*Fundamenta Informaticae 28 (1996) 55–85*
*IOS Press*

# WHY DENOTATIONAL? Remarks on Applied Denotational Semantics[*]

**Andrzej Blikle**[*]
*Institute of Computer Science*
*Polish Academy of Sciences*
*Ordona 21, 01-237 Warsaw*

**Abstract.** This is an essay where the author expresses his views on applied denotational semantics. In the author's opinion, whether a software system has or does not have a sufficiently abstract denotational semantics should be regarded as a pragmatic attribute of the system rather than merely as a mathematical attribute of its description. In a software system with denotational semantics structured programming is feasible and for such systems there is a routine method of developing program-correctness logic. All that may not be the case if denotationality is not ensured. On the other hand, a non-denotational semantics can be always artificially made denotational on the expense of lowering its level of abstraction. This leads to an important pragmatic question: to what extent and in which situations can we sacrifice denotationality and/or abstraction of a semantics? All discussions are carried on an algebraic ground but the paper is not very technical and contains a short introduction to the algebraic theory of denotational semantics.

*Everything which is evident should be given special attention at the beginning in order to avoid all possible misunderstandings in the future.*

a popular wisdom

## 1. Introduction

This essay has been addressed to readers interested in the applications of mathematical semantics in software engineering and is devoted to the discussion of the attribute of *denotationality*. In the author's opinion, whether a software system has or does not have a sufficiently abstract denotational semantics should be regarded as a pragmatic attribute of the system rather then merely as a mathematical attribute of its description. Denotationality in system design is like structurality in programming: it makes the final product easy to understand and to prove correct and therefore constitutes a prerequisite of a sound engineering style.

Our main discussion is preceded by some clarification of concepts. In Sec.2. we point out that the word *semantics* has been used in the literature in at least three different meanings and we try to convince the reader that such attributes of a semantics as *denotational*,

[*]A short conference version of this paper appeared earlier in R.Janicki, W.W.Koczkodaj eds. *Computing and Information, Proc. Int. Conf. on Computing and Information, ICCI'89, Toronto, North-Holland 1989.* The reported research has been sponsored by the Polish Academy of Sciences under grant CPBP 02.17

*operational* and *algebraic* should be regarded as orthogonal rather than as alternative or contrasted. In Sec.3. we define our notation and in Sec.4. we briefly introduce the reader into an algebraic framework of denotational semantics. That framework is then used throughout the paper.

The main part of our discussion starts in Sec.5. from an argument that a software system with denotational semantics provides an adequate ground for structured programming and for a systematic development of a program-correctness logic. Then we show that this need not to be the case if denotationality is not ensured (Sec.6.) This is followed by the analysis of such properties of a semantics that can make the semantics non-denotational. We discuss the trade-off between denotationality and abstraction (Sec.7.) and we show that a non-denotational semantics can be always artificially "made denotational" at the expense of lowering its level of abstraction (Sec.8.) This leads us to an important pragmatic question: to what extent and in which situations can we sacrifice denotationality and/or abstraction of a semantics? A few typical cases of 'spoiled' denotationality are discussed in Sec.9. and Sec.10. One of them is a copy-rule mechanism. We complete the discussion with an example of a structured operational definition, in the style of G.Plotkin, of a denotational semantics of a simple programming language (Sec.11.) The last Sec.12. contains some concluding remarks.

## 2. About the Concept of Semantics

This section is devoted to the clarification of the concept of semantics and to the discussion of the relationship between three common attributes of a semantics: *denotational*, *operational* and *algebraic*. We shall argue that these attributes are orthogonal rather than alternative or contrasted.

Let us start from a remark that the issue of semantics is not restricted — as is frequently thought — to programming languages, but applies to all kinds of software including system software, tools and applications. In fact each software system contains some programming language used for the communication with the system. However, for the sake of simplicity, in all examples of this paper we analyze a toy programming language. Such a language can be made much simpler than a toy operating system, a toy data-base management system or the like.

In any software system we can always identify some *syntax*, which is used to formulate our requests to the system, some *denotations*, which are the meanings of these requests and some *semantics*, which assigns denotations to syntax. In other words, in the mathematical model of a software system we can always identify three following components:

- a set of syntactic objects $Syn$,
- a set of denotations $Den$,
- a function of semantics $S : Syn \rightarrow Den$

Unfortunately in the current literature the word *semantics* is used ambiguously to mean four different things:

1) the functions $S$,
2) the definition of that function,
3) the underlying theory,
4) or even the denotations themselves, like e.g. in: "*... the semantics of commands are state-to-state transformations*"

Case 4) is, of course, only a linguistic sloppiness. Case 3) can be easily recognized from a context and therefore does not cause problems. Cases 1) and 2), however, if not distinguished properly, may lead to a confusion. For instance, if an author says that: "*we call a semantics denotational if it is compositionally defined and tackles recursion with the help of fixed points*", it is not clear if he is talking about a function — which is "*compositionally* *defined...*" — about its definition — which "*tackles recursion...*" — or, maybe, about both at the same time?

In this paper by a *semantics* we always mean a function and when we want to talk about its definition, then we say that explicitly, unless the context indicates clearly what we mean. Below we briefly explain our understanding of the attributes *denotational*, *operational* and *algebraic.* We start from the attribute of denotationality.

*Syn* usually represents a context-free language described either by a context-free grammar, or by BNF equations, or by a set of syntactic domain equations or by a signature of an algebra. In each of these cases one may construct a unique many-sorted algebra over *Syn*. We say that *S is denotational* if it has the property of *compositionality*, i.e. if one can construct such a many-sorted algebra over *Den* that *S* becomes a homomorphism. As we shall see in the sequel (Sec.7.) that understanding of denotationality is a little too weak for applications. In fact, a denotational semantics must be also sufficiently abstract in order to be of a practical use. That is, however, a more pragmatic issue.

In the literature the term *denotational semantics* is usually associated with two specific techniques of constructing the algebra of denotations: *reflexive domains* and *continuations* (see e.g. [29], [13] or [24]). These techniques have been introduced in the early 1970-ties by the pioneers of denotational semantic Dana Scott and Christopher Strachey [25], [27] in order to construct a denotational model of a rather exotic programming mechanism where an untyped lambda-calculus had been mixed with unrestricted goto's. That mechanism appeared in THE programming language of these days — Algol-60.

Reflexive domains and continuations have attracted the attention of researchers so much that for many years it became customary to assume that every denotational semantics must involve these concepts. In fact, however, D.Scott and Ch.Strachey have always emphasized that the most relevant attribute of a denotational semantics is compositionality (cf. [27]):

> *In this approach the semantical functions give mathematical values to expressions — values related to some given model. The values of expressions are determined in such a way that the value of a whole expression depends functionally on the values of its parts — the exact connection being found through the clauses of the syntactical definition of the language.*

The idea of compositionality has been later formalized on an algebraic ground by a group of American authors known as ADJ, see e.g. [31]. It has been also pointed out by Andrzej Blikle and Andrzej Tarlecki [9] that reflexive domains are needed only if we wish to describe *self-applicable functions*, i.e. functions that can assume themselves as arguments. Such functions appear e.g. in Algol'60 — due to the admission of typeless procedural parameters — or in Lisp — due to dynamic recursion. However, since self-applicability has turned out to be an unsafe mechanism, it has been abandoned in all contemporary programming languages. Therefore, the denotational models of such languages may be conveniently constructed in a framework where the domains of denotations are usual sets possibly with a *cpo* ordering.

Continuations have been abandoned in the applications as well, since it has turned out that even such anarchic jumps as in Algol-60 can be described without continuations (cf. [3], [9]).

A set-theoretic continuation-free style has been assumed, therefore, in the 1980's in the majority of projects devoted to the development of software-specification systems based on denotational semantics such as e.g. **MetaSoft** [6], RAISE [21] or BSI/VDM [18]. Readers interested in the technical details of set-theoretic approaches to domains are refered to [30] which describes such an approach for BSI/VDM and to [1] where a domain theory and a corresponding type theory have been constructed for **MetaSoft**.

The discussion which follows in this paper applies essentially to all styles of denotational semantics. However, it refers mainly to — and it has been stimulated by — the mentioned above recent developments of that theory and its applications.

The attribute of denotationality may be associated not only with a function of semantics but also with its definition. When we say that a programming language Pascal has been given a denotational semantics, we mean that the semantics of Pascal has been given a denotational definition, i.e. a definition which expresses the compositionality of $S$ explicitly by the equations of the form:

$$S[op(syn_1, \ldots, syn_n)] = \| \; op \; \| \; (S[syn_1], \ldots, S[syn_n]) \tag{1}$$

where $op$ is an operation in the algebra **Syn** and $\| \; op \; \|$ is the corresponding operation in **Den**. Usually the operations $op$ and $\| \; op \; \|$ are not explicit in these equations but implicit in (meta)expressions which appear on both sides of (1). For instance, if we write:

$$S[com_1; com_2](sta) = S[com_2]((S[com_1](sta))$$

where *com* stands for a command and *sta* for a state, then the operation $\| \; ; \; \|$ defined by:

$$\| \; ; \; \| \; (S[com_1], S[com_2])(sta) = S[com_2]((S[com_1](sta))$$

appears on the right-hand side of the former equation only implicitly.

Since a denotational form of a definition of semantics quarantees that the semantics itself is denotational, a non-denotational semantics cannot be given a denotational definition. If, therefore, a designer of a software system chooses a denotational form of a definition, then he/she may be sure that the semantics of the system will become denotational. If a definition is not denotational, then one may still be able to construct a denotational semantics (cf. Sec.11.)), but in that case one has to prove that the semantics has indeed this property and such a proof may be far from trivial. Also, in such a case there is always a risk that the semantics may "come out of control" and become non-denotational.

Now let us briefly discuss the concept of *operational semantics.* In contrast to *denotationality*, the attribute of *operationality* is not very sharp and applies in the first place to the definition of $S$ rather than to $S$ itself. Moreover, that attribute has never been formalized. We can only point to some techniques and/or metalanguages which are regarded as related to the operational style. For instance the *Vienna Definitional Language* (VDL) [19], the *structured operational semantics* (SOS) [23] or the *natural semantics* [15] belong to that group.

In general the definition of semantics is called *operational* if it describes some algorithm of "executing" the syntax. A code of an interpreter is an example of a very operational definition. Another such example — much more abstract in fact — may be a definition written in G.Plotkin's SOS [23]. Also definitions written in the so called *Danish dialect* of VDM [2] are to a large extent operational in that sense — although they are denotational at the same time (!) — since they usually describe an abstract interpreter.

In contrast to the attribute of denotationality, where we can always formally decide whether a given definition is or is not denotational, whenever we talk about operationality we can only argue about the degree to which our definition is operational in a given context.

In this paper we do not discuss much of the idea of operational semantics. We only wish to express a claim that *operational* should not be contrasted to *denotational.* In author's opinion a definition of a software system must be always operational to some degree, that degree depending on several factors such as e.g. the stage of the system development, the target programming language where the system is to be implemented, the expected reader of the definition, etc.

The attribute *algebraic* — as used today in the literature to name a certain equational-axiomatic style (cf. e.g. [11]) — again refers to the definition of $S$, rather than to $S$ itself, and again is not very sharp. Usually an algebraic definition consists of a set of equational axioms which identify a class of the algebras of denotations **Den**. The corresponding syntax is common for all these algebras and is implicit in their common signature. For any **Den** the function of semantics is the unique homomorphism from the algebra of ground terms over that signature into **Den**. An algebraic definition in this sense guarantees that the defined semantics is denotational.

In the sequel we shall try to convince the reader that a semantics should be always sufficiently denotational and that it should be described at an appropriate level of operationality, the latter depending on the current application. The author also believes that the algebraic framework is most appropriate for the description of software whether or not we are using axiomatic techniques and independently of the degree of operationality of the used semantics.

Whenever in the future we use the term *software system* or *programming language* we always mean a triple which consists of a syntax **Syn**, of a denotation **Den** and of a semantics $S$.

## 3. Basic Notation

Most definitions of semantics which we discuss in this paper are written in a so called *model-oriented style* typical for the Danish dialect of VDM and for **MetaSoft**. In that style the algebras of syntax and of denotations are constructed within some set theory, rather than described axiomatically as e.g. in OBJ [12] or in ACT-ONE [11]. Below we briefly introduce a notation which is used in our definitions. For more details we refer the reader to [5].

