VDM semantics of programming languages: combinators and monads

Peter D. Mosses

Under consideration for publication in Formal Aspects of Computing

VDM Semantics of Programming Languages: Combinators and Monads

Peter D. Mosses

Department of Computer Science, Swansea University, UK
E-mail: p.d.mosses@swan.ac.uk

Abstract. VDM was developed in the early 1970s as a variant of denotational semantics. VDM descriptions of programming languages differ from the original Scott–Strachey style by making extensive use of combinators which have a fixed *operational* interpretation. After recalling the main features of denotational semantics and the Scott–Strachey style, we examine the combinators of the VDM specification language, and relate them to monads, which were introduced more than 15 years later. We also suggest that use of further monadic combinators in VDM could be beneficial. Finally, we provide an overview of published VDM semantic descriptions of major programming languages.

Keywords: Programming language semantics; Denotational semantics; VDM; Combinators; Monads.

1. Introduction

The Vienna Development Method, VDM, is a major framework for the formal specification and rigorous development of software systems. In this paper, we focus on the use of VDM for semantic description of programming languages, which was the original motivation for the framework.

VDM evolved from the operational semantics framework known as VDL, the Vienna Definition Language [Weg72], in the early 1970s [Jon01]. The main change in the transition from VDL to VDM was the adoption of the fundamental principles of denotational semantics, which had already been established by Dana Scott and Christopher Strachey [SS71].

One of the innovations in the VDM style of denotational semantics was the introduction of a number of *combinators* having a fixed *operational* interpretation. We shall see that the combinators introduced in VDM for the definition of PL/I in 1974 [Jon01] are closely related to the *monadic* style of denotational semantics, which was introduced by Eugenio Moggi only in 1989 [Mog89]. The VDM combinators were first defined to give a side-effect monad, and then redefined to give a monad that also supports exception raising and handling – essentially by using a monad *transformer*. In the later *Meta-IV* and *VDM-SL* versions of VDM, however, a seemingly minor difference in the definitions of the combinators meant that they no longer provided an exception monad.

Correspondence and offprint requests to: P. D. Mosses

In fact Scott and Strachey had already introduced combinators that determine a side-effect monad in their seminal paper in 1971 [SS71]. But after the introduction of continuations in 1974 [SW00], combinators were generally avoided in the Scott–Strachey style, since denotations could be expressed easily enough using the corresponding patterns of λ-abstraction and application.

The contribution of this paper is elucidation of the relationship between various styles of denotational semantics that were introduced in the 1970s and '80s. Since readers cannot be assumed to be familiar with the details of these styles, we shall start by giving an introductory overview of each, recalling the main concepts and notation and highlighting the distinguishing features: Sect. 2 recalls the fundamental principles of denotational semantics; Sect. 3 illustrates the original Scott–Strachey style adopted in denotational descriptions at Oxford, both before and after the introduction of continuations; Sect. 4 illustrates the VDM style, explaining the main differences between it and the Scott–Strachey style; and Sect. 5 presents the concepts and notation used in the monadic style of denotational semantics. Section 6 then considers the relationship of monads both to the Scott–Strachey style and to the VDM style; it also examines the possibility of using monadic notation to a greater extent in VDM. Section 7 gives an overview of published VDM semantic descriptions of major programming languages, and proposes to establish an online repository for semantic descriptions of programming languages in all frameworks. Section 8 concludes by summarising the main points.

This is a revised and extended version of a paper published in the Bjørner–Zhou Festschrift [Mos07]. The main changes are as follows: detailed definitions of the mentioned monad transformers have been added in Sect. 5; the previous Sect. 5.1.3 is now Sect. 6, and a suggestion for how an exception monad could be defined in *Meta-IV* has been added; and the main contribution of the paper has been made more explicit.

2. Denotational Semantics

The development of denotational semantics was initiated by Strachey in the 1960s [Str66, Str00]. Originally it was based on mapping phrases of programs to the untyped λ-calculus.[1] At the time, it was conjectured that there was no mathematical model of self-applicable functions, which were allowed by the untyped λ-calculus and used to define the so-called paradoxical combinator Y (needed by Strachey for expressing the semantics of loops and recursive procedures). In 1969, Scott, while on sabbatical and visiting Strachey in Oxford, discovered how to construct the missing model, and developed a theory of domains, providing solid foundations for denotational descriptions [Sco70, SS71].

This section recalls the fundamental principles of denotational semantics. Details specific to the original Scott–Strachey style, and the differences between the VDM and Scott–Strachey styles, will be covered in the following two sections.

A denotational semantics of a programming language maps each phrase of the language to its *denotation*. The denotation represents the contribution of the phrase to the overall behaviour of any complete program that contains it; in particular, the denotation of a complete program represents its entire behaviour when run with particular input. The denotation of a phrase is composed from the denotations of its subphrases, and is independent of its context.

A programming language is essentially just a set of strings (the texts of the syntactically legal programs) together with some criteria for implementations of the language to be regarded as conforming. A language can have many different denotational semantics, depending on the choice of:

- phrase structure: how programs can be uniquely decomposed into phrases;
- program behaviour: when programs are regarded as equivalent; and
- denotations: how contributions to behaviour are represented by abstract entities.

The above differences concern the semantic function that maps phrases to denotations, and are independent of the framework used to specify that function. Let us consider them in a bit more detail.

Phrase structure: A set of strings can have many different phrase structures. The choice of a particular phrase structure determines the compositional structure of denotations, which may in turn affect the possibilities for choosing denotations.

[1] Peter Landin's approach [Lan65] was superficially similar, but involved an extended λ-calculus with imperative features.

For example, consider the set of binary numerals: a string of 0s and 1s could be grouped to the left or to the right (or even both ways). Suppose that the leftmost digit of a binary numeral is the most significant, and that the 'behaviour' of a numeral is its numerical value; then grouping to the left is the obvious choice (since the value of a compound numeral such as `100` can then be computed by doubling the value of `10`, whereas with grouping to the right the value of `100` depends on the length of `00` as well as on its value, so denotations would then be pairs of numbers).

Phrase structure for use in denotational semantics is specified by some form of context-free grammar, together with a correspondence relating program texts uniquely to derivation trees according to the grammar. The grammar could be an unambiguous concrete grammar, involving the symbols used in program texts; but usually it is an *abstract grammar*, defining a set of *abstract syntax trees* whose structure is significantly simpler than that of derivation trees for a concrete grammar. The relationship between program texts and abstract syntax trees is generally left to be inferred from the use of suggestive symbols in the abstract grammar, augmented by some informal explanations.

Semantic functions, mapping phrases to their denotations, are defined on abstract syntax trees. The semantic function for a particular language is specified inductively, by giving for each production of the abstract grammar a semantic equation of the form:

$$\mathcal{M}[\![\cdots V_1 \cdots V_n \cdots]\!] = f(\mathcal{M}[\![V_1]\!], \ldots, \mathcal{M}[\![V_n]\!]) \tag{1}$$

where $V_1, \ldots, V_n$ are metavariables ranging over sets of abstract syntax trees, and f expresses how the denotations $\mathcal{M}[\![V_1]\!], \ldots, \mathcal{M}[\![V_n]\!]$ of the subphrases are composed to give the denotation of the phrase '$\cdots V_1 \cdots V_n \cdots$'. The double brackets '$[\![\ldots]\!]$' enclose the notation expressing syntactic phrases of the described programming language, separating it from the notation used for expressing the mathematical entities used as denotations.

Program behaviour: Program behaviour is an abstract representation of what is supposed to be *observable* when programs are compiled and run. It corresponds to the behaviour exhibited by conforming implementations of the programming language.

Compilation usually involves checking for consistency between declaration and use of identifiers throughout the program; the abstract behaviour might then include a list of error messages, or merely a boolean value.

When running the program, its input and output are regarded as observable, so its abstract behaviour always has to represent the input-output relationship. In contrast, potentially observable properties such as how long it takes to run the program (when it terminates), how much memory is required, which machine is used, etc., are generally regarded as irrelevant to the conformance of implementations, and therefore not included in the abstract behaviour of programs.

Denotations: After a phrase structure has been chosen, the denotations of phrases can be specified, subject only to the following constraints:

- the denotation of each phrase is composed from the denotations of its subphrases, and
- the abstract behaviour of each program is determined by its denotation.

