COMPUTING SCIENCE

An Exegesis of Four Formal Descriptions of ALGOL 60

Cliff B. Jones, Troy K. Astarte

TECHNICAL REPORT SERIES

No. CS-TR-1498 September 2016

TECHNICAL REPORT SERIES

No. CS-TR-1498 **September, 2016**

An Exegesis of Four Formal Descriptions of ALGOL 60

Cliff B. Jones, Troy K. Astarte

Abstract

The programming language ALGOL 60 has been used to illustrate several different styles of formal semantic description. This paper identifies the main challenges in providing a formal semantics for imperative programming languages and reviews the responses to these challenges in four relatively complete formal descriptions of ALGOL 60. The aim is to identify the key concepts rather than get bogged down in the minutiae of notational conventions adopted by their authors.
As well as providing historical pointers and comparisons, the paper attempts to draw some general conclusions about semantic description styles.

© 2015 Newcastle University.
Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

An Exegesis of Four Formal Descriptions of ALGOL 60

Cliff B. Jones, Troy K. Astarte
September 8, 2016

Abstract

The programming language ALGOL 60 has been used to illustrate several different styles of formal semantic description. This paper identifies the main challenges in providing a formal semantics for imperative programming languages and reviews the responses to these challenges in four relatively complete formal descriptions of ALGOL 60. The aim is to identify the key concepts rather than get bogged down in the minutiae of notational conventions adopted by their authors.
As well as providing historical pointers and comparisons, the paper attempts to draw some general conclusions about semantic description styles.

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1498

About the authors

Cliff B. Jones is currently Professor of Computing Science at Newcastle University. As well as his academic career, Cliff has spent over 20 years in industry. His 15 years in IBM saw, among other things, the creation –with colleagues in Vienna– of VDM which is one of the better known "formal methods". Under Tony Hoare, Cliff wrote his doctoral thesis in two years. From Oxford, he moved directly to a chair at Manchester University where he built a world-class Formal Methods group which – among other projects– was the academic lead in the largest Software Engineering project funded by the Alvey programme (IPSE 2.5 created the "mural" (Formal Method) Support Systems theorem proving assistant). In 1996 he moved back into industry with a small software company (Harlequin), directing some 50 developers on Information Management projects and finally became overall Technical Director before leaving to re-join academia in 1999 to take his current chair in Newcastle. Much of his current research focuses on formal (compositional) development methods for concurrent systems and support systems for formal reasoning. Cliff is also a Fellow of the Royal Academy of Engineering (FREng), ACM, BCS, and IET. He has been a member of IFIP Working Group 2.3 (Programming Methodology) since 1973.

Troy Astarte graduated from his undergraduate degree in Computer Science at Newcastle University in 2015, and moved on to a PhD at the same institution under the supervision of Cliff Jones, with assistance from Brian Randell, Martin Campbell-Kelly, and John Tucker. He is researching the history of formal semantics of programming languages.

Suggested keywords

Programming languages; Semantic description; Operational semantics; Denotational semantics; Axiomatic semantics

An Exegesis of Four Formal Descriptions of ALGOL 60

Cliff B. Jones, Troy K. Astarte

September 8, 2016

Abstract

The programming language ALGOL 60 has been used to illustrate several different styles of formal semantic description. This paper identifies the main challenges in providing a formal semantics for imperative programming languages and reviews the responses to these challenges in four relatively complete formal descriptions of ALGOL 60. The aim is to identify the key concepts rather than get bogged down in the minutiae of notational conventions adopted by their authors. As well as providing historical pointers and comparisons, the paper attempts to draw some general conclusions about semantic description styles.

Keywords: Programming languages; Semantic description; Operational semantics; Denotational semantics; Axiomatic semantics

This paper was presented at the *History and Philosophy of Programming* Symposium held in Paris in June 2016. An updated version of this report will be published in the HaPoP proceedings.

1 Introduction

Research on providing formal descriptions[1] of the semantics of programming languages began in the 1960s and remains active. This paper draws on some clear documentary evidence to examine the period up to the mid 1980s.

Several research groups have chosen to use ALGOL 60[2] to illustrate their way of formally describing the semantics of programming languages. The availability of semantic descriptions of broadly the same object language makes for an interesting comparison of various aspects of the approaches. It is also helpful that the authors of the descriptions have often been careful to provide context for their research.

There are some fundamental differences between the proposed approaches but there are also some almost accidental differences (such as the house style on the length of identifiers). This paper emphasises the deeper issues.

This version of the paper follows a broadly historical path. In the introduction, a definition of "semantics" is offered and the reasons for attempting a formal semantics are considered. The importance of ALGOL is discussed along with examples of how its semantics are described informally in the defining Report. A key early reference is identified and finally the dimensions of comparison for each full semantic description are also given.

Then follows a section on each of four complete descriptions, presented in chronological order; in each, a historical background and context is given before deeper semantic points are discussed. Two of the descriptions follow an operational style, and two a denotational approach.

Finally, in the conclusions section, some direct comparisons are made between the semantic approaches covered and some other important semantic descriptions and styles not discussed in the body of the paper are mentioned.

One brief note on citations: some appear as custom cite keys in place of the standard author abbreviation-year. This is to aid in identification of certain key documents whose authors may not be well-known and to ensure grouping of similar items in the bibliography.

1.1 Why it is crucial to be precise about semantics

It is worth briefly reviewing the advantages that a formal semantics brings to give some motivation to the subject of the paper. Computers execute machine code which, although detailed, is relatively easy to understand. The state of the hardware is simple, typically consisting of a huge linear vector of bytes and a small collection of registers. This makes the semantics of individual instructions of machine code fairly easy to follow[3] but programming directly in machine code has long been seen as time-consuming

[1]Many authors use the phase "formal definition"; following Peter Mosses, we reserve "definition" for a document that is an established standard; most formal semantic descriptions are separate from and written after any standard is set.

Jones heard John Reynolds wish that "formal methods should be the midwives of language design rather than the morticians". Section 6.6 lists disappointingly few cases where this has happened.

[2]Henceforth, unqualified references to "ALGOL" are to be taken to refer to ALGOL 60.

[3]Recent explorations of relaxed or weak memory machines have made this much harder, however.

and short sighted. High-level programming languages make the job of the programmer easier[4] but programs written in these languages require translation into machine code before they can be run. This task is typically performed by a compiler or interpreter.

The introduction of new languages does however introduce challenges and these have become more and more onerous as the level of abstraction in programming languages has increased. How can we be sure that the object code into which a program is translated has the meaning of the high level program both in the sense of being a good translation and also an expression of the programmer's wishes? Given that different machines have different low-level instruction sets, how can we be sure that different implementations of the same program perform the same task? Further, if we want to be certain of the effects of a program, we want to be able to perform some reasoning: how do we enable tractable reasoning at the higher-level of abstraction? If the specification or user manual of a programming language is written in natural language, how do we eliminate the ambiguity inherent in long strings of words? And how do we clearly communicate the meanings of the various language constructs between the language designer, programmer and compiler writer? One way to address these concerns is with formal semantics.

One person interested in formal semantics at its inception was Peter Landin; in his pair of papers [Lan65a; Lan65b] in which he draws a correspondence between ALGOL and λ-calculus, he notes:

> The attempt to fit ALGOL 60 into the AE/SECD framework can be considered from two sides. On the one hand, for someone familiar with ALGOL 60 it may clarify some of the features of AEs. [...] On the other hand, AEs illuminate the structure of ALGOL 60.
> — [Lan65a]

1.2 What do we mean by "semantics"?

Most dictionaries define "semantics" as something like "meaning", but this only provides an alternative noun; what is needed is a test that characterises the acceptability of approaches to describing semantics.

This paper is concerned with imperative programming languages (in fact, principally, one particular imperative language) and it is reasonable to think of programs in such languages as having an effect either on an internal machine state or externally visible entities such as files or databases. A semantic description should provide the ability to reason about the *effect* of a program. This is in contrast to the syntax which defines the texts of a language which are of semantic interest.[5]

There must, in any semantic description, be a given set of basic notions. In order to illustrate this, consider the simple example of giving meaning to the string of characters

[4] A panel at the *Mathematical Foundations of Programming Language Semantics* held at CMU in 2004, on which Jones sat, was asked an interesting two-part question by Vaughan Pratt: 1) How much money have high-level programming languages saved the world? 2) Is there a Nobel Prize in Economics for answering 1)?

[5] The further distinction between concrete and abstract syntax is made below; furthermore, the fact that either syntactic description is likely to be context free means that context dependencies have to be recorded separately. Some approaches handle such constraints statically in "context conditions" whereas others detect inconsistent uses of declared variables only dynamically (i.e. in the semantic rules).

111. It is tempting to read this as the decimal representation of "one hundred and eleven"; but it could equally be a string of binary digits and correspond to the decimal number seven; as octal it would have seventy three as its decimal equivalent. For each of these choices, it is straightforward to write a recursive function that takes a string of digits and computes its mathematical value. Such a function defines the semantics of the strings of digits.

To clarify the issue of base concepts, however, it is useful to be careful about two things. First, it is necessary to agree that there is a shared (between reader and writer) notion of interpreting descriptions of functions. Fortunately, the concept of functions, even recursive functions, is widely understood and notations tend to vary only in details of syntax.

The other issue might appear to be overly pedantic but it is important to understand it on a simple example. The base cases of the recursive function need to identify mathematically understood objects for the meaning of the digits. The need is to say that the mathematical concepts of zero, one etc. can be used in the calculation of the value of a string of what are only symbols. (There is, after all, no fundamental reason why one could not reverse the normal notational convention and say that the binary string 101 denotes the decimal value two.) Fortunately, natural numbers can be based on the primitive concepts of zero and the successor function. Moreover, recursive functions can be written for all of the arithmetic operators and Peano's axioms provide a way to prove results about these functions.

The issues above present challenges when defining the semantics of more complicated languages. There will, for example, be a need to be more careful when a single text in language admits more than one effect: a function from the language to its denotations is no longer adequate.

Sections 4 and 5 use denotational semantics and it is a key property of such descriptions that there is a way of reasoning about the objects to which programs are mapped. Without at this point being precise about how it is determined, the requirement is that the denotations are "tractable" in the same way that Peano induction makes it possible to reason about natural numbers.

There are many issues that make it more challenging to define the semantics of programming languages than, say, those of logic. One quintessential issue is the lack of "referential transparency" in programming languages: an identifier denotes different values as a computation proceeds. Moreover, in languages that offer parameter passing by location, the value of one variable can be changed by an assignment to a variable with a different name. Finding suitable techniques to describe the semantics of programming languages requires addressing a whole series of issues of this nature.

Perhaps the most obvious way to describe the effect of a program is to construct an interpreter that takes a program and a starting state and computes (or judges to be acceptable[6]) a final state. This is the essence of the *operational* approach to semantic description. It is however unlikely to be easy to reason about an interpreter written in the machine code of some particular computer. McCarthy (see Section 1.5) used the term "abstract interpreter" for one which is written in a tractable, functional, notation. Section 1.6 outlines issues that make it difficult to achieve ease of reasoning but the

[6]See the discussion in Section 6.1 about non-deterministic languages that require a way of saying that there is more than one valid result to a computation.

basic idea remains that of a mathematically tractable interpretation function.

Central to an operational semantic description is the notion of the (abstract) states that can be changed by the imperative statements of the language being described. In this sense, the term "model-oriented" can be applied to operational semantics. It is also the case that denotational semantics are given in terms of states and such descriptions are in the same sense model-oriented (in contrast to "axiomatic semantics"; see Section 6.7 below). In all model-oriented descriptions, it is desirable to make the states as abstract as possible as every state component brings extra complication in transition functions and makes reasoning more complex.

The key distinction between operational and denotational descriptions is that those in the latter class map programs, or their constituent parts, to functions from states to states. Where an operational semantics requires a program and an initial state, a denotational semantics maps just a program into a mathematical function. The mapping should be homomorphic from the (nested) structure of program components to the space of denotations.[7] It might be argued that this structural requirement encourages the use of smaller, cleaner, states. This argument is evaluated in Section 6.1.

One obvious reservation about operational semantics is the lack of abstraction inherent in interpreting programs statement-by-statement. For example, in the absence of concurrency, a program which adds one twice (in successive assignments) to a specific variable is functionally indistinguishable from one that adds two to the same variable in a single assignment. Since these two program fragments bring about the same state to state transition, a denotational semantics provides a way of reasoning about their equivalence in an established mathematical field: that of functions. The search for "full abstraction" has however proved rather difficult and is still unresolved for concurrent programs.

It is also debatable just how much abstraction is a good thing: to what extent are the two adding programs actually identical? Where is the line drawn between programs such that they become semantically different? In the case of the addition program mentioned above, the equivalence of the two programs seems clear, but must one then also consider two sorting algorithms to be equivalent as they both transform an unsorted array into a sorted one? The desired use of the program semantics may influence the answer to this question and an appropriate level of abstraction employed.

Two approaches to giving the semantics of programming languages attempt to distance themselves from the notion of state. Axiomatic semantics in the style of [Hoa69] provides rules of inference that facilitate proving properties about programs. Programming languages may also be given meaning by defining the equivalences between programs. Both of these approaches might be termed "property oriented" descriptions of semantics and are discussed further in Section 6.7.

All language descriptions ultimately need a universal "meta-language", as named by Fraser Duncan in an after-dinner speech at the Formal Language Description Languages conference discussed in Section 1.5 [Dun66] and this must be natural language, typically English. It is, of course, possible to describe the semantics of a programming language only in natural language and this is exactly what is done in the ALGOL Reports (see discussion in Section 1.4). What marks a description as formal is taking a

[7] Language constructs such as goto statements make this rather difficult!

very small collection of basic notions and then using these to provide the semantics of descriptions of one or more large and complicated languages.

It is also clear that any formal description approach must take a certain collection of basic objects as given (and presumably described in natural language); furthermore, both operational and denotational semantic descriptions rely on the notion of functions (or relations). These "building blocks" are reviewed in Section 6.3.

1.3 Why ALGOL is interesting

ALGOL was designed by the members of IFIP Working Group 2.1; a good account of the process is contained in *History of Programming Languages* [Per81; Nau81b].[8] The resulting ALGOL 60 language is powerful yet clean and it introduced many concepts that have been adopted in other languages. As Tony Hoare has commented [Hoa73]:

> Here is a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors.

The grammar of ALGOL is regular in that it allows, for example, blocks to contain statements and those statements can be blocks. Since blocks define their own name spaces, the same identifier can denote different variables in different scopes.

So-called "strong typing" means that no type errors can occur at run-time. The supporters of strong typing argue that the redundancy inherent in stating the intended way in which any variable is to be used is a key safeguard against minor slips resulting in either latent bugs or wasted time in debugging. ALGOL is very nearly strongly typed: all variables must be declared, but there is no requirement for constrained array or procedure parameter types.

A further challenge is present in ALGOL as defined by the Reports: the ability to declare variables as "own" adds an extra layer of complexity. Upon exit of a phrase (block or procedure), the values of variables are lost[9] and, if the phrase is re-entered, these variables are re-initialised. In contrast, in the case of "own" variables, their value is maintained after phrase exit so that if the phrase is re-entered, the previous value is available. This feature proved rather contentious (e.g. own arrays with dynamic bounds) and many subsets and revisions of ALGOL omit it, as do some of the descriptions discussed below.

Parameters to procedures or functions can be passed "by value" or "by name". The first of these is fairly conventional. In the case where a single identifier is provided as an argument,[10] "by name" parameters behave as what is now normally called "by reference" or "by location". The general form (in which an expression is passed to a "by name" parameter) essentially requires that a closure is formed so that the expression is evaluated as though it was in the calling context. Even the simple "by reference" mode introduces the problem that different identifiers can denote the same variable (or location).

[8] Perlis [HOPL, p91] quips "ALGOL deserves our affection and appreciation. It was a noble **begin** but never intended to be a satisfactory **end**.".

[9] Precisely how this is handled depends on implementation and definition, but the essential idea of these values being non-accessible remains.

[10] The ALGOL literature tends to refer to arguments as "actual parameters" and to parameters as "formal parameters".

This presents a challenge for any semantic model: identifiers with the same name but different values (allowed if they occur in different phrases) must not end up clashing, with some values lost or overwritten. This is avoided in the ALGOL Reports by use of the "copy rule". The idea is fundamentally simple and had been used by mathematicians for decades in any situation involving bound variables: copy the identifiers from their various locations into the target phrase and, if there is an identical name found, simply rename one of the variables. The intuitiveness of the idea belies the complexity underlying it and thus, while some of the descriptions discussed apply this principle, most avoid it by using other methods.

Procedures and functions can be defined by recursion. Although this is now common in languages, it required the invention of implementation techniques such as Dijkstra's "display mechanism". The story of recursion in ALGOL is not wholly straightforward and is presented clearly in [Hov14]. van den Hove argues that, although it seems recursion was sneaked into the language at the last minute, in fact the recursive nature of the language syntax and the substitution rules of procedure semantics make recursion innate in the language.

Furthermore, procedures can be passed as parameters. As is explained in Section 4, this decision presented particular difficulties for denotational semantic descriptions.

Explicit sequencing of execution by goto statements gave rise to considerable controversy after Dijkstra wrote his letter "goto statements considered harmful" [Dij68] (met with Knuth's defence in [KF71]). For better or worse, ALGOL allows label parameters, goto statements to close either blocks or procedures and even introduces further embellishments with switch variables. Modelling this collection of ideas presents interesting problems for the formal descriptions. In an unpublished note *Jumping into and out of Expressions* Christopher Strachey writes:

> Full jumps ... introduce an entirely new feature in programming languages (and one which increases considerably their referential opacity).

Goto statements may be local hops within a phrase, or may be full jumps which cause phrase structures to be closed if the target label is in a containing context. In the latter case, it is necessary to perform and cleanup housekeeping that would have occurred had the phrases (which are abnormally terminated) terminated normally. In ALGOL, such phrases can be either blocks or procedures.

The language makes the situation more complicated because labels can be passed as actual parameters to procedures. With the dubious argument of "orthogonality",[11] ALGOL also allows switch variables to which the programmer can assign labels.

Chris Wadsworth (see Section 4.5.3 on continuations) wrote to his supervisor Christopher Strachey (see Section 4 on denotational semantics)[12] that "Peter's Algol 60 paper ... I must admit I still feel a little surprised it's as long as it is — I guess Algol 60's just not nearly as 'well-behaved' as one tends to think it is."