For any sets $A$ and $B$:

| | |
|---|---|
| $A \mid B$ | denotes the *union* of $A$ and $B$, |
| $A \rightarrow B$ | denotes the set of all *total functions* from $A$ into $B$, |
| $A \overset{\sim}{\rightarrow} B$ | denotes the set of all *partial functions* from $A$ into $B$, |
| $A \underset{\mathrm{m}}{\rightarrow} B$ | denotes the set of all *mappings* from $A$ into $B$, i.e. partial functions defined over a finite subset of $A$, |
| $A^{c*}$ | denotes the set of all *finite strings* (including the empty string) of the elements of $A$. |

Domain equations are written in the form e.g.:

$$sta \ : \ State \ = \ Identifier \rightarrow Integer$$

by which we mean that each state is a total function from identifiers to integers and that a typical element of the set *State* is denoted by *sta* possibly with indices. By $f : A \rightarrow B$ and $f : A \overset{\sim}{\rightarrow} B$ we denote the fact that $f$ is a *total* resp. *partial function* from $A$ to $B$. In our paper the formula $a : A$ is used synonimously with $a \in A$. It can be read as "*$a$ is of type $A$*", which in some software specification languages means more than to be an element of $A$ (cf .Sec.9.) By $dom.f$ we denote the domain of the function $f$. For *curried functions* like $f : A \rightarrow (B \rightarrow (C \rightarrow D))$ we write $f : A \rightarrow B \rightarrow C \rightarrow D$. We also write $f.a$ for $f(a)$ and $f.a.b.c$ for $((f.a).b).c$. For uniformity reasons each many-argument non-curried 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. We also assume — for a better readability of semantic clauses — that the syntactic argument of a function of semantics is always enclosed in square brackets. E.g. we write $C.[x := x + 1]$ rather than $C.x := x + 1$. If $f : A \overset{\sim}{\rightarrow} B$ and $g : B \overset{\sim}{\rightarrow} C$, then $f \bullet g : A \overset{\sim}{\rightarrow} C$ where $f \bullet g = \{\langle a, c \rangle | (\exists b)(f.a = b \ \& \ g.b = c)\}$. In the definitions of functions we frequently use conditional expressions of the form $b \rightarrow c, d$ which stand for:

**if** $b$ **then** $c$ **else** $d$ **fi**

This construction may be nested in which case the expression $b_1 \rightarrow (a_1, (b_2 \rightarrow \ldots (b_n \rightarrow a_n, a_{n+1}) \ldots))$ is written in a column:

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

Sometimes in conditional expressions we are nesting "*local constant declarations*" of the form *Let* $x = exp_1$ **in** $exp$ or *Let* $x_i = exp_i$ **for** $i = 1, ..., n$ **in** $exp$. The scope of such a declaration is the expression $exp$.

For any partial function $f : A \rightsquigarrow B$, by $f[b/a]$, where $a \in A$, $b \in B$, we denote a function $f[b/a] : A \rightsquigarrow B$ such that:

$$f[b/a].x = (x = a \rightarrow b, f.x)$$

By $[b_1/a_1, \ldots, b_n/a_n]$ we denote a total function on $\{a_1, \ldots, a_n\}$ which assigns $b_i$ to $a_i$ for $i = 1, \ldots, n$. Of course, all $a_i'$smust be mutually distinct.

## 4. Denotational Semantics in an Algebraic Framework

As we have already mentioned in Sec.2., a denotational semantics may be understood as a homomorphism between two many-sorted algebras. This section is devoted to a short introduction of a mathematical theory of such semantics. More on an algebraic framework of denotational semantics in our style may be found in [8] and a general introduction to an axiomatic setting of algebraic semantics is [11].

By a *signature* we mean a quadruple:

$$\mathbf{Sig} = \langle Sn, Fn, sort, arity \rangle$$

where $Sn$ is a nonempty set of *sort names*, $Fn$ is a nonempty set of *function names* and:

$$\begin{array}{lll} sort & : & Fn \rightarrow Sn \\ arity & : & Fn \rightarrow Sn^{c*} \end{array}$$

are functions which associate the sorts of the results and of the arguments respectively with any function name. By an *algebra over the signature* **Sig**, or shortly by a **Sig**-*algebra*, we mean a triple $\mathbf{Alg} = \langle \mathbf{Sig}, car, fun \rangle$ where $car$ and $fun$ are functions interpreting sort names as nonempty sets and function names as total functions on these sets respectively. 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.

$$fun.fn : car.sn_1 \times \ldots \times car.sn_n \rightarrow car.sn$$

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

In applications, and also in the examples, algebras are treated a little less formally and are defined as collections of carriers and operations between them. In that case it is understood that the signatures are implicit in the notation used.

Two algebras with the same signature are called *similar.* Given two similar algebras $\mathbf{Alg}_i = \langle \mathbf{Sig}, car_i, fun_i \rangle$for $i = 1, 2$, 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 $fn \in Fn$, $fun_1.fn$ is the restriction of $fun_2.fn$ to the carriers of $\mathbf{Alg}_1$. By a *homomorphism* from $\mathbf{Alg}_1$ (a *source algebra*) into $\mathbf{Alg}_2$ (a similar *target algebra*) we mean an $H$ that assigns to any sort $sn \in Sn$ a function:

$$H.sn \ : \ car_1.sn \rightarrow car_2.sn \tag{2}$$

called the *sn-component* of $H$, such that for any $fn \in Fn$ with $sort.fn = sn$ if $arity.fn = \langle sn_1, \ldots, sn_n \rangle$ with $n \geq 0$, then for any tuple of arguments $\langle a_1, \ldots, a_n \rangle \in car.sn_1 \times \ldots \times car.sn_n$ we have:

$$H.sn.(fun_1.fn.\langle a_1, \ldots, a_n \rangle) = fun_2.fn.\langle H.sn_1.a_1, \ldots, H.sn_n.a_n \rangle \tag{3}$$

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

With every many-sorted function which satisfies (2) we may associate an $Sn$-sorted relation $\equiv_H$in $\mathbf{Alg}_1$:

$$\equiv_H .sn \subseteq car_1.sn \times car_1.sn$$

called the *kernel* of $H$ and defined as follows: for any $a_1, a_2 \in car.sn_1$

$$a_1 \equiv_H .sn \ a_2 \quad \text{iff} \quad H.sn.a_1 = H.sn.a_2.$$

Formally the kernel is a function $\equiv_H$ which to every sort $sn \in Sn_1$ assigns a binary relation $\equiv_H .sn$ in $car_1.sn$. It is a well known fact that each $\equiv_H .sn$ is an equivalence relation. The many-sorted relation $\equiv_H$ is said to be a *congruence* in $\mathbf{Alg}_1$, if for any $fn \in Fn_1$ with $arity_1.fn = \langle sn_1, \ldots, sn_n \rangle$ and $sort_1.fn = sn$ and for any $a_i, b_i \in car_1.sn_i$, $i = 1, \ldots, n$:

$$\begin{array}{ll} \textbf{if} & a_i \equiv_H .sn_i \ b_i \quad \text{for } i = 1, ..., n \\ \textbf{then} & fun_1.fn.\langle a_1, \ldots, a_n \rangle \equiv_H .sn \ fun_1.fn.\langle b_1, \ldots, b_n \rangle \end{array}$$

The property described above is called the *extensionality property* of an equivalence relation. In the sequel we shall frequently refer to the following well-known fact:

**Proposition 4.1.** If $H$ is an arbitrary many-sorted function which satisfies (2) then $\equiv_H$ is a congruence in $\mathbf{Alg}_1$ iff one can construct such an algebra $\mathbf{Alg}_2$ over the many-sorted family of carriers $\{car_2.sn | sn \in Sn\}$ that $H$ becomes a homomorphism from $\mathbf{Alg}_1$ into $\mathbf{Alg}_2$.

An element of an algebra is said to be *reachable* if it can be constructed from the constants of the algebra by means of the operations of the algebra. An element which is not reachable is called *junk*. For instance, in an algebra of integers $\langle Int, 1, + \rangle$ all positive integers are reachable and all non-positive are junk. A subset of $car.sn$ which consists of all reachable elements is called a *reachable carrier* of the sort $sn$. All reachable carriers are closed under the operations of the algebra and therefore, if none of them is empty, then they constitute a subalgebra of that algebra. We call it the *reachable subalgebra*. It is the (unique) least subalgebra of a given algebra. If all the elements of an algebra are reachable, i.e. if the algebra has no junk, then it is called a *reachable algebra*. The following well-known fact is important for the theory of denotational semantics:

**Proposition 4.2.** If $H : \mathbf{Alg}_1 \rightarrow \mathbf{Alg}_2$ is a homomorphism and $\mathbf{Alg}_1$ is reachable, then:

1) the image of $\mathbf{Alg}_1$ in $\mathbf{Alg}_2$ is the reachable subalgebra of $\mathbf{Alg}_2$,
2) $H$ is a unique homomorphism between $\mathbf{Alg}_1$ and $\mathbf{Alg}_2$.

If for a given algebra **Alg** there is exactly one homomorphism into any algebra with the same signature, then we say that **Alg** is *initial*, or — precisely speaking — that it is *initial in the class of all algebras similar with* **Alg**. An algebra is called *unambiguous* if each of its reachable elements may be constructed from the constants of that algebra by using the operations of the algebra in exactly one way. For a more formal definition of the ambiguity of algebras see [8].

**Proposition 4.3.** An algebra is initial iff it is reachable and unambiguous.

For instance, the algebra $\langle Int^{+}, 1, +\rangle$ of positive integers is reachable but not unambiguous, hence not initial, since e.g. 4 may be constructed as $((1+1)+1)+1)$ or as $((1+1)+(1+1))$. If, however, we replace "+" by "+1" (*successor*), then the new algebra is initial.

On an algebraic ground a denotational model of a software system is represented by a triple $\langle \mathbf{Syn}, \mathbf{Den}, S\rangle$ where **Syn** is a reachable algebra of *syntax*, **Den** is an algebra of *denotations* and

$$S : \mathbf{Syn} \rightarrow \mathbf{Den} \tag{4}$$

is a unique corresponding denotational semantics. The algebra of syntax is always reachable — syntax junk makes no practical sense — but does not need to be unambiguous. The admission of ambiguous syntax distinguishes our approach from some other algebraic approaches to denotational semantics. It allows us to regard the algebra of syntax as a close approximation of so called *concrete syntax* rather than merely as an *abstract syntax.* For a discussion of that problem see [8].

Now, let us explain the introduced concepts in the context of a toy programming language. That language constitutes a core for many examples which we discuss in this paper. Let the syntax of our language be defined by the following CF-grammar written in the form of a fixed-point set of equations [4]:

$$\begin{array}{lcll} ide : Ide & = & \{x, y, z\} & \text{(identifiers)} \\ exp : Exp & = & \{1\} \mid Ide \mid \{(\}Exp\{+\}Exp\{)\} & \text{(expressions)} \\ com : Com & = & Ide\{:=\}Exp \mid Com\{;\}Com & \text{(commands)} \end{array} \tag{5}$$

An alternative less formal but simpler notation may be:

$$\begin{array}{lcl} ide : Ide & = & \mathrm{x} \mid \mathrm{y} \mid \mathrm{z} \\ exp : Exp & = & 1 \mid Ide \mid (Exp + Exp) \\ com : Com & = & Ide := Exp \mid Com \end{array} \tag{6}$$

The carriers of the corresponding algebra **Syn** are the three sets (formal languages) defined above and the operations, which are implicit in this grammar, are the following:

$$\begin{array}{lll} set\_x & : \rightarrow Ide & \text{(and the same for } y \text{ and } z\text{)} \\ set\_1 & : \rightarrow Exp & \\ make\_exp & : Ide \rightarrow Exp & \\ plus & : Exp \times Exp \rightarrow Exp & \\ asg & : Ide \times Exp \rightarrow Com & \\ follow & : Com \times Com \rightarrow Com & \end{array}$$

where

$$\begin{array}{ll} set\_x.\langle\rangle & = x \\ set\_1.\langle\rangle & = 1 \\ make\_exp.ide & = ide \quad \text{(transforms ident. into expr.)} \\ plus.\langle exp_1, exp_2\rangle & = (exp_1 + exp_2) \\ asg.\langle ide, exp\rangle & = ide := exp \\ follow.\langle com_1, com_2\rangle & = com_1; com_2 \end{array} \tag{7}$$

In order to define the corresponding algebra of denotations we first define a domain of states:

$$sta : State = Ide \rightarrow Integer$$

and then we define three carriers of **Den**:

$$\begin{array}{lcll} ide : Ide & = & \{x, y, z\} & \text{(the denotations of identifiers)} \\ eva : Evaluator & = & State \rightarrow Integer & \text{(the denotations of expressions)} \\ exe : Executor & = & State \rightarrow State & \text{(the denotations of commands)} \end{array}$$