Denotations are elements of *domains*. The mathematical nature of domains as particular kinds of partially-ordered sets (each having a least element, and closed under limits) is of much theoretical interest, but does not substantially affect how denotational semantic descriptions are formulated in practice. The crucial properties provided by Scott's domain theory [Sco70, SS71] are that both domains and elements of domains can be defined recursively as (least) solutions of systems of equations:

- *Domain equations* involve domain constructors (e.g., domains of continuous functions, tuples, tagged values) and given domains (truth values, integers, etc.). The least solution of a system of domain equations can be understood as the limit of a series of approximations, starting from trivial domains. Recursive domain equations are needed for denotations of phrases involving self-applicable procedures (which are found not only in the untyped λ-calculus, but also in many imperative programming languages).
- *Element equations* are expressed using λ-notation. The least solution of an element equation $x = f(x)$ in a domain D is a fixed point of the function f on D, and can be understood as the limit of the approximations $f^n(\bot_D)$, where $\bot_D$ is the least element of D, representing nontermination or undefinedness. The function Y mapping each function f on D to its least fixed point $Y(f)$ corresponds to the paradoxical

combinator used in Strachey's early work, and is needed for expressing the denotations of loops and recursive procedures.

Two phrases (of the same sort) are *interchangeable* when replacing one of them by the other in any program does not affect the overall program behaviour. Clearly, phrases that have the same denotation are necessarily interchangeable. In the other direction, denotations are said to be *fully abstract* when two interchangeable phrases always have the same denotation. When denotations are less than fully abstract, two phrases with different denotations may in fact be interchangeable. Although full abstraction is desirable, it can be difficult (sometimes even impossible) to obtain using standard frameworks for specifying denotations,[2] and lack of full abstraction does not prevent the use of denotational semantics for defining the class of conforming implementations of a language.

The denotation of a phrase is generally a function of an *environment* $\rho \in Env$ that represents the bindings created by the context of the phrase. Environments are themselves functions, mapping identifiers to the entities to which they are bound. Landin and Strachey's original approach in the 1960s was to map program identifiers to bound variables in the λ-calculus, and to map blocks with local declarations to applications of λ-abstractions; Scott suggested to use explicit environments in 1969, and they were introduced and illustrated in his seminal joint paper with Strachey in 1971 [SS71].

In the semantics of imperative languages, the denotation of a phrase is moreover a function of a *store* $\sigma \in S$ that represents the effects of assignments to variables. Stores generally include functions mapping each (currently allocated) *location* $\alpha \in L$ to the value $\beta \in V$ last stored in it. Simple variable declarations compute environments in which variable identifiers are bound to locations; inspecting the value of a simple variable involves looking up the location to which the identifier is bound in the environment, then looking up the current value of that location in the store.

3. The Scott–Strachey Style

Strachey established the Programming Research Group in Oxford in 1965. He was already developing his own approach to semantics [Str66] (see also [Str00, Sect. 3.3]). Following Scott's discovery of a model for the untyped λ-calculus [Sco70] while on sabbatical at Oxford in 1969, and Strachey's subsequent collaboration with Scott in the early 1970s [SS71], the development of denotational semantics accelerated rapidly. All in Strachey's group – and many outside it – shared his firm conviction that the denotational approach would now do for semantics what BNF had already done for syntax (as witnessed by the *Algol 60 Report* [BBG$^+$63]) and that within a few years, all major programming languages should have been given a denotational semantics.

Initial case studies of denotational descriptions of major programming languages (e.g. Algol 60 [Mos74], Pascal [Ten77]) were encouraging. The distinctive Scott–Strachey style is particularly concise, and it was adopted in many textbooks and articles on semantics, e.g. [Mos90, Sch86, Sto77, Ten76]. The conciseness facilitates (pencil and paper) proofs about semantic properties and is strongly favoured by many theoreticians – but unfortunately, it does not seem to appeal much to practitioners such as compiler writers and programmers. In contrast, the VDM style appears to be relatively palatable to practitioners; we shall consider a possible reason for this difference in Sect. 4. The only currently used language that has a published Scott-Strachey style denotational semantics is *Scheme* [FM09].

The original Scott–Strachey style of denotational semantics described here is sometimes referred to as 'the Oxford style'. However, that could be confusing, since a rather different style of denotational semantics was also developed at Oxford, by Tony Hoare and He Jifeng, in connection with their *Unifying Theories of Programming* (UTP) framework [HH98]. The characteristic features of that approach are quite different from those of the Scott–Strachey style: in UTP, denotations of programs are predicates that relate initial states to final states (rather than state transformation functions), and the treatment of iteration and recursion is not based on Scott's domain theory (although notions of continuity and least fixed points are still required).

Let us now look at some simple examples in the Scott–Strachey style, which have been selected to illustrate the modest extent to which *combinators* were used.

[2] Fully abstract denotations can always be defined as equivalence classes of phrases.

$$\begin{array}{lcll} \gamma & \in & \mathsf{Cmd} & \textit{commands} \\ \varepsilon & \in & \mathsf{Exp} & \textit{expressions} \end{array}$$

$$\begin{array}{lcl} \gamma & ::= & \mathsf{dummy} \mid \gamma_0\,\mathsf{;}\,\gamma_1 \mid \varepsilon\,\mathsf{->}\,\gamma_0\,\mathsf{,}\,\gamma_1 \mid \varepsilon_0\,\mathsf{:=}\,\varepsilon_1 \mid \ldots \\ \varepsilon & ::= & \ldots \end{array}$$

Fig. 1. Abstract syntax fragment, Scott–Strachey style

Table 1. Operations on arbitrary domains

pairing	$P : A \to (B \to (A \times B))$	$(P\ a)(b) = (a, b)$
pair projections	$M_0 : (A \times B) \to A$	$M_0(a, b) = a$
	$M_1 : (A \times B) \to B$	$M_1(a, b) = b$
sum injections	$\_\,\mathsf{in}\,(A{+}B) : A \to (A{+}B)$	
	$\_\,\mathsf{in}\,(A{+}B) : B \to (A{+}B)$	
sum projections	$\_\|A : (A{+}B) \to A$	$(a\,\mathsf{in}\,(A{+}B))\|A = a$
		$(b\,\mathsf{in}\,(A{+}B))\|A = \bot$
	$\_\|B : (A{+}B) \to B$	$(a\,\mathsf{in}\,(A{+}B))\|B = \bot$
		$(b\,\mathsf{in}\,(A{+}B))\|B = b$
least fixed points	$Y : (D \to D) \to D$	$Y(f) = f(Y(f))$
identity	$I : D \to D$	$I(d) = d$

3.1. Abstract Syntax

Recall that abstract syntax trees are essentially derivation trees for an abstract context-free grammar. The Scott–Strachey style of specifying abstract syntax, illustrated in Fig. 1, is to give a simplified BNF-like grammar using the same terminal symbols as in concrete syntax: reserved words, mathematical signs, and separators. This makes the intended mapping from program texts to abstract syntax trees rather easy to imagine, even though there is usually some grouping ambiguity.

The nonterminal symbols of the grammar are written as metavariables ranging over the corresponding sets of abstract syntax trees; metavariables over the same set are distinguished by subscripts and/or primes.

3.2. Domains and Operations

The domain constructors used in the Scott–Strachey style include:

- $A \times B$: *Cartesian product* domain, with elements of the form (a, b) for $a \in A, b \in B$;
- $A + B$: *separated sum* domain, with elements from the disjoint union of A and B, together with a fresh least element;
- $A \to B$: *continuous function* domain, with elements expressed by λ-abstractions of the form $\lambda x.t$, where x is a variable ranging over A, and t expresses elements of B (usually involving x).

The above domains are equipped with some natural operations. Scott and Strachey [SS71] used those shown in Table 1. They also introduced the two *combinators* shown in Table 2, simply as abbreviations. Finally, they introduced the operations shown in Table 3 in connection with the given domain T of truth values and an (unspecified) domain S of stores with locations $\alpha \in L$ and stored values $\beta \in V$. Later papers by other authors introduced considerably more auxiliary notation – mainly to improve the readability of the λ-expressions used to define denotations.