ALGOL initially contained no input/output statements but these were added in [MHW76]. A small collection of "standard" functions are defined for ALGOL, such as a square root function.

[11] Some people argue that, because values of, say, type integer can be assigned to variables, there should be variables to which one can assign label values.

[12] Letter dated 1974-03-26 from Syracuse University (USA) held in the Bodleian archive of Strachey's papers.

1.4 Describing semantics without a formal semantic notation

There are a number of definition documents produced for the various versions of ALGOL, but our main reference is the 1963 'Revised Report on the Algorithmic Lanugage ALGOL 60' [Revised Report]. This was the most modern source at the time of the earliest of the descriptions below (that of Peter Lauer [Lau68]) and is the version upon which he based his semantic description.

1.4.1 Syntax

ALGOL 60 was the first language to be described with a formal, concrete, context-free syntax; this was primarily devised by John Backus and first used in the 1960 'Report' [ALGOL Report]. Note that ALGOL 58 (or International Algorithmic Language, as it was then called) did not have its syntax defined formally, but rather in natural language with examples. It was subjected to some improvements and additions when it was used by Peter Naur in the [Revised Report], as reported by Knuth in [Knu64].

The syntax of the language is defined formally using BNF (Backus–Naur or Backus Normal Form). A full discussion of this method is beyond the scope of the current paper, but represents a way to break down syntactic constructs, defined as their string literals, into their constituent parts. Recursion is used in BNF to express the nested phrase structure of ALGOL.

1.4.2 Context conditions

The grammars of the syntax are context-free, which means that they cannot define errors which are caused by syntactically valid structures used in the wrong context. Bob Floyd proved this in a short and neat article [Flo62], indicating that extra concepts are needed to rule out these errors. They are carefully described in natural English in the 'Revised Report'; some examples are shown below.

> Dynamically this implies the following: at the time of an entry into a block (through the begin since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.
> — [Revised Report, §5]

> The type associated with all variables and procedure identifiers of a left part list must be the same. If the type is Boolean, the expression must likewise be Boolean. If the type is real or integer, the expression must be arithmetic.
> — [Revised Report, §4.2.4]

> A label separated by a colon from a statement, i.e. labelling that statement, behaves as though declared in the head of the smallest embracing block, i.e. the smallest block whose brackets **begin** and **end** enclose that statement.

— [Revised Report, §4.1.3]

The use of careful wording like this does help to elucidate some of the common contextual errors and how to avoid them, but the lack of any kind of formalisation would have made the task of automatically checking for them, or proving their absence, rather tricky. For this to be possible, a more rigorous approach to context conditions is required; an example of an approach to this is discussed in Section 5.3 of this document.[13]

1.4.3 Semantics

Similar to the definition of context conditions, carefully-crafted English is used to provide semantics for the language. This section contains some representative examples.

> Statements are supported by declarations which are not themselves computing instructions, but inform the translator of the existence and certain properties of objects appearing in statements, such as ...
> — [Revised Report, §1]

In this way the meanings of the language are described as carefully as possible, but this necessity makes the definition a little convoluted at times.[14]

Another method the Report uses for semantics is to describe equivalences:

> The operations $\langle term\rangle / \langle factor\rangle$ and $\langle term\rangle \div \langle factor\rangle$ both denote division, to be understood as a multiplication of the term by the reciprocal of the factor. — [Revised Report, §3.3.4.2]

When the language construct to which meaning is to be given is complicated, this is often broken down iteratively into smaller parts, each of which is then subsequently defined. A good example of this is the **for** statement, [Revised Report, §4.6.3], in which the statement is first defined via a simple diagram as: 'Initialize; test; statement S; advance; successor'. Shortly following the diagram is an explanation for each of these terms and following that is a further expansion of terms used.

The semantic meanings are also separated on occasion by different cases; for example, in §4.7.3 the semantics of procedure invocation is given by different explanatory paragraphs depending on whether the statement is call by name or call by value. One example is given below; this serves to illustrate the version of the copy rule (see Section 1.3) used in the Report.

> *Name replacement (call by name).* Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved. — [Revised Report, §4.7.3.2]

[13] Some authors use the term "static semantics" for these context conditions (and "dynamic semantics" for what below is called simply "semantics"). These terms are not employed in this paper.

[14] That said, Peter Naur attacked Henhapl and Jones in [Nau81a] after publication of [HJ78], comparing the complicated mathematics of the formal model unfavourably to the structured English of the [Modified Report].

It can be seen that while this description leaves the reader fairly sure of how the name replacement system works, it provides no opportunity to use a formal reasoning system, or any indication of how these "systematic changes" ought to be accomplished. This may be compared with the function described in Section 3.5.1.

1.5 McCarthy's "Micro-ALGOL" description

The 1964 IFIP Working Conference held in Baden-bei-Wien, entitled "Formal Language Description Languages for Computer Programming", brought together most of the European researchers who were interested in semantic description approaches. The conference was organised by Heinz Zemanek and partially funded by IBM thanks to his influence [Utm64, p. 11.3]. This was the first ever IFIP Working Conference, and was sandwiched between two halves of the fourth meeting of IFIP Working Group 2.1 (concerning "ALGOL x" and "ALGOL y", names used at that time for proposed 1965 and 1970 versions of ALGOL respectively) [ALGOL Bulletin, No. 18]. As a result, the conference was well-attended by most members of WG2.1 and also non-members who had an interest in semantics: thus both the theory of programming languages and its application were well represented.[15] In the The proceedings [FLDL] only appeared in 1966 but are particularly valuable because of the effort that was made by members of the Vienna team to record and transcribe the discussions that followed the presentations.[16]

The first paper in the proceedings was written by John McCarthy: in [McC66] he provides an operational description of a subset of ALGOL that he dubs 'Micro-ALGOL'. This paper is a stimulus to much of the subsequent work on semantics in general and operational descriptions in particular (its influence on the IBM Vienna Lab's "VDL" approach is discussed in Section 2). McCarthy's use of the term 'abstract interpreter' is very useful in explaining the semantic approach.

One interesting observation is McCarthy's choice of subset for "micro-ALGOL": he doesn't take the obvious selection of assignments, conditionals and while statements but does include $goto$ statements. This decision actually forces him to look at retaining the whole text (for backwards $goto$s) and, in a sense, can be seen as the germ of the ULD "control tree" (see Section 2.5.4).

The authors of the ALGOL report had shown how BNF could be used to define the concrete syntax of a language: the production rules defined a set of strings of characters that were to be considered as valid inputs to an ALGOL compiler. With some care in their formulation, such syntactic rules could also be used by a parser or parser generator. In contrast, McCarthy introduced the idea of basing a semantic description on an "abstract syntax" that omits the syntactic marks that are there only to help parsing. He distinguishes: "synthetic syntax", which describes the constructors of the syntax classes and "analytic syntax", which describes their composition. A few items from McCarthy's table of abstract (analytical) syntax are shown in Table 1. He did not include any synthetic syntax in the conference paper; examples can be found in [McC62].

[15] In the Preface to [FLDL], Steel observes "Attendance was limited by invitation to recognised experts in one or more of the various disciplines of linguistics, logic, mathematics, philosophy, and programming whose frontiers converge around the subject of the meeting. The resulting group — 51 individuals from 12 nations — was ideal in size, breadth of experience, and commitment to the enterprise."

[16] Of course, there must have been many informal exchanges that were not captured.

Objects which belong to an abstract syntax class are recognised as such by applying predicates (e.g. $isvar$ in the example) and their components can be accessed by selector functions (e.g. $left, right$ in the example). McCarthy writes specific axioms to relate these functions/predicates.

Predicate	Associated functions		Examples
isvar(τ)			x
isprod(τ)	multiplier(τ)	multiplicand(τ)	$x \times (a + b)$
assignment(s)	left(s)	right(s)	s is "root := $0.5 \times$ (root $+ x/$root)"
			left(s) is "root"
			right(s) is "$0.5 \times$ (root $+ x/$root)

Table 1: Selection of McCarthy's abstract syntax.

The case for McCarthy's use of an abstract syntax for micro-ALGOL is perhaps less compelling than when one is faced with a language such as PL/I or Java in which there are many different ways of writing semantically equivalent texts.

The semantic function in [McC66] takes a program, a store and a program counter as arguments and delivers a store as a result:[17]

$micro\colon Program \times Store \times \mathbb{N} \to Store$

This signature is compared with those of other semantic approaches in Section 6.1.1 (see Fig. 7). The fact that this can be a functional relationship follows from the absence of non-determinism in Micro-ALGOL. The most compelling case for non-determinism in programming languages comes from concurrency but, even in full ALGOL, non-determinism arises from the order of expression evaluation (coupled with the presence of side-effects). The VDL description of ALGOL discussed in Section 2 has to provide a way to model such non-determinism; Section 6.1 explains how the more modern SOS rules define the set of permissible final states of a computation.

1.6 Dimensions of comparison

The intention is that this paper draws attention to the ways in which each of four different formal descriptions of ALGOL tackle the issues raised by the semantics of the language. For each of the approaches covered in Sections 2–5, the following items are discussed:

- the context of the work
- which version of ALGOL was taken as a basis for the description and whether any features were omitted
- syntactic issues (including the choice between concrete and abstract syntax and the handling of context dependant issues)
- the overall semantic style

[17]Strictly, the program counter is Curried but no essential use is made of this higher order idea.

- specific modelling issues (including how jumps are modelled)
- a postscript (including other descriptions in the same style and how the description might have been extended to cope with concurrency)

2 Vienna operational description

Peter Lauer worked at the IBM Laboratory in Vienna until 1972 and was regarded as the specialist logician, according to his colleague Wolfgang Henhapl. During Lauer's time at the Laboratory, he co-authored a number of publications including the guide for use of VDL version 2 [ULD-IIIvII-Meth] and some theoretical work on algorithms [Lau67]. Subsequent to working on the ALGOL description, Lauer obtained a PhD[18] under the supervision of Tony Hoare, who was at that time professor of computing science at Queen's University Belfast. Lauer spent only part of the time in Belfast and finished writing his thesis back in the IBM Vienna Lab. Following his time at IBM, Lauer obtained a lectureship at the University of Newcastle upon Tyne in 1972 and then a professorship at McMaster University in Canada in 1985; he continued to work in the field of theoretical computer science, including programming language design and implementation, until his retirement.

2.1 Background: A brief history of VDL

The story of VDL really starts with IBM's development of the PL/I programming language, described by Fred Brooks as "a universal programming language that would meld and displace FORTRAN and COBOL" [Shu15]. This was an ambitious objective in several ways. Some in IBM assumed that one universal language would free them from the need to maintain two compilers!

Furthermore, an objective of universality, compounded by design by committee, was almost bound to yield something akin to a tower of Babel. (A photograph of Pieter Bruegel's *Tower of Babel* covered an entire wall of the conference room of the IBM Vienna laboratory. At various different points in time, the figures in the bottom left corner were identified with people involved in IBM projects. Figure 1 shows key early members of the Vienna Lab in front of this wall.) The official definition of PL/I was written in natural language and given to the IBM Laboratory in Hursley, England, whose task was to develop a compiler.[19] This specification was initially referred to as "Universal Language Document" [Luc81], but quickly became known as solely as "ULD" without expansion, even in official documents.

At this time, the Vienna Laboratory under Heinz Zemanek was interested in formal definitions of programming languages, energised by the Formal Language Description Languages conference [Luc81]. The conference was organised by IFIP TC 2, of which Zemanek was chair, although interestingly the suggestion to make formal languages the topic of the symposium appears to have been made by Peter Naur [Utm63, p. 7].

[18] Zemanek consistently encouraged his staff to obtain their PhD qualification.

[19] There was a parallel activity in the IBM Böblingen Laboratory to develop a PL/I compiler for smaller IBM/360 machines; see [End13].

Figure 1: From left to right: (standing) Peter Lucas; George Leser; Viktor Kudielka; Kurt Walk; seated: Rutishauser; Kurt Bandat; Heinz Zemanek; Norbert Teufelhart

The IBM Vienna Lab had already implemented an ALGOL 60 compiler for the transistorised Mailüfterl computer that Zemanek's group had designed and built in the Vienna Technical University in 1962 so the group had experience in the area of compiler development.

Zemanek's prescient decision to move the Lab's focus from hardware to software coincided with IBM's development of the PL/I programming language.[20] The PL/I language was considerably more complex than ALGOL and the Vienna group argued for a formal description both to clarify the language and to record its semantics in a precise way.

There was a parallel activity in the IBM UK Lab at Hursley (Hampshire) that led to ULD-II [ULD-II-CS; ULD-II-AS; ULD-II-Trans; ULD-II-Sem] that is described as a semi-formal description. The different motivations of the two teams and their interaction are interesting. The Hursley team was led by David Beech who was a Cambridge trained mathematician.[21] The aim of the Hursley effort was to create a description that was precise but readable by compiler developers. This resulted in (an abstract syntax and) a formally defined state but with most state transitions described in careful prose.

As the Vienna group began the process of understanding the new language, they sent a series of numbered "LDV" notes that contained questions and requests for clarification to colleagues in Hursley who replied with a similarly numbered sequence of "LDH"

[20]As recorded in HOPL, this language was to have been called "New Programming Language" or NPL until the (UK) National Physical Laboratory pointed out their prior use of the abbreviation.

[21]The material here was reinforced by a discussion with Beech when he visited Newcastle on 2016-08-12.

notes. Over time, the technical depth of these questions and answers increased and both groups turned inexorably to formalism as the only way to pin down the decisions adequately. There were also visits between the two groups and Beech recalled having made seven trips to Vienna in one year.

This led to the Vienna Lab taking over the job of formally defining PL/I and developing a method for doing so, which became ULD-III. This description went through three major versions and was, confusingly, often referred to internally as ULD-*version*, although it really ought to have been ULD-III-*version*. The name Vienna Definition Language was coined by the American computer scientist J.A.N. Lee [LD69] and the tag VDL stuck. Peter Wegner's survey article [Weg72] might have played a part in cementing the name VDL.[22]

In the mid-60s, the Vienna group was small. Although the origins of the group were in hardware development at the Technical University of Vienna, Heinz Zemanek was far-sighted enough to insist that the future direction of their scientific work must shift to software. A fuller history can be found in [Luc81] and [Rad81], but it is worth listing here a few of the key steps towards the formal description of PL/I:

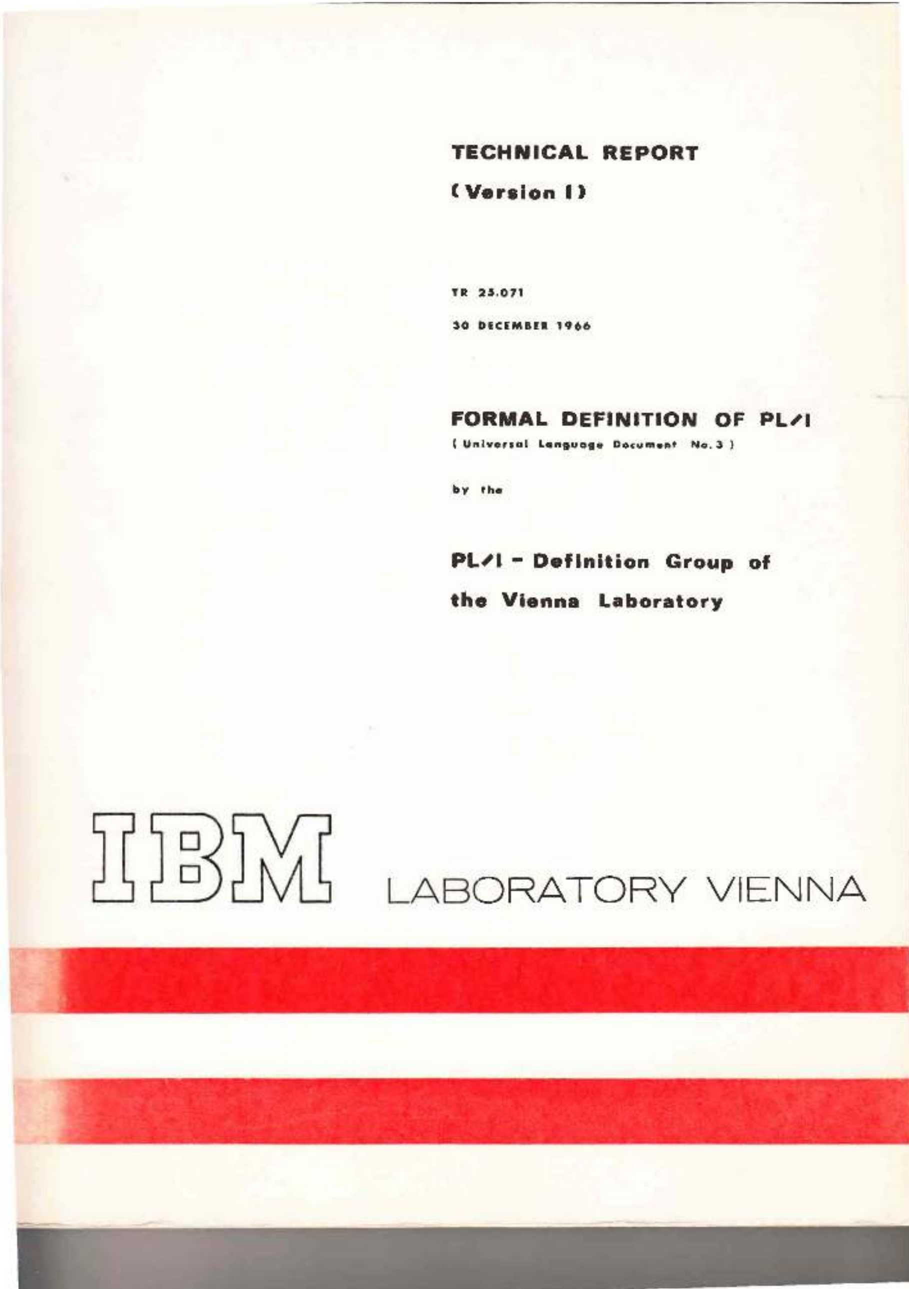
- In [Bek64] Hans Bekič discussed giving the semantics of "mechanical languages" by reducing them to elementary terms.[23] The initial focus is on expression languages but [Bek64, §4] addresses "programming languages" (i.e. those containing "statements") and includes the prescient comment that "a statement can be interpreted as a function mapping states into states".
- The involvement of members of the Vienna group in the September 1964 Baden-bei-Wien IFIP Working conference is mentioned above in Section 1.5.[24]
- Lucas [Luc81] states that "Work on the formal definition of PL/I started in September 1965." but already in July of that year, Kurt Bandat edited a collection of four papers [Ban65] that set out much of the VDL approach.[25]
- ULD-II [ULD-II-CS; ULD-II-AS; ULD-II-Trans; ULD-II-Sem]
- The first version of the complete PL/I description [ULD-IIIvI] appeared in 1966; the cover of the report attributes it to "PL/I – Definition Group of the Vienna Laboratory". The actual authors and their contributions are listed inside the report. See Figures 2 and 3.
- A second version appeared as multiple reports [ULD-IIIvII-Intro; ULD-IIIvII-Meth; ULD-IIIvII-Sem; ULD-IIIvII-CS; ULD-IIIvII-Trans; ULD-IIIvII-CT] in 1968. This version first introduced the axiomatic definition of storage; for a detailed description of this work see [BW71] since it does not relate to ALGOL.
- The final version (which postdates Lauer's ALGOL description) also appeared as a collection of reports [ULD-IIIvIII-Intro; ULD-IIIvIII-Sem; ULD-IIIvIII-CS; ULD-IIIvIII-Trans; ULD-IIIvIII-CT] in 1969.

