Noname manuscript No.
(will be inserted by the editor)

Formal Semantics of ALGOL 60: Four Descriptions in their Historical Context

Troy K. Astarte, Cliff B. Jones

Received: date / Accepted: date

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 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 become distracted by the minutiae of notational conventions adopted by their authors. This paper also explores the historical reasons for the development of these descriptions in particular, and gives some general historical background of the groups involved (the IBM laboratories in Vienna and Hursley, and Oxford's Programming Research Group).

Keywords Programming languages · Semantic description · Operational semantics · Denotational semantics

Version 8.1 Dated: August 24, 2018

Financial support from EPSRC, who are both providing the PhD funding for Astarte and whose "Strata" Platform Grant is funding travel expenses for both authors, is gratefully acknowledged.

T. K. Astarte
Newcastle University
Newcastle upon Tyne
NE1 7RU
E-mail: t.astarte@ncl.ac.uk

C. B. Jones
Newcastle University
E-mail: cliff.jones@ncl.ac.uk

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, specifically four different descriptions of the same language. It must be made clear that the aim is an in-depth study of a narrow topic: for a general historical background to computing machines, the reader could consult [Randell, 2013]; an invaluable source on the history of programming languages is the HOPL conferences [Wexelblat, 1981, Bergin and Gibson, 1996]; a very clear book on the history of programming that sets out a broader context for our focus is [Priestley, 2011]. The closest paper of which the authors are aware to the chosen narrow theme is [Zhang and Xu, 2004] (its focus, however, is a broad and shallow overview of various approaches to the semantics of programming languages, without much history).

Several research groups have chosen ALGOL 60[2] to demonstrate that their way of formally describing semantics scales to realistic programming languages. The availability of these semantic descriptions of broadly the same object language makes for an interesting comparison of various aspects of the methods. The approach here is somewhat similar to that utilised by Knuth and Trabb Pardo in their paper on the early development of programming languages [Knuth and Pardo, 1976]. It is also helpful that the authors of the semantic descriptions have often been careful to record at least some context of their research.

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

The body (Sections 2–6) of this paper follows a broadly historical sequence and attempts to clarify the context and background of the work. This introductory section deviates from the conventions of historical writing by using benefit of hindsight (in particular with respect to the use of modern terminology).[3]

There are two ways of reading the current paper. It is hoped that those involved in research into language semantics will understand the technical details and potentially follow some of our pointers to primary source material.[4] Historians ought be able to skip much of the technical detail and still obtain a useful overview of the origin and flow of ideas. Above all, we hope to have provided useful source material for subsequent study.

In this introduction, a definition of ‘semantics’ is offered and the reasons for attempting a formal semantics are considered. The importance of the ALGOL language is discussed along with examples of how its semantics are described informally in the defining Report and finally the dimensions of comparison for each full semantic description are also given.

A key early reference is described in Section 2. Then follow sections, presented in chronological order, on four complete descriptions; in each, a historical background and context is given before deeper semantic points are discussed. Finally, in the conclusions sec-

[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 the standard (and written after it is set).

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

[3] In places where it is useful to establish a link to later work the forward references are placed in footnotes.

[4] All but one of the ALGOL descriptions are only available as technical reports and there was never an easy path to publishing the descriptions of even larger languages. Where the authors have access to physical copies of historical documents that are difficult to locate, they have made scans available at:
`http://homepages.cs.ncl.ac.uk/cliff.jones/semantics-library/`
In particular, all four descriptions of ALGOL are available.

tion, 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. There is a brief summary of the historical story.

One brief note is in order on citations and references: section numbers preceded by the word ‘Section’ refer to sections in this document; numbers preceded by the section symbol (§) refer to sections in the referenced text.

1.1 Why it is crucial to be precise about semantics

To give some motivation to the subject of the paper, it is worth briefly reviewing the advantages that a formal semantics can bring. Computers execute machine code programs which, although detailed, have easy-to-follow effects on the state of the machine. 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 machine code instructions fairly easy to follow[5] but programming directly in machine code has long been seen as time-consuming and short sighted [Peláez Valdez, 1988, p.4]. High-level programming languages make the job of the programmer easier[6] but programs written in these languages require translation into machine code before they can be executed. This task is typically performed by a compiler[7] or interpreter program.

The introduction of new languages does, however, introduce challenges and these have become increasingly onerous as the level of abstraction in programming languages has increased. How can one 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 questions is with formal semantics.

Another more subtle observation is that many of the languages that have been designed[8] can be judged as poor in that they offer surprises to the programmer and/or unreasonable challenges to their implementers Many researchers who have worked on semantics claim that building an abstract model of a language can help reduce such unintended problems.

[5] More recent ‘relaxed’ (or weak) memory architectures have, however, made this much harder.

[6] A panel, on which Jones sat, at the *Mathematical Foundations of Programming Semantics* held at CMU in 2004 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)?

[7] It is interesting that the term ‘compiler’ is more commonly used than ‘translator’; the former appears to derive from the early attempts to improve re-use of programs by building libraries of routines that could be compiled with some additions to create an application. According to Grace Hopper, a key member of the team that created the very first such compilers, this was a precursor to actual languages where the task was more clearly one of translation [Beyer, 2009, p.223].

[8] An indication of the scale of the challenge is that up to 2010, a web site that attempted to track programming languages had recorded 8512 languages (and this probably excluded myriad ‘domain specific languages’).

Jones heard John Reynolds express the wish at a panel discussion at MFPS 2004 that "semanticists should be the obstetricians rather than the coroners of programming languages". Historically, McCarthy appeared to see the need to record semantics to ensure that different compilers implemented the same language; Floyd—and even more clearly Hoare—put the emphasis on being be able to reason about programs written in a language.