Table 2. Combinators

plain composition	$f \circ g : A \to C$	when $f : B \to C, \quad g : A \to B$
	$(f \circ g)(a) = f(g(a))$	
curried composition	$f * g : A \to C$	when $f : B_0 \to (B_1 \to C), \quad g : A \to (B_0 \times B_1)$
	$(f * g)(a) = f(b_0)(b_1)$	when $g(a) = (b_0, b_1)$

Table 3. Operations on given domains

truth values T	$Cond : (A \times A) \to (T \to A)$	$Cond(a_0, a_1)(true) = a_0$ $Cond(a_0, a_1)(false) = a_1$
stores S	$Contents : L \to (S \to V)$ $Assign : (L \times V) \to (S \to S)$	$Contents(\alpha)(\sigma) =$ the value in σ at α $Assign(\alpha, \beta)(\sigma) = \sigma'$ s.t. the value at α is β, otherwise the same values as in σ

Table 4. Reversed combinators

reverse plain composition	$f \mathbin{\bar{\circ}} g : A \to C$ $(f \mathbin{\bar{\circ}} g)(a) = g(f(a))$	when $f : A \to B, \quad g : B \to C$
reverse curried composition	$f \mathbin{\bar{*}} g : A \to C$ $(f \mathbin{\bar{*}} g)(a) = g(b_0)(b_1)$	when $f : A \to (B_0 \times B_1), \quad g : B_0 \to B_1 \to C$ when $f(a) = (b_0, b_1)$

3.3. Denotations

Let us first recall how Scott and Strachey defined denotations in their joint paper in 1971 [SS71], before reviewing the more commonly used continuation-passing style.

3.3.1. Direct Semantics

Scott and Strachey's choice of denotations in [SS71] is called *direct* semantics. The basic idea is that denotations of commands are functions from environments to store transformers, i.e. elements of $Env \to (S \to S)$. The semantic function $\mathcal{C}$ maps commands to their denotations:[3]

$$\mathcal{C} : Cmd \to Env \to S \to S \tag{2}$$

Similarly, the denotations of expressions (whose evaluations might have side-effects) should be functions from environments to value-returning store transformers. The semantic function $\mathcal{E}$ maps expressions to their denotations:

$$\mathcal{E} : Exp \to Env \to S \to V \times S \tag{3}$$

Using the combinators defined in Table 2, the denotations of various commands and expressions can be expressed without explicit reference to the store σ:

$$\mathcal{C}[\![\gamma_0\,\texttt{;}\,\gamma_1]\!] = \lambda\rho.\ \mathcal{C}[\![\gamma_1]\!](\rho) \circ \mathcal{C}[\![\gamma_0]\!](\rho) \tag{4}$$

$$\mathcal{C}[\![\varepsilon\texttt{->}\gamma_0\,\texttt{,}\,\gamma_1]\!] = \lambda\rho.\ (\lambda\beta.Cond(\mathcal{C}[\![\gamma_0]\!](\rho), \mathcal{C}[\![\gamma_1]\!](\rho))(\beta|T) * \mathcal{E}[\![\varepsilon]\!](\rho) \tag{5}$$

However, Scott and Strachey were apparently not satisfied with the relatively complicated notation required for expressing the denotations of assignment commands, and resorted to an informal explanation of the steps involved. Here is how they might have written the formal definition:[4]

$$\mathcal{C}[\![\varepsilon_0\,\texttt{:=}\,\varepsilon_1]\!] = \lambda\rho.(\lambda\beta_0.(\lambda\beta_1.Assign(\beta_0|L, \beta_1)) * \mathcal{E}[\![\varepsilon_1]\!](\rho)) * \mathcal{E}[\![\varepsilon_0]\!](\rho) \tag{6}$$

Reading the above equation involves associating the values computed by the expressions ε_0 and ε_1 with the λ-abstractions on β_0, respectively β_1, which is not immediately obvious without close inspection of the term.

Suppose, however, that we were to use the *converses* $\bar{\circ}$ and $\bar{*}$ of the combinators $\circ$ and $*$, taking their operands in the *reverse order*, as defined in Table 4. The same denotations that we defined above can now be expressed as follows:

$$\mathcal{C}[\![\gamma_0\,\texttt{;}\,\gamma_1]\!] = \lambda\rho.\ \mathcal{C}[\![\gamma_0]\!](\rho) \mathbin{\bar{\circ}} \mathcal{C}[\![\gamma_1]\!](\rho) \tag{7}$$

$$\mathcal{C}[\![\varepsilon\texttt{->}\gamma_0\,\texttt{,}\,\gamma_1]\!] = \lambda\rho.\ \mathcal{E}[\![\varepsilon]\!](\rho) \mathbin{\bar{*}} \lambda\beta.Cond(\mathcal{C}[\![\gamma_0]\!](\rho), \mathcal{C}[\![\gamma_1]\!](\rho))(\beta|T) \tag{8}$$

$$\mathcal{C}[\![\varepsilon_0\,\texttt{:=}\,\varepsilon_1]\!] = \lambda\rho.\ \mathcal{E}[\![\varepsilon_0]\!](\rho) \mathbin{\bar{*}} \lambda\beta_0.\mathcal{E}[\![\varepsilon_1]\!](\rho) \mathbin{\bar{*}} \lambda\beta_1.Assign(\beta_0|L, \beta_1) \tag{9}$$

[3] Henceforth we exploit the usual convention that $A \to B \to C$ is grouped as $A \to (B \to C)$, and $A \to B \times C$ as $A \to (B \times C)$.
[4] For simplicity, we assume that dereferencing of variables is made explicit in the abstract syntax of expressions.

This minor change of notation has allowed the terms to be read more *operationally*, from left to right, with the λ-abstractions simply naming the values computed by the preceding terms. (It also substantially reduces the number of required parentheses.) In Sect. 6.1 we shall consider a further aspect of these combinators.

3.3.2. Continuation Semantics

Scott and Strachey's original denotations for commands and expressions, based on store transformers, can represent both normal termination and nonterminating behaviour. To represent abrupt termination, due to escapes (such as break or return) and jumps to labels, the denotations need to be enriched. The standard technique in the Scott–Strachey style (often used also for languages that do not involve abrupt termination) is to replace store transformers by *continuation* transformers, where continuations are themselves some kind of functions on stores [SW00]. The semantics of abrupt termination involves ignoring the argument continuation and applying a different one. As we shall see in the next section, the VDM style avoids the use of continuations by letting store transformers return extra values that indicate whether termination is normal or abrupt, and by introducing combinators to propagate and detect the extra values; see [Jon82] for a detailed comparison of the two techniques.

Any ordinary store transformer θ can be converted to a continuation transformer which, given a continuation θ', returns the continuation that maps any store σ to the result of $\theta'(\theta(\sigma))$. This continuation transformer can be expressed by $\lambda\theta'.\ \theta' \circ \theta$. The original store transformer can be retrieved from the continuation transformer by applying it to the identity continuation.

Denotations of commands now map environments to continuation transformers:

$$\mathcal{C} : Cmd \to Env \to C \to C$$

where the domain C of command continuations θ can be defined as $S \to A$ for any domain A (e.g. A could simply be S). The denotation of command sequencing using continuations is defined as follows:

$$\mathcal{C}[\![\gamma_0\,;\gamma_1]\!] = \lambda\rho.\lambda\theta.\ \mathcal{C}[\![\gamma_0]\!](\rho)\{\mathcal{C}[\![\gamma_1]\!](\rho)\{\theta\}\} \tag{10}$$

(Continuation arguments are conventionally grouped using braces '$\{\ldots\}$' instead of ordinary parentheses.) The domain K of expression continuations κ is defined as $V \to C$: the continuation is normally applied to the value of the expression. Denotations of expressions are given by the semantic function:

$$\mathcal{E} : Exp \to Env \to K \to C$$

The continuation semantics of conditional commands is defined by:

$$\mathcal{C}[\![\varepsilon\texttt{->}\gamma_0\,,\gamma_1]\!] = \lambda\rho.\lambda\theta.\ \mathcal{E}[\![\varepsilon]\!](\rho)\{\lambda\beta.Cond(\mathcal{C}[\![\gamma_0]\!](\rho),\mathcal{C}[\![\gamma_1]\!](\rho))(\beta|T)\{\theta\}\} \tag{11}$$

and that of assignment commands by:

$$\mathcal{C}[\![\varepsilon_0\,\texttt{:=}\,\varepsilon_1]\!] = \lambda\rho.\lambda\theta.\ \mathcal{E}[\![\varepsilon_0]\!](\rho)\{\lambda\beta_0.\mathcal{E}[\![\varepsilon_1]\!](\rho)\{\lambda\beta_1.Assign'(\beta_0|L,\beta_1)\}\} \tag{12}$$

where $Assign' : L \times V \to C \to C$ is the continuation-passing version of $Assign$.