[22]This survey also puts an emphasis on the notion of (VDL) "objects" that might surprise a current reader of the material.

[23]It is interesting to compare this to the idea that is discussed in Section 6.7 of giving semantics by defining equivalences.

[24]Lucas in [Luc81] notes "Members of the IBM Vienna Laboratory, involved in the preparation of the conference, had the opportunity to become acquainted with the subject and the leading scientists."

[25]Peter Lucas also presented a paper at the IBM (Internal) Programming Symposium at Skytop, Pennsylvania but this basically reiterates the material in [Ban65, §3].



TECHNICAL REPORT

(Version I)

TR 25.071

30 DECEMBER 1966

FORMAL DEFINITION OF PL/I

(Universal Language Document No. 3)

by the

PL/I - Definition Group of

the Vienna Laboratory

IBM LABORATORY VIENNA

Figure 2: Copy of the cover of [ULD-IIIvI]

The authors and their contributions

In the following, the authors are listed according to their main contributions by chapters.

Method and Notation
K. WALK: 1, 2.8, 3.2
P. LUCAS: 1, 2.1 to 2.7, 3.1 (except 3.1.35 and 3.1.7)
Expression Evaluation, Assignment, Storage Allocation
K. WALK: 4 (except 4.3, 4.8, 4.10, and 4.11), 5.3, 5.7, 5.8
G. CHROUST: 4.3
H. BEKIC: 4.8, 5.11
Flow of Control Statements, Prepass
P. LUCAS: 4.10, 4.11, 5.1, 5.2, 5.4, 5.5, 5.9, 5.10
Conditions, Tasks
K. BANDAT: 3.1.35, 5.6, 5.12, 7
Input, Output
P. OLIVA: 6.1 to 6.3, 6.5, 3.1.7
V. KUDIELKA: 6.4
Abstract Syntax
K. ALBER: Appendix I

Figure 3: Copy of the author list of [ULD-IIIvI]

By the time Lauer initiated work on the ALGOL description [Lau68], VDL had been used successfully to define the entirety of PL/I, a considerably larger language. The second version of ULD-III was available, indicating that the technique was quite mature by this stage. It is interesting to note that Zemanek was very keen on ALGOL 60[26] and a strong critic of ALGOL 68, so it is perhaps not unrelated that the description of ALGOL 60 came out the same year as ALGOL 68 and both around the end of the year.

It is likely that Lauer was chosen for the task of defining ALGOL in order to help him familiarise himself with VDL.[27] ALGOL was probably chosen for description due to its simplicity and elegance (particularly compared to PL/I) and to fight back against the critics of the VDL descriptions of PL/I who had claimed that they were too large and unwieldy.

2.2 Extent of ALGOL described

The VDL description of ALGOL covers all essential features of the language, as described in more detail below, including ‘own’ variables and non-determinacy of expression evaluation. The version of ALGOL defined is that of the ‘Revised Report’ [Revised Report].

[26]As evidenced by his choice of an ALGOL 60 compiler for Mailüfterl as a demonstration of its capability of handling high-level languages.

[27]Lauer acknowledges scientific contributions from P. Lucas, K. Alber, H. Bekič, and M. Fleck.

2.3 Syntactic issues

2.3.1 Concrete vs. abstract syntax

See [Lau68, §2 & 5]

As with the other languages defined in VDL, the semantic description is based on an abstract syntax which is given via a series of recursively defined identity predicates (beginning is-). For example, declarations are defined as variables, procedures, labels, or switches (see p. 2-3, equation 2.4).[28]

The notation for more complex syntactic constructs is based on the Vienna notion of objects: everything is either an elementary object (typically represented in upper case), or a composite object, with selectors to other objects (which may be elementary or composite themselves). A fuller explanation may be found in [ULD-IIIvII-Meth].

The VDL style of abstract syntax follows on from McCarthy's (see Section 1.5) by defining the compositional and constructional aspects separately; however, rather than having a separate constructor for each syntactic construction, the universal μ_0 function can create any object. Another change from McCarthy's explicit approach to abstract syntax is in the selection functions: where McCarthy states which selections are present for each syntactic construct, the VDL approach implicitly allows the use of any selector in a composite object to be applied to that object.

Abstract syntactic objects which comprise multiple parts are represented as a list of <selector:type> pairs. See, for example, p. 2-4, equation 2.17, in which an array is defined as the combination of a lower bound type expression, an upper bound type expression and a data attribute list of elements. This can be compared with the more straightforward notation of such composite objects in VDM; see Section 5.

A system for the translation of the abstract syntax into a concrete string representation is given in the final chapter of Lauer's report. Mapping in this direction works for ALGOL given its relative paucity of syntactic redundancy; for more complex languages a homomorphism from the larger set of concrete strings to the smaller set of abstract objects is more natural.

2.3.2 Handling context dependencies

All error checking in the description is performed dynamically at 'run-time' via the semantic rules; there is no attempt to catch any context dependancy errors statically. This is partially due to the abstracted nature of the syntax description preventing easy symbol checking; but compare with Section 3.3.3 which follows a similar system but does have some static checking.

During interpretation statements, errors are typically produced by distinguished cases, often default cases, and some explanatory English sentences are often written underneath the formulae.

[28] Page numbers in the report are split by chapter.

2.4 Overall semantic style

VDL uses a small-step operational semantics described in terms of the actions of an abstract interpreting machine. Control flows via the abstract concept of a control tree. At any point in time, the leaves of the control tree are (equally valid) candidates for execution at the next step. In ALGOL this makes it possible to define the non-determinism involved in expression evaluation; it would have a larger role in a language that offered concurrency (as, of course, was the case in PL/I).

The structure of VDL descriptions uses objects to represent all values: either elementary objects (the base types and Ω for the 'empty object') or composed of named selectors to other objects. Interpretation is performed by a series of nested functions, often defined by cases. Strictly, the non-determinism means that the semantics has to allow a set of possible results:[29]

$int\text{-}program$: $AP \times \Xi \rightarrow \Xi$-**set**

but, in common with other VDL definitions, the description tends towards "non-deterministic functions". This signature is illustrated in Figure 8 and contrasted with other approaches in Section 6.1.1. In particular, Plotkin's "Structural Operational Semantics" (SOS) makes explicit the recognition of the semantics as a relation (see Fig. 10).

Following standard VDL style, Lauer does not indicate the types of any of his functions, which makes the reading somewhat difficult.

The semantics of each language construct is given by an abstract interpretation or evaluation function. These are commonly split by cases and either modify state objects directly or call on other interpretation or evaluation functions.

2.5 Specific points

2.5.1 Environment/State

See [Lau68, §3].

The environment and store of the abstract interpreting machine are separate in the description; a stack of environments is, however, one component of the overall state, for it is a very large state.[30] The state (Lauer's report uses ξ for members of Ξ) is split into six components: the denotation directory `DN`, the environment `E`, the dump `D`, the unique name counter `UN`, the control `C`, and the control information `CI`.

Environments link program text identifiers with globally-unique names; only one environment is active at any given time.

The dump is a stack of environments for phrases (blocks and procedures) which have been entered but not terminated. The first step in the interpretation of a block or procedure statement is to push the current environment onto the dump and create a new

[29] In contrast to the signature for McCarthy's which used Σ for the set of all possible stores, Ξ is used here to emphasise that VDL states contain much more than the store (see below).

[30] This approach is often referred to within semantics literature as 'grand state' and is discussed further in Section 3.

environment; the final step is to make the top element of the dump the current environment. (Clearly, there need to be special actions where a construct ends abnormally.)

The denotation directory links global unique names with the values (for variables) or declarations (for procedures) which they denote. Associated type information is also included. Note that no values are ever removed from the $\underline{\mathtt{DN}}$; old values no longer present in the $\underline{\mathtt{E}}$ or $\underline{\mathtt{D}}$ are simply inaccessible.

The unique name counter is an integer value which increments every time a new identifier is detected and thus handles assigning unique names to all identifiers globally.

The control part of the state contains the set of source statements that are to be executed by the interpreting machine, which can be considered as an abstract tree. Each instruction may have a set of successor instructions and leaf nodes are candidates for execution. Interpretation of certain instructions may cause changes to the state of the machine, including the control tree.

Finally, the control information contains three parts: the whole program text; an index part which is either an integer pointing to the particular part of the program text which is next to be executed or a special constant when the active text part is a **for** statement; and a control dump which operates similarly to $\underline{\mathtt{D}}$ but handles the return control parts for nested expressions such as procedure calls embedded within expressions.

Documents on the VDL method (e.g. [Luc81; ULD-IIIvII-Meth]) often cite the influence of Peter Landin, and this can be clearly seen in the composition of the state. Landin's SECD machine [Lan64] bears a strong similarity to Ξ. The environment and dump have essentially identical functionality in both Landin's machine and the VDL state and the combination of the control and control information in VDL share functions with the stack and control in Landin's approach. It is interesting to note, however, that despite this similarity in data structures the essential approach to semantics is quite different in that Landin is giving a semantics to "imperative applicative expressions" which are used as the denotations of ALGOL programs (see Figure 5 and discussion in Section 6).

2.5.2 Shared name space

The $\underline{\mathtt{DN}}$ component of the state contains the value (as well as the typing information) of every variable declared in the program up to the current execution point, and the texts and parameter information of procedures. It is global to the whole program regardless of environment. This enables sharing of values between environments as long as ids are passed.

2.5.3 "Own" variables

See [Lau68, §4.2].

"Own" variables (see Section 1.3) are handled in Lauer's ALGOL description. A prepass executed before program interpretation replaces all instances of "own" variable identifiers with uniquely generated integer identifiers (see p. 4-3, equation 4.1). This ensures there are no name clashes between "own" variables.

"Own" variables are not interpreted or evaluated any differently from normal variables. The difference in the evaluation occurs at the block interpretation level, where the **update-env** function has separate cases for "own" and non-own variables. Normal variables are assigned a new unique name each time the block is entered, but "own" variables keep the same id they were assigned by the pre-pass. This allows access to the previous value of the "own" variable still stored in the DN.

2.5.4 Handling of jumps

See [Lau68, §4.5]

The handling of jumps in the control tree context is the cause of a lot of the complexity in VDL descriptions. It is also the necessity of handling jumps which leads to the placing of a stack of environments in the state. The germ of the idea is conceptually the same as the way in which [McC66] interprets jumps, but complicated by the phrase structure of ALGOL.

There are four parts to the interpretation of jumps (see p. 4-19, equation 4.44–53):

- Close environments in the dump until the environment containing the id for the destination label is found.
- Close control dump elements until the labelled statement is found.
- Advance control information index pointer to labelled part (number, FOR, or conditional).
- Update control information with index and resume sequential interpretation.

2.5.5 Procedure value handling

See [Lau68, §4.4.2].

In operational descriptions such as Lauer's, procedures are simply represented by their texts so there is none of the complexity about higher-order functions that has to be faced in denotational semantics. The returning of values from type procedure execution environments to their calling environments is not handled during the procedure execution interpretation (**int-proc-st**; p. 4-9, equation 4.17) but instead during the evaluation of the procedure call (p. 4-26, equation 4.68). At this point a unique name is created as an identifier for the value to be returned and stored in the calling environment. It is passed into the interpretation function as a parameter and when the value is calculated during the procedure call it is stored in that environment under the same id. As the value has the same id in both the calling and procedure environments, it is accessible by both in the DN.

2.6 Postscript on VDL

This description of ALGOL has no problem with higher order functions; procedure denotations are simply a text and an environment and so it is simple to pass one procedure as a parameter to another with its denotation text being operated upon.

Although there is no concurrency in ALGOL, the control tree model can be used to handle non-determinism, as had been shown in the definition for PL/I. Each leaf in the control tree is an equally valid candidate for next execution and so these can be picked in any arbitrary order to model concurrency.

Lauer's PhD [Lau71], under the supervision of Tony Hoare, followed the ALGOL description and showed the fundamental consistency of various semantic methods. These were classified into constructive (such as VDL) and implicit (such as Hoare's axiomatic) approaches.

In addition to the descriptions of PL/I, ALGOL and large parts of FORTRAN [Zim69], J.A.N. Lee published a description of BASIC in 1972 in classic VDL style [Lee72] and a semantics of Prolog is given in [AB87].

3 Vienna functional description

Cliff Jones went on assignment for two years to the IBM Vienna laboratory in August 1968 with the aim of seeing whether the difficulties seen in the development of the PL/I F compiler could be avoided by basing compiler design on a formal description. Peter Lucas had already written the twin machine report [Luc68] linking the models used in the IBM Hursley definition of PL/I (worked on by Dave Allen, an author on this ALGOL description) with the Vienna definition. One of Jones' first activities was to read Lauer's ALGOL description prior to its printing, so he had a good degree of familiarity with the style and approach used in Vienna. Jones also worked on the notion of objects, as in for example abstract syntax (see Section 6.3).

More relevant to the ALGOL description was the paper from Jones and Lucas on proving implementation correctness. The proof of Lemma 10 of [JL71][31] was gratuitously difficult: it stated that the environment was unchanged over the execution of a single statement; this is complicated by the fact that the statement could be a block or a procedure call. It was thus clear, even in operational semantics, that a 'small state' approach would be preferable to the traditional VDL 'grand state'.

After finishing his first period in Vienna, Jones returned to IBM Hursley to take over the Ad Tech group, amongst whom were the Dave Allen mentioned above, Dave Chapman and Peter Gershon. Jones was, at this time, pushing the 'exit' concept mentioned below, and the Ad Tech group's first project was this ALGOL description using that idea. Later, they worked on an early "Formal Development Support System"[32] before Jones returned to Vienna in early 1973.

3.1 Background: Why 'functional'

An important motivation for this shift in definition style was to move away from the messy control tree hacking for explicit sequencing; Jones' view was that jumps shouldn't "take the interpretation by surprise." [ACJ72, §2.1]

[31] First available as a Vienna Lab technical report (TR25.110) in August 1970 immediately before Jones moved back to the UK. Another relevant report is [HJ70b].

[32] FDSS was an attempt to build support for program verification using proof obligations for relational post conditions that eventually crystallised in program development aspects of VDM.

The exit idea was to pre-plan a way of capturing abnormal termination: it was first published in a report by Henhapl and Jones [HJ70a]. This concept also played a major part in the VDM style of denotational semantics (see Section 5), but in the current description it was presented in a rather verbose form.

The main aim of the exit approach was to address gotos without breaking the inherent stack nature of the phrase structure of ALGOL. This provided the main impetus for the new description and a number of changes percolate through based on this, such as the inclusion of sets and the ability to handle non-determinism in expression evaluation.

The functional semantics discussed here also differ from previous VDL practice by using a small state, although this is less obvious than in [BBHJL74; HJ78] because of the use here of the "copy rule" (see Section 3.5.1 below). To a large extent, the decision to move to a small state was a reaction to the difficulty of proving the difficult twin machine lemma.

3.2 Extent of ALGOL described

The language defined was the ECMA Subset of ALGOL, which was described in 1963 [Dun63] and published in April 1965 by the European Computer Manufacturers' Association [ECM65].

This is a smaller version of ALGOL 60, designed to be simpler and easier to implement across multiple computers. Many of the more contentious elements of ALGOL are removed, such as "own" variables and recursion (See Section 1.3). Although this description does omit "own" variables, recursion is kept in: the stated aim is to avoid some of the less clearly-defined features, while defining a language more oriented to static compilation [ACJ72, §1]. Standard functions are included.

Non-determinism in expression evaluation is handled, though the fundamental part of this process is left undefined and there is no cohesive story of how this fits with a functional view of semantics (see Section 3.5.4). The remaining two descriptions also duck non-determinism to some extent.

3.3 Syntactic issues

3.3.1 Concrete vs. abstract syntax

See [ACJ72, §5.1.1]

As with the previous VDL description, an abstract syntax is used. Rather than a function to turn abstract syntax into concrete, as presented in [Lau68], there is a translation function discussed which takes concrete syntax strings and translates them into abstract objects. There are a number of comments about this translator scattered through the description but the translator itself is not defined. Any string of correct syntax, as defined by the ALGOL Report, will translate into an abstract object defined by $is\text{-}program$.

See [ACJ72, §3]

The same notation style for abstract syntax from the previous VDL description is maintained (see Section 2.3). Essentially, syntax is described by a series of nested identity

predicates. These are actually used in the definitions of some of the semantic functions, providing a type signature, which makes them considerably easier to follow. Interestingly, the description includes a large section on notation, which essentially just duplicates the information in [ULD-IIIvII-Meth].

Once again, the essential building blocks of the description are objects, although the view is shifted somewhat by the inclusion of sets.

3.3.2 Inclusion of sets

See [ACJ72, §3.7–8]

The move to the exit approach requires the keeping of labels with their statements, rather than the use of abstract index pointers as used in Lauer's description (see Section 2.5.4). There can be multiple labels associated with one statement and they can change dynamically due to switch variables, so to cover this the definition language is extended to include sets. This brings the non-deterministic `for` `some` construction which picks an arbitrary member of a set and also path-els which are 'selectors' for any given set element. path-els are composed into paths, which represent the unique location of any given part within the program as a whole. Neither of these is defined fully but are presumed to exist.