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 describe and 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.[9]

A fundamental requirement for a formal semantics is a set of basic notions.[10] A semantics is essentially a description of one ('object') language in another ('meta') language; in order for the exercise to add value in the sense of clarifying the object language, the meta language must be understood by both the reader and the writer, and at its core this is a question of the base concepts.

To illustrate this, consider a simple example of an object language which includes as one of its texts a string of three vertical marks III. Its interpretation is not obvious and to some extent depends on preconceptions. One option would be to view the string as the decimal representation of the number *one hundred and eleven*; however, it could also be the binary representation of the number *seven* or the unary representation of the number *three*. It could even be Roman numerals (also for *three*) or indeed three capital letter I's.

In order to fix on one of these interpretations (or denotations), two things are required: there must be some base concepts in terms of which the meaning of simple symbols is fixed; then there must be an (understood) notation for fixing the meaning of a string of symbols in terms of the meaning of its constituents.

For numbers, Peano's axioms give the notion of *zero* and *successor* from which the denotations of the binary digits 0 and 1 can be constructed. The meaning of strings of binary digits can be defined by a recursive function that uses the meaning of individual digits and multiplication to reflect the meaning of the position of the digits. The same basic objects (Peano numbers) suffice as the basis for decimal numbers but the signs commonly written as 2, 3 etc. must also be given meaning and the recursive function for strings of digits now needs to reflect the fact that each shift in position multiplies the value of the next digit by *ten*.

[9] There is a distinction to be made between concrete and abstract syntax. Concrete syntax defines the valid texts of a language in terms of strings of symbols of the language. Abstract syntax uses a different notation to describe the syntax of a language purely in terms of its composition from subcomponents. 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 dynamically (i.e. in the semantic rules).

[10] See the discussion raised by McCarthy in a paper presented in 1964 [McCarthy, 1966, §8].

Numbers, either as strings of binary or decimal digits, are so familiar that the above discussion might appear to be unnecessary. However, it is important to be clear about the distinction between numerals and the numbers they denote. Consider the symbols used in binary, 1 and 0, and the mathematical concepts of the numbers *one* and *zero*: while there are conventional links between these things, there is no fundamental reason why one could not use the symbol '!' to mean *one* and 'o' to mean *zero*, or even to flip the normal convention around and thus make 101 denote the number *two*.

So there are two important notions in a formal semantics that must both be understood in the same way by reader and writer: the meanings of the bases (in this case symbols) and the meanings of the interpretive transformations (in this case functions).

The issues of what concepts can be used in the meta-language and what are the basic (understood) objects 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 a language admits more than one effect (such as occurs when non-determinism is introduced): a function from the language to its denotations is no longer adequate.

There are many issues that make it more challenging to define the semantics of programming languages than, say, logic languages. 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 (i.e. by reference to a space in storage), 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[11]) 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. John McCarthy used the term 'abstract interpreter' for one which is written in a tractable, functional, notation; a semantic approach developed by him is outlined in Section 2. Section 1.4 outlines issues that make it difficult to achieve ease of reasoning about ALGOL but the basic idea remains that of a mathematically tractable interpretation function.

Central to most semantic descriptions is the notion of choosing 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 the semantic approaches described in this paper (in contrast to 'axiomatic' semantics). 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.

Another important and influential semantic description style is today normally known as *denotational*, and two of the descriptions discussed in this paper follow a denotational approach. The key distinction from operational descriptions is that denotational descriptions abstract away from the concept of a machine, and instead map programs, or their constituent parts, to functions from states to states. Where the interpreting function of an operational semantics requires a program and an initial state, a denotational semantics embeds the state into the domains and codomains of the denotation functions, thereby pushing notions of state onto the other side of the interpretation mapping. This mapping should be homomorphic

[11] See the discussion in Section 7.1 about non-deterministic languages that require a way of saying that there is more than one valid result to a computation.

from the (nested) structure of program components to the space of denotations.[12] It might be argued that this structural requirement encourages the use of smaller, cleaner, states. This argument is evaluated in Section 7.1.

Sections 5 and 6 describe denotational approaches to semantics; 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.

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 Tony Hoare's 'Axiomatic basis' paper [Hoare, 1969] 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 but are not discussed further in this paper.[13]

All language descriptions ultimately need a universal 'meta-language', as named by Fraser Duncan in an after-dinner speech [Duncan, 1966] at the *Formal Language Description Languages* conference discussed in Section 2, and this must be natural language, typically English. It is, of course, possible to describe the semantics of a programming language using only natural language and this is exactly what is done in the ALGOL Reports (see discussion in Section 1.5) and, indeed, most language specifications. The hallmark of a formal definition is that it takes a very small collection of basic notions and then combines 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 base concepts as axiomatic (and presumably described in natural language); furthermore, both operational and denotational semantic descriptions rely on the notion of functions (or relations). The 'building blocks' used in the four ALGOL descriptions in this paper are reviewed in Section 7.3.

[12] Some language constructs such as goto statements make this rather difficult, and extra care is needed when handling those. See Sections 5.5.4 and 6.5.4 for two different approaches.

[13] For reasons of length, several topics that are covered in the Technical Report version [Jones and Astarte, 2016] are omitted here.

1.3 Dimensions of comparison

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 3–6, 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
– 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)[14]

1.4 Why ALGOL is interesting

This section emphasises the historical and intellectual importance of ALGOL, but it is also a *leitmotiv* of this paper that creating semantic description methods whose capabilities extend beyond simple toy languages is important; ALGOL represents a sufficient semantic challenge without requiring book length descriptions.

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*, Chapter 3 [Perlis, 1981, Naur, 1981a].[15] The resulting ALGOL[16] 60 language is powerful yet clean and it introduced many concepts that have been adopted in other languages. As Tony Hoare has commented in his paper ‘Hints on programming language design’ [Hoare, 1973]:

> 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.