Observe that the denotations of identifiers are the identifiers themselves. Now we define the operations of **Den**. Since **Den** is to be a homomorphic image of **Syn**, for each operation *op* from **Syn** we define an operation $\| op \|$ in **Den**, such that when $\| op \|$ is applied to the denotations of the arguments of *op*, it gives the denotation of the corresponding value of *op*:

$$\begin{array}{lll}
\| \mathit{set\_x} \| & : \ \rightarrow \mathit{Ide} & \text{(and the same for } y \text{ and } z\text{)} \\
\| \mathit{set\_1} \| & : \ \rightarrow \mathit{Evaluator} & \\
\| \mathit{make\_exp} \| & : \ \mathit{Ide} \rightarrow \mathit{Evaluator} & \\
\| \mathit{plus} \| & : \ \mathit{Evaluator} \times \mathit{Evaluator} \rightarrow \mathit{Evaluator} & \\
\| \mathit{asg} \| & : \ \mathit{Ide} \times \mathit{Evaluator} \rightarrow \mathit{Executor} & \\
\| \mathit{follow} \| & : \ \mathit{Executor} \times \mathit{Executor} \rightarrow \mathit{Executor} &
\end{array}$$

where

$$\begin{array}{ll}
\|\mathit{set\_x}\|.\langle\rangle & = x \\
\|\mathit{set\_1}\|.\langle\rangle & = 1.0 \\
\|\mathit{make\_exp}\|.\mathit{ide}.\mathit{sta} & = \mathit{sta}.\mathit{ide} \\
\|\mathit{plus}\|.\langle \mathit{eva}_1, \mathit{eva}_2\rangle.\mathit{sta} & = (\mathit{eva}_1.\mathit{sta}) \oplus (\mathit{eva}_2.\mathit{sta}) \\
\|\mathit{asg}\|.\langle \mathit{ide}, \mathit{eva}\rangle.\mathit{sta} & = \mathit{sta}[(\mathit{eva}.\mathit{sta})/\mathit{ide}] \\
\|\mathit{follow}\|.\langle \mathit{exe}_1, \mathit{exe}_2\rangle.\mathit{sta} & = \mathit{exe}_2.(\mathit{exe}_1.\mathit{sta})
\end{array}$$

In these equations *1.0* denotes the number which corresponds to the numeral (name) "1" and $\oplus$ denotes the arithmetical operation of addition (we are distinguishing here between the syntactic element '+', which is just a symbol and the corresponding arithmetical operation $\oplus$). The equations are written in an implicit lambda-notation. For instance, the third equation can be read as follows: $\|\mathit{make\_exp}\|$ is a function that given an identifier *ide* returns an evaluator $\|\mathit{make\_exp}\|.\mathit{ide}$ which given a state *sta* returns that integer which has been stored under *ide* in *sta*.

As is easy to prove, the algebra **Syn** is reachable and therefore a homomorphism $S$ : **Syn** $\rightarrow$ **Den** is unique if it exists. The proof of the existence of $S$ is, however, not quite trivial (cf. [8]) since the grammar which underlies (6) is ambiguous and therefore so is **Syn**.

Our homomorphism is a many-sorted function and therefore it may be regarded as a triple of functions $\langle I, E, C\rangle$ where:

$$\begin{array}{lll}
I & : & \mathit{Ide} \rightarrow \mathit{Ide} \\
E & : & \mathit{Exp} \rightarrow \mathit{Evaluator} \\
C & : & \mathit{Com} \rightarrow \mathit{Executor}
\end{array}$$

The definitions of these functions are of the form (3) and therefore are implicit in the correspondence between $\| op \|$ and *op*. Hence we do not need to write them explicitly. In a more traditional approach, however, these functions are usually defined explicitly whereas the definitions of $\| op \|$'s are implicit. In such a case we write:

$$\begin{array}{ll}
I.[\mathit{ide}] & = \mathit{ide} \\
E.[1].\mathit{sta} & = 1.0 \\
E.[\mathit{ide}].\mathit{sta} & = \mathit{sta}.\mathit{ide} \\
E.[(\mathit{exp}_1 + \mathit{exp}_2)].\mathit{sta} & = E.[\mathit{exp}_1].\mathit{sta} \oplus E.[\mathit{exp}_2].\mathit{sta} \\
C.[\mathit{ide} := \mathit{exp}].\mathit{sta} & = \mathit{sta}[(E.[\mathit{exp}].\mathit{sta})/\mathit{ide}] \\
C.[\mathit{com}_1; \mathit{com}_2].\mathit{sta} & = C.[\mathit{com}_2].(C.[\mathit{com}_1].\mathit{sta})
\end{array}$$

where the syntactic arguments of the function of semantics are traditionally enclosed in square brackets (cf.Sec.3.)

If we are developing a denotational model of a software system in a traditional order — i.e. first **Syn**, then **Den** and finally $S$ — then we have to prove that $S$ is indeed a homomorphism. In real-life situations this need not be a simple task. Besides, we can easily make a mistake and construct an $S$ which is not compositional. In the sequel we shall see some typical sources of such mistakes. If, therefore, we want to be sure that the mathematical model of a software system becomes denotational, we should start the development of that model from the algebra **Den**. In that case we can always develop a custom-made syntax **Syn** such that the existence of the corresponding unique homomorphism (denotational semantics) is guaranteed by the way in which **Syn** has been developed. A systematic method of software design along these lines has been described in [8].

## 5. Why Denotational?

The role of *denotationality* in software engineering is similar to the role of structurality in programming. Both improve the readability, the comprehensibility and the maintainability of a final product and both allow for the decomposition of a large task into a number of independent subtasks. Also in both cases the non-convinced may give thousands of "clever examples" where the principle of denotationality, respectively structurality, has been violated in the derivation of a "smart" program. It has been known for years, however, that in large-scale applications an anarchic cleverness brings always more disasters than benefits.

Readability is, of course, a property of a definition of semantics. The advantages of writing the definitions of semantics in a denotational form are widely known — even if not widely appreciated — and therefore we shall not discuss them here. This section is devoted to a claim that the decision whether the semantics itself, i.e. the function $S : \mathbf{Syn} \rightarrow \mathbf{Den}$, is to be compositional, should be regarded as a design decision since it implies relevant properties of the software system in question.

**Claim 5.1.** The denotationality of semantics allows structured top-down programming in the corresponding syntax.

Consider an arbitrary software system represented by a denotational model

$$S : \mathbf{Syn} \rightarrow \mathbf{Den}$$

where for the sake of simplicity we assume that the algebras **Syn** (of syntax) and **Den** (of denotations) are one-sorted and that *Syn* and *Den* denote the corresponding unique carriers. Each programming task in our system consists of the construction of a syntactic object $syn \in Syn$ which for a given prespecified denotation $den \in Den$ satisfies the equation:

$$S.[syn] = den$$

Of course, if *den* is to be programmable at all (i.e. if it is to be a denotation of some syntax), it must belong to the reachable part of *Den*, since the image of *Syn* in *Den* is always reachable (Fact 4.2.) This means that there must be $den_1, \ldots, den_n$ in *Den*, $syn_1, \ldots, syn_n$ in *Syn* and an operation

$$op : Syn \times \ldots \times Syn \rightarrow Syn$$

such that:

$$\begin{array}{ll} 1)\ den = \| op \| .\langle den_1, \ldots, den_n \rangle & \text{and} \\ 2)\ S.[syn_i] = den_i & \text{for } i = 1, \ldots, n. \end{array}$$

This means in turn that the task of finding a program for *den* may be split into the subtasks of finding programs for each of $den_i$. These subtasks may be assigned to independently working groups of programmers, and when programming in the groups is completed, the denotationality of $S$ guarantees that

$$op.\langle syn_1, \ldots, syn_n \rangle$$

realizes the global task, i.e. that

$$S.[op.\langle syn_1, \ldots, syn_n \rangle] = \| op \| .\langle den_1, \ldots, den_n \rangle = den$$

Of course, the problem of splitting *den* into the "right" $den_i$'s need not to be easy since in general there exist splits where $den_i$'s are not reachable. A correct split, however, always exists. As we shall see in an example which comes in Sec.6., if $S$ is not denotational, then it may be impossible to split some programming tasks into independently programmable subtasks.

**Claim 5.2.** For each software systems with denotational semantics there is a systematic (although not algorithmic!) method of deriving a sound and a relatively complete set of program-correctness proof-rules.

Below we give a rough, half-formal, justification of that claim. A full discussion, which requires the introduction of many technical arguments, would go outside the scope of the present paper.

To say that a program is correct is to say that its denotation possesses a certain property. Mathematically the properties of denotations are represented by n-ary total functions on denotations called *predicates*:

$$pred : Predicate = Den_1 \times \ldots \times Den_n \rightarrow Bool$$

where $Bool = \{true, false\}$ and where $Den_i = Den$ for $i = 1, ..., n$. Of course, in general we may deal with more than one domain of predicates.

In order to expand the language of a software system by claims about programs we add the domains of predicates to the algebra of denotations and we define some constructors on them. These constructors identify a class of properties that we want to express by means of predicates, such as e.g. partial or total correctness of sequential programs, liveness and deadlock freeness of concurrent systems etc. As we shall see in an example which follows, this may also require the introduction of some auxiliary carriers. The new algebra of denotations corresponds to a language in which besides writing programs one can also express their properties.

When we are done with the extended algebra we proceed to establishing proof rules which most frequently are lemmas of, roughly, the following form:

$$\begin{array}{l} \text{for any reachable } pred \text{ and any reachable } den_1, \ldots, den_n : \\ \quad pred.(\| \; op \; \| \; .\langle den_1, \ldots, den_n \rangle) = true \\ \quad\quad \textbf{iff} \\ \quad\quad \text{there exist predicates } pred_1, \ldots, pred_n \text{ such that} \\ \quad\quad\quad 1) \; pred_i.den_i = true, \;\; i = 1, ..., n \\ \quad\quad\quad 2) \; \Phi.\langle pred, pred_1, \ldots, pred_n \rangle \end{array} \tag{8}$$

where $\Phi.\langle pred, pred_1, \ldots, pred_n \rangle$ expresses a certain relationship between predicates. We establish such a lemma for each operation $op : Syn \times \ldots \times Syn \rightarrow Syn$ of the algebra of syntax (notice that $\| \; op \; \|$ denotes the counterpart of *op* in **Den**). Of course, if $op : \rightarrow Syn$, then $i = 0$ and "**iff**" is followed only by $\Phi.pred$.

Lemmas constructed in that way can be used in structured-inductive proofs of programs' correctness. Each of them enables the reduction of a global correctness problem — for a compound object — to a number of local correctness problems — for the components of that object. Of course, we can also develop a formalized proof system (a logic) in which our lemmas become inference rules. The "**if**" part of each lemma guarantees the soundness of such a rule and the "**only if**" — a relative completeness in the sense close to that of [10].

In general, by a *completeness of a logic* we mean the fact that every true statement which can be expressed in the language of that logic can be proved. In the case of a logic of programs we can only expect a so called *relative completeness*, since in every concrete situation the applicability of each of our lemmas depends on the following three factors:

(i) that the required $pred_i$'s are reachable, i.e. expressible in our logic,
(ii) that we are able to express condition 2) of (8) in our logic,

(iii) that we are able to prove that condition in our logic, i.e. that we can prove it on the ground of the corresponding theory of data (such as e.g. the theory of integers, of records, etc.)