3.3.3 Handling context dependencies

See [ACJ72, §2.4 & 4.3]

Unlike in the previous VDL description, where all error checking was performed dynamically, this description begins the process of static error checking which leads to the context conditions seen in the VDM Denotational description (see Section 5.3). As many type errors as possible are trapped before the semantic function is applied; the document observes that the aim was "basically to check those things which rely only on symbol matching and omit those checks which, in general, rely on values of symbols".

Some of this error checking is presumed to be done by the translator (see Section 3.3.1). Notes are given by some constructs for the translation process: these are typically in the form of predicates featuring implications and state some of the syntactically-valid programs to which it is not possible to give semantics.

Static error checking is aided by the use of the $desc$ function, which, given a path to an id and the text containing that id, provides contextual information: specifier, description or label description.

3.4 Overall semantic style

The main semantic style is an operational, small state approach. The functional signature is roughly $Program \times \Sigma \rightarrow \Sigma$, where Σ contains the state-like components vl, dn, and Abn. In this way, it is most similar to the simplicity of McCarthy's semantics (see Figure 7 in Section 6.1.1). Laying aside notational differences and some specific points (as below), the semantic style is not much changed from Lauer's description. Meaning is given by a series of recursive interpreter functions, nested down from $int\text{-}program$.

This function only has an effect if there is a block which starts the program, reflecting the ALGOL procedural approach.

The layout of the document is unusual: it is printed on landscape oriented paper in order to accommodate long formulae, and has large vertical gaps. This is so that the description can be placed alongside the ALGOL Report and the formula will match the sections of the Report. This also means that, unlike in Lauer's description, abstract syntax and semantic functions for each construct are grouped together, rather than being separated into different sections. Important functions are provided with type signatures and there is a cross reference of functions and abstract syntax provided at the end of the document, linking the declaration of each entry with its uses.

3.5 Specific points

3.5.1 Environment/state

See [ACJ72, §2.2–2.3 & 4.4–4.5]

The issue of state vs. environment is actually rather hidden in this description. The semantics for the phrases of the grammar (such as blocks) work in a similar way to how mathematicians describe the lambda calculus' bound variables. The "copy rule" as described in the ALGOL Report (See Section 1.3) is followed here: variables carried into phrases (parameters into procedures and existing values into blocks) are simply kept with their current identifiers, unless clashes are detected, in which case name changes are made as appropriate using the $change\text{-}text$ function (see [ACJ72, §5.4.7]). This makes it difficult to make a direct comparison with descriptions which use an environment.

So there is no broad, globally accessible state as such. Instead, two variables dn and vl act as state-like components. The dn is a set of pairs between ids and denotations (which are either types, or meta-components like labels, arrays, and procedures) and the vl is a set of pairs between ids and values. The same ids are used in both dn and vl and thus information on each variable is preserved.

3.5.2 Shared name space

The dn and vl are passed around most of the semantic functions and so are accessible wherever needed. The key idea is to restrict these state components to only the parts needed at any given. Thus, while most statement interpretation functions take both dn and vl, they only return a vl because the meta information on variables will be unchanged. Smaller and auxiliary functions tend to use only specific parts of these components.[33]

The "copy rule" described in the previous section prevents clashing of ids in the shared name space; it is applied whenever blocks are entered or procedures activated.

[33] This is precisely the reaction to the problems discussed above with respect to the proofs in [JL71].

3.5.3 Handling of jumps

See [ACJ72, §2.1, 4.1, & 5.4]

The exit mechanism, as first proposed in [HJ70a], is used. The essential idea is that functions $\Sigma \rightarrow \Sigma$ become functions $\Sigma \rightarrow \Sigma \times Abn$, where Abn represents an abnormal exit. It is Ω (the null object) if none are encountered and it contains the label of the statement to be jumped to when a goto is encountered.

The abnormal component is checked for and handled by many of the interpreting functions but, without any "combinators" to hide this away (see Section 5.4), the approach seems clumsy and long-winded. Nevertheless, the description serves as a good proof of concept: the exit idea works for a real language.

The interpretation of goto statements is very simple: when one is encountered, the label of the destination statement is placed into the abnormal part and simply returned from the function where the calling $int\text{-}st$ function can handle it. The only catch is that if a label already exists in the abnormal part (as may happen if a goto occurs during expression evaluation) it stays there.

The $int\text{-}st$ function handles the majority of the work: first, it checks the 'locality' of the label in the abnormal part, determining whether the destination is within the current phrase. If the label is not local, the current phrase's interpretation is simply halted and the current vl and existing abnormal part are returned to the calling $int\text{-}st$ function, where locality can be checked again. In ALGOL, jumps can only be made to destinations in the current phrase or a containing phrase, so this approach allows all allowable localities to be checked.

Once the locality of the label in the abnormal part is reached, the $cue\text{-}int\text{-}st$ function checks whether the current statement has the label in question; if it does, $int\text{-}st$ is called and interpretation proceeds as normal. Otherwise, $cue\text{-}int\text{-}unlab\text{-}st$ checks through the rest of the phrase's statement list for the id of the label in question and passes it back to $cue\text{-}int\text{-}st$.

3.5.4 Non-determinism in expression evaluation

See [ACJ72, §4.2 & 5.3]

Although the expression evaluation order is well-defined in ALGOL for numerical operations, there are certain subexpression evaluation orders which are not defined. These include the evaluation order of actual parameter (argument) lists to procedures. Some of these, conditional and especially switch expressions, can have side effects and so their order matters. The function for evaluation of expressions, therefore, has some non-determinism which uses the `for some` construction from a 'ready set' of subexpression parts. Further complication comes from the potential inclusion of labels here and so the expression evaluation also has to return an abnormal part.

3.5.5 Procedure value handling

See [ACJ72, §4.2, 4.4, & 5.3]

As mentioned in Section 3.5.1, a version of the ALGOL "copy rule" is used to model the movement of variables into and out of procedures. The decision to use this approach, rather than the shared denotation directory and environment of classic VDL, is due to both the desire for a smaller state and an attempt to follow the ALGOL Report more closely.

The process for handling procedures is a little involved and is worth breaking down in some detail.

1. All actual parameters are evaluated, including those which require procedure evaluation (the process may be recursive but ultimately simple values will be obtained).
2. The match between formal and actual parameters is tested for type errors.
3. Pairs are built up of local formal parameter id and evaluated actual parameter (in the case of by value parameters) or local formal parameter id and external id (in the case of by name parameters).
4. If the procedure is typed, a declaration for the return value is inserted into the program text.
5. A modified version of the procedure text is created with the actual parameters inserted, which is then interpreted and the resulting vl passed back out.

After this process, an id exists outside the phrase of the procedure for the returned value and the vl contains the value of this id and thus the procedure's result can be accessed.

Once the procedure is completed, an $epilogue$ function deletes all variables used in the procedure from both vl and dn, so only the returned value from type procedures is kept. This also applies to closed blocks and is another part of the general effort to make the state smaller. By contrast, in Lauer's description, values are never deleted and thus contribute to the grandness of state. If at any time during procedure evaluation a label appears in the abnormal portion (which is passed between all procedure evaluation functions), the epilogue function is called early and the jump interpretation starts.

3.6 Postscript on Functional Semantics

The later use in VDM of the exit "combinator" (see Section 5) avoids the heaviness of the many case distinctions in [ACJ72].

4 Oxford denotational description

One approach to semantics not explored in the preceding sections is the denotational method, developed in Oxford in 1969 primarily by Christopher Strachey and Dana Scott. In 1974, a PhD student under Strachey, Peter Mosses, took on the task of providing a denotational semantics of ALGOL in the Oxford style. It is interesting to note that Strachey's opinion of ALGOL was not high (indeed, he went so far as to co-write with Wilkes a paper [SW61] outlining the many faults he perceived in the language) which prompts the question of why this language was chosen. One clue comes from Mosses' acknowledgements in [Mos74] which start with:

> The original inspiration for this report came from reading [1] (=[ACJ72]) and [2] (=[Lan65a; Lan65b]), as it was felt that a shorter and less algorithmic description of ALGOL 60 could be formulated in the Scott-Strachey semantics.

So ALGOL was already being seen as a standard on which language description methods could be demonstrated and compared. As the system for using continuations to handle jumps had just been worked out, there was a desire amongst the Programming Research Group (PRG) community in Oxford to provide a full semantics of a language with jumps and ALGOL was an obvious choice.

Another contextual comment is that Mosses' actual thesis topic [Mos75a] was the design of a system that would enable the generation of compilers from a semantic language description; this required formalising the syntax and grammar of the semantic metalanguage. Although the ALGOL description was written in this MSL style (also presented in [Mos75b]), and used somewhat as a proof-of-concept, it was never actually run on this "semantics implementation system" (personal communication June 2016). It is noted in Section 4.1 below that a denotational description of the Sal language must have been largely worked out (by Strachey and Milne in their Adams Essay, finished in 1973) before Mosses wrote his PRG monograph on ALGOL but the latter presented much the most ambitious description task tackled by the PRG group at that point in time or, in fact, thereafter.

4.1 Background: Brief history of 'denotational' concept

This is not the place to attempt a full history of the development of what is variously referred to as Scott-Strachey, Mathematical or Denotational Semantics. The current authors consider that, for the current purposes, the beautifully clear exposition in [Sto77] and [Sto80], Campbell-Kelly's insightful [CK85] and the issue of *Higher-Order and Symbolic Computation* dedicated to Strachey (Volume 13, Issue 1) absolve them of the need to attempt such a history. However, to facilitate merging the time sequence of the evolution of ideas, it is worth noting the following events and their dates:

- both Scott [Sco00] and Penrose [Pen00][34] record that Roger Penrose suggested to Strachey in around 1958 looking at Church's Lambda Calculus.
- It is widely claimed that Lisp 1.5 was based on the Lambda Calculus. This is not, of course, the same as arguing that McCarthy envisioned using the Lambda Calculus in giving semantics and the evidence in his approach described in Section 1.5 is certainly to the contrary.[35]

[34] Penrose records "I cannot clearly remember at what stage I tried to persuade Christopher Strachey of the virtues on the Lambda Calculus. As I recall it, my own ideas were along the lines that that the operations of the Lambda Calculus should somehow be 'hard-wired' into the computer itself, rather than that the calculus should feature importantly in a programming language. In any case, my recollections are that Strachey was initially rather cool about the whole idea. However, at some point his interest must have picked up, because he borrowed my copy of Church's book and did not return it for a long time, perhaps even years later. When I eventually learnt that he and Dana Scott had picked up on the ideas of Lambda Calculus, it came as something of a surprise to me, as I do not recall his mentioning to me that he had taken a serious interest in Church's procedures."

[35] Furthermore, McCarthy frankly writes in [McC81] "And so, the way in which to do that was to borrow from Church's Lambda Calculus, to borrow the lambda notation. Now, having borrowed this notation, one of the myths concerning LISP that people think up or invent for themselves becomes apparent, and that is that LISP is somehow a realization of the Lambda Calculus, or that was the intention. The truth is that I

- Peter Landin noted the "correspondence" between fragments of ALGOL and Lambda Calculus in [Lan65a; Lan65b].
- In [Lan66] (recall the conference was 1964), Landin gives a mapping from ALGOL to "imperative applicative expressions", a modified form of Lambda Calculus notation which uses an SECD machine [Lan64] to give semantics (see Fig. 5 in Section 6.1).
- Strachey's paper [Str66] at the 1964 conference[36] discussed how an early version of this approach (as presented in [Lan64]) could be applied to CPL (a language developed by Strachey and others [BBHS63]).
- Scott and Strachey first met at the April 1969 meeting of IFIP WG 2.2 in Vienna and Scott was immediately "struck by Strachey's striving to isolate clear-cut general principles" and found his approach "the most sympathetic of the members of the group" [Sco00].
- Scott visited the IBM Vienna lab in August of the same year and presented the work [BS69] that he had been doing at the Mathematisch Centrum in Amsterdam (now Centrum Wiskunde & Informatica).
- Scott spent one semester in Oxford (around October 1969–January 1970): initially, he believed that it was impossible to construct a mathematical model of the type-free Lambda Calculus [Sco69][37] but after a sudden inspiration a succession of foundational papers were written initially as PRG monographs [Sco70; Sco71b; Sco71a; Sco73]. In [ST15] Scott says this about his inspiration: "If when you go from a space to the function space it seems more complicated, maybe there's a space such that when you go to the function space it isn't more complicated, so a space can be isomorphic to its function space."
- Strachey visits Scott in Princeton [SS70].
- Scott accepted the new Professor of Mathematical Logic chair and returns to Oxford in 1972; sadly, Strachey and Scott (in Scott's words) "had so many obligations and duties as new professors at Oxford that [their] joint work could never again be so concentrated" [as it was in 1969].
- Scott continued to refine models for the untyped Lambda Calculus e.g. [Sco73].
- The technical concept of "continuations" as a way to handle abnormal termination was invented by Chris Wadsworth at the PRG[38] and published in a joint paper with Strachey [SW74].
- Robert Milne and Strachey wrote their submission [MS74] to the Cambridge Adams prize competition in the period from early 1973 right up until the sub-

didn't understand the Lambda Calculus, really." (There is a specific discussion in the same article (P.180) on getting the binding rules wrong.)

[36] A draft of this paper is contained in the archive of Strachey's papers in the Bodleian Library and it is clear that it was completely written prior to the meeting in 1964.

[37] One major issue is the "cardinality problem": the number of functions $\mathbb{N} \to \mathbb{N}$ must have a higher cardinality than that of $\mathbb{N}$. Thus, there are more procedures over $\mathbb{N}$ than $\aleph_0$. If one associates an untyped lamda-defined function with procedures that can be passed as arguments to themselves a paradox is likely. Scott resolved the problem by posing suitable restrictions on functions so that domains could be constructed that can be viewed as partially ordered lattices.

[38] Most denotational semantics publications (e.g. [Sto77]) credit Wadsworth and also Lockwood Morris independently; the story is, however, slightly more complicated: see [Rey93] for a fuller history.

mission deadline in December 1974. This was intended as a comprehensive account of the fundamental concepts in programming languages, how they may be modelled using a denotational semantics, illustrated by a full denotational definition of a significant language, SAL, and a method for giving implementations of languages from the semantics, together with proofs of equivalence and correctness.

- Strachey died suddenly of hepatitis in May 1975, shortly after hearing that his Adams Essay submission was unsuccessful.
- Milne completed his (Cambridge) PhD [Mil74].
- Milne rewrote the Adams essay in book form as [MS76a; MS76b].
- Joe Stoy, a lecturer at the PRG who worked closely with Strachey and was the internal examiner for Mosses' DPhil, published a textbook on the denotational semantics style intended as an easier-to-read alternative to the Milne-Strachey book in 1977 [Sto77].

4.2 Extent of ALGOL described

See [Mos74, p.3 & C9]

Mosses declined to model own variables claiming (with justification) that they were ill-thought out at that time. He does mention (and indicate where to add) standard functions.

4.3 Syntactic issues

See [Mos74, p.5 & C2]

Mosses bases his semantic description on a concrete syntax of ALGOL using "annotated deduction trees" in the words of [SS71], which are tagged with labels that correspond with fragments of concrete syntax. This has some of the advantages that are claimed for using an abstract syntax. Whether one likes or dislikes the inclusion of syntactic parsing clues such as **begin/end/if/then/else**, or prefers distinct records such as $Block$, If, $Assign$ as proposed by McCarthy and deployed by the Vienna group, is probably just a matter of taste. Interestingly, Mosses does use constructed objects such as $makeArray$, $makeBounds$ (and their associated implicit selectors) but not for the syntactic classes.

Mosses also makes the point about not needing to worry about parsing. One could argue that using an abstract syntax fits Strachey's dictum "that one should work out what one wants to say before fixing on how to say it".

There are no context conditions in [Mos74]; so the semantics has to trap type errors dynamically that could have been detected statically.

4.4 Overall semantic style

By 1974, it was accepted that the basic space of denotations should be functions from stores to stores and Mosses employs these as the basic type. The situation is somewhat complicated by the use of "continuations" to handle abnormal exit from phrase structures (see Section 4.5.3 below).

The semantic function is, as far as possible, a homomorphic mapping from phrases of ALGOL to the aforementioned denotations. The store-to-store denotations are, of necessity, unnamed functions and have to be defined by lambda expressions. In Fig. 9 in Section 6.1.1, this is suggested by showing λ creating functions out of the basic set Σ.

The great advantage of making the basic semantic blocks functions in the purely mathematical sense is that they are well-known mathematical objects with well-known properties. This tends to make reasoning about them more straightforward and tractable than in an operational semantics where reasoning has to be performed over the steps of the interpreting machine.

There are two parts to the monograph: the first contains a brief introduction and references plus 30 pages of formulae constituting the formal description itself; the second provides a 20 page commentary thereon.

Perhaps abiding by Strachey's comment that one can do much more with an equation that fits on one line, Mosses uses identifiers for functions and their arguments that are often single Greek letters. Although he provides a decoding of these names in his commentary, these offer little intuition to the reader. It might be argued that this approach to brevity would not scale to a larger language description and one might even ask whether the decision was optimal for that of ALGOL.

The semantic functions are represented in the description with cursive capital letters, and the fragments of deduction tree upon which they operate are enclosed within "Strachey" brackets: $\mathcal{A}[\![t]\!]\rho\theta$. These functions take multiple arguments (a term, an environment and a continuation) but, rather than having a tuple, the arguments are Curried. [Sto77] argues that this allows varying levels of detail to be supplied for a slightly different meaning:

$\mathcal{A}[\![t]\!]$ is the meaning of a command *in vacuo*;
$\mathcal{A}[\![t]\!]\rho$ instantiates the variables by adding in an environment;
$\mathcal{A}[\![t]\!]\rho\theta$ adds a continuation, making the command "ready to go";
$\mathcal{A}[\![t]\!]\rho\theta\sigma$ is a particular execution of the command in a particular state.

4.5 Specific technical points

4.5.1 Environment/Store

See [Mos74, p.9, C4 & C18]

The store (*Map*) is a "small state" object which associates locations with values; in addition there is an $Area$ that indicates which locations are in use. The $Area$ would appear to be needed because *Map* is a general function from the entire infinite set of locations and $Area$ tracks the busy locations.