Mark Priestley makes a strong argument [Priestley, 2011, Chap. 9] that ALGOL had a far wider intellectual impact than its relatively limited application in industry would lead one to expect.[17] It is of course also true that some dialects such as JOVIAL acquired followers and, academically, Pascal can be viewed (via ALGOL-W) as a successor that was chosen as the first language for teaching in many universities.

In her thesis ‘A gift from Pandora’s box’, Peláez Valdez describes the advent of ALGOL as a paradigm shift from existing high-level programming languages [Peláez Valdez, 1988].

[14] These sections also deviate from historical convention by making references to subsequent research that was affected by work being discussed.

[15] Perlis [p.91] quips “ALGOL deserves our affection and appreciation. It was a noble **begin** but never intended to be a satisfactory **end**.”.

[16] Peláez Valdez also notes that ‘Algol’ is the name of a fixed star in the Perseus constellation; it is an Arabic word whose meaning is ‘the demon’. This pun was not lost on the original language designers! [Peláez Valdez, 1988, p.26].

[17] A major impetus to research on *Programming Methodology* arose from the divisions surrounding ALGOL-68: IFIP WG 2.3 was formed around distinguished figures such as Dahl, Dijkstra and Hoare who signed the “minority report” in WG 2.1. (This is not the place to repeat this story—see the ALGOL 68 session in the second History of Programming Languages conference [Bergin and Gibson, 1996]; another account is given in Chapter 7 of Peláez Valdez’ thesis [Peláez Valdez, 1988]).

ALGOL was designed specifically for machine independence and universality, in contrast to other languages which were designed with performance on particular machines in mind. There was at first a great deal of enthusiasm from IBM and its usergroups, because although FORTRAN was IBM owned, it was designed for one particular machine and a machine-independent language might be more suited to future IBM computers. Ultimately, however, there was less support for ALGOL than hoped for in industry, primarily because of the large amount of existing FORTRAN code which users were reluctant to either give up or painstakingly re-encode.

An important technical feature of ALGOL is its grammar, which 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. This presents a challenge to the language describer, as considerations must be made of the 'environment' (i.e. block or procedure) in which a statement is executed in order to determine the value of a variable.

A 'strong typing' system is intended to prevent type errors from occurring 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[18] 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 (especially for '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 straightforward: the value of the variable is evaluated, and this raw value is passed into the procedure. In the case where a single identifier is provided as an argument,[19] '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). 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, as that could result in some values being 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 in ALGOL can be defined by recursion. Although this is now common in languages, it required the invention of implementation techniques such as Dijk-

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

[19] The ALGOL literature tends to refer to arguments as 'actual parameters' and to parameters as 'formal parameters'.

stra's 'display' mechanism for accessing values of identifiers in a stack structure. The story of recursion in ALGOL is not wholly straightforward and is discussed in van den Hove's article 'On the origin of recursive procedures' [van den Hove, 2014]. He 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 into other procedures. As is explained in Section 5, 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 famous letter to the Communications of the ACM 'Goto statements considered harmful' [Dijkstra, 1968] (met with Knuth's defence in [Knuth and Floyd, 1971]). For better or worse, ALGOL allows label parameters, goto statements closing 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* [Strachey, 1970] Christopher Strachey[20] 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 housekeeping that would have occurred had the abnormally terminated phrases 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',[21] ALGOL also allows switch variables which can hold different labels under different conditions.

Chris Wadsworth [22] wrote to his supervisor Christopher Strachey[23] about Peter Mosses' ALGOL 60 paper (see Section 5) "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 was designed partly as a publication language for algorithms and, as such, initially contained no input/output statements; these were added in de Morgan, Hill, and Wichmann's 'Supplement to the ALGOL 60 Revised Report' [de Morgan et al., 1976b]. A small collection of 'standard' functions such as a square root function are defined for ALGOL.

1.5 Describing ALGOL without a formal meta-language

There are a number of definition documents produced for the various versions of ALGOL, but the main reference used for this paper is the 1963 'Revised Report on the Algorithmic

[20] Christopher S. Strachey (1916–75); leading British computer scientist who published little but each item was a polished gem (e.g. 'The varieties of programming language' [Strachey, 1973], which became something of a manual to those interested in programming theory); see Martin Campbell-Kelly's 'Biographical Note' [Campbell-Kelly, 1985] for an excellent biographical summary.

[21] Some people argue that because labels are types, and values of other types (such as integer) can be assigned to variables, there should be variables to which one can assign label values.

[22] Christopher P. Wadsworth D.Phil. [Wadsworth, 1971] student under Strachey's supervision at Oxford and inventor of continuations in denotational semantics (see Section 5.5.4).

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

Lanugage ALGOL 60' [Backus et al., 1963]. This was the most modern source at the time of the earliest of the descriptions below (that of Peter Lauer, discussed in Section 3) and is the version upon which that semantic description is based.

1.5.1 Syntax

The 'Syntax and Semantics of the Proposed International Algebraic Language' paper prepared by Backus describing the language that came to be known as ALGOL 58 has an interesting introduction to its second section [Backus, 1959]. The author explains some of the problems associated with an informal language definition and acknowledges a desire to present a fully formal definition of the language; however, only the 'description of legal programs', i.e. the syntax, is actually given, and a subsequent paper giving a formal treatment to the semantics is promised. This never appeared; however, it is interesting to note that as early as 1958 the arguments for formalism in language presentation, if not design, were clearly understood.

ALGOL 58, then, was the first language to be fully specified with a formal, concrete, context-free syntax, but the description is short and not given in the main body of the paper. Instead, natural language descriptions and examples are given for the language syntax.

It was not until the 1960 'Report on the algorithmic language ALGOL 60' [Backus et al., 1960] that the formalised syntax was given pride of place throughout the definition. Backus' technique was subjected to some improvements and additions when it was used by Peter Naur in the 1963 'Revised Report on the algorithmic language ALGOL 60' [Backus et al., 1963], as reported by Knuth in a letter to the ACM [Knuth, 1964].