An alternative to the use of braces is to introduce an infix operation corresponding to application, but grouping to the right. Bob Tennent [Ten76] and Mike Gordon [Gor79] used semicolons for this purpose, writing e.g.:

$$\mathcal{C}[\![\gamma_0\,;\gamma_1]\!] = \lambda\rho.\lambda\theta.\ \mathcal{C}[\![\gamma_0]\!](\rho);\mathcal{C}[\![\gamma_1]\!](\rho);\theta \tag{13}$$

However, we shall see that fundamentally, this operation is of quite a different nature from the combinators introduced by Scott and Strachey.

4. The VDM Style

This section focusses on the distinctive features of the VDM style of denotational semantics, which was already quite stable by 1974 [BBH$^+$84]. The illustrations and explanations given here are based primarily on the presentation of VDM in the book by Dines Bjørner and Cliff Jones from 1982 [BJ82], since it is essentially that version of the VDM specification language, known as *Meta-IV*, which has been used for almost all published VDM semantic descriptions of major programming languages (see Sect. 7 for references). A

```
Stmt = Compound | If | Assign | ...
Expr = ...
...
Compound :: Stmt*
If       :: Expr Stmt Stmt
Assign   :: Expr Expr
...
```

Fig. 2. Abstract syntax fragment, VDM style

subsequent version, *VDM-SL*, was standardised by ISO in 1996 (see [PL92]), and used for defining the formal semantics of Modula-2 (see [PS96]). Minor differences between *Meta-IV* and *VDM-SL* will be indicated below when introducing the notation.

In contrast to the Scott–Strachey style of denotational semantics, illustrated in the previous section, the VDM style is quite verbose, generally using (abbreviated) English words rather than Greek letters and mathematical signs. Another stylistic difference is that in VDM, the notation used for abstract syntax (inherited from VDL) does not involve the concrete symbols of the described language. The VDM style appears to have been more successful than the Scott–Strachey style for describing larger programming languages. The author's conjecture is that this is a consequence primarily of the much greater use of combinators in VDM, together with their fixed operational interpretation. However, both styles suffer from a lack of explicit modular structure; this issue has been addressed by the monadic style of denotational semantics, which is discussed in Sect. 5.

4.1. Abstract Syntax

The VDM style of specifying abstract syntax is illustrated in Fig. 2. The absence of terminal symbols from the concrete syntax of the described language makes the mapping from program texts to abstract syntax trees somewhat less obvious than in the Scott–Strachey style, although the words used as nonterminal symbols in the abstract syntax are usually quite suggestive.

Another difference from the Scott–Strachey style is that the nonterminal symbols of the abstract grammar are the names of the sets of abstract syntax trees themselves, rather than metavariables over those sets. Moreover, VDM requires a separate nonterminal to be introduced for each kind of statement, expression, etc. A grammar rule of the form `N = N1 | ... | Nm` defines `N` to be the union of `N1`, ..., `Nm`; in contrast, a rule of the form `N :: N1...Nm` defines `N` to be the set of trees constructed by terms of the form `mk-N(t1,...,tm)` – essentially, the roots of these trees are labelled by the name `N`. The use of `Stmt*` in the abstract grammar is reminiscent of regular expressions in concrete syntax, but it is interpreted as the set of tuples of `Stmt` trees, so the trees in the set `Compound` are constructed by terms of the form `mk-Compound(s1,...,sn)` for all $n \geq 0$.

An interesting feature of the VDM treatment of abstract syntax (not illustrated here) is that it allows trees to have sets and maps as components, as well as tuples. Sets are used when the order of the components of a node is (semantically) irrelevant; maps can be used to reflect that declarations or formal parameters bind distinct identifiers.

4.2. Domains and Operations

The domain constructions available in VDM (both *Meta-IV* and *VDM-SL*, except where indicated) include:

- `A` × `B`: a *smash product* domain, whose non-⊥ elements are of the form `<a,b>` for non-⊥ $a \in A$, $b \in B$;
- `A | B`: a *union* domain, whose non-⊥ elements are of those of `A` and `B` (when `A` and `B` are 'union-compatible');
- `[A]`: an *optional* domain, containing the elements of `A` and the special element $\underline{nil}$ (used to indicate the absence of an element of `A`);
- `D :: A1...An`: a rule declaring a *tree* domain `D`, whose non-⊥ elements are of the form `mk-D(a1,...,an)` for non-⊥ $a1 \in A1, \ldots, an \in An$;

- $A \to B$ and $A \xrightarrow{\sim} B$: *total*, respectively *partial function* domains,[5] with elements expressed by λ-abstractions of the form $\lambda x.\, t$ where x is a variable ranging over A, and t expresses elements of B (usually involving x); and
- $A \xrightarrow[\mathrm{m}]{} B$: a *finite map* domain,[6] with elements of the form $[a1 \mapsto b1, \ldots, an \mapsto bn]$ for non-$\bot$ $a1, \ldots, an \in A$ and $b1, \ldots, bn \in B$ (when A is flat).

VDM provides also various further domain constructions, as well as some basic domains (booleans, numbers, characters and tokens).

A distinctive feature of VDM is its imperative combinators, which are used for expressing state transformations, i.e. functions from *STATE* to *STATE* (where *STATE* generally includes *STORE*, mapping locations to their assigned values, as a component). Continuations are not normally used in VDM semantics [Jon82].

A significant difference between these combinators and those introduced by Scott and Strachey is that each combinator provided by VDM has a *fixed operational interpretation*, whereas its definition in λ-notation varies according to what kind of transformations are to be composed. In contrast, each combinator used in the Scott–Strachey style has a *fixed definition in λ-notation*, but its operational interpretation varies.

For example, the VDM combinator written '*s1;s2*' *always* represents sequential composition of state transformations (starting from *s1*, and continuing with *s2* when *s1* terminates normally) and its definition depends on that of the domain of state transformations. In the Scott–Strachey style, in contrast, the combinator $f \circ g$ is defined as an abbreviation for $\lambda x.f(g(x))$, and what it represents operationally *varies*. Tennent [Ten76] and Gordon [Gor79] introduced an operation '$f; g$', which might appear to express sequencing in continuation semantics; but in fact it is merely an alternative notation for application of a continuation transformer f to a continuation g, and not associative, so it does not correspond to VDM's '*s1;s2*'.

The abbreviation '*=>*' in the *Meta-IV* version of VDM stands for the domain of pure state transformations, and '*=>R*' stands for the domain of transformations that return values in *R*. The domain of functions from *D* to *=>* is written '*D=>*'; similarly, the domain of functions from *D* to *=>R* is written '*D=>R*'.[7]

The main imperative combinators provided by the VDM style of denotational semantics are as follows (in *Meta-IV* notation), assuming that *s1*, *s2*, ... are in *=>*, *e* is in *=>V*, and *f* is in *V=>* or *V=>R*:

- sequencing '*def x: e; f(x)*' applies the transformation *e*, followed by the transformation obtained by applying *f* to the value *x* returned by *e*;
- '*return v*' simply returns the value *v* without transforming the state;
- '*I*' is the identity transformation on states;
- assignment '*r:=e*' first applies *e*, then replaces the component of the state selected by *r* by the value returned by *e*;
- contents '*c r*' returns the component of the state selected by *r* without transforming the state;
- sequencing '*s1; s2*' applies *s2* to the state obtained by applying *s1*;
- conditional '*if b then s1 else s2*' applies *s1* or *s2*, depending on whether the boolean value *b* is *true* or *false*;
- iteration '*for i = m to n do s(i)*' abbreviates '*sm; ...; sn*'.

Some further combinators are used in connection with abrupt termination:

- '*exit v*' terminates abruptly, returning a non-*nil* abnormal value *v*;
- '*trap x with f(x) in s*' handles abrupt termination of *s* with the transformation obtained by applying *f* to the abnormal value *x* returned by *s*;
- '*tixe m in s*' handles abrupt termination of *s* by applying the transformation to which the abnormal value returned by *s* is mapped by *m* (repeatedly), propagating the abrupt termination if the value is not in the domain of *m*;
- '*always s2 in s1*' applies *s1*, then applies *s2*, regardless of whether termination of *s1* was normal or abnormal (*s2* is not supposed to terminate abnormally).

---

[5] *VDM-SL* uses the notation $A \xrightarrow{t} B$ for *total* function domains, and interprets $A \to B$ as a domain of *partial* functions.

[6] *VDM-SL* uses the notation $A \xrightarrow{m} B$ for finite map domains.

[7] In *VDM-SL*, state transformations are called *operations*, and domains of operations are written using $\xrightarrow{o}$ instead of *=>*.