An environment associates identifiers with their denotations. In the case of simple scalar variables, the denotations are locations ($Locn$). For arrays, the denotations ($Array$) are sequences of bounds and of locations. This decision is again presumably because finite mappings are not considered to be basic objects.

Other sorts of denotations are discussed below.

4.5.2 Procedure values

See [Mos74, p.14 & C12]

As one would expect, the denotations of functions and procedures are full-blown functions (again somewhat complicated by continuations). Thus, there is no need to add a mechanism for returning parameters, as the denotations of type procedures are functions which return a value (and optionally modify the state, if the procedure has side-effects), or simply modify the state, for non-type procedures.

4.5.3 Handling of jumps

See [Mos74, p.13, 24, C18 & C19]

The story of continuations deserves a separate historical account. Fortunately, this has been provided by Reynolds in [Rey93] who updated this somewhat in his December 2004 talk at the Computer Conservation Society in London.[39] For the current purposes, the joint paper by Wadsworth and Strachey [SW74] is used as the reference point.

Providing a homomorphic model of the goto statement is a key issue in denotational semantics precisely because the content of such a statement is just a label and there is no obvious way in which its meaning is contained in something derived from that content. The idea of the continuations method is to say that the denotation of such a label is the computation that will arise if computation begins at that label. Unfortunately, this means that all of the obvious denotations for statements need to take their potential completions as an extra argument.

These "potential completions" are referred to as continuations and are arguments to (almost) every semantic function. Typically they are referred to with θ in denotational semantics and Mosses' ALGOL description follows suit. The continuation is the denotation of the semantically-following statement and, as that contains its own continuation, the continuation of the first statement in a program contains the denotations of every statement up until the end of the program.

Thus, in a small example taken from [Sto77] (any example taken from the actual ALGOL description would be somewhat larger):

$$\mathcal{L}[\![\Gamma_1;\Gamma_2]\!]\rho\theta = \mathcal{L}[\![\Gamma_1]\!]\rho\{\mathcal{L}[\![\Gamma_2]\!]\theta\}$$

Commands are supplied with continuations simply so they have the option of being ignored should an abnormal termination occur.

[39] Video recordings exist of this event and the earlier one organised in the same way in June 2001.

In the ALGOL description, the denotation of goto statements involves determining whether the label is within the current phrase and using the appropriate auxiliary Hop or $Jump$ function (following Strachey's names for gotos within and outside the current phrase respectively). Both functions alter the continuation to become the meaning of the labelled statement; $Jump$ uses another auxiliary function to modify the environment as appropriate first.

4.5.4 Non-determinism in expression evaluation

See [Mos74, p.C14 & C16]

Because denotations are functional, Mosses' description (like that from Vienna which is discussed in Section 5) cannot handle the non-determinism permitted for expression evaluation in ALGOL within the denotations. Instead, in some places the semantic evaluation functions force a left-to-right order and, in others, the order is simply left unspecified.

4.6 Postscript on Oxford Denotational Semantics

ALGOL offered no way of writing concurrent programs. Any attempt to tackle concurrency would bring with it non-determinism and neither the version of denotational semantics in use in Oxford at the time of Mosses' description nor that used by the Vienna group in the 1970s would have a way of modelling concurrency. One way forward that evolved later was the use of power domains; Bekič also made suggestions before his tragic death [Bek71].

A clear description of denotational semantics is given in [Ten76] which also contains a formal description of Reynolds' Gadanken language.

Mosses has continued to work in the field of formal semantics throughout his career, going on to devise, for example, a form of SOS called "Modular Operational Semantics" [Mos04], and "Action Semantics" ... [Mos05] based on Gul Agha's actor systems for concurrency. Recently, Mosses has been working on a "Funkons" approach [MV14] to mechanising formal semantic descriptions based on defining a small set of "fundamental concepts" and providing mappings from language constructs into these.

5 VDM denotational description

The technical arguments for moving away from 'grand state' semantics were clear in 1970 (see Section 3 above); the evolving 'denotational' ideas were understood; what was needed was the opportunity for the Vienna Lab to tackle a significant language description to try out their own combination of these ideas. This came about in 1973.

5.1 Background: Vienna denotational semantics

Roughly corresponding to the period 1971–72, the Lab had been asked to work on finding automatic ways of detecting potential parallelism in sequential FORTRAN pro-

grams. But, in 1972, IBM began an ambitious plan to design a machine architecture that was radically different from that of the 360 range that had dominated the 1960s. The aim of the 'Future System' (FS) project was to make computers far easier to use and included concepts such as a one-level address space, unforgeable pointers and in-built support for what were essentially procedure calls. Because the project to build FS machines was eventually cancelled, little is published about it but [RS76] gives some hint of the novelty of the ideas that were explored. The Vienna Lab was asked to design a PL/I compiler for FS. Furthermore, there were no constraints put on the design methods to be used. Unsurprisingly the Lab decided that the first task was to write a formal description of the version of PL/I that they were to support: the ECMA/ANSI version of PL/I.

As regards the approach to description, there had been an exchange of letters between Bekič, Lucas and Jones during 1972 that explored how to fit some of their own ideas into a denotational mould.[40] Jones moved back to Vienna on a 'permanent transfer' in early 1973. Much of the technical detail about 'VDM'[41] is covered in [Jon99] but it is worth adding that the task of designing a compiler for a machine whose architecture was both novel and evolving presented considerable challenges.[42]

Overall control of the PL/I for FS project was by Kurt Walk. Initially there were two sub-groups with Viktor Kudielka managing the front-end and Peter Lucas the back-end. When Lucas transferred to IBM Research in Yorktown Heights, Kudielka became manager of the project and Jones became 'Chief Programmer' around April 1974. (By this time, Zemanek had been made an IBM Fellow and Walk was Lab director.) The project occupied most of the 20 or so professional members of the Lab.

In 1974, a full VDM denotational description of the ECMA/ANSI subset of PL/I had been constructed and was printed as a Technical Report [BBHJL74]; a collection of further reports discussing aspects of developing compilers from such descriptions were written (see Section 5.6). The authors listed for TR25.139 are Hans Bekič, Dines Bjørner, Wolfgang Henhapl, Cliff Jones and Peter Lucas (ten further colleagues are acknowledged for contributions including detailed reviews).

On St Valentine's day 1975 the FS machine project was cancelled [Gra06] and it became clear that the next mission of the Vienna Lab would be the development of conventional IBM products. Many of the key researchers began to leave the Lab: Bjørner back to a chair at the Technical University of Denmark, Henhapl to a chair in Darmstadt, Germany and Jones moved to IBM's European System Research Centre in La

[40] Bekič had spent the academic year 1978/9 with Landin at Queen Mary College London and Jones had attended some of Strachey's Oxford lectures in 1971/2.

[41] The name "Vienna Development Method" was actually coined rather late in the project. There is also a certain ambigutiy: to many people, VDM refers to a development method for all forms of computer system (this aspect is placed in a historical context in [Jon03]); in the current paper, VDM is taken to refer specifically to the technique for language description that evolved in the Vienna Lab between 1972 and 1976.

[42] An interesting cautionary tale about formal descriptions relates to that of the FS architecture itself. As indicated, the machine was intended to have one-level store addressing, novel call/return instructions, unforgeable pointers etc. Clearly, to design a compiler, it was necessary to have a clear description of the (evolving) architecture. A small team in the IBM Lab in Poughkeepsie (New York State) wrote a formal description that initially used rather abstract types and implicit definitions. This was not, of course, executable. Management suggested that since this had involved a lot of work (and thus expense) it would be better if it could execute FS instructions. The team laboured to achieve this and then to respond to a subsequent request that it should be optimised to run at a more acceptable speed. At the end of this process, the description was of little use to the Vienna Lab as a basis for understanding the machine and Hans Bekič had to write a short formal description to guide the code generation work.

Hulpe, Belgium.

Papers are often cited more than they are read ([Flo67] is almost certainly an example); technical reports are perhaps more read than cited. Certainly [BBHJL74] has had more influence than its relatively low citation count would suggest. After the cancellation of FS and thus the PL/I compiler project, Bjørner and Jones agreed to try to preserve and promulgate the VDM denotational style by cajoling their former colleagues to contribute to [BJ78] which includes the description [HJ78] of ALGOL that is the subject of this section. The Table of Contents of [BJ78] is reproduced in Figure 4.[43]

So, here again, the description of ALGOL followed that of the larger PL/I language; the simpler task being undertaken so as to illustrate the method on a language whose description would fit in a chapter of a book.

Just as Mosses in [Mos74] provides a slightly backhanded acknowledgement to [ACJ72], [HJ78] has in its acknowledgement:

> Returning the compliment to Peter Mosses, one of the authors would like to acknowledge that a part of the incentive to write this definition was the hope to provide an equally abstract but more readable definition than that in [Mos74].

5.2 Extent of ALGOL described

The authors of [HJ78] claim to cover all of ALGOL as given in [MHW76] that cleared up obscurities in the [Revised Report]. In particular, the VDM description does handle "own" variables, input/output and the so-called "standard functions" (See Section 1.3). A few comments are offered in the introduction to [HJ78] that suggest yet further improvements to ALGOL itself.

As made clear in the introduction, non-determinism in expression evaluation is not described. It is stated:

> As has been discussed elsewhere in this volume, the definition of arbitrary order of evaluation has not been addressed . . .

5.3 Syntactic issues

5.3.1 Concrete vs. abstract syntax

The semantic description is based on an abstract syntax; some comments on the translation from concrete to abstract syntax are given but not a full description of the process. The movement away from the purely object-based view of the world in the classic VDL style (see Section 2.3) that is seen with the inclusion of sets as first-class objects in [ACJ72] had by this time developed into the rich VDM notation. Now, there were

[43] There is a coda to this story. At that time, Springer Verlag appeared to take the attitude that once an LNCS volume had sold its initial print run that their task was complete. When they declined to reprint LNCS 61, Tony Hoare came to the rescue and offered to have a suitably updated collection of papers reprinted in his prestigious 'red and white' Prentice-Hall series: [BJ82] contains among other contributions a revised description of ALGOL [HJ82]. The revision differs mainly in the order of presentation as discussed in Section 5.6 below.

CONTENTS

Introduction V
Acknowledgements XVII
Addresses of All Authors XIX

ON THE FORMALIZATION OF PROGRAMMING LANGUAGES:
EARLY HISTORY AND MAIN APPROACHES 1
Peter Lucas

PROGRAMMING IN THE META-LANGUAGE: A TUTORIAL 24
Dines Bjørner

THE META-LANGUAGE: A REFERENCE MANUAL 218
Cliff B.Jones

DENOTATIONAL SEMANTICS OF GOTO: AN EXIT FORMULATION
AND ITS RELATION TO CONTINUATIONS 278
Cliff B.Jones

A FORMAL DEFINITION OF ALGOL 60 AS DESCRIBED
IN THE 1975 MODIFIED REPORT 305
Wolfgang Henhapl & Cliff B.Jones

SOFTWARE ABSTRACTION PRINCIPLES:
-- Tutorial Examples of: An Operating System
Command Language Specification, and a PL/I-like
On-Condition Language Definition 337
Dines Bjørner

References & Bibliography 375

All papers lists their CONTENTS at their very beginning.

Figure 4: Copy of the Table of Contents of [BJ78]

a number of different types which were all equally fundamental and linked by a series of known operators. VDM includes sets, sequences, maps and records as basic types, which allows sophisticated abstract constructs to be described rather succinctly. Each type comes with a set of functions to construct, select and transform them. These associated functions are implicitly included in the definition of the type, in contrast with the explicit use of constructors and selectors in McCarthy's style or the universal construction and modification operator seen in VDL.

For example, the abstract syntax for prefix operators is as follows:

$$
\begin{array}{lll}
\mathit{Prefixexpr} :: & \mathit{s\text{-}opr} & : \mathit{Prefixopr} \\
 & \mathit{s\text{-}op} & : \mathit{Expr}
\end{array}
$$

Objects of this type can then be constructed with the function $\mathit{mk\text{-}Prefixexpr}(a, b)$, and the operand can be selected with $\mathit{s\text{-}op}(E)$. The real power comes in the equivalence of a mk- expression with an object constructed in this way, which allows the easy naming of components in a function.

5.3.2 Handling context dependencies

In common with other VDM descriptions (particularly [BBHJL74]), as many meaningless programs as possible are eliminated by defining 'context conditions': a family of predicates $\mathit{is\text{-}wf\text{-}}\theta$ for each syntactic class θ that determines if it makes sense with respect to the declared types of variables. In ALGOL, these checks cannot be totally static because of array parameter bounds and procedure parameters. The PL/I Subset definition cited above appears to be the first published used of a completely formalised static error checking system.

As an example, the predicate for prefix operators checks that for expressions prefixed with `NOT` the type of the expression is Boolean and for other prefix operators, the type is arithmetic.

5.4 Overall semantic style

Vienna had moved completely to a denotational approach to semantics but the appearance of their descriptions differs greatly from those from Oxford denotational descriptions. One reason for this is not of any depth: faced with a large language like PL/I, it was completely clear that single (Greek) letters would not be useful for the names of either functions or their parameters.This decision is however about the surface appearance and does not signify a difference in approach to semantics.

Much the most significant difference between Oxford and Vienna denotational descriptions can be termed "exits versus continuations". Section 4.4 explains how continuations are used to model exceptional sequencing such as is required by goto statements. The Vienna group chose to pick up the exit idea described in Section 3 as a simpler mechanism for describing exceptional termination of phrase structures.

For languages without exceptional sequencing such as goto statements, functions from states to states ($\Sigma \to \Sigma$) can be used for the space of denotations. The denotation of the sequential composition of statements in the object language is mapped into the composition of the denotations of the separate constructs; fixed points can be used to

define (homomorphically) the denotation of repetition in terms of the denotation of the body of the repetition.

The exit idea is used in a denotational setting by making the basic denotations functions from states to pairs of states and an optional abnormal component ($\Sigma \to \Sigma \times \left[Abn\right]$). The denotation of say $s1; s2$ is now derived in a slightly more complicated way from their separate denotations:

- when the abnormal part of the pair for the denotation of $s1$ is **nil** the denotation of composition passes the state part of the denotation of $s1$ to the denotation of $s2$.
- if however the abnormal part of the denotation of $s1$ is non-**nil**, the pair from $s1$ is the result of the composition of $s1; s2$ — thus effectively ignoring $s2$

This slightly complicated form of composition is made readable in semantic descriptions by defining a "combinator" whose representation was chosen to be a semicolon. Had the Vienna group tried to emulate the compactness of the Oxford descriptions, they could have written something like:[44]

$M[\![s1; s2]\!] \triangleq M[\![s1]\!]; M[\![s2]\!]$

In fact, rather more readable names (e.g. $i\text{-}stmt$, $i\text{-}block$) were used for semantic functions in [HJ78] but the essential point is the use of exits and making them palatable by defining appropriate combinators.

The denotation of a goto statement makes no change to the state but defines an abnormal value which is defined using an $exit$ combinator. The propagation of abnormal values has to be caught somewhere and this requires one more combinator for which the name was chosen by writing "exit" backwards ($tixe$). Further details of how the exit concept was used in modelling ALGOL are given in Section 5.5.3 below.

One of the Vienna reservations about continuations is that they are too powerful for the task of modelling exceptional exits from phrases of an object language; so it is not claimed that exits and continuations are equivalent. It is however possible to show that, for a language similar to ALGOL, an exit model gives the same semantics as one using continuations; such a proof is given in [Jon78] and this is one of the chapters that was significantly revised in the 1982 volume giving [Jon82]. The proofs are interesting because they tease apart different aspects of how labels are modelled.

One last observation is worth making about combinators and that is that it is possible to read them operationally: although the semicolon above is defined as a combinator of functions, it can be interpreted as an operational definition that first performs the computation before the semicolon followed by that after it.

Peter Mosses in [Mos11] points out that the use of combinators in VDM is similar to the later development of Moggi's "monads" [Mog89]. The use of combinators also makes denotational descriptions in VDM look different from those written in Oxford where arguments (with very short names) are passed to Curried functions.

The overall semantic function of this approach is very similar to that of the Oxford approach: the signature can be seen as $AP \to (\Sigma \to \Sigma \times \left[Abn\right])$. As in the Oxford style, the denotations are formed from λ-expressions on Σ. See Figure 9 in Section 6.1.1.

[44]This is close to the style of [HJ82] but [HJ78] used a more long-winded notation.

5.5 Specific points

As mentioned above, the discussion here revolves around [HJ78]; the basic model in [HJ82] is the same but the description in the latter paper is organised by language construct (as in the Functional description; see 3.4) rather than collecting all of the abstract syntax for all constructs following that with all of the context conditions and finishing with all of the semantic descriptions. However, direct pointers in this section are made to [HJ82] as it is probably easier for the reader to obtain.

5.5.1 Environment/State

See [HJ82, §6.0].

As is normal in small state descriptions, there is a clear separation between environments and states. In the simplest case, the Env maps identifiers corresponding to scalar identifiers to internally generated scalar locations (Sc-loc) and the $Storage$ maps scalar locations to scalar values which are values of the elementary types (Booleans, integers and reals). The model of arrays is straightforward: a dense mapping from indices to scalar locations.

Statements can change the store but their denotations depend on environments which are not then shown as results. This makes immediately apparent the property that the environment of $s2$ in $[\![s1; s2]\!]$ is identical with that of $s1$; this property required a non-trivial proof of a lemma in grand state descriptions. Unfortunately, for any language that allows side effects (including ALGOL), expression evaluation can also change the state.

Following this line of what can be changed on statement evaluation, the overall state (Σ) has to contain the current values of every $Channel$ for the model of ALGOL's input/output statements.

5.5.2 "Own" variables

See [HJ82, §6.0.4].

As mentioned above, "own" variables are handled by having a separate mapping from their identifiers to additional unique locations. This is held in a separate environment component named own-env which is only used in the denotations of "own" variables. Furthermore, internal unique names are generated to avoid name clashes. As discussed in [HJ78, p 307], this model is given in detail because the topic of own variables had been a subject of controversy.

5.5.3 Handling of jumps

See [HJ82, §6.4.4 & 6.1.1].