This formal syntax description method is referred to as BNF for Backus Normal or Backus–Naur 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.

1.5.2 Context dependencies

The grammars for 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 [Floyd, 1962], 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.
> — [Backus et al., 1963, §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.
> — [Backus et al., 1963, §4.2.4]

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 can bring advantages; an example of an approach to this is discussed in Section 6.3 of this document.[24]

1.5.3 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 …
> — [Backus et al., 1963, §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.[25]

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.
> — [Backus et al., 1963, §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, [Backus et al., 1963, §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 [Backus et al., 1963, §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 1.4) 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.
> — [Backus et al., 1963, §4.7.3.2]

[24] 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.

[25] That said, Peter Naur attacked Henhapl and Jones in his 1981 paper 'Formalization in program development' [Naur, 1981b], after publication of the duo's ALGOL description [Henhapl and Jones, 1978] (see Section 6), comparing the complicated mathematics of the formal model unfavourably to the structured English of the 1976 'Modified Report' [de Morgan et al., 1976a].

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 used to implement the copy rule in the 'functional' description of ALGOL described in Section 4.5.1 below.

2 McCarthy's 'Micro-ALGOL' description

As an introduction to the 'complete' descriptions of ALGOL, it is worth reviewing the formal description of a severe subset of ALGOL written by John McCarthy[26] and presented at the 1964 IFIP Working Conference held in Baden-bei-Wien. Important and influential concepts in formal semantics were presented in this talk and other presentations at that conference. The *Formal Language Description Languages for Computer Programming* conference brought together most of the researchers who were interested in semantic description approaches. The conference was organised by IFIP TC-2 chair and IBM Laboratory Vienna director Heinz Zemanek[27] and partially funded by IBM thanks to his influence, as recorded in the minutes of TC-2 meeting dated May 1964 [Utman, 1964, p. 11.3]. Interestingly the suggestion to make formal languages the topic of the symposium appears to have been made by Peter Naur at the previous TC-2 meeting dated September 1963 [Utman, 1963, p. 7].

This was in fact the first IFIP 'Working Conference',[28] 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), as number 18 of the ALGOL Bulletin records [Wichmann, 2004, No. 18]. As a result, the conference was well-attended by 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.[29] The proceedings [Steel, 1966] appeared two years later in 1966 and are particularly valuable because of the effort that was made by members of the IBM Vienna team to record and transcribe the discussions that followed the presentations.

McCarthy's paper was the first given at the conference: in *A formal description of a subset of ALGOL* [McCarthy, 1966] he provides an operational description of a subset of ALGOL that he dubs 'micro-ALGOL'. This paper was 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 3). 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: he does not take the obvious selection of assignments, conditionals, and while statements, but does include goto statements. This decision forces him to retain the whole text (for backwards gotos).[30]

[26] 1927–2011; AI pioneer and inventor of LISP. See `http://news.stanford.edu/news/2011/october/john-mccarthy-obit-102511.html` for one obituary.

[27] 1920–2014; Austrian computing pioneer; see [Fröschl et al., 2015] for an *In Memoriam*.

[28] The start of an influential series—for decades, IFIP Working Conferences provided one of the main drivers for researchers.

[29] In the Preface to the proceedings of the conference [Steel, 1966], the editor Tom 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."

[30] In a sense, this can be seen as the germ of the ULD 'control tree' (see Section 3.5.4).

The authors of the ALGOL report had shown how BNF could be used to define the concrete syntax of a language: the production rules define a set of strings of characters that are 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 his earlier paper 'Towards a mathematical science of computation'. 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, but McCarthy explains the value of using an abstract syntax as follows: "Questions of notation are separated from semantic questions and postponed until the concrete syntax has to be defined."

The semantics of micro-ALGOL are given via an interpreting function *micro*, which takes a program, a store and a program counter as arguments and delivers a store as a result.[31] This function is also given two concrete representations in McCarthy's paper: a LISP S-expression "suitable for use inside a machine"; and a concrete syntax.

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

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 possibility of side-effects).

Although McCarthy did not continue working in the field of formal semantics long, an interesting piece of work from him and his student James Painter applies the principles of this operational semantics to the development of a compiler [McCarthy and Painter, 1966]. By constructing the compiler around a formal description, the proofs of correctness are shown to be surprisingly straightforward (at least for the relatively simple language demonstrated).

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

3 Vienna operational description

In the mid 1960s, the IBM Laboratory in Vienna under Heinz Zemanek became heavily engaged in research into the formal semantics of programming languages, ultimately developing a series of operational descriptions of programming languages. Peter Lauer is the listed author[32] of [Lauer, 1968a] which presents a Vienna style operational semantics of ALGOL. The notation used later became known as the Vienna Definition Language. VDL had evolved as part of a group effort that solved many problems in scaling the basic idea of McCarthy's approach to describe the huge PL/I language.

Peter Lauer worked at the IBM Vienna Lab 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 a guide for use of VDL [Lucas et al., 1968b] and some theoretical work on algorithms [Lauer, 1967, 1968b].

Subsequent to working on the ALGOL description, Lauer obtained a Ph.D.[33] 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.

3.1 Background: A brief history of VDL

The story of VDL starts with IBM's development of the PL/I programming language,[34] described by Fred Brooks in an interview as "a universal programming language that would meld and displace FORTRAN and COBOL" [Shustek, 2015].[35] 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 whose compromises undermined its elegance. This and the general situation with programming languages relates to the decision to have a photograph of Pieter Bruegel's *Tower of Babel* covering an entire wall of the conference room of the IBM Vienna laboratory. Sometimes, after difficult meetings, people involved in IBM projects were linked to the figures in the bottom left corner of the painting. Figure 1 shows key early members of the Vienna Lab in front of this wall. Heinz Zemanek was a particular fan of this picture, and frequently used it in publications—the programme of the previously-mentioned *Formal Language Description Languages* conference was adorned with a reproduction of the *Tower*.