All the above combinators are defined by translation to λ-notation, making the state explicit. In the absence of abrupt termination, '`=>`' and '`=>V`' are defined as follows:

```
=>   = STATE ~→ STATE
=>V  = STATE ~→ STATE × V
```

When the possibility of abrupt termination is introduced, '`=>`' is redefined as:

```
=>   = STATE ~→ STATE × [ABNORMAL]
```

In the VDM semantics of PL/I from 1974 [BBH+84], '`=>V`' is redefined using a disjoint union:

```
=>V  = STATE ~→ STATE × (res V | abn ABNORMAL)
```

In the later books by Bjørner and Jones [BJ78, BJ82], however, '`=>V`' is redefined somewhat differently:

```
=>V  = STATE ~→ STATE × [ABNORMAL] × V
```

In both cases, redefining '`=>`' and '`=>V`' requires redefinition of all the imperative combinators, but the operational interpretation and usage of the combinators in the semantic equations does *not* change.

In Sect. 6.2, we shall see that some of the VDM combinators are closely related to standard operations of the *monads* used in the monadic style of denotational semantics. Moreover, the redefinitions required when abrupt termination is introduced in [BBH+84] correspond to the so-called *lifting* of operations when applying the standard *monad transformer* for exception-handling. Thus it appears that VDM, right from the start in the early 1970s, was already using significant elements of the monadic style that was developed by Moggi in the late 1980s [Mog89].

4.3. Denotations

The following examples illustrate the use of the most basic VDM combinators for defining the denotations of the syntactic constructs shown in Fig. 2, which correspond directly to those used to illustrate the Scott–Strachey style in Sect. 3.

As in the Scott–Strachey style, denotations of statements (i.e. commands) are functions of environments. The semantic function `M` maps statements to their denotations:

```
M : Stmt → ENV =>
```

The same semantic function also maps expressions to their denotations:

```
M : Expr → ENV => VALUE
```

The abbreviations '`=>`' and '`=> VALUE`' indicate that both statements and expressions are modelled as state transformations. Whether these transformations involve the possibility of abnormal termination does not need to be specified until later, since it does not affect how denotations are expressed. However, to specify the denotations of assignment statements, we shall need to know how to select the store from a state. This is specified by indicating the name of the selector function next to the store component of the state:

```
STATE :: STR:STORE ...
```

(The elided further components might support input and output, for example.)

The denotation of a compound statement involves combining the denotations of an arbitrary number of sub-statements, which can be expressed using the VDM combinator corresponding to a definite iteration:

```
M[mk-Compound(sl)](env) = for i = 1 to len sl do M[sl](env)
```

The following is a special case of the above, and is equivalent to the definition of binary statement sequencing in the Scott–Strachey direct semantics style, given in Sect. 3.3:

```
M[mk-Compound(<s1,s2>)](env) = M[s1](env); M[s2](env)
```

Also the VDM definition of the denotations of conditional statements is rather similar to the corresponding definition in the Scott–Strachey style:

```
M[mk-If(e,th,el)](env) =
    def b: M[e](env);
    if b then M[th](env) else M[el](env)
```

Our final illustration of the VDM style of denotational semantics is the assignment statement:

```
M[mk-Assign(lrs,rhs)](env) =
    def l: M[lhs](env);
    def v: M[rhs](env);
    STR := c STR + [l ↦ v]
```

5. The Monadic Style

This style of denotational semantics was introduced by Moggi at the end of the 1980s [Mog89, Mog91]. His original motivation was to generalise the categorical semantics of partiality to "other notions of computation"; he subsequently realised that it also allows a much higher degree of modularity and extensibility in semantic descriptions.

The main technical innovations were to let denotations be elements of *monads*, and to construct monads incrementally using *monad transformers*. Monads and monad transformers provide various combinators, which are closely related to some of those used by Scott and Strachey, and even more closely to some of those provided by VDM. Like the latter, the monadic combinators have a simple operational reading. The monadic style of denotational semantics has been exploited in several theoretical studies [LH96, WH97] but it appears that, despite its advantages regarding modularity, it has not yet been used to describe any major programming language.

5.1. Domains and Operations

The monads used in the monadic style of denotational semantics provide the denotations of phrases of programs, and are generally defined in terms of domains.

5.1.1. Monads

The category-theoretic concept of a *monad* was defined by Mac Lane [ML71], in terms of functors and natural transformations. However, monads are in 1-1 correspondence with *Kleisli triples* [Man76], which in turn can be presented simply as domain constructors T, mapping *value* domains D to *computation* domains $T(D)$, together with two polymorphic operations:

- $\mathit{Return} : D \to T(D)$;
- $\gg\!= : T(A) \times (A \to T(B)) \to T(B)$

The trivial computation $\mathit{Return}(d)$ simply returns the value d as its result. When the computation e computes the value a and f is a function mapping values to computations, $e \gg\!= f$ follows the computation e with the computation $f(a)$. (The symbol '$\gg\!=$', pronounced 'bind', is from the notation provided by the language Haskell.) The operations *Return* and $\gg\!=$ are required to satisfy three laws:

$$(\mathit{Return}(d) \gg\!= f) = f(d) \tag{14}$$

$$(e \gg\!= \mathit{Return}) = e \tag{15}$$

$$((e \gg\!= f) \gg\!= g) = (e \gg\!= \lambda x.(f(x) \gg\!= g)) \tag{16}$$

where x must not be free in f or g in the last law above. Let us follow common practice and identify a triple $(T, \mathit{Return}, \gg\!=)$ that satisfies the above laws with the monad determined by the corresponding Kleisli triple.

The Identity Monad: The simplest possible example of a monad is the identity monad $\mathsf{id} = (T, \mathit{Return}, \gg\!=)$ where $T(D) = D$, $\mathit{Return}(d) = d$, and $(e \gg\!= f) = f(e)$.

5.1.2. Monad Transformers

Particular kinds of monads provide further operations. Such monads can often be constructed by applying standard monad transformers [Mog89, LH96] to existing monads.

Side-Effect Monads: Given domains L of locations α and V of values β, let $S = L \to V$ be the domain of stores σ (representing the values currently stored at the locations). Given any monad T, a *side-effect* (or *state*) monad $\mathsf{se}\,T$ with operations $\mathit{Update}_{\mathsf{se}\,T} : L \times V \to \mathsf{se}\,T()$ and $\mathit{Inspect}_{\mathsf{se}\,T} : L \to \mathsf{se}\,T(V)$ is constructed by defining:

$$\mathsf{se}\,T(D) = S \to T(D \times S) \tag{17}$$
$$\mathit{Return}_{\mathsf{se}\,T}(d) = \lambda\sigma.\ \mathit{Return}(d, \sigma) \tag{18}$$
$$e \gg\!=_{\mathsf{se}\,T} f = \lambda\sigma.\ e(\sigma) \gg\!= \lambda(d, \sigma').f(d)(\sigma') \tag{19}$$
$$\mathit{Update}_{\mathsf{se}\,T}(\alpha, \beta) = \lambda\sigma.\ \mathit{Return}((), \sigma[\alpha \mapsto \beta]) \tag{20}$$
$$\mathit{Inspect}_{\mathsf{se}\,T}(\alpha) = \lambda\sigma.\ \mathit{Return}(\sigma(\alpha), \sigma) \tag{21}$$

Environment Monads: Given domains I of identifiers and V of values, let $\mathit{Env} = I \to V$ be the domain of environments ρ (representing the values currently bound to the identifiers). Given any monad T, an *environment* (or *read-only state*) monad $\mathsf{en}\,T$ with operations $\mathit{GetEnv}_{\mathsf{en}\,T} : \mathsf{en}\,T(\mathit{Env})$ and $\mathit{UseEnv}_{\mathsf{en}\,T} : \mathit{Env} \to \mathsf{en}\,T(D) \to \mathsf{en}\,T(D)$ is constructed by defining:

$$\mathsf{en}\,T(D) = \mathit{Env} \to T(D) \tag{22}$$
$$\mathit{Return}_{\mathsf{en}\,T}(d) = \lambda\rho.\ \mathit{Return}(d) \tag{23}$$
$$e \gg\!=_{\mathsf{en}\,T} f = \lambda\rho.\ e(\rho) \gg\!= \lambda d.f(d)(\rho) \tag{24}$$
$$\mathit{GetEnv}_{\mathsf{en}\,T} = \lambda\rho.\ \mathit{Return}(\rho) \tag{25}$$
$$\mathit{UseEnv}_{\mathsf{en}\,T}(\rho')(e) = \lambda\rho.\ e(\rho') \tag{26}$$

