \textit{block}, \textit{post}, \textit{mode}, \textit{reg}, \text{OK}\rangle,\\
&\quad \textit{mode} = \text{OVE} \rightarrow \langle \textit{pre}\hat{}\langle a\rangle, \textit{block}, \textit{tail.post}, \textit{mode}, \textit{reg}, \text{OK}\rangle.
\end{aligned}
$$

This operation is available only in the DEV regime and depending on the actual mode it either inserts an "a" at the current position or overtypes with "a" the current character. We assume to have such an operation for each character separately. This means that the user is able to type any character from the keyboard into the

document.

in-toggle.⟨*pre*, *block*, *post*, *mode*, *reg*, *mes*⟩ =
reg = MARK → ⟨*pre*, *block*, *post*, *mode*, *reg*, INVALID OPERATION⟩,
mode = INS → ⟨*pre*, *block*, *post*, OVE, *reg*, *mes*⟩,
mode = OVE → ⟨*pre*, *block*, *post*, INS, *reg*, *mes*⟩.

This operation is available in the DEV regime and toggles between the modes INS and OVE.

set-mark.⟨*pre*, *block*, *post*, *mode*, *reg*, *mes*⟩ =
reg = DEV → ⟨*pre*, ⟨ ⟩, *post*, *mode*, MARK, OK⟩,
reg = MARK → ⟨*pre*, *block*, *post*, *mode*, *reg*,
YOU ARE ALREADY IN MARK REGIME⟩.

This operation is available in the DEV regime. It switches the regime into MARK and empties the block.

f-shift.⟨*pre*, *block*, *post*, *mode*, *reg*, *mes*⟩ =
reg = DEV → ⟨*pre*^(*head.post*), *block*, *tail.post*, *mode*, *reg*, OK⟩,
reg = MARK → ⟨*pre*, *block*^(*head.post*), *tail.post*, *mode*, *reg*, OK⟩.

This operation is available in both regimes. It shifts the cursor one character forwards unless the postfix is empty. In the DEV mode this results the shifting of the first character from the postfix to the prefix; in the MARK mode—the shifting of the same character to the block. In the latter case the part of the text stored in the block is highlighted on the screen. The operation of shifting the cursor backwards is defined analogously. Of course, in the MARK regime backwards shifts demark the formerly marked text.

The operations *copy*, *move* and *del* are available only in the regime MARK and all of them change the regime to DEV. The first adds the content of the block to the prefix (i.e. leaves the portion of the formerly highlighted text in the document) and stores the block for the future use:

copy.⟨*pre*, *block*, *post*, *mode*, *reg*, *mes*⟩ =
reg = MARK → ⟨*pre*^*block*, *block*, *post*, *mode*, DEV, OK⟩,
reg = DEV → ⟨*pre*, *block*, *post*, *mode*, *reg*, INVALID OPERATION⟩.

The operation *move* stores the block but does not add it to the prefix, and the operation *del* only empties the block. Formal definitions are left to the reader. The operation *paste* is available in the DEV regime and copies the block into the document at the current position. Then it empties the block:

paste.⟨*pre*, *block*, *post*, *mode*, *reg*, *mes*⟩ =
reg = DEV → ⟨*pre*^*block*, ⟨ ⟩, *post*, *mode*, DEV, OK⟩,
reg = MARK → ⟨*pre*, *block*, *post*, *mode*, *reg*, INVALID OPERATION⟩.

Observe that all our operations do not depend on the message component of the document and whenever an error message is generated the other components of the

document remain unchanged. This means that the only reaction of the system to a user's error is to display an error message without doing nothing with the rest of the document. An error does not suspend the system and any next correct move of the user generates an "OK" message.