Now, let us discuss an example. Consider the programming language defined in Sec.4. We shall construct a corresponding Hoare-like proof-system for the partial correctness of commands. First we expand the algebra *Den* of our language by two new carriers: *Condition* — which constitutes an auxiliary carrier — and *Predicate.* The new algebra has five carriers:

$$\begin{array}{lcl} ide & : & Ide = \{x, y, z\} \\ eva & : & Evaluator = State \rightarrow Real \\ exe & : & Executor = State \rightarrow State \\ con & : & Condition = State \rightarrow Bool \\ pred & : & Predicate = Executor \rightarrow Bool \end{array}$$

Then we define the constructors of conditions and predicates. As the constructors of conditions we choose e.g.:

$$\begin{array}{lcl} \|less\| & : & Evaluator \times Evaluator \rightarrow Condition \\ \|and\| & : & Condition \times Condition \rightarrow Condition \end{array}$$

etc., where:

$$\|less\|.\langle eva_1, eva_2\rangle.sta = eva_1.sta < eva_2.sta$$
$$\|and\|.\langle con_1, con_2\rangle.sta = con_1.sta \ \& \ con_2.sta$$

Of course, in both definitions '=' denotes the equality in *Bool.* For predicates we define just one constructor, which corresponds to the property of partial correctness of commands. This constructor is of the type:

$$\|parcor\| : Condition \times Condition \rightarrow Predicate$$

and since commands in our language represent total functions, is defined as follows:

$$\begin{array}{l} \|parcor\|.\langle con_1, con_2\rangle.exe = \\ \quad (\forall sta \in State)[\textbf{if } con_1.sta = true \textbf{ then } con_2.(exe.sta) = true] \end{array}$$

Now we can formulate our lemmas. For a better readability we assume that:

$$\textbf{pre } con_1 \ : exe \textbf{ post } con_2$$

stands for $\|parcor\|.\langle con_1, con_2\rangle.exe = true.$ We formulate one lemma for each of our two constructors of commands — "*follow*" and "*asg*" — introduced in Sec.4.:

$$\begin{array}{l} \text{for any } con_1,\ con_2, \text{ and any } exe_1,\ exe_2 : \\ \quad \textbf{pre } con_1 \ : follow.\langle exe_1, exe_2\rangle \textbf{ post } con_2 \\ \quad \textbf{iff} \\ \quad \text{there exist conditions } con_{11}, \text{ and } con_{12} \text{ such that} \\ \quad\quad 1)\ \textbf{pre } con_1 \ : exe_1 \textbf{ post } con_{11} \\ \quad\quad\quad \textbf{pre } con_{12} \ : exe_2 \textbf{ post } con_2 \\ \quad\quad 2)\ (\forall sta)(\textbf{if } con_{11}.sta = true \textbf{ then } con_{12}.sta = true) \end{array} \tag{9}$$

$$\begin{array}{l} \text{for any } con_1,\ con_2 \text{ and for any } ide,\ eva : \\ \quad \textbf{pre } con_1 \ : asg.\langle ide, eva\rangle \textbf{ post } con_2 \\ \quad \textbf{iff} \\ \quad (\forall sta)(\textbf{if } con_1.sta = true \textbf{ then } con_2.sta[(eva.sta)/ide] = true) \end{array} \tag{10}$$

Observe that the form of (10) is such as if the syntactic operation *asg* were nullary, which is, of course, not the case. In our example we have not introduced predicates for identifiers and evaluators, and therefore we do not introduce any statement that corresponds to 1) of (8), and in the clause corresponding to 2) of (8) we refer directly to *ide* and *eva* rather than to the predicates on them.

It should be also noticed that in (9) and (10) we have omitted an explicit assumption about the reachability of predicates. This makes our lemmas a little stronger than in the general case (8). In order to make them exact analogues of (8) we should have assumed that our $con_i$'s are reachable, which in this particular case is not necessary. On the other hand all the predicates that appear in our lemmas are of the form:

$$\|parcor\|.\langle con_1, con_2\rangle$$

and hence they are not quite arbitrary.

As has been already said earlier, our lemmas may be used in the construction of a program-correctness logic — in this case a Hoare's partial-correctness logic. Assume that the expanded syntax of our programming language includes the following syntax of formulas and of correctness statements:

$$\begin{array}{llll} for : For & = & Exp\{\textbf{less}\}Exp \mid For\{\textbf{and}\}For & \text{(formulas)} \\ cst : Cst & = & \{\textbf{pre}\}For\{:\}Com\{\textbf{post}\}For & \text{(correctness statements)} \end{array}$$

This syntax has an obvious semantics and the proof rules which correspond to (9) and (10) are the following:

$$\frac{\begin{array}{l}\textbf{pre}\ for_1\ : com_1\ \textbf{post}\ for_{11} \\ \textbf{pre}\ for_{12}\ : com_2\ \textbf{post}\ for_2 \\ for_{11} \Rightarrow\ for_{12}\end{array}}{\textbf{pre}\ for_1\ : com_1; com_2\ \textbf{post}\ for_2}$$

$$\frac{for_1\ \Rightarrow\ for_2[exp/ide]}{\textbf{pre}\ for_1\ : ide := exp\ \textbf{post}\ for_2}$$

In these rules the operator '$\Rightarrow$' corresponds to a usual implication, but in a more realistic case where conditions are partial- or three-valued functions this is a so called superpredicate "*stronger than*", cf. [7] and [16]. Of course, $for_2[exp/ide]$ denotes a formula which results from $for_2$ by substituting *exp* for all free occurrences of *ide*. Each "enumerator" expresses a conjunction of metaformulas from which one may infer the "denominator". Observe that since in these rules we are talking about formulas rather than about conditions we have now implicitly introduced the assumption about the reachability of the involved predicates.

Readers interested in the mathematical problems related to the construction of logics for the denotational models of software may find more technical material in [5], [7] and [16].

## 6. Why Not Non-Denotational?

In the former section we have discussed some advantages of denotational semantics. Here we show that these advantages may disappear if a semantics is not denotational. Consider as an example our little programming language of Sec.4. which we now modify by setting:

$$com\ :\ Com\ =\ Ide\{:=\}Exp \mid \{(\}Com\{;\}Com\{)\}$$

and

$$\begin{array}{ll} C.[(com_1; com_2)] = & \\ fai.com_1 \neq fai.com_2 \rightarrow C.[com_1] \bullet C.[com_2] & \\ \text{TRUE} \qquad\qquad\qquad \rightarrow nullsta & \end{array} \qquad (11)$$

where

*fai.com* = first assignable identifier in *com*, i.e. the left-hand side identifier in the first assignment of *com*,
*nullsta* = a function that transforms any state to [0.0/x,0.0/y,0.0/z]

Similarly to our notation for 1.0vs. 1 (Sec.4.) 0.0stands for the number "*zero*", whereas "0" denotes the corresponding syntax (symbol).

In the new version of the language the syntax has been modified by introducing parentheses into compound commands (since otherwise the definition (11) of the semantics of commands would be ambiguous) and the semantics has been modified by making the effect of commands dependent on *fai*'s. The latter modification makes our semantics not denotational since now $C.[(com_1; com_2)]$ depends on more than just $C.[com_1]$ and $C.[com_2]$. We prove that fact by showing that the equivalence relation $\equiv_C$is not a congruence (cf. Proposition 4.1.). Indeed:

$$\begin{array}{lll} C.[(x := 1; y := 2)] & = C.[(y := 2; x := 1)] & \text{but} \\ C.[((x := 1; y := 2); y = (x + y))] & \neq C.[((y := 2; x := 1); y = (x + y))] & \end{array}$$

Although our semantics is not denotational, its definition is still in a structural inductive form, hence it is easy to understand and implement. Why then should we bother about the non-denotationality? As we are going to see, structured programming and structured-inductive proofs are not possible in the new language.

Consider first the problem with structured programming and take as an example a task of writing a program *com* that loads 2.0to $x$ and to $y$, i.e. a program which satisfies the equation:

$$C.[com].sta = sta[2.0/x, 2.0/y] \tag{12}$$

Now assume that we want to split this task into two new ones, described by two following specifications:

$$\begin{array}{l} C.[com_1].sta = sta[1.0/x, 1.0/y] \\ C.[com_2].sta = sta[(E.[(x + y)].sta)/x,\ E.[(x + y)].sta)/y] \end{array}$$

If we assign these subtasks to two programmers, then the first has a choice between at least two following commands (we use a simplified intuitive notation for commands):

$$\text{either} \quad (x := 1\ ;\ y := 1) \quad \text{or} \quad (y := 1\ ;\ x := 1)$$

and the other has a choice between at least the following commands:

$$\text{either} \quad (y := (x + y)\ ;\ x := y) \quad \text{or} \quad (x := (x + y)\ ;\ y := x)$$

As is easy to see, unless our programmers communicate about the syntax of their target programs, they cannot guarantee that $com_1; com_2$ will satisfy (12). Hence in our language top-down structured programming is not feasible.

Now consider the problem of structured-inductive proofs. First observe that in the new language none of the two implications in (9) — i.e. the "*if*" and the "*only if*" — is true. Indeed, although

$$\textbf{pre}\ true\ : C.[(x := 1; x := 1)]\ \textbf{post}\ x = 0$$

is certainly satisfied, there are no intermediate assertions which can be used in a proof of that fact based on (9), since

$$\textbf{pre}\ con\ : C.[x := 1]\ \textbf{post}\ x = 0$$

is false for any $con \in Condition$. Similarly, although both

$$\begin{array}{l} \textbf{pre}\ true\ : C.[x := 1]\ \textbf{post}\ x = 1 \\ \textbf{pre}\ x = 1\ : C.[x := 1]\ \textbf{post}\ x = 1 \end{array}$$

are true, the statement

$$\textbf{pre}\ true\ : C.[(x := 1; x := 1)]\ \textbf{post}\ x = 1$$

is not true.

```
PROGRAM ONE:
type
    object =record
                no,size : integer;
            end
    item   =record
                no,size : integer;
            end;
var
    x : object;
    y : item
begin
    x := y
end
```

```
PROGRAM TWO:
type
    object =record
                no,size : integer;
            end
    item   =object


var
    x : object;
    y : item
begin
    x := y
end
```

Figure 1 Two non-equivalent Pascal programs

It is also not very easy to modify the logic of Sec.5. to the new language. The major problem consists in the fact that in the present case we do not have — and we cannot have — an algebra of denotations. We cannot apply, therefore, the routine way for the construction of logic described in Sec.5. In fact, we cannot use here any of the known techniques of formal logic, since all these techniques are inherently based on the assumption that the underlying language of terms and formulas has a denotational semantics. In the usual formal logic — whether classical, algorithmic, temporal or any other — we never talk about the properties of (the syntax of) formulas. And here we should have to do that in order to formulate a proof rule for ";".

In the opinion of the author the only rational way of solving the problem consists of "repairing" a non-denotational semantics by making it denotational and then applying the method of Sec.5. We shall discuss this solution in Sec.8.

Our example of a non-denotational programming language has been made a little artificial in order to be sufficiently simple. Similar examples may be shown, however, on a more natural ground. Take, for instance, Pascal and its concept of a type. As we can read in [14]: "*A data type determines a set of values which variables of that type may assume...*". This suggests that types in Pascal are just the sets of values. If we assume that, then the type definitions of the programs on Figure 1 have equal denotations. At the same time, however — according to the standard of Pascal — the first program generates a type error whereas the second does not. This leads to a conclusion that the equivalence relation which corresponds to our semantics is not a congruence, hence that our semantics is not denotational.

Our example should not be understood as an argument that Pascal cannot be given a denotational semantics. That example only indicates that in the context of Pascal the interpretation of types as sets is too abstract to be denotational. Two next sections are devoted to a general discussion of the relationship between the denotationality and the abstraction of a semantics.

## 7. Denotationality Versus Abstraction

Each semantics describes the effect of the execution of syntax. Of course, such an effect may be described in a more or less detailed way. For instance, the effect of the execution of an imperative program may be described by the set of sequences of memory states generated by all executions of that program, or — more abstractly — by a corresponding input-output function on states, or — even more abstractly — by a function on states truncated to the global variables of the program. The more abstract is a semantics, the less information is carried by denotations. The class of all semantics of a given syntax may be viewed as a spectrum *preordered* by a reflexive and a transitive relation of *abstraction*. On one end of that spectrum we have the least abstract semantics — which maps each syntactic object identically onto itself — and on the other end, the most abstract semantics, which maps all syntactic objects of the same sort onto a common denotation.

Of course, both extremes of our spectrum are trivial. A good semantics should provide all the relevant information about the e f f e c t of the execution of a piece of syntax, but at the same time it should hide all the irrelevant details of the execution itself. Of course, what is relevant and what is not depends on the current application. We should also be aware of the fact that if a semantics is to be denotational, then the abstraction levels of its components (e.g. of $I$, $E$, $C$, in the example of Sec.4.) must be mutually balanced. For instance, if the denotations of expressions do not carry enough information in order to compute the denotation of commands in which they appear, then the semantics is not denotational.

In Sec.6. we have seen two examples of non-denotational semantics. In the first example the denotations of commands do not include an information about *fai*'s, although that information is relevant when we execute compound commands. It seems intuitively rather clear that since we have made the execution of commands dependent on *fai*'s, we should have put an appropriate information on *fai*'s into the denotations, i.e. we should have made our semantics less abstract. In the example with Pascal, the set-theoretical meaning of a type is not sufficiently informative since each compiler of Pascal discriminates between two compound types with different names, unless they have been explicitly declared to be equal. The compilers of Pascal compare the definitions of types rather than their set-theoretical interpretations. We need, therefore, more information about a type — more than just the corresponding set of values — in order to predict the effect of the execution of a program where that type has been used. Again, in order to make our semantics denotational we have to make it less abstract.