Exception Monads: Given a domain X of exception identifiers and any monad T, an *exception* monad $\mathsf{ex}\,T$ with operations $\mathit{Raise} : X \to \mathsf{ex}\,T(D)$ and $\mathit{Handle} : X \times \mathsf{ex}\,T(D) \times \mathsf{ex}\,T(D) \to \mathsf{ex}\,T(D)$ is constructed by defining:

$$\mathsf{ex}\,T(D) = T(D + X) \tag{27}$$
$$\mathit{Return}_{\mathsf{ex}\,T}(d) = \mathit{Return}(d\,\mathsf{in}\,(D{+}X)) \tag{28}$$
$$e \gg\!=_{\mathsf{ex}\,T} f = e \gg\!= \mathsf{cases}(\lambda d.f(d), \lambda x.\mathit{Return}(x\,\mathsf{in}\,(D{+}X))) \tag{29}$$
$$\mathit{Raise}_{\mathsf{ex}\,T}(x) = \mathit{Return}(x\,\mathsf{in}\,(D{+}X)) \tag{30}$$
$$\mathit{Handle}_{\mathsf{ex}\,T}(x, e_1, e_2) = e_1 \gg\!= \mathsf{cases}(\lambda d.\mathit{Return}(d\,\mathsf{in}\,(D{+}X)), \lambda x'.\mathit{Cond}(e_2, \mathit{Return}(x'\,\mathsf{in}\,(D{+}X)))(x = x')) \tag{31}$$

where $\mathsf{cases} : (A \to C) \times (B \to C) \to ((A{+}B) \to C)$ is defined by $\mathsf{cases}(f, g)(a\,\mathsf{in}\,(A{+}B)) = f(a)$ and $\mathsf{cases}(f, g)(b\,\mathsf{in}\,(A{+}B)) = g(b)$.

Continuation Monads: Given a domain A of answers and any monad T, a *continuation* monad $\mathsf{cc}\,T$ with operation $\mathit{CallCC} : ((D \to \mathsf{cc}\,T(D)) \to \mathsf{cc}\,T(D)) \to \mathsf{cc}\,T(D)$ is constructed by defining:

$$\mathsf{cc}\,T(D) = (D \to T(A)) \to T(A) \tag{32}$$
$$\mathit{Return}_{\mathsf{cc}\,T}(d) = \lambda\kappa.\ \kappa(d) \tag{33}$$
$$e \gg\!=_{\mathsf{cc}\,T} f = \lambda\kappa.\ e(\lambda d.\ f(d)(\kappa)) \tag{34}$$
$$\mathit{CallCC}_{\mathsf{cc}\,T}(g) = \lambda\kappa.\ g(\lambda d.\lambda\kappa'.\kappa(d))(\kappa) \tag{35}$$

When a monad transformer is applied to a monad, the required operations *Return* and $\gg\!=$ are *lifted* to the monad resulting from the transformation. However, all the other operations defined on the argument monad need to be lifted too, which can be problematic; this is one of the drawbacks of the use of monad transformers.

The operations defined above satisfy some axioms, which allow algebraic reasoning about equivalence. Gordon Plotkin and John Power develop such axioms, distinguishing between *algebraic operations* and *effects*; see [PP04] for an overview of their approach. The exact statement of the axioms that characterise particular classes of monads is highly non-trivial, and out of the scope of this article.

5.2. Denotations

Let us conclude this section by showing how simple it is to define the denotations of our illustrative phrases in the monadic style. We do not need to define the monad T, since the details of its definition do not affect how denotations are expressed, nor their operational understanding. We do however assume that T is a side-effect monad, so that the operation $\mathit{Update}(\alpha, \beta)$ is available.

Let $\mathcal{C} : \mathit{Cmd} \to T()$ and $\mathcal{E} : \mathit{Exp} \to T(V)$, where V is a domain of values whose definition depends on the language being described, but is here assumed to include both L (locations) and T (truth values) as summands. Thanks to the use of monadic notation, the denotations of various commands and expressions can be defined without notational clutter regarding propagation of the environment ρ and the store σ, and the possibility of abrupt termination:

$$\mathcal{C}[\![\gamma_0\,;\gamma_1]\!] = \mathcal{C}[\![\gamma_0]\!] \gg\!= \lambda().\ \mathcal{C}[\![\gamma_1]\!] \tag{36}$$

$$\mathcal{C}[\![\varepsilon\texttt{->}\gamma_0\,,\gamma_1]\!] = \mathcal{E}[\![\varepsilon]\!] \gg\!= \lambda\beta.\ \mathit{Cond}(\mathcal{C}[\![\gamma_0]\!], \mathcal{C}[\![\gamma_1]\!])(\beta|T) \tag{37}$$

$$\mathcal{C}[\![\varepsilon_0\texttt{:=}\varepsilon_1]\!] = \mathcal{E}[\![\varepsilon_0]\!] \gg\!= \lambda\beta_0.\ \mathcal{E}[\![\varepsilon_1]\!] \gg\!= \lambda\beta_1.\ \mathit{Update}(\beta_0|L, \beta_1) \tag{38}$$

The reader is invited to compare the above semantic equations with those given in the Scott–Strachey style (Sect. 3.3) and in the VDM style (Sect. 4.3).

6. Monads in the Scott–Strachey and VDM Styles

Let us now review some examples of notation defined by the Scott-Strachey and VDM styles of denontational semantics, and see how closely they correspond to monads that can be constructed using the monad transformers presented in Sect. 5.

6.1. Scott–Strachey Style

6.1.1. Direct Semantics

Scott and Strachey [SS71] used the domain $S \to V \times S$, which is provided by the monad $\mathsf{se\,id}$ obtained by applying the side-effect monad transformer to the identity monad. Scott and Strachey's combinators P and $*$ are special cases of *Return* and $\gg\!=$ (the latter with the arguments reversed, corresponding directly to $\bar{*}$ as defined in Sect. 3.3.1). They also used the domain $S \to S$, which is isomorphic to the domain $S \to () \times S$ provided by the monad, and their $\mathit{Assign}(\alpha, \beta)$ operation corresponds to the provided $\mathit{Update}(\alpha, \beta)$ operation under that isomorphism.

Scott and Strachey did not introduce any combinators in connection with environments, apparently preferring to exhibit the propagation of environments. However, it would be straightforward to lift their notation to the environment monad constructed by $\mathsf{en\,se\,id}$, and remove the explicit environment arguments from the semantic equations shown in Sect. 3.3.1.

6.1.2. Continuation Semantics

The domain of denotations of expressions in Sect. 3.3.2 was $\mathit{Env} \to K \to C$ where $K = V \to C$ and $C = S \to A$. This domain is provided by the monad constructed by $\mathsf{en\,cc\,se\,id}$. However, operations corresponding to *Return* and $\gg\!=$ were never introduced in continuation semantics, since the direct use of function application in λ-notation was regarded as sufficiently succinct and perspicuous. As previously mentioned, the operation an operation '$f; g$' introduced by Tennent [Ten76] and Gordon [Gor79] does *not* correspond to a monadic combinator.

6.2. VDM Style

6.2.1. Normal Termination

Let $T(D)$ be `=>D`, which, in the absence of abnormal termination, is `STATE ~→ STATE × D`. Let *Return* be the VDM combinator `return`, and let $e \gg\!= f$ be defined as `def x: e; f(x)`. This provides a monad. Assuming that the store component of the state is selected by `STR`, *Update*(l, v) can be defined to be `STR := (c STR)+ [l ↦ v]`, and *Inspect*(l) to be `return((c STR)(l))`. The result is a side-effect monad, corresponding closely to the monad `se id`, and has been exploited in VDM since the early 1970s.

6.2.2. Abnormal Termination

The redefinition of `=>D` to allow abrupt termination given in the PL/I semantics from 1974 [BBH+84] is:

```
=>V  = STATE ~→ STATE × (res V | abn ABNORMAL)
```

The values are tagged with `res` or `abn` to distinguish between normal and abrupt termination. The redefined combinators `return` and `def` still provide a monad: this is essentially an instance of applying a monad transformer, and all the original combinators are redefined to take account of the new domains. Moreover, the resulting monad is easily made into an exception monad: define *Raise*(x) to be `exit x`, and *Handle*(x, e, e') as `trap x with e in e'`. It is remarkable that VDM was already using monads and monad transformers – albeit unaware of their mathematical foundations – more than 15 years before the monadic style of denotational semantics had been developed.