With this we have completed the development of the algebra **Dat** of data. Now we proceed to the second stage where we define a computer system based on that algebra. We assume that in this system we should be able to keep a current document in computer memory, to perform all defined operations on it, to save the textual part of the document for the future use and to retrieve the formerly saved file for elaboration. Later we shall also show how to enrich our system by the mechanism of procedures. So far we define the following domains:

$$sto : Store = Identifier \to_m Text,$$
$$ide : Identifier = Character,$$
$$sta : State = Document \times Name \times Store,$$
$$name : Name = Identifier,$$
$$exe : Executor = State \to State,$$
$$ini : Initiator = Store \to State,$$
$$ter : Terminator = State \to Store.$$

In the stores we store text files named by identifiers. For simplicity we assume that identifiers are single characters. A state (of the system) consists of a document, a name of that document and a store. Actions which our system may perform are of three types: *executors* which modify states and which are used during a session with the system, *initiators* which create states from stores and which are used to initiate a session and *terminators* which reduce a state to a store and which are used to close a session.

Below we define the constructors of executors, initiators and terminators. Analogously as in the Example 5.1 (cf. (5.1)) we first "lift" all operations of the algebra **Dat** to the level of the constructors of executors. For each non-zero-ary operation on documents,

$$oper : Document \to Document,$$

we define the following zero-ary constructor of executors:

$$co\text{-}oper : \to Executor,$$
$$co\text{-}oper.\langle\,\rangle.\langle doc, name, sto\rangle = \langle oper.doc, name, sto\rangle.$$

Each such *a* constructor generates an executor which modifies only the document component of a state and does that by applying the corresponding operation on documents. We have, therefore, the following executor constructors: *co-type-a*, *co-in-toggle*, *co-copy*, etc. We do not assign such a constructor to *create-ed* since this operation will be used later to define an initiator.

The constructors which have been defined so far are zero-ary, i.e. they correspond to primitive executors. Each such an executor is "activated" by the user by only

communicating the name of the corresponding constructor to the system, but without giving any arguments. Below we define two constructors which are not lifted operations of **Dat** and which have nontrivial arities:

rename : *Identifier* → *Executor*
(rename the current file),
extcopy : *Identifier* → *Executor*
(external copy);

$rename.ide.\langle doc, name, sto\rangle = \langle doc, ide, sto\rangle;$

$extcopy.ide.\langle doc, name, sto\rangle =$
let $\langle pre, block, post, mode, reg, mes\rangle = doc$ **in**
$sto.ide = ? \to \langle\langle pre, block, post, mode, reg, \text{NO SUCH FILE}\rangle, name, sto\rangle,$
$\text{TRUE} \to \langle\langle pre\hat{}(sto.ide), block, post, mode, reg, \text{OK}\rangle, name, sto\rangle.$

Here $sto.ide = ?$ is a shorthand for "**not** $sta \in dom.sto$" and therefore is satisfied whenever *ide* does not belong to the domain of *sto*. The first operation changes the name of the current document, the second copies an indicated textfile into the current position of the current document.

Now we proceed to the constructors of initiators and terminators. We assume to have four following constructors:

create : *Identifier* → *Initiator*
(create a new document),
edit : *Identifier* → *Initiator*
(adit an existing document),
save : → *Terminator*
(save the current document),
quit : → *Terminator*
(quit without saving).

For the sake of brevity we give only two of the corresponding definitions:

$create.ide.sto = \langle create\text{-}ed.\langle\,\rangle, ide, sto\rangle,$

$save.\langle\langle pre, block, post, mode, reg, mes\rangle, name, sto\rangle =$
$sto[\,pre\hat{}post/name\,].$

First operation creates a state with an empty document. The other, stores the concatenation of the prefix and the postfix of the current document under the given name in the store.