The overall idea of the exit mechanism is explained above in Section 5.4. Denotations of labels obviously contain the label identifier but, to make them unique, an "activation identifier" is appended. The semantics of a goto statement is then simple: evaluate the denotation corresponding to the label expression (if any) and perform an $exit$. As each

phrase structure is closed, a $tixe$ operation catches an abnormal part when present and uses $Mcue$ functions to determine the correct place to resume giving meaning to the program.

5.5.4 Procedure denotations

See [HJ82, §6.2.2].

As one would expect from a denotational description, procedures are denoted by functions that are ultimately of type $\Sigma \to \Sigma \times \left[Sc\text{-}val\right]$. They are Curried to require the denotations of the actual parameters (arguments) and a set of activation identifiers (see Section 5.5.3). The $Sc\text{-}val$ is present in the case of functions and **nil** in the case of procedures.

5.6 Postscript on VDM

As in any functional or denotational semantics, non-determinism cannot be handled. This means that this description, in common with those in Sections 3, 4 and 5, fails to describe the option to evaluate expressions in arbitrary order.

The differences between the models in [HJ78] and [HJ82] are minor; the main organisational difference is in the order of presentation. In the earlier paper, all the abstract syntax is grouped together, followed by the context conditions and then finally the semantics; in the later paper, these are grouped by language construct.

The description in [HJ82] employs constructor functions in parameters to overloaded function names. This pattern matching idea makes the description easier to read.

Connected with the PL/I for FS compiler project, other than formal description of PL/I itself [BBHJL74], a number of other technical reports (e.g. [Wei75; Izb75; BIJW75; Jon76]) describe aspects of compiler development from VDM language descriptions.

There are a number of language descriptions in the same VDM style including:

- PRTV (essentially SQL) [Han76].
- Database programming languages [Wel82; Wel84].
- Smalltalk [Wol88]
- In the description of Pascal in the same style [AH82], an interesting issue that significantly complicates the model of Pascal is the modelling of so-called variant records. This was a feature of Wirth's Pascal language that supported unions in a type description. Tagged variant records contained information recording the option and these are fairly easy to model. There is the possibility, however, of having untagged variant records. Furthermore, variant records can be passed as parameters. The amount of extra checking that has to be put into the formal model to distinguish incorrect handling of untagged variant records is considerable.
- The Modula-II standard (and for once, this is really the defining document) is given in [AGLP88].

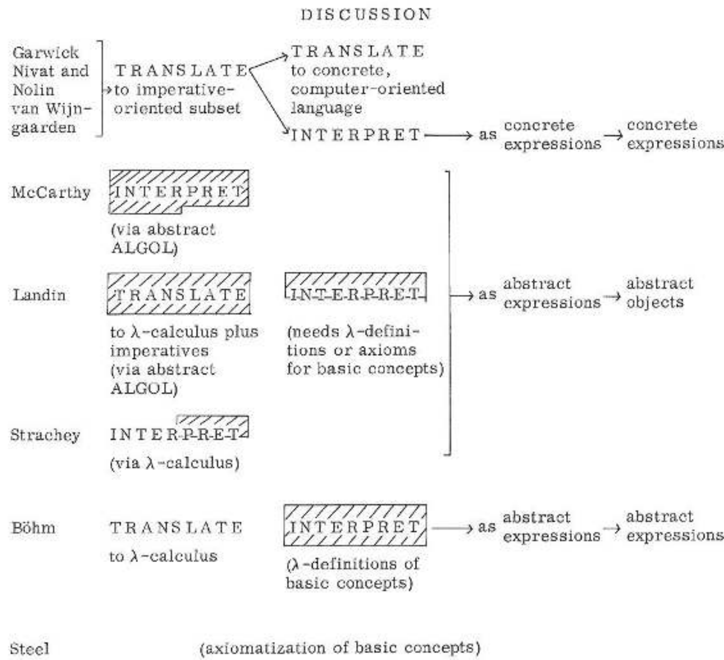


Figure 5: Landin's figure ([FLDL, p.290]) "categorization chart"

- The PL/I standard [ANS76] uses the concepts of a VDM model but makes the unfortunate choice to present all but the abstract syntax and state in words rather than formulae; furthermore the authors took the position that while sequences would be familiar to their readers, sets might be too abstract. Mathematically literate readers are faced with having to scan the whole description to ascertain whether the order of elements in a sequence actually influences the semantics and reading hundreds of pages of "English" that tries, but fails, to be as precise as conventional function notation.

6 Conclusions

Sections 6.1–6.4 offer a more explicit comparison between distinctions made in the major Sections 2–5 above; Section 6.5 reviews the (limited) tool support used in the creation of the ALGOL descriptions; Section 6.6 lists some other significant formal language descriptions in the model-oriented camp; finally, Section 6.7 briefly discusses property-oriented descriptions.

6.1 Operational vs. denotational

The general inclusion of the discussions in [FLDL] has been praised above. It is however worth drawing attention to one specific figure: Landin prepared a "categorization

chart" for the final discussion and this is printed in the proceedings and reproduced here as Figure 5. The categorisation choices made here do not necessarily reflect the views of the current authors (in particular, it seems that Strachey is not at all 'interpreting' in the same way as McCarthy or Landin) but it is included as an interesting comparison. In particular, it is clear that there is a two-step process in Landin's work, one half of which (the translation) goes on to influence denotational semantics and the other half (interpreting) operational semantics.

An obvious distinction between the four ALGOL descriptions in the current paper is that Lauer (Section 2) and Allen et al. (in Section 3) use an operational semantics approach whereas Mosses (Section 4) and Henhapl et al. (in Section 5) are denotational. It is however worth adding that the move to a small state semantics makes a radical difference to both the readability and tractability of a semantic description. Lauer's VDL description followed nearly all of the decisions that had been made in the VDL descriptions of PL/I (ULD-III versions I–III). In particular, almost anything which could affect the computation was placed in the grand state. As a consequence, it is unclear when such items can be changed in the evolution of the state. In contrast, small state descriptions attempt to show things such as environments (mapping identifiers to their locations) as arguments to the semantic description and make the major transitions from stores (mapping locations to values) to stores. Allen et al., as outlined in Section 3, is a small state description. This observation is important if one considers how a "Structural Operational Semantics" (SOS) [Plo81] (reprinted as [Plo04a]) of ALGOL would be presented.

One reason for raising the issue of SOS is that the denotational approach does inflict some rather heavy foundational lifting on both writer and reader. The load becomes particularly heavy for languages that allow concurrency. Plotkin had published the fundamental contribution that proposed power domains as a model for concurrency [Plo76] but made the decision to teach an operational approach in his Aarhus course in 1981. His reflections [Plo04b] that accompany the republication [Plo04a] of his Aarhus lecture notes [Plo81] offer a useful perspective.

He writes:[45]

> I remember attending a seminar at Edinburgh where the intricacies of their PL/I abstract machine were explained. The states of these machines are tuples of various kinds of complex trees and there is also a stack of environments; the transition rules involve much tree traversal to access syntactical control points, handle jumps, and to manage concurrency. I recall not much liking this way of doing operational semantics. It seemed far too complex, burying essential semantical ideas in masses of detail; further, the machine states were too big.

Advocates of denotational semantics also make much of the rule that the mapping from syntax to semantic objects should be homomorphic in the sense that the denotation of a composite object should be some function of the semantics of its components. It has been seen above that this rule can be problematic with constructs such as goto

[45] Interestingly, also in [Plo04b] he writes: "I recall Dana was sceptical regarding the latter point and, in that connection, he asked a good question: why call it operational semantics? What is operational about it? It would be interesting to know the origins of the term 'operational semantics'; an early use is in a paper of Dana's [...] written in the context of discussions with Christopher Strachey where they came up with the denotational/operational distinction. The Vienna group did discuss operations in their publications, meaning the operations of the abstract interpreting machine, but do not seem to have used the term itself."

statements. It is also worth observing that a guideline suggesting that there should be one SOS rule per composite object has a similar effect.[46] Of course, the difficult cases such as abnormal termination remain. Furthermore, it is often useful to make case distinctions in SOS by providing different hypotheses (e.g. the two cases for the evaluation of the conditional expression in an if statement). Finally, SOS does cope naturally with non-determinacy by moving from functions to relations and this can result in the hypotheses of multiple rules matching a given configuration. Despite these caveats, much of the structural clarity of the homomorphic rule can be preserved in SOS description.

6.1.1 ADJ diagrams

The differences between the various approaches can be presented by moving beyond simple signatures and employing a more graphic presentation. ADJ diagrams[47] are a way to picture the signatures of functions by drawing arcs between their domain and range sets. They are used in this paper because their comparison emphasises the differences in the approaches better than mere signatures. Meaning is given typically to abstract programs (AP) although in the Oxford style concrete programs are used. The diagram in Figure 6 simply indicates that abstract programs are derived from actual programs, which are strings of tokens, in some usually-unspecified way which may be thought of as a translator.

The Greek letter Σ is used in the remaining figures to indicate a *store*, a smaller state which is essentially little more than a mapping between identifiers (or their surrogates such as locations) and values. Similarly, Ξ is used in Figure 8 for classic VDL and is a much larger state with more components (some of which do not change as frequently as the store). More information is given in the 'Overall semantic style' in each description's section.

6.2 Modelling decisions

All three of the descriptions in Sections 2, 3 and 5 base the descriptions on an abstract syntax; the description in Section 4 presents the semantic rules being applied to concrete ALGOL phrases. That having been said, Mosses achieves some of the advantages of an abstract syntax by reducing the grammar for ALGOL to a (highly ambiguous) normal form.

It is also worth noting that the notion of abstraction in an abstract syntax is not absolute. For example, an abstract syntax in VDM might well represent integer constants as $\mathbb{N}$; perhaps more questionably, floating point numbers might be shown as $\mathbb{R}$. In either

[46]Again from [Plo04b]: "A realisation struck me around then. I, and others, were writing papers on denotational semantics, proving adequacy relative to an operational semantics. But the rule-based operational semantics was both simple and given by elementary mathematical means. So why not consider dropping denotational semantics and, once again, take operational semantics seriously as a specification method for the semantics of programming languages?"

And again: "The second idea was that the rules should be syntax-directed; this is reflected in the title of the Aarhus notes: the operational semantics is structural, not, as some took it, structured. In denotational semantics one follows an ideal of compositionality, where the meaning of a compound phrase is given as a function of the meaning of its parts. "

[47]Named after the ADJ group (Thatcher, Wagner and Wright) who first used them.

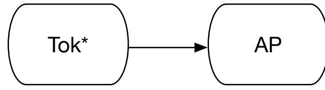


Figure 6: Syntax translator

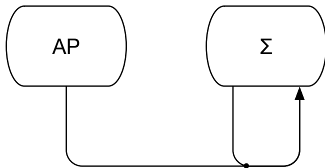


Figure 7: McCarthy-style simple operational semantics

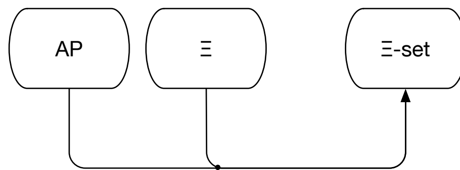


Figure 8: Classic VDL operational semantics

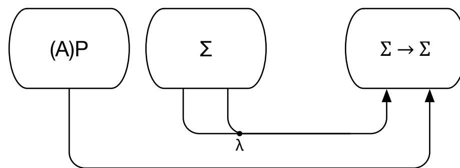


Figure 9: Denotational semantics

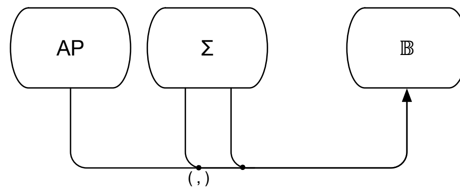


Figure 10: Structured operational semantics

case, there is of course a translation problem from sequences of digits. Although this translation is non-trivial in the case of floating point numbers, it is possible to argue that it is a problem that is usefully separated from the main semantic description.

The use of static checking of context dependencies, as hinted at in [ACJ72] and fully exploited in [HJ78], provides a significant advantage in the process of giving semantic meaning to programs by limiting the set of texts for which it is worth trying to give meaning. Separating out the context conditions from semantics facilitates shorter, easier to read semantic functions and groups together error checks that compiler-writers should emulate.

The denotational descriptions of ALGOL from Oxford and Vienna take different approaches to modelling goto constructs: Mosses uses the continuation concept most of whose "discoveries" (see [Rey93]) have an Oxford connection; the "exit" approach originated in the Vienna Lab and is deployed in [HJ78]. This can be viewed as a modelling decision because either route fits with the overall denotational approach. Indeed, [Jon78] establishes the equivalence of the approaches on a language fragment that presents the essential challenges of ALGOL.

"Own" variables were always a contentious subject in ALGOL and two of the descriptions decline to cover them at all. It is quite clear that doing so removes a good deal of complication from the modelling process: in [Lau68] a pass of the entire program is required and in [HJ78] an entirely separate environment is created.

6.3 Fundamental objects

An interesting dimension for comparison of semantic approaches is in their choice of fundamental abstract objects. The evolution of richness in Vienna semantics can be seen starting from their inspiration with McCarthy, through the use of purely objects in VDL, the addition of sets in [ACJ72] and finally the rich collection of basic types in VDM. The Oxford story is different, as no abstract syntax is used and the whole issue is somewhat side-stepped.[48] McCarthy's approach to abstract syntax uses explicitly-defined constructor and selector functions (see Section 1.5 for some examples), with predicates describing language constructs as the basic types of the metalanguage.

In classic VDL, as seen in Section 2, this concept is expanded somewhat and used along with the Vienna concept of objects. All fundamental blocks in the VDL style are such objects and they come with selector functions implicit in the construction of composite objects. There are explicitly-defined construction and modification functions which operate over these objects.

[ACJ72] maintains the objects focus but the addition of sets as basic components adds an extra layer of richness to the notation. This also brings requisite non-determinacy in selectors, which is co-opted in the non-deterministic expression evaluation.

The Oxford focus is essentially on functions, which are organised in the mathematically complex lattices and retracts. The use of these objects allows the use of a number of their useful properties in proofs, but brings complications in the combining of types.

During the development of VDM, one suggestion for improvement of the VDL method by Jones concerned abstract objects. In contrast to McCarthy's explicit relation be-

[48]But this issue actually goes beyond abstract syntax.

tween constructor functions, predicates and selectors, the Vienna group henceforth took a definition such as

$$X :: a : TypeA \\ b : TypeB$$

to define implicitly the constructor and selector functions:

$mk\text{-}X$: $TypeA \times TypeB \rightarrow X$
a: $X \rightarrow TypeA$
b: $X \rightarrow TypeB$

and $x \in X$ could only be true if x was built with $mk\text{-}X$. This proposal is contained in [Jon69, §1].

One useful direct comparison that can be made is in the treatment of maps: constructions associating keys with values. These form the central part of the store or state of most semantics descriptions, those in this paper included, as they associate variables with their current values.

The VDL approach is to use sets of simple pair objects. An example is in the denotation directory, which is defined as follows:

$$is\text{-}dn = (\{< n: is\text{-}den >|| is\text{-}n(n)\})$$

Thus a set is built up of simple composite objects comprising a selector with an elementary name pointing to a denotation part. Selection uses a simple application-like syntax $n(dn)$ which returns the object where n corresponds to the selector part. This is not fully defined anywhere in the description nor its associated method and notation guide [ULD-IIIvII-Meth]. In fact, rather heavy weather is made of the concept of "abstract objects" in [Zem68; Oll71]. A comparison with a modern (e.g. VDM) view of records with constructors, selectors and predicates is however somewhat unfair because in VDL so much of the work had to be done using "composite selectors" to locate things in trees and to prune those trees.

In the VDL Functional description, the basic idea is somewhat similar: a set of pairs is used.

$$is\text{-}dn = (\{< is\text{-}id, is\text{-}den >\})$$

However, rather than composite objects in the VDL style, they are more akin to tuples in classical mathematics. Selection from these is performed implicitly with auxiliary functions such as *firsts*:

$$firsts(pr\text{-}set) \quad \triangleq \quad \{ob\text{-}1 \mid < ob\text{-}l, ob\text{-}2 > \in pr\text{-}set\}$$

In the Oxford denotational description, Mosses skips the issue entirely, simply stating:

```
Map      (associating locations with values)
```

In the commentary, he does note that a model for storage could be formulated with $Map = N \rightarrow V$ (where N is the integer domain and V the domain of ALGOL-allowable values). This would then presumably be a partial function allowing the selection of values by the passing of identifiers.

The VDM approach allows maps as powerful fundamental objects in their own right. The environment thus is:

$ENV = Id \xrightarrow{m} DEN$

This comes implicitly with application for selection, thus $ENV(x)$ would return the DEN associated with x, as well as the auxiliary functions **dom** and **rng** returning sets of the domain and range of the map respectively. Thus, **dom** ENV would be of type Id-**set**, and **rng** ENV would be DEN-**set**.

6.4 Superficial differences

An obvious superficial difference between the descriptions in Sections 4 and 5 is the notation style. In the former, the semantics of ALGOL assignments begins:

```
case"Var := AssL":
    let χ = Main(𝒥var⟦Var⟧ρ) in 𝒜⟦t⟧ρχ⟨⟩ ∥ θ
```

in the latter:

```
i-assign-stmt(mk-assign-stmt(dl,e),<,env,cas>) =
   let dl:<e-left-part(s-tg(dl(i)),<env,cas>)|1≤i≤lendl>;
   let v:  e-expr(e,<env,cas>);
   for i=1 to lendl do
       (let vc:conv(v,s-tp(dl(i)));
        assign(vc,dl(i)))
```

The shorter identifiers and function names in the Oxford style make for a more compact semantics, but the use of single (often Greek) letters can make it harder to follow quickly.

6.5 Tool support

Each of the ALGOL descriptions contain a significant number of formulae. In none of the four cases were these subjected to significant checking by tools that today might be thought of as standard. The preparation of the early VDL descriptions[49] used a system called “Formula/360” [SZ66] that was driven from a concrete syntax and thus checked for simple errors. It also had a simple but extremely useful formatting algorithm that makes line breaks in long formulae by cutting at the highest point in the parse tree.

The one description that could have been processed by a tool that did more than syntax checking is that from Mosses who was at that time working on his Semantics Implementation System [Mos75a]. SIS would have not only type checked the description

[49]The decision not to record the types of the semantic functions makes checking VDL definitions more tedious that it needed to be.

but could also have provided a prototype implementation. Mosses however informed the current authors[50] that his description of ALGOL was never processed by SIS.