However, a different redefinition of `=>D` was given in *Meta-IV* in 1978 [BJ78] and again in 1982 [BJ82]:

```
=>V  = STATE ~→ STATE × [ABNORMAL] × V
```

This domain construction does *not* allow the combinators `return` and `def` to be defined to give a monad.[8] The problem is that when e terminates abruptly, so does $e \gg\!= f$, with the same result; but if e is in `=>A` and f is in `A=>B`, the combination $e \gg\!= f$ is in `=>B`, and this gives rise to inconsistency when `A` and `B` are disjoint.

To obtain a monad, it appears that the VDM definition of `=>V` used in the PL/I semantics would have to be reformulated in *Meta-IV*[9] as follows (since the tagged union domain constructor is not provided):

```
      =>V   =  STATE ~→ STATE × OUTCOME(V)
OUTCOME(V)  =  RES(V) | ABN
    RES(V)  :: V
       ABN  :: ABNORMAL
```

6.3. Comparison

The basic monadic combinators provided in the original Scott–Strachey style of denotational semantics allowed it to get close to the simplicity of the monadic style illustrated in Sect. 5. However, the subsequent adoption of continuations led to the use of combinators being abandoned, until Moggi realised their significance in connection with monads, and reintroduced them in a general and systematic way.

VDM has the advantage of a fixed operational interpretation for its combinators, and originally used what is essentially a monad transformer when adding the possibility of abnormal termination. Moreover, it seems that the domains defined for abnormal termination in *Meta-IV* and *VDM-SL* could be adjusted to restore the monadic properties, as suggested above. VDM already includes notation corresponding to side-effect and exception monads; adding notation for environment monads would allow omission of propagated '`env`' arguments, and support for further monad transformers would significantly enhance the (inherent) modularity of VDM semantics of programming languages.

---

[8] Thanks to Moggi for drawing attention to this point.

[9] Corresponding domain constructors could presumably be defined in *VDM-SL* using parameterised modules.

7. Published VDM Semantic Descriptions

VDM was originally developed for giving a formal definition of PL/I and providing a formal basis for the development of a compiler [Jon01]. Other major programming languages that have been described using VDM include Algol 60, Pascal, Ada, and Modula-2. The aim here is merely to give a general overview of the cited descriptions. In general, the descriptions are out of print; this section concludes by proposing the establishment of an online repository for semantic descriptions, so that these major contributions can be preserved and made more easily accessible to the research community.

PL/I: The technical report *A Formal Definition of a PL/I Subset* by Hans Bekič, Dines Bjørner, Wolfgang Henhapl, Cliff Jones, and Peter Lucas was published by the IBM Laboratory in Vienna in 1974 [BBH$^+$84]. Covering 201 pages, its length was modest compared to the size of the language described (and to that of the earlier definition of PL/I given in the operational VDL framework). The described subset is essentially the version of PL/I described in the ECMA/ANSI standard (which does not include tasking) but omitting Input/Output. Section 4 of the chapter on Notation introduces and defines the VDM combinators concerned with state transformations, imperative variables, exit, and arbitrary ordering.

Although the original technical report is out of print, 61 pages from it were reprinted in a volume of LNCS containing a selection of papers by Bekič [BBH$^+$84]. The development of a compiler based on the semantic description stopped prematurely in 1975, when IBM cancelled the project to build the intended target machine [Jon01].

Algol 60: A VDM semantics for Algol 60 by Wolfgang Henhapl and Cliff Jones was published as Chapter 6 of the book *Formal Specification and Software Development* in 1982 [HJ82]; it is a revision of a previous paper by the same authors published in the 1978 LNCS volume on VDM [BJ78]. In 33 pages it specifies the abstract syntax and semantics of the language described in the 1975 *Modified Report on Algol 60*, and provides a list of abbreviations as well as an index of object and function definitions. The specifications of the abstract syntax, static semantics and dynamic semantics are interleaved, so that the static and dynamic semantics for the same construct are given close to each other.

The definition of the arbitrary order of evaluation allowed by Algol 60 is deliberately not addressed; a few other minor deviations from the intended semantics of Algol 60 are indicated in comments. Neither the concrete syntax nor its translation to the abstract syntax are given, although some of the comments refer to various expansions made by "the translator".

Pascal: A VDM semantics for Pascal by Derek Andrews and Wolfgang Henhapl was published as Chapter 7 of the book *Formal Specification and Software Development* in 1982 [AH82]. As with the VDM semantics for Algol 60 that it follows, it is a revision of a previous paper published in the 1978 LNCS volume on VDM [BJ78]. After an introduction commenting on various aspects of the VDM semantics of Pascal, it takes 60 pages to specify the abstract syntax and semantics of the language described in the BSI/ISO Standard for Pascal.

In contrast to the VDM semantics of Algol 60, the specifications of the abstract syntax, static semantics, and dynamic semantics of Pascal are not interleaved. The abstract syntax was chosen to be "fairly close" to the concrete syntax of Pascal, making their relationship "more obvious". The specification of the abstract syntax is about 6 pages, including some detailed notes about the intended concrete to abstract translation (which involves the introduction of fresh identifiers "not used elsewhere"). The static semantics takes 22 pages, and the dynamic semantics 30 pages.

Ada: A VDM semantics for full Ada was initially developed by Dines Bjørner and several MSc students under his supervision at the Danish Technical University, and published as a volume of LNCS with the title *Towards a Formal Description of Ada* in 1980 [BO80]. This description was subsequently revised and finalised in a series of technical reports, published by Dansk Datamatik Center (DDC) in 1981–2, which provided the basis for the rigorous development of an Ada compiler [CO84]. The compiler was released in 1983, and became renowned not only for its quality, but also as commercially successful. The unqualified success of this application of VDM was a clear vindication of Bjørner's trust in the suitability of the VDM specification language for describing the semantics of large languages such as Ada, as well as a welcome, highly visible demonstration of the potential usefulness of research in formal semantics.

VDM was later used also in the official *Draft Formal Definition of Ada*, but only for the static semantics

[BSP87]. The dynamic semantics [BNK87, GMRZ86] was specified using *SMoLCS* [AR87], which uses a "VDM-like" style of denotational semantics to map Ada programs into a semantic algebra, where behaviour (including concurrency) is specified using a combination of labelled transition rules and algebraic axioms. The semantic algebra includes the combinators used in VDM.

Modula-2: VDM was used, along with English text, for defining the semantics of Modula-2 in its ISO/IEC base standard, which was developed from 1987 to 1996. The formal definition and the English text are regarded as having equal importance. According to an article about the process of producing the standard [PS96], it contains about "200 type definitions, 1800 function and operation definitions and some 20,000 lines of *VDM-SL* code". All the *VDM-SL* specifications were "tested for syntactical accuracy and semantic constraints" using a front end for *VDM-SL* developed at Delft University of Technology. ISO/IEC did not allow publication of the standard on the web [PS03], although a draft version is available.[10]

Online access: The VDM Portal[11] provides online access to many VDM documents, including examples of specifications in *VDM-SL* and *VDM++*. However, it appears that only *two* VDM semantic descriptions of programming languages are currently available through the portal: one for a language called *NewSpeak* from 1994, the other for a tutorial-style example of static and dynamic semantics of a simple programming language. Fortunately the VDM descriptions of PL/I, Algol 60 and Pascal, together with the original VDM description of Ada, have all been scanned to pdf, and are currently available online.[12]

A different and more general problem with providing online access to large VDM specifications is to allow efficient *searching* for particular items of interest. For semantic descriptions of programming languages, it would be useful to search for the parts concerning particular (concrete or abstract) constructs. Searching for mathematical formulae is inherently difficult, and addressed on the web by using special markup languages such as MathML; but this would not help with existing, older documents. A possible solution might be to add bookmarks to pdfs, identifying the pages concerned with particular constructs.

There is also the issue of copyright. Presumably all authors of semantic descriptions would be happy to see their work made accessible online, but some publishers seem unlikely to allow open access in the near future. A compromise might be to provide open access only to summary information about semantic descriptions, sufficient to support searching for descriptions of particular constructs, but require login as a registered user to obtain the pdf of the description itself.

The author is currently investigating the possibility of establishing a repository for semantic descriptions of programming languages in *all* major frameworks; VDM would be among the first to be covered. Readers who have copies of significant semantic descriptions of programming languages (in any format) are kindly requested to contact the author, indicating what they could provide, and who holds the copyright.