At the end we have to define the constructors of identifiers since the latter appear as the arguments of some of our formerly defined constructors. Since identifiers are just characters, we assugn with each character a corresponding constructor:

makeide-a : → *Identifier*
$makeide\text{-}a.\langle\,\rangle = a$
⋮

This completes the development of the algebra **Sys**. Now we select **Den** in **Sys** by selecting the operations which we want to make accessible to the user:

$$
\begin{aligned}
&\textit{makeide-a} : \rightarrow \textit{Identifier} \quad \text{(analogously for other characters)},\\
&\textit{co-type-a} : \rightarrow \textit{Executor} \quad \text{(analogously for other characters)},\\
&\vdots\\
&\textit{co-paste} : \rightarrow \textit{Executor},\\
&\textit{rename} : \textit{Identifier} \rightarrow \textit{Executor},\\
&\textit{extcopy} : \textit{Identifier} \rightarrow \textit{Executor},\\
&\textit{create} : \textit{Identifier} \rightarrow \textit{Initiator},\\
&\textit{edit} : \textit{Identifier} \rightarrow \textit{Initiator},\\
&\textit{save} : \rightarrow \textit{Terminator},\\
&\textit{quit} : \rightarrow \textit{Terminator}.
\end{aligned}
\tag{8.2}
$$

In contrast to the signature (5.2) in our former example, the present signature gives rise to a very poor syntax reduced to a finite list of the names of actions available in the system. It is so because there are no programming facilities in the system, e.g. there are no constructors neither of executors nor of initiators, nor of terminators that take these actions as arguments. In such a case the derivation of syntax is trivial and therefore we omit this step in our example.

Now we may proceed to extending of our model by procedures. We assume that by a procedure we shall mean any sequence of executors stored for a future execution. Of course, in the abstract model we shall not *store* the sequences of executors themselves, but their intended effects, hence state-to-state transformations. We introduce, therefore, two new domains:

$$
\begin{aligned}
&\textit{proc} : \textit{Procedure} = \textit{State} \rightarrow \textit{State},\\
&\textit{p-sto} : \textit{Proc-store} = \textit{Identifier} \rightarrow_{\mathrm{m}} \textit{Procedure},
\end{aligned}
$$

and we rename the former domain *Store* to

$$
\textit{f-sto} : \textit{File-store} = \textit{Identifier} \rightarrow_{\mathrm{m}} \textit{Text}.
$$

Here the reader may wonder why we have introduced separate domains of stores for procedures and for textfiles. The reason is such that procedures should have no access to procedure stores (should not be self-applicable), whereas executors must have such an access in order to execute procedure calls. This will be seen better when we define our new domains and constructors:

$$
\begin{aligned}
&\textit{sta} : \textit{State} = \textit{Document} \times \textit{Name} \times \textit{File-store},\\
&\textit{env} : \textit{Environment} = \textit{State} \times \textit{Proc-store},\\
&\textit{sto} : \textit{Store} = \textit{File-store} \times \textit{Proc-store};
\end{aligned}
$$

$$
\begin{aligned}
&\textit{exe} : \textit{Executor} = \textit{Environment} \rightarrow \textit{Environment},\\
&\textit{dec} : \textit{Declarator} = \textit{Environment} \rightarrow \textit{Environment},\\
&\textit{p-body} : \textit{Proc-body} = \textit{Environment} \rightarrow \textit{Environment},\\
&\textit{ini} : \textit{Initiator} = \textit{Store} \rightarrow \textit{Environment},\\
&\textit{ter} : \textit{Terminator} = \textit{Environment} \rightarrow \textit{Store}.
\end{aligned}
$$

States in the new sense are the same as before (since File-store is the same as the former Store) and environments consist of a state and a procedure store. Now, executors have access to the whole environment but procedures only to its state component. This protects procedures against self-applicability and makes our set of domain equations solvable in set theory (see [10] and [9]). The domains of executors, declarators and procedure-bodies are identical, but since they are going to have different associated constructors, their corresponding reachable parts in the algebra of denotations will be different.