This section is devoted to a formal discussion of a trade-off between denotationality and abstraction. Let us start from introducing a few basic concepts. Consider two, not necessarily denotational, semantics of the same syntax:

$$\begin{aligned} S &: \mathbf{Syn} \to \mathbf{Den} \\ S' &: \mathbf{Syn} \to \mathbf{Den}' \end{aligned}$$

where $\mathbf{Syn} = \langle \mathbf{Sig}, car\_s, fun\_s \rangle$ and $\mathbf{Sig} = \langle Sn, Fn, sort, arity \rangle$. We assume that if $S$ or $S'$ are not denotational, then **Den** respectively **Den**' are just families of sets rather than algebras. If for any $sn \in Sn$ and any $syn_1, syn_2 \in car\_s.sn$:

$$S'.sn.syn_1 = S'.sn.syn_2 \ \ \mathbf{implies} \ \ S.sn.syn_1 = S.sn.syn_2 \qquad (13)$$

i.e. if $\equiv_{S'} \subseteq \equiv_S$in a componentwise way, then we say that:

$$S' \text{ is } \textit{less abstract} \text{ than } S$$

We also say that $S'$ *is adequate* for $S$, since $S'$ bears — in a certain sense — at least as much information as $S$. Whenever we change a semantics by enriching denotations, the new semantics becomes less abstract (i.e. more informative) than the former.

We say that $S$ and $S'$ are *equally abstract* if $\equiv_{S'} = \equiv_S$. Of course, equally abstract semantics need not be equal. However, if $S$ and $S'$ are equally abstract and one of them is denotational, then so must be the other. In other words: we cannot adequately repair non-denotationality without a loss of abstraction.

**Proposition 7.1.** For any $S$ there exists a maximally abstract $S'$ which is both denotational and adequate for $S$.

**Proof:**
Let $\equiv^*$be the transitive closure of the union of all congruences which are less abstract than (which are the subsets of) $\equiv_S$. This relation is the largest congruence included in $\equiv_S$. The corresponding homomorphism $S' : \mathbf{Syn} \rightarrow \mathbf{Syn}/\equiv^*$ is the maximally abstract denotational semantics which is adequate for $S$. □

Of course, the semantics $S'$ constructed in this proof is not the unique semantics that satisfies Proposition 7.1. There are many such semantics, but all of them are equally abstract with $S'$.

When we construct a semantics we should care not only about its global level of abstraction, but also about a balance between the abstraction levels of the semantics assigned to different sorts of **Syn**. If syntactic objects of sort $sn_1$ are used in the construction of syntactic objects of sort $sn_2$, then on one hand the denotations of sort $sn_1$ should be sufficiently informative in order to enable the calculation of the denotations of sort $sn_2$, but on the other hand they should not carry more information than necessary, i.e. they should be as abstract as possible. In order to express that claim in a more formal way we introduce a few technical concepts.

Let **Syn** be a many-sorted algebra of syntax. By a *context* of arity $\langle sn_1 \rangle$ and sort $sn_2$ we mean a function of the type

$$ct \ : \ car\_s.sn_1 \rightarrow car\_s.sn_2$$

(where $car\_s.sn_i$'s are the carriers of **Syn** ) which represents a '*term with a hole*' like e.g. (cf.Sec.4.):

$$\begin{array}{ll}
(\lambda com)(com; x := 1) & : Com \rightarrow Com \\
(\lambda exp)(y := exp; x := 1) & : Exp \rightarrow Com \\
(\lambda ide)(y := ide + x; z := 1) & : Ide \rightarrow Com \\
(\lambda ide)(ide + x) & : Ide \rightarrow Exp
\end{array}$$

For a more formal definition of that concept, based on the notion of *derived operators*, see e.g. [28]. Now, let

$$S \ : \ \mathbf{Syn} \rightarrow \mathbf{Den}$$

be an arbitrary — not necessarily denotational — semantics. Two syntactic objects $syn_1$ and $syn_2$ of the same sort $sn_1$ are said to be *context-equivalent* w.r.t. a sort $sn_2$, in symbols

$$syn_1 \ CE.\langle sn_1, sn_2 \rangle \ syn_2$$

if one of them may be replaced by the other in any context of sort $sn_2$ without changing the meaning of the whole phrase, i.e. if for any context $ct : car\_s.sn_1 \rightarrow car\_s.sn_2$

$$S.sn_2.(ct.syn_1) = S.sn_2.(ct.syn_2)$$

Given two sorts $sn_1$ and $sn_2$ we say that the semantics $S.sn_1$ of $car\_s.sn_1$ is *sufficiently abstract* w.r.t. the semantics $S.sn_2$ of $car\_s.sn_2$, if any two syntactic objects of the sort $sn_1$ which are context-equivalent w.r.t. $sn_2$, have the same meaning, i.e. if for any $syn_1, syn_2 \in car\_s.sn_1$:

$$syn_1 \ CE.\langle sn_1, sn_2 \rangle \ syn_2 \ \ \textbf{implies} \ \ S.sn_1.syn_1 = S.sn_1.syn_2 \tag{14}$$

For instance, we say that the semantics of expressions of a programming language is sufficiently abstract w.r.t. the semantics of commands, if any two expressions which are context-equivalent in commands have the same denotation.

If in (14) the opposite implication holds, then we say that $S.sn_1$ is *sufficiently informative* w.r.t. $S.sn_2$.

**Proposition 7.2.** A semantics $S$ is denotational **iff** any two of its components $S.sn_i$ and $S.sn_j$ are sufficiently informative w.r.t. each other.

The proof of that proposition is quite routine, but since it requires the formalization of the concept of a context we omit it.

In many software systems there exists a sort — usually called '*programs*' — such that the syntactic objects of that sort may be executed without any context, whereas the objects of all other sorts — e.g. *expressions*, *declarations* or *commands* — can be executed only in the context of *programs*. The user of such a system is, of course, mainly interested in the execution of programs and in the semantics he wants to see the effect of that execution at an abstraction level adequate for the intended applications. For instance, in a general-purpose programming language an adequate abstraction level for programs may be represented by I/O functions, whereas in a programing language for controlling robots — by sets of sequences of states. The designer of the system should, therefore, choose an adequate abstraction level for the semantics of programs in the first place, and then he should 'tune' to it the abstraction level of all other component semantics in such a way that the whole semantics becomes denotational and maximally abstract.

If in a semantics $S$ all its components $S.sn_i$, including the component for programs, are sufficiently abstract w.r.t. the semantics of programs, then R.Milner [20] calls $S$ *fully abstract*. For instance, the semantics of our toy programming language of Sec.4. is fully abstract, if we assume that commands play the role of programs, but a semantics of a programming language with blocks, where the denotations of blocks include an information about local variables, is in general not fully abstract. In the latter case we can construct two blocks which have different denotations although they are interchangeable in any program.

The term '*full abstraction*' has not been chosen very adequately with respect to what the word '*full*' means in colloquial English. When we say '*fully abstract*' we could have expected that the semantics in question is 'as abstract as it can be'. In fact, however, it is only sufficiently abstract with respect to the semantics of programs. If, therefore, the semantics of programs is little abstract, then the other semantics may be also little abstract. For instance, a trivial semantics

$$S^s : \mathbf{Syn} \to \mathbf{Syn}$$

which maps syntax identically into itself (the supersctipt '$s$' in $S^s$ stands for '*syntax*') is fully abstract. Since it is also denotational and adequate with respect to any other semantics of the same syntax, we can formulate the following trivial proposition about the reparation of non-denotational semantics:

**Proposition 7.3.** For any semantics $S : \mathbf{Syn} \to \mathbf{Den}$ there exists another semantics $S' : \mathbf{Syn} \to \mathbf{Den}'$ such that:

1) $S'$ is denotational,
2) $S'$ is adequate w.r.t. S,
3) $S'$ is fully abstract.

Of course, 3) above makes only sense if the common signature of **Syn** and **Den** contains a sort of programs. For the sake of further investigations (in Sec.8.) we slightly generalize the concept introduced by R.Milner and say that a given semantics S is *fully abstract* w.r.t. a given sort $sn$ if all $S.sn_i$'s are sufficiently abstract w.r.t. $S.sn$. We also assume that the concept of full abstraction is applicable to any semantics rather than only to denotational semantics, as it was the case with Milner's definition.

## 8. Repairing Semantics

In this section we discuss the problems of repairing non-denotational or not fully abstract semantics. We start from non-denotationality.

As Proposition 7.3. indicates, the repairability problem for non-denotational semantics should be formulated with a a certain care, since otherwise it may become trivial. Even if we request that besides satisfying (1)-(3) of that proposition the new semantics preserves all the information that was carried by $S$, we can still find a trivial solution, namely

$$S^{d+s} : Syn \to Den \times Syn$$

where '$d+s$' stands for '*denotations plus syntax*' and where $S^{d+s}.syn = \langle S.syn, syn\rangle$. The semantics $S^{d+s}$ is as little abstract as $S^s$ of Sec. 7., i.e. $\equiv_{S^{d+s}} = \equiv_{S^s}$. The latter is denotational at the expense that it says nothing about the execution of programs. The former says everything that $S$ does, but it adds the whole syntax to that information. If we want to use $S^{d+s}$ in order to tell a programmer what his program is supposed to do, we have to give him that program explicitly. The semantic $S^{d+s}$ is therefore as useless as $S^s$.

Although of no value for applications, $S^{d+s}$ indicates a certain way of searching for the most abstract denotational semantics which is adequate for $S$ (cf. Proposition 7.1.) In fact, we can always try to repair $S$ by a semantics $S^r$, ('$r$' stands for '*repair*') where $S^r.syn = \langle S.syn, A.syn\rangle$ and where $A.syn$ provides the lacking information which makes $S^r$ denotational. In the worst case $A.syn = syn$, but frequently we can do much better.

Consider as an example the non-denotational language described in Sec. 6. and let, a little informally, $S = \langle I, E, C\rangle$. In this semantics $C$ is not sufficiently informative w.r.t. itself since the denotations of commands lack the information about the first assignable identifier and therefore the denotation of a compound command cannot be "computed" from the denotations of its subcommands. In order to repair $S$ we have to add the missing information to the denotations of commands. This leads to a semantics $S^r = \langle I, E, C^r\rangle$, where $I$ and $E$ are the same as in $S$ and where:

$$C^r : Com \to Executor \times Ide$$

$$\begin{aligned} &C^r.[ide := exp] = \\ &\quad \langle (\lambda sta) sta[E.[exp].sta/ide], ide\rangle \end{aligned} \tag{15}$$

$$\begin{aligned} &C^r.[(com_1; com_2)] = \\ &\quad \textbf{let } \langle exe_i, ide_i\rangle = C^r.[com_i] \textbf{ for } i = 1,2 \textbf{ in} \\ &\quad ide_1 \neq ide_2 \to \langle exe_1 \bullet exe_2, ide_1\rangle \\ &\quad \text{TRUE} \to \langle (\lambda sta) nullsta, ide_1\rangle \end{aligned} \tag{16}$$

The semantics $S^r$ is, of course, both adequate for the former one and denotational. We can also show that it is a maximally abstract such semantics. Indeed, assume that $S' = (I', E', C')$ is denotational, inherently more abstract than $S^r$ and adequate for $S$. By these assumptions $I'$ and $E'$ must be equally abstract with $I$ and $E$ respectively. Therefore $C'$ must be inherently more abstract than $C^r$, i.e. there must be two commands $com_1$ and $com_2$ such that:

$$\begin{aligned} &1)\ C^r.[com_1] \neq C^r.[com_2] \quad \text{and} \\ &2)\ C'.[com_1] = C'.[com_2] \end{aligned}$$

Let $C^r.[com_i] = \langle exe_i, ide_i\rangle$ for $i = 1, 2$. From the adequacy of $C'$ for $C$ and by 2) we may conclude that $C.[com_1] = C.[com_2]$, i.e. $exe_1 = exe_2$. Therefore, by 1), $ide_1 \neq ide_2$. In that case:

$$C.[(com_1 ; ide_2 := 1)] \neq C.[(com_2 ; ide_2 := 1)]$$

and hence, again by the adequacy of $C'$ for $C$,

$$C'.[(com_1 ; ide_2 := 1)] \neq C'.[(com_2 ; ide_2 := 1)]$$

which in the virtue of 2) contradicts the denotationality of $S'$.

It is also not difficult to show that our semantics $S^r$ is fully abstract w.r.t. commands. One should only prove that any two identifiers, any two expressions and any two commands which have different denotations can be discriminated by a certain command-context. An easy proof is left to the reader.