8. Conclusion

The fundamental principles of denotational semantics were established by Scott and Strachey at Oxford in the early 1970s. VDM adopted these principles, but also introduced some innovations: in particular, VDM made much greater use of combinators than was usual in the original Scott–Strachey style. Significantly, each combinator in VDM has a fixed operational interpretation, whereas its definition in λ-notation can vary; see also [Mos77]. We have seen that some of the VDM combinators introduced in the early 1970s correspond directly to operations provided in the monadic style of denotational semantics, which was developed at the end of the 1980s; moreover, the way their definitions vary corresponded to the lifting of operations by particular monad transformers.

VDM semantic descriptions of some major programming languages have been published, including PL/I, Algol 60, Pascal, Ada, and Modula-2. They are valuable sources of examples of how to describe a wide range of programming constructs using VDM, and deserve to be much more easily accessible to researchers and students than at present; including them in the proposed online repository of semantic descriptions would not only make them electronically available, but should also allow searching for descriptions of particular kinds of constructs.

[10] `ftp://ftp.mathematik.uni-ulm.de/pub/soft/modula/standard/draft4/`
[11] `http://www.vdmportal.org/`
[12] `http://homepages.cs.ncl.ac.uk/cliff.jones/semantics-library.html`

Acknowledgment This is a revised and extended version of an essay written in honour of Dines Bjørner and Zhou Chaochen on the occasion of their 70th birthdays.

As one of the originators of VDM, through his many articles and books about VDM, and by his use of VDM in major projects such as the formal descriptions of PL/I and Ada, Dines has had a profound influence on the development and practical application of denotational semantics. Personally, I have benefited immensely from contact with him since we first met in Denmark in the late 1970s. His expertise, friendship and hospitality have always seemed limitless.

Thanks to Cliff Jones, Eugenio Moggi and Gordon Plotkin for helpful comments related to previous versions of this paper, and to the anonymous referees for constructive suggestions for improvement of the submitted version.

References

[AH82] Derek J. Andrews and Wolfgang Henhapl. Pascal. In *Formal Specification and Software Development* [BJ82], chapter 7, pages 175–252.

[AR87] Egidio Astesiano and Gianna Reggio. Direct semantics of concurrent languages in the SMoLCS approach. *IBM J. Res. Dev.*, 31(5):512–534, 1987.

[BBG+63] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *Commun. ACM*, 6(1):1–17, 1963.

[BBH+84] Hans Bekič, Dines Bjørner, Wolfgang Henhapl, Cliff B. Jones, and Peter Lucas. On the formal definition of a PL/I subset (selected parts). In *Programming Languages and Their Definition – Hans Bekič (1936–1982)*, volume 177 of *LNCS*, pages 107–155. Springer, 1984. Available at `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/LNCS177-Bekic/`. Full version published as Technical Report 25.139, IBM Lab. Vienna, Dec. 1974; available at `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/TR25139/`.

[BJ78] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.

[BJ82] Dines Bjørner and Cliff B. Jones. *Formal Specification and Software Development*. Computer Science Series. Prentice-Hall Int., 1982. Available at `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/BjornerJones1982/`.

[BNK87] Claus Bendix Nielsen and E. W. Karlsen. The draft formal definition of Ada, the dynamic semantics definition, vols 1–3. Technical report, Dansk Datamatik Center, Lyngby, Denmark, January 1987.

[BO80] Dines Bjørner and Ole N. Oest. *Towards a Formal Description of Ada*, volume 98 of *LNCS*. Springer, 1980.

[BSP87] N. Botta and Jan Storbank Pedersen. The draft formal definition of Ada, the static semantics definition, vols 1–4. Technical report, Dansk Datamatik Center, Lyngby, Denmark, January 1987.

[CO84] Geert B. Clemmensen and Ole N. Oest. Formal specification and development of an Ada compiler – a VDM case study. In *ICSE '84: Proc. 7th Int. Conf. on Software Engineering*, pages 430–440. IEEE Press, 1984.

[FM09] Robert Bruce Findler and Jacob Matthews. Revised^6 report on the algorithmic language Scheme, App. A: Formal semantics. *J. Funct. Program.*, 19(S1):125–145, 2009.

[GMRZ86] Alessandro Giovini, Franco Mazzanti, Gianna Reggio, and Elena Zucca. The draft formal definition of Ada, the dynamic semantics definition, vol 4. Technical report, Dansk Datamatik Center, Lyngby, Denmark, December 1986.

[Gor79] Michael J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction*. Springer, 1979.

[HH98] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[HJ82] Wolfgang Henhapl and Cliff B. Jones. ALGOL 60. In *Formal Specification and Software Development* [BJ82], chapter 6, pages 141–173.

[Jon82] Cliff B. Jones. More on exception mechanisms. In *Formal Specification and Software Development* [BJ82], chapter 5, pages 125–140.

[Jon01] Cliff B. Jones. The transition from VDL to VDM. *J. UCS*, 7(8):631–640, 2001.

[Lan65] Peter J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Commun. ACM*, 8(2):89–101, 1965.

[LH96] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *ESOP'96*, volume 1058 of *LNCS*, pages 219–234. Springer, 1996.

[Man76] Ernest G. Manes. *Algebraic Theories*, volume 26 of *Graduate Texts in Mathematics*. Springer, 1976.

[ML71] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer, 1971.

[Mog89] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh Univ., 1989.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[Mos74] Peter D. Mosses. The mathematical semantics of Algol60. Technical Monograph PRG-12, Oxford Univ. Comp. Lab., 1974.

[Mos77] Peter D. Mosses. Making denotational semantics less concrete. In *Proc. Int. Workshop on Semantics of Programming Languages, Bad Honnef*, number 41 in Bericht, pages 102–109. Abteilung Informatik, Universität Dortmund, 1977.

[Mos90] Peter D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 11. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[Mos07] Peter D. Mosses. VDM semantics of programming languages: Combinators and monads. In *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *LNCS*, pages 483–503. Springer, 2007.

[PL92] Nico Plat and Peter Gorm Larsen. An overview of the ISO/VDM-SL standard. *SIGPLAN Not.*, 27(8):76–82, 1992.

[PP04] Gordon D. Plotkin and A. John Power. Computational effects and operations: An overview. In *Proc. Workshop on Domains VI*, volume 73 of *Electr. Notes Theor. Comput. Sci.*, pages 149–163. Elsevier, 2004.

[PS96] C. Pronk and M. Schönhacker. ISO/IEC 105141, the standard for Modula-2: Process aspects. *SIGPLAN Not.*, 31(8):74–83, 1996.

[PS03] C. Pronk and M. Schönhacker. Formal definition of programming language standards. *SIGPLAN Not.*, 38(8):20–21, 2003.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986. Available at `http://people.cis.ksu.edu/~schmidt/text/densem.html`.

[Sco70] Dana S. Scott. Outline of a mathematical theory of computation. In *Proc. Fourth Annual Princeton Conf. on Information Sciences and Systems*, pages 169–176. Princeton University, 1970. Superseded by Technical Monograph PRG-2, Oxford Univ. Comp. Lab., Nov. 1970.

[SS71] Dana S. Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. In *Proc. Symp. on Computers and Automata*, volume 21 of *Microwave Research Inst. Symposia*, pages 19–46. Polytechnic Institute of Brooklyn, 1971. Also Technical Monograph PRG-6, Oxford Univ. Comp. Lab., Aug. 1971.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, volume 1 of *The MIT Press Series in Computer Science.* The MIT Press, 1977.

[Str66] Christopher Strachey. Towards a formal semantics. In T. B. Steel, Jr., editor, *Formal Language Description Languages for Computer Programming, Proc. IFIP Work. Conf., Vienna, 1964*, pages 198–220. North-Holland, Amsterdam, 1966.

[Str00] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):11–49, 2000. Originally lecture notes, NATO Copenhagen Summer School, 1967.

[SW00] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher Order Symbol. Comput.*, 13(1-2):135–152, 2000. Originally published as Technical Monograph PRG-11, Oxford Univ. Comput. Lab., Jan. 1974.

[Ten76] Robert D. Tennent. The denotational semantics of programming languages. *Commun. ACM*, 19(8):437–453, 1976.

[Ten77] Robert D. Tennent. A denotational definition of the programming language Pascal. Technical Report 77-47, Queen's Univ., Kingston, Ont., 1977.

[Weg72] Peter Wegner. The Vienna Definition Language. *ACM Comput. Surv.*, 4(1):5–63, 1972.

[WH97] Keith Wansbrough and John Hamer. A modular monadic action semantics. In *DSL'97.* USENIX, 1997.