[32] Contributions are acknowledged from other members of the laboratory: Lucas, Alber, Bekič, and Fleck.

[33] Zemanek consistently encouraged his staff to obtain their doctorates. Erich Neuhold (private communication December 2016) gratefully recalled the major influence that this had on his career.

[34] A history of the PL/I language can be found in *History of Programming Languages* [Radin, 1981]; Peter Lucas also wrote a history of the VDL semantics method in [Lucas, 1981].

[35] Bo Evans is franker: in his unpublished autobiographical notes, he writes "IBM undertook to specify and develop a single high-level language, PL/I, to serve both types of applications, something that could replace FORTRAN and COBOL".

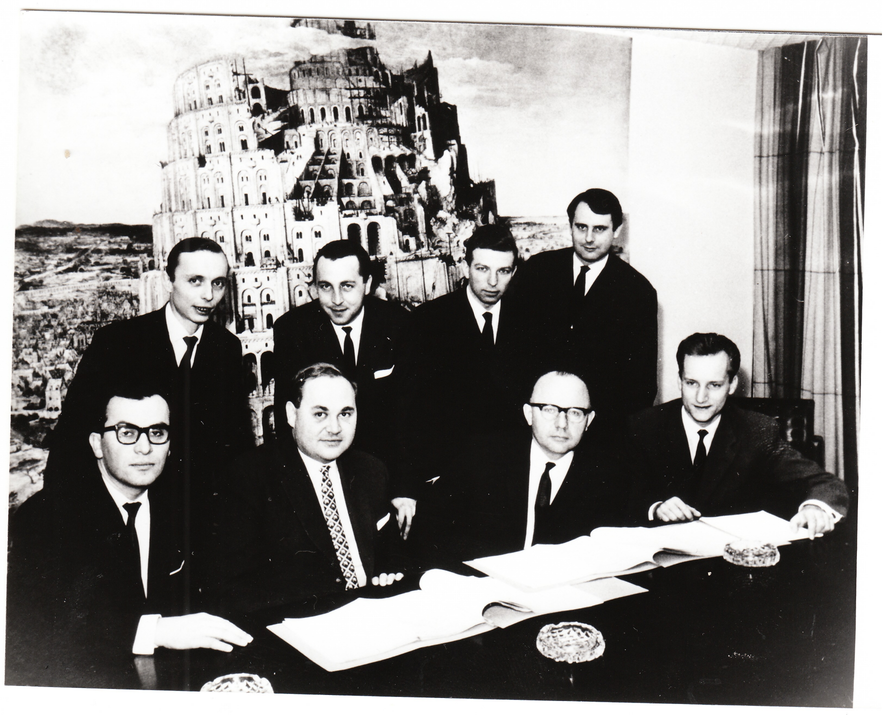


Fig. 1 From left to right: (standing) Peter Lucas; George Leser; Viktor Kudielka; Kurt Walk; seated: Ernst Rothauser; Kurt Bandat; Heinz Zemanek; Norbert Teufelhart.

The official IBM 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.[36] This specification was initially referred to as "Universal Language Document", according to Peter Lucas' history of VDL [Lucas, 1981], but quickly became known 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, as Lucas describes [Lucas, 1981]. According to Hermann Maurer,[37] members of the Lab at the time were a mix of engineers and mathematicians, and knowledge of semantics prior to the Baden-bei-Wien conference was limited; certainly no work on formal semantics had taken place. People who were members of the Lab[38] agree that Peter Lucas and Hans Bekič were the main drivers behind the majority of the technical concepts in the early semantics work; Maurer recalls that Walk, one level higher in the management structure, was adept at recognising and promoting talent in other members of the Lab.

[36] There was a parallel activity in the IBM Böblingen Laboratory to develop a PL/I compiler for smaller IBM/360 machines; see Albert Endres' history of early language development in Europe at IBM [Endres, 2013].

[37] Personal communication, 2016.

[38] Maurer; Kurt Walk; Erich Neuhold (personal discussions in December 2016).

Kurt Walk was the only member of the lab to speak at the 1964 conference, although the conference programme [IFIP, 1964] shows that many of the others served as 'scientific secretaries' and minders to the speakers. Neuhold remembers this conference as a great learning experience for the members of the laboratory, as they had the opportunity to be exposed to some of the best minds in the field. Peter Lucas agrees, noting in his history of VDL [Lucas, 1981] that "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."

One of the Vienna group's first publications on semantics was written very shortly after the Baden-bei-Wien conference: in [Bekič, 1964] Hans Bekič discussed giving the semantics of 'mechanical languages' by reducing them to elementary terms. The initial focus is on expression languages but Section 4 of the report 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". Lucas' history 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 [Bandat, 1965] that set out much of the VDL approach. Lucas also presented a paper on the topic at the IBM (Internal) Programming Symposium at Skytop, Pennsylvania, but that essentially reiterated the material in the Bandat-edited papers.

Zemanek's group had been based initially at the Technical University of Vienna and they had designed and constructed the transistorised Mailüfterl computer. Zemanek however realised that a small group in Vienna could not compete with major efforts on the hardware front and made the wise decision to move the focus of the newly formed Lab from hardware to software. The group had already implemented an ALGOL 60 compiler for Mailüfterl so they (particularly Bekič and Lucas) had experience in compiler development.

The relocation of the group to IBM coincided with IBM's development of the PL/I programming language.[39] The PL/I language was far 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 an overlapping activity in the IBM UK Lab at Hursley (Hampshire) that led to what they themselves dubbed as a *semi*-formal description, published at the end of 1966 in four parts [Beech et al., 1966b, 1967, 1966a, Allen et al., 1966]. The different motivations of the two teams and their interaction are interesting. Documents from the Zemanek archive at the Technical University of Vienna from this period show that the plan was for the Hursley team to first make a shorter, less formal description, to be called ULD-II, and then the Vienna group would create a longer and fully formal definition, to be called ULD-III, with input from Hursley [de Vere Roberts, 1965, Larner and Nicholls, 1965, Bandat et al., 1965].