Now we are ready to specify the signature of the new algebra **Den**. It consists of the signature (8.2) of the former **Den**—but with the new meanings of domains—plus the following operations:

$$\begin{aligned}
&\mathit{make\text{-}body} : \mathit{Executor} \to \mathit{Proc\text{-}body},\\
&\mathit{continue} : \mathit{Proc\text{-}body} \times \mathit{Proc\text{-}body} \to \mathit{Proc\text{-}body},\\
&\mathit{call} : \mathit{Identifier} \to \mathit{Executor},\\
&\mathit{declare} : \mathit{Identifier} \times \mathit{Proc\text{-}body} \to \mathit{Declarator}.
\end{aligned}$$

Given this new signature we have to redefine the "old" constructors and to define the new ones. The former task is quite routine since the new constructors define executors, initiators and terminators which "do the same as before", but now may receive in the arguments and/or return in the values also procedure-stores. E.g. in the new version the constructor *rename*, given an identifier produces an executor which does the same as before with the state and keeps the procedure-store component unchanged:

$$\begin{aligned}
&\mathit{rename.ide.}\langle\langle \mathit{doc}, \mathit{name}, \mathit{f\text{-}sto}\rangle, \mathit{p\text{-}sto}\rangle =\\
&\quad \langle\langle \mathit{doc}, \mathit{ide}, \mathit{f\text{-}sto}\rangle, \mathit{p\text{-}sto}\rangle.
\end{aligned}$$

Now we define new constructors. The first makes a procedure from an executor:

$$\mathit{make\text{-}body.com} = \mathit{com}.$$

Observe that this is not an identity function, since the type of its argument is different from the type of its value.

$$\mathit{continue.}\langle \mathit{p\text{-}body}_1, \mathit{p\text{-}body}_2\rangle = \mathit{p\text{-}body}_1 \cdot \mathit{p\text{-}body}_2.$$

This is the usual ";" operation (as e.g. in Example 5.1) just that in our case it is applicable to procedure bodies rather than to executors. This means that we may declare as a procedure a sequence of executors to be executed one after another, but such a sequence is not itself an executor.

Procedure calls are defined as follows:

$$\begin{aligned}
&\mathit{call.ide.}\langle \mathit{sta}, \mathit{p\text{-}sto}\rangle =\\
&\quad \textbf{let } \langle\langle \mathit{pre}, \mathit{block}, \mathit{post}, \mathit{mod}, \mathit{reg}, \mathit{mes}\rangle, \mathit{name}, \mathit{f\text{-}sto}\rangle = \mathit{sta} \textbf{ in}\\
&\quad \mathit{p\text{-}sto.ide} = ? \to \langle\langle\langle \mathit{pre}, \mathit{block}, \mathit{post}, \mathit{mod}, \mathit{reg}, \text{NO SUCH PROCEDURE}\rangle,\\
&\qquad\qquad \mathit{name}, \mathit{f\text{-}sto}\rangle, \mathit{p\text{-}sto}\rangle,\\
&\quad \textbf{let } \mathit{proc} = \mathit{p\text{-}sto.ide} \textbf{ in}\\
&\quad \langle \mathit{proc.sta}, \mathit{p\text{-}sto}\rangle.
\end{aligned}$$

A procedure call generates an executor that executes a procedure stored under *ide* in the current procedure-store. If there is no such procedure then the call issues an appropriate error message.

Now we come to the problem of defining procedure declarators. Since procedure calls are executors, they may appear in procedure bodies and therefore we have to decide if they are to be interpreted as nonrecursive calls or as recursive calls. Let us discuss both possibilities. In the former case we define:

$$\begin{aligned}&\textit{declare}.\langle \textit{ide}, \textit{p-body}\rangle.\langle \textit{sta}, \textit{p-sto}\rangle = \\ &\quad \textbf{let } \textit{proc.sta} = \textit{first}.(\textit{p-body}.\langle \textit{sta}, \textit{p-sto}\rangle) \textbf{ in} \\ &\quad \langle \textit{sta}, \textit{p-sto}[\textit{proc}/\textit{idea}]\rangle.\end{aligned} \tag{8.3}$$