6.6 Other significant formal descriptions of semantics

This section maintains the emphasis on procedural programming languages. In particular, the authors of this paper are aware that they have omitted mention of extensive work on the semantics of process algebras.

Other descriptions of ALGOL include:

- Landin's [Lan66] (remember: presented in 1964) is an introduction to his later pair of papers [Lan65a; Lan65b] which present a correspondence between ALGOL and lambda notation. This is achieved by way of an abstract object language into which both a lambda-based model and ALGOL are mapped to 'applicative expressions'. The interpretation of these AEs is given by a machine referred to as the Stack-Environment-Control-Dump machine. This machine is cited in [LW69] as an inspiration for the state of VDL.
- [Bak65]
- Rod Burstall's [Bur70] "set of sentences in first order logic" describes a major part[51] of ALGOL 60. Burstall acknowledges the (largely program verification) work of Floyd, Hoare, Manna and others and generalises this approach to giving the semantics of whole programming languages. The method is to describe the rules that translate ALGOL commands into these sentences. One advantage of this method, the author claims, is that the resulting sentences can be fed into theorem provers and thus be used both to debug programs written in the language more easily and indeed even to debug the language itself.

In addition to the semantic descriptions listed in "Postscript" sections (2.6, 3.6, 4.6, 5.6) and without wishing to claim that they are the earliest or most important, other significant early formal descriptions of programming languages include:[52]

- CLU [Sch78]
- CHILL semantic description [HB80] 1980 denotational style in VDM with exit mechanism and parallel processes modelled in abstract formulae. Context conditions also used.
- Veronique Donzeau-Gouge's semantics [DGKL80; INR80] of the sequential parts

[50] Personal communication June 2016.

[51] The description omits call by name, procedures and arrays as parameters, own variables and switches.

[52] The authors would be grateful to hear of other early semantic descriptions that are not listed here.

of Ada[53] [54]

- Ada in VDM but including concurrency by using SMoLCS [BO80]
- SML [MTH90]
- Actors [HBS73] and their semantics [AMST97]
- SPARK Ada was given a formal definition [O'N94] because of the intention to use the language in the development of safety critical systems.
- COLD-K [FJ87]

6.7 Property-oriented descriptions

The ALGOL descriptions discussed above can all be viewed as being based on a model. In particular, whether operational or denotational, the concept of a state or store is central to fixing the semantics. In contrast, it is possible to attempt to fix the semantics of a language by defining properties without such an explicit notion of state.

Much the better known property-oriented approach is to give ways of reasoning about assertions that should hold of a program. Furthermore, Hoare's "axiomatic basis", [Hoa69] the most widely cited reference on reasoning about valid assertions,[55] specifically mentions the technique's suitability for defining the semantics of programming languages.[56]

Hoare and Wirth claim to provide a description of Pascal in the style in [HW73] but it is difficult to understand how this copes with parameter passing "by variable" (in other words "by location"); a clearer way forward with this feature might be [AB77]. The only language designed around an axiomatic description is "Turing" [HMRC88].

As Bertrand Russell observes, there are dangers of writing inconsistent postulates

[53] There was a series of requirements (Woodman, Tinman, Ironman) from the DoD for language proposals. The Tinman version contains the following text (VI.C.2) for **Unambiguous Definition**:

> A complete and unambiguous definition of a common language is essential. Otherwise, each translator will resolve the ambiguities and fill in the gaps in its own unique way. There are currently a variety of methods for formal specification of programming language semantics, but it remains a major effort to produce a rigorous formal description, and the resulting products are of questionable practical value. The real value in attempting a formal definition is that it reveals incomplete and ambiguous specification. An attempt will be made to provide a formal definition of any language selected, but success in that effort should not be requisite to its selection. Formal specification of the language might take the form of an axiomatic definition, use of the Vienna Definition Language, definition by an interpreter (à la Lisp), or use of some other formal semantic system.

[54] The story of Ada and the formal semantics from INRIA is itself interesting. In the Foreword to [INR80], it is stated that the formal definition using denotational semantics had a deep influence on the language. From his dismissive comments at the PhD jury of Donzeau-Gouge, it was obvious that Jean Ichbiah who was the leader of the team that developed the chosen contender (Green) was far less positive about formal semantics.

[55] Extensive historical notes on program verification are given in [Jon03].

[56] Hoare, in an interview recorded in 2015 for the ACM Turing Award series, stated "Well, I had the idea that it would be a good idea to define programming languages in a way that didnt say too much about what the computer actually did, because in those days anyway all computers were doing things slightly differently, but gave enough information to the user of a programming language to be able to predict whether the computer would do what the programmer wanted it to do."

> what we want has many advantages; they are the same as the advantages
> of theft over honest toil

Two papers that establish the consistency of axiomatic descriptions with respect to model-oriented descriptions are [Lau71; Don76].

Another model-oriented approach to semantic language description is to define equivalences over texts of a language. This idea was considered very early in the history of semantics. Bekič calls this "defining a language in its own terms" in [Bek64]; Mike Patterson's PhD thesis [Pat67] examines the power of program schemas[57] (see also [LPP70]).

The idea of pinning down the semantics of a language by defining equivalences over the language has been given new life by research on "(Concurrent) Kleene Algebras". In, for example, [Hoa69; HMSW09; HMSW11], equivalences are shown to fit Kleene Algebras. Hoare in his video for the ACM Turing Laureate series goes so far as to suggest that this is a more profound link than his axiomatic semantics and in recent seminars has shown how axiomatic, operational and denotational views can all be derived from this algebraic view.

Acknowledgements

The authors have benefited from discussions with Peter Mosses, David Beech and John Tucker; they are also grateful to Martin Campbell-Kelly for detailed comments on a draft of this report. The second author would particularly like to thank the audience of the *History and Philosophy of Programming 3* symposium in Paris in June 2016 for their useful questions and feedback after presentation of this work. The first author began the detailed analysis of the ALGOL descriptions whilst collaborating with Peter Mosses on his PLanCompS project. Thanks are also due to the Bodleian Library in Oxford for their archiving and curating of the Strachey papers and granting access to the same.

The authors would also like to acknowledge the financial support of EPSRC, who are both providing the PhD funding for the slightly earlier-career member of the authorship team and whose Strata Platform Grant is funding some of the travel expenses for both authors.

[57] There was an interesting exchange between McCarthy and Patterson at the 1969 MTOC conference in IBM's Yorktown Heights Lab. Essentially, McCarthy was completely won over by the ideas presented by Zohar Manna and could see little virtue in the program schema approach. Patterson gave a robust reply.

References

[AB77] Krzysztof R Apt and Jacobus Willem de Bakker. "Semantics and proof theory of PASCAL procedures". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 1977, pp. 30–44.

[AB87] Bijan Arbab and Daniel M Berry. "Operational and denotational semantics of Prolog". In: *The Journal of Logic Programming* 4.4 (1987), pp. 309–329.

[ACJ72] C. D. Allen, D. N. Chapman, and Cliff B. Jones. *A Formal Definition of ALGOL 60*. Tech. rep. 12.105. IBM Laboratory Hursley, 1972. URL: `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/TR12.105.pdf`.

[AGLP88] D.J. Andrews, A. Garg, S.P.A. Lau, and J.R. Pitchers. "The formal definition of Modula-2 and its associated interpreter". In: *VDM '88 VDM — The Way Ahead*. Ed. by Robin E. Bloomfield, Lynn S. Marshall, and Roger B. Jones. Vol. 328. Lecture Notes in Computer Science. Springer, 1988, pp. 167–177.

[AH82] Derek Andrews and Wolfgang Henhapl. "Pascal". In: *Formal Specification and Software Development*. Ed. by Dines Bjørner and Cliff B. Jones. Prentice Hall International, 1982. Chap. 6, pp. 175–252. ISBN: 0-13-329003-4. URL: `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/BjornerJones1982/Chapter-7.pdf`.

[ALGOL Bulletin] Brian Wichmann. *The ALGOL Bulletin*. 2004. URL: `http://archive.computerhistory.org/resources/text/algol/algol_bulletin/`.

[ALGOL Report] John W. Backus, Friedrich L. Bauer, Julien. Green, C Katz, John McCarthy, Peter Naur, Alan J. Perlis, H Rutishauser, K Samelson, B Vauquois, et al. "Report on the algorithmic language ALGOL 60". In: *Numerische Mathematik* 2.1 (1960), pp. 106–136.

[AMST97] Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. "A foundation for actor computation". In: *Journal of Functional Programming* 7.01 (1997), pp. 1–72.

[ANS76] ANSI. *Programming Language PL/I*. Tech. rep. X3.53-1976. American National Standard, 1976.

[Bak65] J. W. de Bakker. *Formal Definition of Algorithmic Languages, with an application to the definition of ALGOL 60*. Tech. rep. MR–74. Stichting Mathematisch Centrum, 1965.

[Ban65] Kurt Bandat. *Tentative Steps towards a Formal Description of PL/I*. Tech. rep. 25.056. IBM Laboratory Vienna, 1965.

[BBHJL74] Hans Bekič, Dines Bjørner, Wolfgang Henhapl, Cliff B. Jones, and Peter Lucas. *A Formal Definition of a PL/I Subset*. Tech. rep. 25.139. IBM Laboratory Vienna, 1974. URL: `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/TR25139/`.

[BBHS63] D. W. Barron, J. N. Buxton, D. F. Hartley, and Christopher Strachey. "The Main Features of CPL". In: *Computer Journal* 6 (1963), pp. 134–143.

[Bek64] Hans Bekič. *Defining a Language in its own terms*. Tech. rep. 25.3.016. IBM Laboratory Vienna, 1964.

[Bek71] Hans Bekič. *Towards a Mathematical Theory of Processes*. Tech. rep. TR 25.125. see also [BJ84]. IBM Lab. Vienna, 1971.

[BIJW75] Hans Bekič, H. Izbicki, Cliff B. Jones, and F. Weissenböck. *Some Experiments with Using a Formal Language Definition in Compiler Development*. Tech. rep. LN 25.3.107. IBM Laboratory, Vienna, 1975.

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

[BJ82] Dines Bjørner and Cliff B. Jones, eds. *Formal Specification and Software Development*. Prentice Hall International, 1982. ISBN: 0-13-329003-4. URL: `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/BjornerJones1982`.

[BJ84] Hans Bekič and Cliff B. Jones, eds. *Programming Languages and Their Definition: Selected Papers of H. Bekič*. Vol. 177. LNCS. Springer-Verlag, 1984.

[BO80] Dines Bjørner and Ole Nybye Oest. *Towards a formal description of Ada*. Vol. 98. LNCS. Springer, 1980.

[BS69] J. W. de Bakker and Dana Scott. "A Theory of Programs". Manuscript notes for IBM Seminar, Vienna. 1969.

[Bur70] Rod M. Burstall. "Formal description of program structure and semantics in first-order logic". In: *Machine Intelligence* 5 (1970), pp. 79–98.

[BW71] Hans Bekič and Kurt Walk. "Formalization of storage properties". In: *Symposium on Semantics of Algorithmic Languages*. Ed. by E. Engeler. Vol. 188. Lecture Notes in Mathematics. Springer-Verlag, 1971, pp. 28–61.

[CK85] Martin Campbell-Kelly. "Christopher Strachey, 1916–1975: A Biographical Note". In: *Annals of the History of Computing* 7 (1985), pp. 19–42.

[DGKL80] Veronique Donzeau-Gouge, Gilles Kahn, and Bernard Lang. "On the formal definition of ADA". In: *Semantics-Directed Compiler Generation*. Springer, 1980, pp. 475–489.

[Dij68] Edsger W. Dijkstra. "Letters to the editor: Go To statement considered harmful". In: *Communications of the ACM* 11.3 (1968), pp. 147–148.

[Don76] James Edward Donahue. *Complementary Definitions of Programming Language Semantics*. Springer-Verlag New York, Inc., 1976. ISBN: 038707628X.

[Dun63] Fraser G. Duncan. "ECMA Subset of ALGOL 60". In: *Communications of the ACM* 6.10 (Oct. 1963), pp. 595–599.

[Dun66] Fraser G. Duncan. "Our ultimate metalanguage: an afterdinner talk". In: *Formal Language Description Languages for Computer Programming*. North-Holland, 1966, pp. 295–295.

[ECM65] ECMA. *Standard ECMA-2 for a subset of ALGOL*. Tech. rep. European Computer Manufacturers' Association, 1965.

[End13] Albert Endres. "Early language and compiler developments at IBM Europe: A personal retrospection". In: *Annals of the History of Computing, IEEE* 35.4 (2013), pp. 18–30.

[Eng71] E. Engeler. *Symposium on Semantics of Algorithmic Languages*. Lecture Notes in Mathematics 188. Springer-Verlag, 1971.

[FJ87] L. M. G. Feijs and H. B. M. Jonkers. *Formal Definition of the Design Language COLD-K*. Tech. rep. METEOR/t7/PRLE/7. Preliminary Edition. Philips Research Labs, The Netherlands, 1987.

[FLDL] Thomas B. Steel. *Formal Language Description Languages for Computer Programming*. North-Holland Publishing Company New York, 1966.

[Flo62] Robert W. Floyd. "On the nonexistence of a phrase structure grammar for ALGOL 60". In: *Communications of the ACM* 5.9 (Sept. 1962), pp. 483–484.

[Flo67] Robert W. Floyd. "Assigning Meanings to Programs". In: *Proc. Symp. in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science*. American Mathematical Society, 1967, pp. 19–32.

[Gra06] Burton Grad. *Oral History of Richard Case*. Online. 2006. URL: `http://archive.computerhistory.org/resources/text/Oral_History/Case_Richard/Case_Richard_1.oral_history.2006.102658006.pdf`.

[Han76] A. Hansal. *A Formal Definition of a Relational Data Base System*. Tech. rep. UKSC 0080. IBM UK Scientific Centre, Peterlee, Co. Durham, 1976.

[HB80] P Haff and Dines Bjørner. *A Formal Definition of CHILL. A Supplement to the CCITT Recommendation Z. 200*. Tech. rep. 1980.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. "A universal modular actor formalism for artificial intelligence". In: *Proceedings of the 3rd international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc. 1973, pp. 235–245.

[HJ70a] Wolfgang Henhapl and Cliff B. Jones. *On the Interpretation of GOTO Statements in the ULD*. Tech. rep. 25.3.065. IBM Laboratory, Vienna, 1970.

[HJ70b] Wolfgang Henhapl and Cliff B. Jones. *The Block Concept and Some Possible Implementations, with Proofs of Equivalence*. Tech. rep. 25.104. IBM Laboratory Vienna, 1970.

[HJ78] Wolfgang Henhapl and Cliff B. Jones. "A Formal Definition of ALGOL 60 as Described in the 1975 Modified Report". In: *The Vienna Development Method: The Meta-Language*. Ed. by Dines Bjørner and Cliff B. Jones. Vol. 61. LNCS. Springer-Verlag, 1978, pp. 305–336. URL: `http://homepages.cs.ncl.ac.uk/cliff.jones/publications/OCRd/HJ82.pdf`.

[HJ82] Wolfgang Henhapl and Cliff B. Jones. "ALGOL 60". In: *Formal Specification and Software Development*. Ed. by Dines Bjørner and Cliff B. Jones. Prentice Hall International, 1982. Chap. 6,

pp. 141–174. ISBN: 0-13-329003-4. URL: http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/BjornerJones1982.

[HMRC88] Richard C. Holt, Philip A. Matthews, J. Alan Rosselet, and James R. Cordy. *The Turing Programming Language: Design and Defintion*. Prentice Hall International, 1988.

[HMSW09] C. A. R. Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. "Concurrent Kleene Algebra". In: *CONCUR 2009 - Concurrency Theory*. Ed. by Mario Bravetti and Gianluigi Zavattaro. Vol. 5710. LNCS. Springer, 2009, pp. 399–414.

[HMSW11] C. A. R. Hoare, B. Möller, G. Struth, and I. Wehrman. "Concurrent Kleene Algebra and its Foundations". In: *J. Log. Algebr. Program.* 80.6 (2011), pp. 266–296.

[Hoa69] C. A. R. Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[Hoa73] C. A. R. Hoare. *Hints on Programming Language Design*. Tech. rep. Stanford, CA, USA: Stanford University, 1973.

[HOPL] Richard L. Wexelblat, ed. *History of programming languages*. Academic Press, 1981.

[Hov14] Gauthier van den Hove. "On the Origin of Recursive Procedures". In: *The Computer Journal* (2014).

[HW73] Charles Anthony Robert Hoare and Niklaus Wirth. "An axiomatic definition of the programming language PASCAL". In: *Acta Informatica* 2.4 (1973), pp. 335–355.

[INR80] INRIA. *Formal Definition of the Ada Programming Language*. Tech. rep. INRIA, 1980.

[Izb75] H. Izbicki. *On a Consistency Proof of a Chapter of a Formal Definition of a PL/I Subset*. Tech. rep. 25.142. IBM Laboratory Vienna, 1975.

[JL71] Cliff B. Jones and Peter Lucas. "Proving Correctness of Implementation Techniques". In: *A Symposium on Algorithmic Languages*. Ed. by E. Engeler. Vol. 188. Lecture Notes in Mathematics. Springer-Verlag, 1971, pp. 178–211.

[Jon03] Cliff B. Jones. "The Early Search for Tractable Ways of Reasoning about Programs". In: *IEEE, Annals of the History of Computing* 25.2 (2003), pp. 26–49.

[Jon69] Cliff B. Jones. *A Comparison of Two Approaches to Language Definition as Bases for the Construction of Proofs*. Tech. rep. 25.3.050. IBM Laboratory, Vienna, 1969.

[Jon76] Cliff B. Jones. *Formal Definition in Compiler Development*. Tech. rep. 25.145. IBM Laboratory Vienna, 1976.

[Jon78] Cliff B. Jones. "Denotational Semantics of Goto: An Exit Formulation and its Relation to Continuations". In: *The Vienna Development Method: The Meta-Language*. Ed. by D. Bjørner and C. B. Jones. Vol. 61. LNCS. Springer-Verlag, 1978, pp. 278–304.