The Hursley team was led by David Beech who was a Cambridge trained mathematician.[40] The aim of the Hursley effort was to create a description that was precise but readable by compiler developers. This resulted in a description with 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

[39] As recorded in *History of Programming Languages* [Radin, 1981], 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. Note also the title of an early paper discussing the language: 'NPL: highlights of a new programming language' [Radin and Rogoway, 1965].

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

colleagues in Hursley who replied with a numbered sequence of ‘LDH’ notes. As the technical depth of the questions and answers increased, the Vienna group increasingly turned to the formalism they were developing to try to pin down the answers they were receiving from the Hursley definition and Language Control teams. There were also many visits between the two groups and Beech recalled having made seven trips to Vienna in one year (being a keen muscician probably made this more acceptable).

The Vienna PL/I definition 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 [Lee and Delmore, 1969] and the tag VDL stuck. Peter Wegner’s survey article [Wegner, 1972] might have played a part in cementing the name VDL.[41]

The first version of the complete PL/I description in VDL style [PL/I Definition Group of the Vienna Laboratory, 1966] appeared in December 1966; the cover of the report attributes authorship 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 [Lucas et al., 1968a,b, Walk et al., 1968, Alber et al., 1968, Alber and Oliva, 1968, Fleck and Neuhold, 1968] in 1968. This version corrected a number of errors in the first version, updated the object language to include new features developed in the intervening period, and included modelling of some concepts not included previously, such as the axiomatic definition of storage.[42] The final version, ULD-IIIvIII (which postdates Lauer’s ALGOL description) also appeared as a collection of reports [Alber et al., 1969, Walk et al., 1969, Urschler, 1969a,b, Fleck, 1969] in 1969.[43]

So by the time Lauer initiated work on the ALGOL description [Lauer, 1968a], 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[44] 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. 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 large and unwieldy. By tackling a smaller but certainly non-trivial language, it was hoped that the viability of fixing the semantics of languages by abstract interpreters could be established.

3.2 Extent of ALGOL described

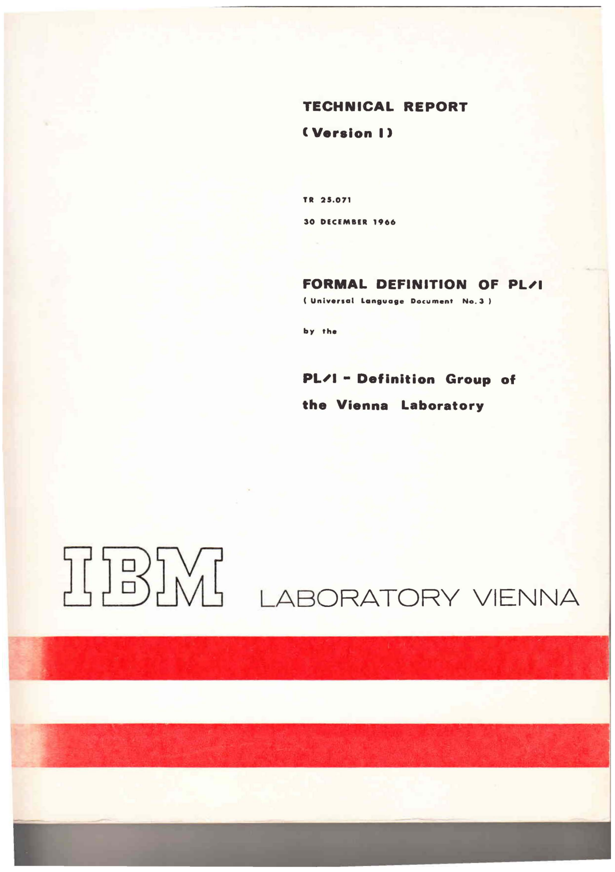
As described in more detail below, Lauer’s VDL description of ALGOL covers all essential features of the language including the complicated issue of ‘own’ variables, and the non-

[41] This survey also puts an emphasis on the notion of VDL ‘objects’ that might surprise a current reader of the material.

[42] For a detailed description of this work on storage formalisation, see a paper presented by Bekič and Walk in 1971 [Bekič and Walk, 1971] since it does not relate to ALGOL.

[43] Versions II and III of ULD-III (as well as ULD-II) have been scanned and are available (see footnote 4); in addition there were a number of revisions made to ULD-III v-III, these are also on the web pages.

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


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

Fig. 2 Copy of the cover of ULD-IIIvI [PL/I Definition Group of the Vienna Laboratory, 1966].

deterministic order of expression evaluation. The version of ALGOL defined is that of the 'Revised Report' [Backus et al., 1963].

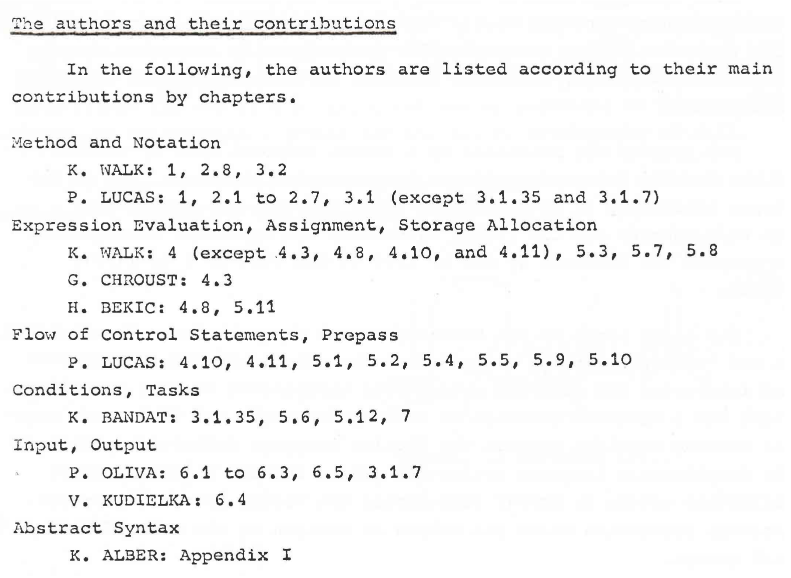
3.3 Syntactic issues

The VDL description of ALGOL follows the approach used for syntax in the definition of PL/I. This section explores some of those issues.