Observe that the metavariable *sta*, which appears in the **let–in** subdefinition, is bound and therefore represents a call-time state, whereas *p-sto* in the same subdefinition is free and therefore represents a declaration-time store. The declared procedure when applied to a call-time state expands it to an environment by adding the declaration-time procedure-store and then applies the body to that environment. From the resulting environment it takes the state-component as the new state. We emphasize that all procedure names which appear in the inner calls of *proc* are referred to the declaration-time *p-sto* and therefore have *static bindings*. Consequently, if we try to call a procedure which calls itself, then we either have an error signal "NO SUCH PROCEDURE", or we call another procedure with the same name, if such has been declared prior to the declaration of our procedure. In any case such a call does not lead to a recursion.

If we want to allow for recursive procedures, we define our constructor as follows:

$$\begin{aligned}&\textit{declare}.\langle \textit{ide}, \textit{p-body}\rangle.\langle \textit{sta}, \textit{p-sto}\rangle = \\ &\quad \textbf{letrec } \textit{proc.sta} = \textit{first}.(\textit{p-body}.\langle \textit{sta}, \textit{p-sto}[\textit{proc}/\textit{ide}]\rangle) \textbf{ in} \\ &\quad \langle \textit{sta}, \textit{p-sto}[\textit{proc}/\textit{ide}]\rangle.\end{aligned} \tag{8.4}$$

In this case *proc* is defined by a fixed-point equation with respect to its call or calls. This is, of course, quite a routine definition. It corresponds to the mechanism of so called *static recursion* since, as in the former case, the binding of all procedure names in the body is static.

In the present model a procedure may call recursively only itself. Mutually recursive declarations will result in an error signal at the time of the call. If we want to allow for mutual recursion, we have to define a new declarator-constructor which takes a tuple of procedure-bodies and elaborates them all at once. We may also expand our model by allowing procedures with parameters [10].

With this remark we have completed the definition of our new algebra **Den**. Since the derivation of syntax does not lead, in this case, to any problems which have not been discussed in Example 6.1, we leave this step to the reader as an exercise.

Acknowledgement

Special thanks should be addressed to Marek Ryćko, who pointed out a technical mistake in [7], and to Brian Monahan who's comments about the early version of the paper cannot be overestimated. The author also wishes to acknowledge many very useful remarks communicated to him by the referees of *Science of Computer Programming*. Last but not least the members of the Group MetaSoft contributed by their constructive criticism about the whole approach to the present shape of the paper.