It should be stressed that although in our example the most abstract denotational semantics that adequately repairs $S$ turned out to be fully abstract, it does not need to be so in general. In fact, $S^r$ is fully abstract because the original semantics $S$ was so. Assume, however, that we repair a semantics which is similar to that of Sec. 6., but where the denotations of expressions include also their syntax. In that semantics $E$ is not sufficiently abstract w.r.t. $C$ and that property will be preserved in the corresponding most abstract $S^r$, since on the way from $S$ to $S^r$ we are lowering the level of abstraction.

The denotationality and the full abstraction of a semantics may be both repaired (or spoiled) either by lowering or by raising the abstraction level. It is so since both these properties require a certain balance between the abstraction levels of the components of a semantics. Let us illustrate this claim by an example where we compare four different semantics.

Assume first that we extend the syntax of the language defined in Sec. 4. by a new sort called *programs*:

$$prog \; : \; Program \; = \; \{\mathbf{begin}\} \; Com \; \{\mathbf{end}\}$$

For the extended syntax we define four different semantics:

1. Semantics $S$, denotational and fully abstract, which results from the (natural) semantics defined in Sec. 4. by setting:

$$P : Program \rightarrow Executor$$
$$P.[\mathbf{begin}\; com\; \mathbf{end}] = C.[com]$$

2. Semantics $S^{nd}$, not denotational but fully abstract, which results from the semantics of (1) by assuming that $C^{nd}$ has been spoiled as in (11), by making the denotation of a compound command dependent on first assignable identifiers. This semantics is not denotational since $C^{nd}$ is not sufficiently informative with respect to itself; $C^{nd}$ does not give enough information for predicting the effect of the execution of a command in the context of another command.
3. Semantics $S^{nfa}$, denotational but not fully abstract, which results from the semantics of (1) by assuming the following:

$$C^{nfa} \; : \; Com \rightarrow Executor \times Ide$$
$$P^{nfa} \; : \; Program \rightarrow Executor$$

$$C^{nfa}.[ide := exp] = \langle(\lambda sta)sta[(E.[exp].sta)/ide], ide\rangle$$
$$C^{nfa}.[(com_1; com_2)] = \mathbf{let}\; \langle exe_i, ide_i\rangle = C^{nfa}.[com_i] \;\mathbf{for}\; i = 1, 2 \; In \; \langle exe_1 \bullet exe_2, ide_1\rangle$$

$$P^{nfa}.[\mathbf{begin}\; com\; \mathbf{end}] \; = \; first.(C^{nfa}.[com])$$

Here the denotations of commands include the information about *fai*'s, but unlike in (15) this information is irrelevant for the "executional effect" of commands. The semantics $S^{nfa}$ is not fully abstract since $C^{nfa}$ is not sufficiently abstract with respect to $P^{nfa}$. Indeed, $C^{nfa}$ gives more information then one needs to calculate the denotations of programs.
4. Semantics $S^r$, denotational and fully abstract, which results from the semantics defined at the beginning of this section (i.e. $C^r$ is defined by (15) and (16)) by setting the semantics of programs as follows:

$$P^r \; : \; Program \rightarrow Executor \times Ide$$

$$P^r.[\mathbf{begin}\; com\; \mathbf{end}] = C^r.[com]$$

$$\begin{tikzcd}
 & & \begin{array}{c}(Exe, Exe)\\ d\&fa\end{array} & & \\
 & & S & & \\
\begin{array}{c}(Exe, Exe)\\ nd\&fa\end{array} & S^{nd} \arrow[ur, "\text{impl.}"] & & S^{nfa} \arrow[ul, "\text{den.}"'] & \begin{array}{c}(Exe \times Ide, Exe)\\ d\&nfa\end{array} \\
 & & S^{r} \arrow[ul, "\text{den.}"] \arrow[ur, "\text{den. and impl.}"'] & & \\
 & & \begin{array}{c}(Exe \times Ide, Exe \times Ide)\\ d\&fa\end{array} & &
\end{tikzcd}$$

Figure 2 A diamond of semantics

If by $S_1 \longrightarrow S_2$ we denote the fact that $S_1$ is inherently less abstract than $S_2$, then our four semantics constitute the diagram of Fig. 2, where in parentheses we indicate the current domains of the denotations of commands and of programs, respectively.

Let us analyze this diagram more carefully since it puts some light onto the nature of denotationality and full abstraction. In our discussion we shall use an intuitive concept of an *implementation* of a language (i.e. of its function of semantics) by which we do not mean the literal code of an interpreter or a compiler, but rather the way in which software is executed, e.g. whether we do or we do not care about *fai*'s.

When we pass from semantics $S^r$ to $S^{nd}$ we change the denotations of commands from *Executor* $\times$ *Ide* to *Executor* without changing their implementation. We loose the information about *fai*'s, while the semantics (implementation) still needs it. This makes $C^r$ not sufficiently informative w.r.t. itself, which makes $S^{nd}$ not denotational. Full abstraction is preserved.

In passing from $S^{nd}$ to $S$ we change the implementation of commands without changing their denotations. We recover the denotationality of semantics by making it (and the implementation) independent on *fai*'s. The semantics of commands becomes sufficiently informative w.r.t. itself. Full abstraction is preserved.

On the way from $S^r$ to $S^{nfa}$ we change the implementation of commands and the denotations of programs. Full abstraction is spoiled by the overhead of information in the denotations of commands. We keep there the information about *fai*'s although that information is not used anymore by the semantics (and the implementation) of commands and programs. Observe that the semantics $S^{nfa}$ which is inherently more abstract than a fully abstract semantics $S^r$ is not fully abstract itself. Denotationality is preserved since new denotations provide all information that is needed to make semantics compositional: the semantics of programs does not refer to *fai*'s.

Finally, when we pass from $S^{nfa}$ to $S$ we only change denotations. We repair the full abstraction of $S^{nfa}$ by removing the overhead of information from the denotations of commands. Of course, the removal of an unnecessary information does not spoil denotationality.

As we already said earlier, both non-denotationality and non-full-abstraction reflect a lack of ballance between the abstraction levels of the components of a semantics. In addition to that our analysis indicates that they also reflect an imballance between the amount of information which is needed by the implementation and which is included in the denotations. There are, therefore, two strategies of repairing semantics. One consists of tuning the implementation to what we are willing to talk about in semantics, e.g.:

- we repair denotationality by passing from $S^{nd}$ to $S$, or
- we repair full abstraction by passing from $S^{nfa}$ to $S^r$

and the other one consists of tuning the denotations to what the implementation needs, e.g.:

- we repair denotationality by passing from $S^{nd}$ to $S^r$, or
- we repare full abstraction by passing from $S^{nfa}$ to $S$.

The choice between these strategies is, in each concrete case, a pragmatic issue. We change denotations whenever we feel that the executional effect of software meets our expectations and that all we want to do is to describe that effect in a denotational resp. fully abstract way. We change implementation if we decide that the current denotations adequately express the intended effect of the execution of software, but the implementation does not meet that expectation.

In the case of $S^{nd}$ the designer of the language should probably conclude that the denotations of commands are adequate and therefore he should correct his semantics by changing it to $S$ — i.e. by changing the implementation — rather than to $S^r$. The case where the other strategy seems to be more appropriate is that of Pascal (Sec. 6.) In that case the designer should change the denotations rather than the implementation. Pascal types should not be regarded as sets since they have to be compared during the execution of programs, and the comparison of sets may be computationally too expensive, if computable at all. Readers interested in a denotational model of Pascal types are refered to [5]).

To complete our discussion let us emphasise that the problem of repairing a semantics is inherent to the traditional style of constructing denotational models of software, where syntax is given in the first place and semantics is defined for it later. As was already pointed out in Sec. 4., when we design a software system from a scratch, a safe systematic way of constructing a denotational semantics consists in starting the designing process from the algebra of denotations. In that case we have no syntax around and therefore we have no chance of making the constructors of denotations dependent on syntax, hence of making the future semantics not denotational. Once the algebra of denotations is ready, we derive an appropriate syntax for it and the derivation method guarantees that the resulting semantics is denotational. E.g. it is technically impossible to construct $S^{nd}$ in using that method.

Now let us briefly comment on the construction of a logic for the non-denotational language defined in Sec. 6. As we have mentioned there, the only rational way of tackling the problem of logic in such a case seemed to first repair the non-denotationality of the semantics and then to construct the logic in the usual way, i.e. as described in Sec. 5. For the sake of the present discussion let us assume that we repair the semantics $S^{nd}$ by changing it to $S^r$, since otherwise — i.e. in the case of $S$ — the logic has been described already in Sec. 5. For simplicity we omit programs in our investigations.

Let us start from defining explicitly two constructors of command denotations for which we intend to develop our proof rules. Let

$$\begin{aligned}
\|asg\|^r &: Ide \times Evaluator \rightarrow Executor \times Ide \\
\|follow\|^r &: Executor \times Ide \rightarrow Executor \times Ide
\end{aligned}$$

be defined by:

$$\begin{aligned}
&\|asg\|^r.\langle ide, eva\rangle = \langle \|asg\|.\langle ide, eva\rangle, ide\rangle \\
&\|follow\|^r.\langle\langle exe_1, ide_1\rangle, \langle exe_2, ide_2\rangle\rangle = \\
&\quad ide_1 \neq ide_2 \rightarrow \langle exe_1 \bullet exe_2, ide_1\rangle \\
&\quad ide_1 = ide_2 \rightarrow \langle(\lambda sta) nullsta, ide_1\rangle
\end{aligned}$$

In order to construct a Hoare-like logic of partial correctness for the new semantics we have to modify the concept of partial correctness in such a way that it captures also an information about *fai*'s. Let:

$$\|parcorfai\| \ : \ Condition \times Condition \times Ide \rightarrow Predicate$$

$$\|parcorfai\|.\langle con_1, con_2, ide\rangle.\langle exe, ide_e\rangle = \\ (\forall sta \in State)[con_1.sta = true \textbf{ implies } con_2.(exe.sta) = true] \textbf{ and } ide = ide_e$$

and let

$$\textbf{pre } con_1 \ : exe \textbf{ post } con_2 \textbf{ fai } ide$$

stand for

$$\|parcorfai\|.\langle con_1, con_2, ide\rangle.\langle exe, ide_e\rangle = true$$

Let $allzero \in Condition$ where:

$$allzero.sta = true \textbf{ iff } (\forall ide)(sta.ide = 0.0)$$

The modified rules are as follows:

$$\begin{array}{l} \text{for any } con_i \text{ and } \langle exe_i, ide_i\rangle \text{ for } i = 1,2 : \\ \quad \textbf{pre } con_1 \ : \|follow\|^r.\langle\langle exe_1, ide_1\rangle.\langle exe_2, ide_2\rangle\rangle \textbf{ post } con_2 \textbf{ fai } ide_1 \\ \quad\quad \textbf{iff} \\ \quad \text{there exist conditions } con_{11},\ con_{12} \text{ such that :} \\ \quad\quad - \textbf{pre } con_1 \ : exe_1 \textbf{ post } con_{11} \textbf{ fai } ide_1 \\ \quad\quad - \textbf{pre } con_{12} \ : exe_2 \textbf{ post } con_2 \textbf{ fai } ide_2 \\ \quad\quad - \textbf{if } ide_1 \neq ide_2 \\ \quad\quad\quad \textbf{then } (\forall sta)(con_{11}.sta = true \textbf{ implies } con_{12}.sta = true)] \\ \quad\quad - \textbf{if } ide_1 = ide_2 \\ \quad\quad\quad \textbf{then } (\forall sta)(con_2.sta = true \textbf{ implies } allzero.sta = true)] \end{array} \tag{17}$$

$$\begin{array}{l} \text{for any } con_1,\ con_2 \text{ and for any } \langle ide, eva\rangle : \\ \quad \textbf{pre } con_1 \ : \|asg\|^r.\langle ide, eva\rangle \textbf{ post } con_2 \textbf{ fai } ide \\ \quad\quad \textbf{iff} \\ \quad (\forall sta)(con_1.sta = true \ \textbf{ implies } con_2.sta[(eva.sta)/ide] = true) \end{array} \tag{18}$$

For the sake of brevity we shall not transform these rules into formalized inference rules as in Sec. 5. Let us only mention that although the rules developed here correspond in some sense to the language of Sec. 6., they are not formally applicable to that language. The reason is that

$$\textbf{pre } con_1 \ : exe \textbf{ post } con_2 \textbf{ fai } ide$$

describes a property of $\langle exe, ide\rangle$ rather than of $exe$, whereas $exe$, rather than $\langle exe, ide\rangle$, are the denotations of commands in the language of Sec. 6. We can also see quite clearly now that neither for $S^r$, hence also nor for $S^{nd}$, there exist predicates on executors that could be used in the proofs of their partial correctness. This means that there is no logic of structured-inductive proofs of the partial correctness of commands for the language of Sec. 6.