3.3.1 Concrete vs. abstract syntax

See [Lauer, 1968a, §2 & 5]

As with the other languages defined using 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


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

Fig. 3 Copy of the author list of ULD-IIIvI [PL/I Definition Group of the Vienna Laboratory, 1966].

(see p. 2-3,[45] equation 2.4). 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 yielding other objects (which may themselves be elementary or composite). A fuller explanation may be found in the method and notation manual produced with ULD-IIIvII [Lucas et al., 1968b].

The VDL style of abstract syntax follows on from McCarthy's (see Section 2) 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 selector 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.[46]

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.

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

[46] This can be compared with the more straightforward notation for such composite objects in the Vienna group's later VDM style; see Section 6.

3.3.2 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.[47]

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

3.4 Overall semantic style

Operational semantic descriptions are based on the notion of a state which affects the computation and is changed by it. McCarthy used the term “grand state” (in contrast to “small state”) for descriptions in which the state contains information that is not necessary in the sense that it cannot be changed by the state transitions. In a small state description, information that affects but cannot be changed by state transitions is passed as a separate argument to the semantic functions.[48] Lauer follows the VDL norm of using a grand state (see below).

Although there is no concurrency in the ALGOL language, the order of expression evaluation is non-deterministic and functions that can be called from expressions can have side effects; Lauer did therefore have to cope with non-determinism. To model this, he used the concept of a control tree present in VDL descriptions of PL/I. At any point in time, the leaves of the control tree are (equally valid) candidates for execution at the next step. This is known as a “small step” operational semantics because any partially executed function can be interrupted by computation elsewhere in the control tree.

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. Strictly, the non-determinism means that the semantics has to allow a set of possible results:[49]

$$\textit{int-program}\colon AP \times \Xi \to \Xi\textbf{-set}$$

but, in common with other VDL definitions, the description tends towards “non-deterministic functions”.

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

3.5 Specific points

This section explores some of the deeper semantic points in the description.

[47] The later ‘functional’ description discussed in Section 4.3.3 follows a similar system but does have some static checking.

[48] A technical penalty for using grand state descriptions is discussed in Section 4.

[49] 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).

3.5.1 Environment/state

See [Lauer, 1968a, §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 (grand) state. The state (Lauer's report uses ξ for members of Ξ) is split into six components: the denotation directory $\underline{\mathtt{DN}}$, the environment $\underline{\mathtt{E}}$, the dump $\underline{\mathtt{D}}$, the unique name counter $\underline{\mathtt{UN}}$, the control $\underline{\mathtt{C}}$, and the control information $\underline{\mathtt{CI}}$.

Environments link program 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 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 globally-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 sub- or 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 that 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. Lucas' history [Lucas, 1981] and the method and notation guide [Lucas et al., 1968b]) often cite the influence of Peter Landin, and this can be clearly seen in the composition of the state. Landin's SECD (*Stack-Environment-Control-Dump*) machine [Landin, 1964] bears a strong similarity to Ξ. The VDL environment and dump state components have essentially the same functionality as the environment and dump components in Landin's machine; the combination of the control and control information VDL state components share functionality with the stack and control combination 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.[50]

3.5.2 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 param-

[50] See Figure 5 and discussion in Section 7.

eter information of procedures. It is global to the whole program regardless of environment. This enables sharing of values between environments as long as identifiers are passed.

3.5.3 ‘Own’ variables

See [Lauer, 1968a, §4.2].

‘Own’ variables (see Section 1.4) are handled in Lauer’s ALGOL description. A pre-pass 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 accessed or changed any differently from normal variables. The difference in the their handling 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 $\underline{\texttt{DN}}$.

3.5.4 Handling of jumps

See [Lauer, 1968a, §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 McCarthy’s micro-ALGOL description [McCarthy, 1966] interprets jumps (see footnote 30), but is considerably complicated by the phrase structure of full 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.

3.5.5 Procedure value handling

See [Lauer, 1968a, §4.4.2].

The returning of values from type procedure execution environments to their calling environments is not handled during the procedure execution interpretation (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 $\underline{\texttt{DN}}$.

3.5.6 Non-determinism in expression evaluation

The use of the control tree to handle the flow of control within interpretation of a program allows the easy management of non-determinism. This mechanism was invented to cope with the complex concurrency present within PL/I, and so the simpler expression evaluation order question poses no problems. All parts of an expression to be evaluated become leaf nodes in the control tree at the point of evaluation, and therefore any could be correctly chosen as the next interpretation step.

3.6 Postscript on VDL

The VDL description of ALGOL has no problem with higher order functions because procedure denotations are simply a text and an environment and so it is simple to pass one procedure as an argument to another with its denotation text being interpreted as needed.

Although there is no concurrency in ALGOL, the control tree model can—as had been shown in the definition for PL/I—be used to handle non-determinism. This approach does bring considerable (and perhaps unnecessary) complexity to the modelling of a language without concurrency, but would allow handling of concurrency without any addition to the method. Any leaf in the control tree is an equally valid candidate for next execution and so these can be interleaved in any arbitrary order to model concurrency.