[Jon82] Cliff B. Jones. "More on Exception Mechanisms". In: *Formal Specification and Software Development*. Ed. by Dines Bjørner and Cliff B. Jones. Prentice Hall International, 1982. Chap. 5,

pp. 125–140. ISBN: 0-13-329003-4. URL: http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/BjornerJones1982.

[Jon99] Cliff B. Jones. “Scientific Decisions which Characterize VDM”. In: *FM’99 – Formal Methods*. Vol. 1708. LNCS. Springer-Verlag, 1999, pp. 28–47.

[KF71] Donald E. Knuth and Robert W. Floyd. “Notes on avoiding “go to” statements”. In: *Information processing letters* 1.1 (1971), pp. 23–31.

[Knu64] Donald E. Knuth. “Backus Normal Form vs. Backus Naur Form”. In: *Communications of the ACM* 7.12 (Dec. 1964), pp. 735–736.

[Lan64] Peter J. Landin. “The mechanical evaluation of expressions”. In: *The Computer Journal* 6.4 (1964), pp. 308–320.

[Lan65a] Peter J. Landin. “A Correspondence Between ALGOL 60 and Church’s Lambda-notation: Part I”. In: *Communications of the ACM* 8.2 (Feb. 1965), pp. 89–101.

[Lan65b] Peter J. Landin. “A Correspondence Between ALGOL 60 and Church’s Lambda-notation: Part II”. In: *Communications of the ACM* 8.3 (Mar. 1965), pp. 158–167.

[Lan66] Peter J. Landin. “A formal description of ALGOL 60”. In: *Formal Language Description Languages for Computer Programming*. North Holland, 1966, pp. 266–290.

[Lau67] Peter E. Lauer. *The formal explicates of the notion of algorithm: an introduction to the theory of computability with special emphasis on the various formalisms underlying the alternate explicates*. Tech. rep. 25.072. IBM Laboratory Vienna, 1967.

[Lau68] Peter E. Lauer. *Formal definition of ALGOL 60*. Tech. rep. 25.088. IBM Laboratory Vienna, 1968. URL: http://homepages.cs.ncl.ac.uk/cliff.jones/publications/OCRd/Lau68.pdf.

[Lau71] Peter E. Lauer. “Consistent Formal Theories of the Semantics of Programming Languages”. Printed as TR 25.121, IBM Laboratory Vienna. PhD thesis. Queen’s University of Belfast, 1971.

[LD69] John A. N. Lee and W. Delmore. “The Vienna Definition Language, a generalization of instruction definitions”. In: *SIGPLAN Symposium on Programming Language Definitions, San Francisco*. 1969.

[Lee72] John A. N. Lee. “The formal definition of the BASIC language”. In: *The Computer Journal* 15.1 (1972), pp. 37–41.

[LPP70] D. C. Luckham, David M. R. Park, and M. S. Paterson. “On Formalised Computer Programs”. In: *Journal of Computer and System Sciences* 4 (1970), pp. 220–249.

[Luc68] Peter Lucas. *Two constructive realisations of the block concept and their equivalence*. Tech. rep. 25.085. IBM Laboratory Vienna, 1968.

[Luc81] Peter Lucas. “Formal Semantics of Programming Languages: VDL.” In: *IBM Journal of Research and Development* 25.5 (1981), pp. 549–561.

[LW69] Peter Lucas and Kurt Walk. “On the formal description of PL/I”. In: *Annual Review in Automatic Programming* 6 (1969), pp. 105–182.

[McC62] John McCarthy. "Towards a Mathematical Science of Computation". In: *IFIP Congress*. 1962, pp. 21–28.

[McC66] John McCarthy. "A formal description of a subset of ALGOL". In: *Formal Language Description Languages for Computer Programming*. North-Holland, 1966, pp. 1–12.

[McC81] John McCarthy. "History of LISP". In: *History of programming languages*. Ed. by Richard L. Wexelblat. Academic Press, 1981. Chap. 4, pp. 173–183.

[MHW76] R. M. de Morgan, I. D. Hill, and B. A. Wichmann. "A supplement to the ALGOL 60 Revised Report". In: *The Computer Journal* 19.3 (1976), pp. 276–288.

[Mil74] R. E. Milne. "The Formal Semantics of Computer Languages and their Implementation". PhD thesis. Cambridge University, 1974.

[Modified Report] R. M. de Morgan, I. D. Hill, and B. A. Wichmann. "Modified Report on the Algorithmic Language ALGOL 60". In: *The Computer Journal* 19.4 (1976), pp. 364–379.

[Mog89] Eugenio Moggi. *An Abstract View of Programming Languages*. Tech. rep. ECS-LFCS-90-113. Edinburgh University Laboratory for the Foundation of Computer Science, 1989.

[Mos04] Peter D Mosses. "Modular structural operational semantics". In: *The Journal of Logic and Algebraic Programming* 60 (2004), pp. 195–228.

[Mos05] Peter Mosses. *Action semantics*. Vol. 26. Cambridge University Press, 2005.

[Mos11] Peter D. Mosses. "VDM semantics of programming languages: combinators and monads". In: *Formal Aspects of Computing* 23.2 (2011), pp. 221–238.

[Mos74] P. D. Mosses. *The Mathematical Semantics of ALGOL 60*. Technical Monograph PRG-12. Oxford University Computing Laboratory, Programming Research Group, 1974. URL: `http://homepages.cs.ncl.ac.uk/cliff.jones/publications/OCRd/Mosses74.pdf`.

[Mos75a] P. D. Mosses. "Mathematical Semantics and Compiler Generation". PhD thesis. University of Oxford, 1975.

[Mos75b] Peter David Mosses. "The semantics of semantic equations". In: *Mathematical Foundations of Computer Science: 3rd Symposium at Jadwisin near Warsaw, June 17–22, 1974*. Ed. by A. Blikle. Berlin, Heidelberg: Springer Berlin Heidelberg, 1975, pp. 409–422.

[MS74] R. Milne and Christopher Strachey. *A Theory of Programming Language Semantics*. An essay submitted for the Adams Prize 1973–74. 1974.

[MS76a] R. Milne and Christopher Strachey. *A Theory of Programming Language Semantics*. Part A: Indices and Appendices, Fundamental Concepts and Mathematical Foundations. Chapman and Hall, 1976.

[MS76b] R. Milne and Christopher Strachey. *A Theory of Programming Language Semantics*. Part B: Standard Semantics, Store Semantics and Stack Semantics. Chapman and Hall, 1976.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT press, 1990.

[MV14] Peter D Mosses and Ferdinand Vesely. "Funkons: Component-based semantics in K". In: *Rewriting Logic and Its Applications*. Springer, 2014, pp. 213–229.

[Nau81a] Peter Naur. "Formalization in program development". In: *BIT Numerical Mathematics* 22.4 (1981), pp. 437–453.

[Nau81b] Peter Naur. "The European side of the last phase of the development of ALGOL 60". In: *History of programming languages*. Ed. by Richard L. Wexelblat. Academic Press, 1981. Chap. 3, pp. 92–137.

[O'N94] Ian O'Neill. *Formal Semantics of SPARK Static Semantics*. Tech. rep. PVL SPARK_DEFN/STATIC V1.3. Program Validation Ltd., 1994.

[Oll71] A. Ollongren. *A Theory for the Objects of the Vienna Definition Language*. Tech. rep. 25.123. IBM Laboratory Vienna, 1971.

[Pat67] M. S. Paterson. "Equivalence Problems in a Model of Computation". PhD thesis. University of Cambridge, 1967.

[Pen00] Roger Penrose. "Reminiscences of Christopher Strachey". In: *Higher-Order and Symbolic Computation* 13.1 (2000), pp. 83–84.

[Per81] Alan J. Perlis. "The American Side of the Development of ALGOL". In: *History of programming languages*. Ed. by Richard L. Wexelblat. Academic Press, 1981. Chap. 3, pp. 75–91.

[Plo04a] Gordon D. Plotkin. "A structural approach to operational semantics". In: *Journal of Logic and Algebraic Programming* 60–61 (2004), pp. 17–139.

[Plo04b] Gordon D. Plotkin. "The origins of structural operational semantics". In: *Journal of Logic and Algebraic Programming* 60–61 (2004), pp. 3–15.

[Plo76] Gordon D. Plotkin. "A Powerdomain Construction". In: *SIAM Journal on Computing* 5 (1976), pp. 452–487.

[Plo81] Gordon D. Plotkin. *A Structural Approach to Operational Semantics*. Tech. rep. Aarhus University, 1981.

[Rad81] George Radin. "The early history and characteristics of PL/I". In: *History of programming languages*. Ed. by Richard L. Wexelblat. Academic Press, 1981, pp. 551–589.

[Revised Report] John W. Backus, Friedrich L. Bauer, Julien Green, C. Katz, John McCarthy, Alan J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, Aadrian van Wijngaarden, and Michael Woodger. "Revised Report on the Algorithm Language ALGOL 60". In: *Communications of the ACM* 6.1 (Jan. 1963). Ed. by P. Naur, pp. 1–17. URL: `http://homepages.cs.ncl.ac.uk/cliff.jones/publications/OCRd/BBG63.pdf`.

[Rey93] John C. Reynolds. "The discoveries of continuations". In: *Lisp and symbolic computation* 6.3-4 (1993), pp. 233–247.

[RS76] G. Radin and P.R. Schneider. *An Architecture for an Extended Machine with Protected Addressing*. Tech. rep. 00.2757. IBM Poughkeepsie Lab, 1976.

[Sch78] R. W. Scheifler. *A Donotational Semantics of CLU*. Tech. rep. MIT/LCS/TR-201. Cambridge, Mass.: Laboratory for Computer Science, Massachusetts Institute of Technology, 1978.

[Sco00] Dana Scott. “Some reflections on Strachey and his work”. In: *Higher-Order and Symbolic Computation* 13.1 (2000), pp. 103–114.

[Sco69] Dana Scott. “A Type-Theoretical Alternative to CUCH, ISWIM, OWHY”. Typed script – Oxford. 1969.

[Sco70] Dana Scott. *The Lattice of Flow Diagrams*. Tech. rep. PRG-3. Oxford University Computing Laboratory, Programming Research Group, 1970.

[Sco71a] Dana Scott. *Continuous Lattices*. Tech. rep. PRG-7. Oxford University Computing Laboratory, Programming Research Group, 1971.

[Sco71b] Dana Scott. “The Lattice of Flow Diagrams”. In: *[Eng71]*. 1971, pp. 311–366.

[Sco73] Dana Scott. “Models for Various Type-Free Calculi”. In: *Studies in Logic and Foundations of Mathematics Vol. 74 (Proc. of the 4th International Congress for Logic, Methodology and Philosophy of Science, Bucharest, 1971)*. Ed. by P. Suppes, L. Henkin, A. Joja, and Gr.C. Moisil. North Holland Publishing Company, 1973, pp. 158–187.

[Shu15] Len Shustek. “An Interview with Fred Brooks”. In: *Communications of the ACM* 58.11 (Oct. 2015), pp. 36–40.

[SS70] Christopher Strachey and Dana Scott. *Mathematical Semantics for Two Simple Languages*. Paper read at Princeton. 1970.

[SS71] Dana Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages*. Technical Monograph PRG-6. Oxford University Computing Laboratory, Programming Research Group, 1971.

[ST15] Dana Scott and Tanja Traxler. *Logic Lounge with Dana Scott*. Online. Video interview. 2015. URL: `https://www.youtube.com/watch?v=nhc94A829qI`.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[Sto80] Joseph E. Stoy. “Foundations of denotational semantics”. In: *Abstract Software Specifications: 1979 Copenhagen Winter School January 22 – February 2, 1979 Proceedings*. Ed. by Dines Bjørner. Springer Berlin Heidelberg, 1980, pp. 43–99.

[Str66] Christopher Strachey. “Towards a Formal Semantics”. In: *Formal Language Description Languages for Computer Programming*. North Holland, 1966.

[SW61] Christopher Strachey and Maurice V. Wilkes. “Some Proposals for Improving the Efficiency of ALGOL 60”. In: *Communications of the ACM* 4.11 (Nov. 1961), pp. 488–491.

[SW74] Christopher Strachey and Christopher P. Wadsworth. *Continuations – A Mathematical Semantics for Handling Jumps*. Monograph PRG-11. Oxford University Computing Laboratory, Programming Research Group, 1974.

[SZ66] F. Schwarzenberger and H. Zemanek. *Editing Algorithms for Texts over Formal Grammars*. Tech. rep. 25.066. IBM Laboratory Vienna, 1966.

[Ten76] R. D. Tennent. "The Denotational Semantics of Programming Languages". In: *Communications of the ACM* 19 (1976), pp. 437–453.

[ULD-II-AS] D. Beech, R. Rowe, R. A. Larner, and J. E. Nicholls. *Abstract Syntax of PL/I*. Tech. rep. TN 3002. IBM Laboratory Hursley, ULD Version II, 1967.

[ULD-II-CS] D. Beech, R. Rowe, R. A. Larner, and J. E. Nicholls. *Concrete Syntax of PL/I*. Tech. rep. TN 3001. IBM Laboratory Hursley, ULD Version II, 1966.

[ULD-II-Sem] C. D. Allen, D. Beech, J. E. Nicholls, and R. Rowe. *An Abstract Interpreter of PL/I*. Tech. rep. TN 3004. IBM Laboratory Hursley, ULD Version II, 1966.

[ULD-II-Trans] D. Beech, J. E. Nicholls, and R. Rowe. *A PL/I Translator*. Tech. rep. TN 3003. IBM Laboratory Hursley, ULD Version II, 1966.

[ULD-IIIvI] PL/I Definition Group of the Vienna Laboratory. *Formal Definition of PL/I (Universal Language Document No. 3)*. Tech. rep. 25.071. IBM Laboratory Vienna, ULD Version I, 1966.

[ULD-IIIvII-CS] K. Alber, P. Oliva, and G. Urschler. *Concrete Syntax of PL/I*. Tech. rep. 25.084. IBM Laboratory Vienna, ULD Version II, 1968.

[ULD-IIIvII-CT] M. Fleck and Erich Neuhold. *Formal Definition of the PL/I Compile Time Facilities*. Tech. rep. 25.080. IBM Laboratory Vienna, ULD Version II, 1968.

[ULD-IIIvII-Intro] Peter Lucas, K. Alber, Kurt Bandat, Hans Bekič, P. Oliva, Kurt Walk, and G. Zeisel. *Informal Introduction to the Abstract Syntax and Interpretation of PL/I*. Tech. rep. 25.083. IBM Laboratory Vienna, ULD Version II, 1968.

[ULD-IIIvII-Meth] Peter Lucas, Peter E. Lauer, and H. Stigleitner. *Method and notation for the formal definition of programming languages*. Tech. rep. 25.087. IBM Laboratory Vienna, ULD Version II, 1968. URL: `http://homepages.cs.ncl.ac.uk/cliff.jones/publications/VDL-TRs/TR25.087.pdf`.

[ULD-IIIvII-Sem] Kurt Walk, K. Alber, K. Bandat, Hans Bekič, Gerhard Chroust, Viktor Kudielka, P. Oliva, and G. Zeisel. *Abstract Syntax and Interpretation of PL/I*. Tech. rep. 25.082. IBM Laboratory Vienna, ULD Version II, 1968.

[ULD-IIIvII-Trans] K. Alber and P. Oliva. *Translation of PL/I into Abstract Syntax*. Tech. rep. 25.086. IBM Laboratory Vienna, ULD Version II, 1968.

[ULD-IIIvIII-CS] G. Urschler. *Concrete Syntax of PL/I*. Tech. rep. 25.096. IBM Laboratory Vienna, ULD Version III, 1969.

[ULD-IIIvIII-CT] M. Fleck. *Formal Definition of the PL/I Compile Time Facilities*. Tech. rep. 25.095. IBM Laboratory Vienna, ULD Version III, 1969.

[ULD-IIIvIII-Intro] K. Alber, H. Goldmann, Peter E. Lauer, Peter Lucas, P. Oliva, H. Stigleitner, Kurt Walk, and G. Zeisel. *Informal Introduction*

to the Abstract Syntax and Interpretation of PL/I. Tech. rep. 25.099. IBM Laboratory Vienna, ULD Version III, 1969.

[ULD-IIIvIII-Sem] Kurt Walk, K. Alber, M. Fleck, H. Goldmann, Peter E. Lauer, E. Moser, P. Oliva, H. Stigleitner, and G. Zeisel. *Abstract Syntax and Interpretation of PL/I*. Tech. rep. 25.098. IBM Laboratory Vienna, ULD Version III, 1969.

[ULD-IIIvIII-Trans] G. Urschler. *Translation of PL/I into Abstract Syntax*. Tech. rep. 25.097. IBM Laboratory Vienna, ULD Version III, 1969.

[Utm63] R. E. Utman. *Minutes of the 3rd meeting of IFIP TC2*. Online. Chaired by H. Zemanek. Archived by Andrei Ershov. 1963. URL: `http://ershov-arc.iis.nsk.su/archive/eaindex.asp?did=41825`.

[Utm64] R. E. Utman. *Minutes of the 4th meeting of IFIP TC2*. Online. Chaired by H. Zemanek. Archived by Andrei Ershov. 1964. URL: `http://ershov-arc.iis.nsk.su/archive/eaindex.asp?did=41826`.

[Weg72] Peter Wegner. "The Vienna definition language". In: *ACM Computing Surveys (CSUR)* 4.1 (1972), pp. 5–63.

[Wei75] F. Weissenböck. *A Formal Interface Specification*. Tech. rep. 25.141. IBM Laboratory Vienna, 1975.

[Wel82] A. Welsh. "The Specification, Design and Implementation of NDB". Also published as technical report UMCS-82-10-1. MA thesis. Department of Computer Science, University of Manchester, 1982.

[Wel84] A. Welsh. "A Database Programming Language: Definition, Implementation and Correctness Proofs". Also published as technical report UMCS-84-10-1. PhD thesis. Department of Computer Science, University of Manchester, 1984.

[Wol88] M. I. Wolczko. "Semantics of Object-Oriented Languages". Also published as Technical Report UMCS-88-6-1. PhD thesis. Department of Computer Science, University of Manchester, 1988.

[Zem68] H. Zemanek. "Abstrakte Objekte". In: *Elektron. Rechenanl.* 5 (1968), pp. 208–217.

[Zim69] K. Zimmermann. *Outline of a formal definition of FORTRAN*. Tech. rep. 25.3.053. IBM Laboratory Vienna Vienna, 1969.