## 9. On the Borderline of Denotationality

No theory can capture all the reality. In some applications a very orthodox attitude to the principle of denotationality may lead to overcomplicated or to non-implementable models. In such cases a pragmatic solution may consist of adding a non-denotational supplement to a denotational core of the model. As long as the bulk of the system is described in a compositional way, such a style may be still acceptable. Below we discuss two typical examples. A third, rather singular, is discussed in Sec. 10.

Our first example is related to types in programming languages. As we have mentioned already in Sec. 5. and Sec. 6., types that we want to think about may be sets with a non-computable equality relation, whereas types that we implement are usually less abstract objects, e.g. some equivalence classes of expressions with a computable equality relation. The former are called *domain types* and the latter are called *symbolic types.* Each domain type may be, in general, represented by many symbolic types, but each symbolic type represents exactly one domain type. The reader is referred to [5] for an example of the use of symbolic- versus domain types in the semantics of Pascal, and to [1] for a more general treatment of that problem.

There are two different strategies of constructing a mathematical semantics of a programming language with types. One — which is probably most common today — consists of assuming that the domain types appear explicitly in the semantics, i.e. are assigned to variables, whereas the symbolic types are implicit. In that case, whenever we compare the types of two variables we check if a certain equivalence relation between the definitions of these types is satisfied. That strategy is most frequently associated with the technique of partitioning a semantics into a static semantics and a dynamic semantics, which is typical for VDM (see e.g. [2]). The corresponding semantics is then in general not denotational (cf.Sec. 6.)

The second strategy is dual to the former and consists of assuming that symbolic types are explicit in the semantics, whereas domain types are implicit. In that case variables are typed by symbolic types and whenever we compare the types of two variables we compare the corresponding symbolic types rather than their definitions. This makes our semantics denotational. In addition to the definition of semantics we define in that case a function

$$D \ : \ \mathit{SymbolicType} \rightarrow \mathit{DomainType}$$

which tells the user what the symbolic types stand for. Of course, our language should have the following adequacy property:

> "If in the execution of a program a variable $v$ has been declared to be of a type *symtype*, then everywhere in the scope of that declaration the value of $v$ belongs to $D$.*symtype*."

In general *SymbolicType* and *DomainType* constitute two algebras over the same signature. That signature usually contains several sorts which correspond to different classes of types, such as e.g. *scalartypes*, *recordtypes*, *filetypes*, etc., plus a sort posessing the same carrier *Bool* in both algebras. The latter sort is needed in order to define an equivalence relation, a subtype relation, etc. between types. Now, if we forget about the boolean sort, then $D$ is usually a homomorphism, hence it may be regarded as a denotational semantics of symbolic types. However, in the full algebra of types $D$ is not a homomorphism since two non-equal symbolic types may denote the same domain type. The (partial) non-compositionality of $D$ is inherent to the difference between symbolic- and domain types and therefore it cannot be avoided.

The definition of $D$ may be regarded as a non-denotational supplement of the main definition of semantics. This fact is not very harmful since it neither affects the feasibility of structured programming in the language nor the construction of the corresponding logic.

Another typical situation where we may wish to slightly relax the principle of denotationality corresponds to the case where we modify an existing syntax by introducing some notational conventions. For instance, we may wish to allow for the optionality of parentheses in expressions while introducing some priority rules for operators. In that case the mathematical model of our system is usually described by the following diagram:

$$\mathbf{SynE} \xrightarrow[\mathrm{P}]{} \mathbf{Syn} \xrightarrow[\mathrm{D}]{} \mathbf{Den}$$

where **SynE** denotes the extended syntax and $P$ is a preprocessing function. In general $P$ is not a homomorphism in the strict sense of the word since **SynE** may have a different signature than **Syn**. However, $P$ is usually a homomorphism in a generalized sense, namely a homomorpism over a morphism of signatures. We shall not go here into any technical details, but explain our remark on a simple example. Consider two syntaxes of arithmetic expressions described by the following equational grammars (cf.[4]):

| **SynE** | **Syn** |
|---|---|
| $ExpE = Cpn \mid Cpn\{+\}ExpE$ | $Exp = \{x\}$ |
| $Cpn \;\; = Fac \mid Fac\{*\}Cpn$ | $\mid(Exp\{+\}Exp)$ |
| $Fac \;\; = \{x\} \mid (ExpE)$ | $\mid(Exp\{*\}Exp)$ |

Each of these grammars defines unambiguously an algebra of syntax (cf. [8]). The former has three carriers: *ExpE* — *extended expressions*, *Cpn* — *components* and *Fac* — *factors*. The latter has only one carrier, *Exp* — *expressions*. Notice that **SynE** imposes a priority of $*$ over $+$. Now, the preprocessing $P$ is a many-sorted function that mapps all the three carriers of **SynE** into the unique carrier of **Syn**. Formally, it is represented by three functions:

$$\begin{aligned} E &: ExpE \rightarrow Exp \\ C &: Cpn \rightarrow Exp \\ F &: Fac \rightarrow Exp \end{aligned}$$

defined by the following equations:

$$\begin{aligned} E.[cpn] &= C.[cpn] \\ E.[cpn + expe] &= (C.[cpn] + E.[expe]) \\ C.[fac] &= F.[fac] \\ C.[fac * cpn] &= (F.[fac] * C.[cpn]) \\ F.[x] &= x \\ F.[(expe)] &= E.[expe] \end{aligned}$$

Observe that although $P = \langle E, C, F\rangle$ is not a homomorphism in a strict sense, it certainly has a compositional character and in fact is not very far from a usual homomorphims.

The moral of our two stories is that if the major mechanisms of a system are described within a denotational model, then some "peripheral" information may be given in a not strictly compositional way. Of course, what is peripheral and what is not is an informal question and therefore in all such cases the designer has to rely on his/her own professional experience and common sense. Little non-denotationalities are not too harmful, but it is clear that a too rich pre- or post-processing of the "main semantics" may completely destroy the compositional effect of the latter.

## 10. Copy-Rule Semantics

A copy-rule semantics has been known for many years as a technique for providing a mathematical semantics of a typeless lambda-calculus in which a function may take itself as an argument. In applications this allows one to formalize such programming mechanisms as e.g. Algol-60 procedural parameters or Lisp dynamic recursion. The idea of copy rule is rather simple (cf.[17]), and corresponds closely to the way in which the original informal semantics of Algol-60 was described. A procedure declaration assigns the text of the procedure body — rather than the corresponding state-transition function — to the procedure name in the environment. At the call time this text is retrieved, and its denotation is applied to the current state.

Copy-rule semantics is not denotational. A denotational semantics of lambda-calculus may be defined using Scott's model of reflexive domains [25] or one of its later versions, e.g. information systems [26]. Although mathematically very elegant, these models are not simple and therefore — at least in the opinion of some authors — not very convenient in applications. Consequently, the majority of software specification systems, such as e.g. BSI/VDM [18], **MetaSoft** [5, 1] or RAISE [21] are based on set-theoretic domains rather than on Scott's models. In all such systems one cannot define a denotational semantics of Algol-60 or of Lisp, but one can give them a copy-rule semantics.

Although the use of self-applicable functions in software systems is certainly not recommendable, and the problem of giving a mathematical semantics to Algol-60 or Lisp is today only of a historical nature, it may be of some interest to know what is the price of using a copy-rule semantics. As we shall try to argue, that price — measured by the loss of denotationality and abstraction — does not seem very high, especially if the only alternative are reflexive domains.

Let us analyze an example of a programming language with self-applicable procedures, i.e. a language in which every procedure may take any other procedure — even itself — is an actual parameter. To simplify our example we assume that every procedure has exactly one parameter which is always a procedure, and that all variables in a procedure body are global. The syntax of our language is the following:

$$\begin{array}{llll} ide : Ide & = a \mid b \mid ... \mid z & & \text{identifiers} \\ exp : Exp & = Ide \mid (Exp + Exp) \mid \ldots & & \text{expressions} \\ dec : Dec & = \mathbf{proc}\ Ide(Ide) = Com \mid Dec; Dec & & \text{declarations} \\ com : Com & = Ide := Exp \mid \mathbf{call}\ Ide(Ide) \mid Com; Com \mid \ldots & & \text{commands} \\ pro : Pro & = \mathbf{begin}\ Dec : Com\ \mathbf{end} \mid Pro; Pro & & \text{programs} \end{array}$$

*States* in our language are triples consisting of an *environment*, used for storing procedures, a *store*, for storing values and a *message* which is either an OK-*message* or an *error message*:

$$\begin{array}{lll} sta : State & = Environment \times Store \times Message \\ env : Environment & = Ide \xrightarrow[m]{} Procedure \\ sto : Store & = Ide \xrightarrow[m]{} Value \\ mes : Message & = \{\text{OK}, \text{ERROR}\} \\ prc : Procedure & = Ide \times Com \\ val : Value & = Integer \mid \ldots \end{array}$$

Observe that procedures are pairs consisting of an identifier (formal parameter) and a command (body). The functions of semantics have the following signatures:

$$\begin{array}{ll} I & : Ide \rightarrow Ide \\ E & : Exp \rightarrow Store \rightarrow Value \\ D & : Dec \rightarrow Environment \rightarrow Environment \\ C & : Com \rightarrow State \rightsquigarrow State \\ P & : Pro \rightarrow Store \rightsquigarrow Store \end{array}$$

Below we analyze only the interesting semantic clauses i.e. the clauses for declarations, for calls and for single-block programs. All others have the usual form.

$$\begin{array}{ll} D.[\mathbf{proc}\ ide_{pn}(ide_{fp}) = com].env = env[\langle ide_{fp}, com\rangle / ide_{pn}] \\ D.[dec_1; dec_2].env & = D.[dec_2].(D.[dec_1].env) \end{array}$$

A procedure declaration assigns the corresponding procedure, i.e. both the formal parameter and the body, to the procedure name in the current environment. Semicolon is interpreted in the usual way.

```
$C.[\textbf{call}\ ide_{pn}(ide_{ap})].\langle env, sto, mes\rangle =$
    $mes = \text{ERROR} \qquad\qquad \rightarrow \langle env, sto, \text{ERROR}\rangle,$
    $\text{NOT}\ \ ide_{pn} \in dom.env \rightarrow \langle env, sto, \text{ERROR}\rangle,$
    $\text{NOT}\ \ ide_{ap} \in dom.env \rightarrow \langle env, sto, \text{ERROR}\rangle,$
    $\textbf{let}\ \langle ide_{fp}, com\rangle = env.ide_{pn}\ \ \textbf{in}$
    $\textbf{let}\ prc = env.ide_{ap}\ \ \textbf{in}$
    $\text{TRUE} \qquad\qquad\qquad \rightarrow C.[com].\langle env[prc/ide_{fp}], sto, mes\rangle$
```

After all necessary error checks a procedure call applies the denotation $C.[com]$ of the corresponding procedure body to a state in which the current environment has been modified by assigning the value of the actual parameter to the formal parameter. We recall that all identifiers in the procedure body are global and therefore there is no need to rename them.

```
$P.[\textbf{begin}\ dec : com\ \textbf{end}].sto =$
    $\textbf{let}\ env = D.[dec].[]\ \textbf{in}$
    $C.[com].\langle env, sto, \text{OK}\rangle$
```

The execution of a program consists of three steps. First an environment is created by applying the declaration part of the program to an empty environment. Then a state is created by combining that environment with the current store and the OK message. Finally, the denotation of the command part of the program is applied to that state.

As is not difficult to see, our semantics has the following two properties:

1) $E$ and $C$ are not sufficiently informative w.r.t. $D$, since the denotation of a declaration depends on the syntax rather than on the denotations of its components.
2) $D$ is not sufficiently abstract w.r.t. $P$, because the denotation of a program depends on the denotation of a procedure body rather than on that procedure body itself.

By 1), our semantics is not denotational, and, by 2), it is not fully abstract. If we assume that the implicit part of the language follows the usual style of e.g. Algol-60, then 1) and 2) are the only violations of denotationality and full abstraction, respectively. What does that mean for the user of the language?

From a formal viewpoint, our language does not provide a fully adequate framework for structured programming (cf. Sec.5.) For instance, it is impossible to define a programmer's task of writing a declaration of a procedure $\langle ide, com\rangle$ by giving only *ide* and $C.[com]$ and it is not worthwhile to do the same with a program **begin** *dec* : *com* **end** by giving $D.[dec]$ and $C.[com]$. However a "local structured programming" is possible. A project coordinator may split the task of writing a compound command with a given denotation into the subtasks of writing subcommands (and/or subexpressions) with appropriate denotations or to split the task of writing a compound program into the tasks of writing a sequence of subprograms. Moreover, he/she may split the task of writing a program **begin** *dec* : *com* **end** into a task of writing *dec* with a given denotation $C.[com_b]$ of the procedure body and a given denotation $C.[com]$. Both these tasks may be further structurally decomposed.