References

[1] A.V. Aho, S.C. Johnson and J.D. Ullman, Deterministic parsing of ambiguous grammars, *Comm. ACM* **18** (1975) 441-452.
[2] D. Bjørner and C.B. Jones, *The Vienna Development Method: The Meta Language*, Lecture Notes in Computer Science **61** (Springer, Berlin, 1978).
[3] D. Bjørner and C.B. Jones, *Formal Specification and Software Development* (Prentice-Hall, Englewood Cliffs, NJ, 1982).
[4] D. Bjørner and O.N. Oest, eds., *Towards a Formal Description of ADA*, Lecture Notes in Computer Science **98** (Springer, Berlin, 1980).
[5] A. Blikle, Noninitial algebraic semantics, in: D. Bjørner, ed., *Formal Software Development Methods: Combining Specification Methods* (Springer, Berlin, 1984).
[6] A. Blikle, *MetaSoft Primer: Toward a Metalanguage for Applied Denotational Semantics*, Lecture Notes in Computer Science **288** (Springer, Berlin, 1987).
[7] A. Blikle, Denotational engineering or from denotations to syntax, in: D. Bjørner, C.B. Jones, M. Mac an Airchinnigh and E.J. Neuhold, eds., *VDM: A Formal Method at Work*, Lecture Notes in Computer Science **252** (Springer, Berlin, 1987).
[8] A. Blikle, Denotational engineering or from denotations to syntax, ICS PAS Reports no 605, Institute of Computer Science, Polish Academy of Science, Warsaw (1987).
[9] A. Blikle, A guided tour of the mathematics of MetaSoft '88, *Inform. Process. Lett.* **29** (1988) 81-86.
[10] A. Blikle and A. Tarlecki, Naive denotational semantics, in: R.E.A. Manson, ed., *Information Processing 83* (North-Holland, Amsterdam, 1983).
[11] P.M. Cohn, *Universal Algebra* (Reidel, Dordrecht, Netherlands, 1981).
[12] N. Dershowitz, Computing with rewrite systems, *Inform. Control* **65** (1985) 122-157.
[13] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification* **1** (Springer, Berlin, 1985).
[14] J.A. Goguen, Abstract errors for abstract data types, in: E. Neuhold, ed., *Formal Description of Programming Concepts* (North-Holland, Amsterdam, 1978).
[15] J. Goguen, J. Meseguer and D. Plaisted, Programming with parameterized abstract objects in OBJ, in: D. Ferrari, M. Bolognani and J. Goguen, eds., *Theory and Practice of Software Technology* (North-Holland, Amsterdam, 1983) 163-194.
[16] J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright, Initial algebra semantics and continuous algebras, *J. ACM* **24** (1977) 68-95.
[17] M.J.C. Gordon, *The Denotational Description of Programming Languages* (Springer, Berlin, 1979).
[18] M.A. Harrison, *Introduction to Formal Language Theory* (Addison-Wesley, Reading, MA, 1978).
[19] G. Huet, Confluent reductions: Abstract properties and applications of term rewriting systems, *J. ACM* **27** (1980) 797-821.
[20] J.W. Klop, Term rewriting systems: A tutorial, *Bull. EATCS* **32** (1987) 143-182.
[21] D. Knuth and P. Bendix, Simple word problems in universal algebras, in: J. Leech, ed. *Computational Problems in Abstract Algebra* (Pergamon Press, Elmsford, 1970) 263-279.
[22] J. McCarthy, A basis for a mathematical theory of computation, in: P. Braffort and D. Hirschberg, eds., *Computer Programming and Formal Systems* (North-Holland, Amsterdam, 1967) 33-70.

[23] J. Meseguer and A.J. Goguen, Initiality, induction and computability, in: M. Nivat and J. Reynolds, eds., *Application of Algebra to Language Definition and Compilation* (Cambridge University Press, Cambridge, 1985) 459-541.

[24] P. Mosses, The mathematical semantics of Algol 60, Technical Monograph PRG-12, Oxford University (1974).

[25] P. Mosses and D.A. Watt, The use of action semantics, in: M. Wirsing, ed., *Formal Description of Programming Concepts* **III** (North-Holland, Amsterdam, 1987).

[26] D.T. Sannella and A. Tarlecki, An observational equivalence and algebraic specification, in: *Proceedings 10th Colloquium on Trees in Algebra and Programming, Joint Conference TAPSOFT*, Lecture Notes in Computer Science **185** (Springer, Berlin, 1985) 209-263.

[27] D. Schmidt, *Denotational Semantics: A Methodology for Language Development* (Allyn and Bacon, Boston, MA, 1986).

[28] D. Scott and Ch. Strachey, Toward a mathematical semantics for computer languages, Technical Monograph PRG-6, Oxford University (1971).

[29] J.E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory* (MIT Press, Cambridge, MA, 1977).

[30] R.D. Tennent, The denotational semantics of programming languages, *Comm. ACM* **19** (1976) 437-453.

[31] G.C. Titterington, Application of formal methods in an industrial environment (Final project report), Report to the Alvey Directorate (Software Engineering), Software Sciences Ltd., Project no TH726DC, Reference no 5510 (1986).