Regarding the logic for our language, the related problems and their solutions are similar to the former. We can infer the properties of a compound command from the properties of its component expressions and commands, and the properties of a compound program from the properties of its subprograms. We cannot infer the properties of a (denotation of a) declaration from the properties of its components, but we can infer the properties of a program from the properties of the components of its declaration. Readers interested in a detailed discussion of Hoare-like logics for different copy-rule schemes are referred to a very elegant paper [22].

## 11. Operational Definition of a Denotational Semantics

As was mentioned in the Introduction, a denotational semantics may have a non-denotational definition. In this section we briefly discuss a typical example of such a definition written in the style of structured operational semantics (SOS) introduced in G.Plotkin [23].

Consider our little programming language and its semantics as defined in Sec.8. We shall discuss an SOS-definition of the semantics of commands $C$. For that sake we introduce a few auxiliary concepts. Let:

$$\begin{array}{lll} pcom : Pseudocommand & = & Com \mid \{nil\} \\ conf : Configuration & = & Pseudocommand \times State \\ \quad\;\; TerminalConf & = & \{nil\} \mid State \end{array}$$

A configuration may be interpreted as a global memory state of a von Neuman machine where we store both data and programs. By a *transition relation* between configurations we mean the least transitive and reflexive relation

$$\longrightarrow \subseteq Configuration \times Configuration$$

with three following properties:

$$\langle ide = exp, sta\rangle \longrightarrow \langle nil, sta[(E.[exp].sta)/ide]\rangle \tag{19}$$

$$\frac{\langle com_1, sta\rangle \longrightarrow \langle com_1', sta'\rangle}{\langle com_1; com_2, sta\rangle \longrightarrow \langle com_1'; com_2, sta'\rangle} \tag{20}$$

$$\frac{\langle com_1, sta\rangle \longrightarrow \langle nil, sta'\rangle}{\langle com_1\ com_2, sta\rangle \longrightarrow \langle com_2, sta'\rangle} \tag{21}$$

The latter two formulas should be, of course, read as top-down implications. The transition relation describes a way in which our machine transforms a global state when executing a program. If $conf_1 \longrightarrow conf_2$ holds, then we say that $conf_1$ *reduces to* $conf_2$. A sequence of configurations:

$$conf_1 \longrightarrow conf_2 \longrightarrow \ldots \longrightarrow conf_n$$

is called a *computation*. If $conf_n$ is a terminal configuration, then the above sequence is called a *terminating computation*. Below we show a simple example of a terminating computation:

```
pseudocommand  state
⟨ x:=1;y:=2;z:=x+y, sta⟩
⟨      y:=2;z:=x+y, sta[1.0/x]⟩
⟨           z:=x+y, sta[1.0/x, 2.0/y]⟩
⟨              nil, sta[1.0/x, 2.0/y, 3.0/z]⟩
```

Properties (19) and (21) imply — by structural induction — that for each non-terminal configuration $\langle com, sta\rangle$ there is exactly one terminal configuration $\langle nil, sta'\rangle$ such that:

$$\langle com, sta\rangle \longrightarrow \langle nil, sta'\rangle$$

This proves the existence of a function of semantics:

$$\begin{array}{l} C : Com \rightarrow State \rightarrow State \\ C.[com].sta = sta' \quad iff \quad \langle com, sta\rangle \longrightarrow \langle nil, sta'\rangle \end{array}$$

Now, (19) and (21) imply (respectively) two following properties of $C$:

$$\begin{aligned} &C.[ide \; = exp].sta \; = sta[(E.[exp].sta)/ide] \\ &C.[com_1 \; com_2].sta = C.[com_2].(C.[com_1].sta) \end{aligned}$$

This means that our function is a component of a homomorphism $\langle I, E, C\rangle$ between **Syn** and **Den** as defined in Sec.4.

Some authors prefer the SOS style as more appealing to the intuition than denotational equations. When defining a denotational semantics in the SOS style one has to remember, however, about two proof obligations:

1. that the corresponding function of semantics exists and is total,
2. that the function of semantics has the compositionality property.

Of course, in practical situations both these proofs may be far from trivial.

## 12. Final Remarks

In the Introduction we have formulated a claim that the semantics of a software system should be denotational, unless we agree to give up structured programming and structured-induction correctness-proofs. We have discussed some arguments that denotationality does guarantee these possibilities, and an example indicating that this may not be the case if denotationality is not ensured. At the same time, however, we have shown that the principle of denotationality may be always trivially insured by adding syntax to denotations. Doesn't that mean that denotationality is merely a property of the definition of a system rather than — as we have claimed earlier — a property of the system itself?

Each software system is a tool used for the construction of some applications (programs), and therefore an adequate mathematical model of a system should provide a ground for a convenient description and validation of these applications. Whether a model is sufficiently adequate depends, generally speaking, on two factors: on the choice of denotations and on the compositionality of semantics. Compositionality is important but it is not a goal in itself. It is worth of a care only if denotations adequately express the behavior of programs and of its components. And once we fix denotations, the compositionality of semantics becomes a property of a system rather than of its definition.

Our requirement of denotationality should be understood as a pragmatic rule. By choosing some "clever" denotations we may be able to construct an elegant algebraic model for a mechanism which is neither elegant nor algebraic. We have seen a simple example of such a situation in Sec.8. Some more interesting examples are related to the well-known technique of continuations. Using that technique one may construct a denotational model of a very unstructured programming language with most anarchic goto's. Formally, such a language guarantees the feasibility of structured programming but this is on the price that the denotations of programs are even more difficult to read, decompose and analyze than unstructured flowcharts. By the use of continuations the principle of denotationality is "cheated" in a very subtle way by putting all the conceptual mess of a language deep into denotations.

One of the advantages of a formal mathematical semantics is the possibility of discovering awkward mechanisms of a software system at the stage of its design rather than at the stage of its use. It has been rather generally agreed that a complicated definition of a mechanism should be regarded as a warning that the use of such a mechanism may be not easy. However, the simplicity of a definition — especially if this is a local simplicity, as e.g. in the case of a continuation-style definition of goto's — does not guarantee a sufficient simplicity of applications. The latter may be adequately estimated by analyzing the corresponding proof rule. Therefore the construction of a program-correctness logic should be regarded as an inherent part of the process of system design. Whenever we cheat on the subject of denotationality too much, the price to be paid is the complexity of proof-rules.

## Acknowledgements

The author expresses his thanks to Marek Gondzio, Beata Konikowska, Jacek Leszczyłowski, Peter Mosses and Wiesław Pawłowski for many relevant comments about the early drafts of this paper. Special thanks are addressed to Andrzej Tarlecki who suggested the inclusion of Proposition 7.1.

As was already mentioned at the front page a short conference version of this paper appeared earlier in R.Janicki, W.W.Koczkodaj eds. *Computing and Information, Proc. Int. Conf. on Computing and Information, ICCI'89, Toronto, North-Holland 1989.* A year later a full version of the paper was submitted to *Formal Aspects of Computing*, where it was accepted but never published, since in the meantime the author decided to take over his family business and that for four years has stopped him to give a final touch to the paper. Today the author expresses his thanks to Cliff Jones, the Editor in Chief of FAC, for the permision of publishing this paper in the present volume. The referees of FAC are also acknowledged for their thorough and constructive criticism.

Last but not least the authors thanks the referee of the present edition of the paper for a very thorough reading of the paper.

## References

[1] M. Bednarczyk, A. Borzyszkowski, and W. Pawłowski. Towards the semantics of the definitional language of MetaSoft. In D. Bjørner, editor, *VDM'90, VDM & Z: Formal Methods in Software Development, 3rd VDM-Europe Symposium, Kiel 1990*, pages 477–503. Lecture Notes in Computer Science, vol. 428, Springer-Verlag, 1990.

[2] D. Bjørner and C.B. Jones. *Formal Specification of Software Development.* Prentice Hall, Englewood Cliffs, NJ, 1982.

[3] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science.* Springer, Berlin, 1978.

[4] A. Blikle. Equational languages. *Information and Control*, 21:134–147, 1972.

[5] A. Blikle. *MetaSoft Primer, Towards a Metalanguage for Applied Denotational Semantics*, volume 288 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[6] A. Blikle. A guided tour of the mathematics of metasoft. *Information Processing Letters*, 29:81–86, 1988. North-Holland.

[7] A. Blikle. Three-valued predicates for software specification and validation. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM'88, VDM: The Way Ahead, Proc. 2nd VDM-Europe Symposium, Dublin, September 1988*, pages 243–266. Lecture Notes in Computer Science, vol. 328, Springer-Verlag, 1988.

[8] A. Blikle. Denotational engineering. *Science of Computer Programming*, 12:207–253, 1989.

[9] A. Blikle and A. Tarlecki. Naive denotational semantics. In R.E.A.Manson, editor, *Information Processing 83, Proc. IFIP World Congress, Paris 1983*, pages 345–355. North Holland, 1983.

[10] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7:70–90, 1978.

[11] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin, 1985.

[12] J. Goguen, J. Messeguer, and D. Plaisted. Programming with parameterized abstract objects in OBJ. In D. Ferrari, M. Bolognani, and J.Goguen, editors, *Theory and Practice of Software Technology*, pages 163–194. North-Holland, Amsterdam, 1983.

[13] M. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, Berlin, 1979.

[14] K. Jensen and N. Wirth. *Pascal: User Manual and Report.* Springer-Verlag, New York, Heidelberg, Berlin, second edition edition, 1978.

[15] G. Kahn. Natural semantics. Raport de Recherche 601, INRIA Center Sophia Antipolis, 1987.

[16] B. Konikowska, A. Tarlecki, and A. Blikle. A three-valued logic for software specification and validation. In R. Bloomfield, R. Jones, and L. Marshall, editors, *VDM'88, VDM: The Way Ahead, Proc. 2nd VDM-Europe Symposium, Dublin, September 1988*, pages 218–242. Lecture Notes in Computer Science, vol. 328, Springer-Verlag, 1988.

[17] P. Landin. The mechanical evaluation of expressions. *BCS Computer Journal*, 6:308–320, 1964.

[18] P. G. Larsen, M. M. Arentoft, S.Bear, and B. Q. Monahan. The mathematical semantics of the BSI/VDM specification language. In *Proceedings IFIP'89 World Congress, San Francisco, August 1989.* North Holland, 1989.

[19] P. Lucas and K. Walk. *On the Formal Description of PL/1*, volume 6 of *Annual Review in Automatic Programming.* Pergamon Press, 1969.

[20] R. Milner. Fully abstract semantics of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[21] M. Nielsen, K. Havelund, K. R. Wagner, and C. George. The RAISE language. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM — The Way Ahead (Proc. 2nd VDM-Europe Symposium, Dublin 1988)*, pages 376–405. Lecture Notes in Computer Science, Springer-Verlag, 1988.

[22] E. Olderog. Sound and complete Hoare-like calculi based on copy rules. *Acta Informatica*, 16:161–197, 1981.

[23] G. Plotkin. A structural approach to operational semantics. Århus University, typescript, 1981.

[24] D. Schmidt. *Denotational Semantics.* Allyn and Bacon, Boston, 1986.

[25] D. Scott. Data types as lattices. *SIAM Journal on Computing*, 5:522–587, 1976.

[26] D. Scott. Domains for denotational semantics. In M. Nielsen and E. M. Schmidt, editors, *Proc. ICALP 82*, pages 577–613. Lecture Notes in Computer Science, Springer-Verlag, 1982.

[27] D. Scott and C. Strachey. Towards a mathematical semantics of computer languages. Technical Monographs PRG-6, Oxford University, Oxford, 1971.

[28] A. Stoughton. *Fully Abstract Models of Programming Languages.* Research Notes in Theoretical Computer Science. Pitman and John Willey & Sons Inc., 1988.

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

[30] A. Tarlecki and M. Wieth. A naive domain universe for VDM. In D. Bjørner, C. Hoare, and H. Langmaack, editors, *VDM'90; VDM and Z — Formal Methods in Software Development*, pages 552–579. Lecture Notes in Computer Science, Springer-Verlag, April 1990.

[31] J. W. Thacher, E. G. Wagner, and J. B. Wright. Notes on algebraic fundamentals of theoretical computer science. In *Proc. of The 3rd Advanced Course on Foundations of Computer Science*, 1978.