Lauer's Ph.D. [Lauer, 1971], under the supervision of Tony Hoare, followed the ALGOL description and showed the consistency of an axiomatic semantics with respect to an operational model.

In addition to the descriptions of PL/I, ALGOL and large parts of FORTRAN [Zimmermann, 1969], J.A.N. Lee published a description of BASIC in 1972 in classic VDL style [Lee, 1972] and a semantics of Prolog was given by Arbab and Berry in 1987 [Arbab and Berry, 1987].

The Vienna Lab was not at this time involved in compiler development but there were certainly thoughts about using formal semantic descriptions as the basis for justifying compiler correctness arguments. The prospect of using a VDL description of an entire language as a hypothesis to a theorem about compiler correctness was unrealistic and Peter Lucas was instead talking about considering separate 'language concepts' and treating correctness issues of compiling individual aspects of languages.

The issue of referring to variables in block structured languages such as ALGOL or PL/I is complicated by both the phrase structure itself and the ability to call procedures (or functions) with the intended semantics that non-local identifiers used within the procedure definition are bound by its statically embracing context. Already in 1968 Lucas had written the *twin machine* report [Lucas, 1968] linking the models used for referring to stack variables in the Hursley ULD-II and the Vienna ULD-III definitions of PL/I.[51] The idea of considering separate language concepts was to bear fruit later; a specific difficulty resulting from the grand state style of operational semantics prompted the next ALGOL description.

Wolfgang Henhapl did use a VDL model as the basis for a *post facto* proof about the mechanism that had been developed by the Hursley Lab (for the PL/I F compiler) to achieve reference to stack variables. The story is perhaps a useful illustration. A number of bugs

[51] Technically, Lucas' approach to establishing equivalence was to combine the states of the two algorithms and link them by what we would call today a data type invariant; having then shown that the combined machine preserves this invariant, Peter used the lovely phrase that one could then "erase the algorithm that one no longer required".

were identified in the attempt to establish correctness of an algorithm which differed from Dijkstra's classic 'display' model. Although the bugs were subsequently corrected, the attitude of the UK developers was that a couple of months of a mathematician was a high price to pay. This might be true if the exercise is evaluated after design but the invention of the original algorithm certainly took more time and yielded a flawed implementation.

4 Vienna functional description

Cliff Jones[52] went on assignment to the IBM Vienna laboratory in August 1968. Before this, he worked in IBM's Hursley laboratory on testing the first PL/I compiler. It must be remembered that PL/I was an extremely large and complicated language but it would not be unfair to say that the experience taught Jones that testing was not a viable way to create a quality product before he had heard Dijkstra's most famous aphorism [in Naur and Randell, 1969, p.21]:

> Testing shows the presence, not the absence of bugs.

The aim of Jones' two-year assignment was to ascertain whether the difficulties seen in the development of the PL/I F compiler could be avoided by basing compiler design on a formal description.[53] A fruit of this investigation was the paper from Jones and Lucas on proving implementation correctness [Jones and Lucas, 1971].[54] In this paper, the proof was that Dijkstra's 'display' mechanism was a valid implementation of a VDL model of referencing stack variables.

Although this showed that proofs about compiling techniques for language concepts could be based on VDL descriptions, it also indicated that it was more difficult than need be: an essential step and key lemma (Lemma 10) in that paper has to show that the environment is the same for the execution of successive statements in a given block even though the first statement could be a nested block or a procedure call whose interpretation requires that a new environment is temporarily used. Because, in a grand state description, a stack of environments is part of the state, the proof of this lemma was gratuitously difficult. It was thus clear, even in operational semantics, that the traditional VDL 'grand state' made reasoning difficult and that a 'small state' approach would be preferable. Thus one stimulus for writing another description of ALGOL semantics was to investigate small state operational semantics.

After finishing his first stay in Vienna, Jones returned to IBM Hursley to take over an 'Advanced Technology' group. Dave Allen joined the group as did Dave Chapman and Peter Gershon. Jones was, at this time, pushing the 'exit' concept explained below, and insisted

[52] b. 1944; worked in the computer industry (including 15 years at IBM) straight out of school and later completed a belated D.Phil. under Tony Hoare at Oxford (Wolfson College) in 1981.

[53] One of Jones' first activities was to review Lauer's ALGOL description prior to its printing, so he had a good degree of familiarity with the VDL.

[54] First available as a Vienna Lab technical report (TR25.110) in August 1970 immediately before Jones moved back to the UK. Two other relevant reports that explored some alternative implementations of the block concept are [Henhapl and Jones, 1970a, 1971]. In 2017, Jones was surprised to note that [Jones and Lucas, 1971] still used Lucas' twin machine approach because he had noted the benefits of a functional relation in [Jones, 1970] (Milner used the more mathematical term 'homomorphism' in [Milner, 1971] and Hoare also used a functional connection in [Hoare, 1972]). VDM later used the term 'retrieve functions' for the connection between representation and abstraction.

that the group's first project[55] should be an ALGOL description using that concept as an illustration of its application to a realistic language: the key outcome was an IBM Technical Report authored by Allen, Chapman, and Jones.[56]

4.1 Background: Why 'functional'

An important motivation for this shift in definition style was to move away from the messy control tree manipulation needed for explicit sequencing; Jones' view, stated in the introduction to the description, was that jumps shouldn't "take the machine by surprise" [Allen et al., 1972, §2.1]. The 'exit' idea was to pre-plan a way of capturing abnormal termination and was first published in a report by Henhapl and Jones in 1970 [Henhapl and Jones, 1970b].[57]

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 a key 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 later VDM models such as those of PL/I [Bekič et al., 1974] and ALGOL [Henhapl and Jones, 1978] because of the use here of the 'copy rule' (see Section 4.5.1 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.

The term 'functional' is used by the authors of the description to distinguish it from the previous VDL work: although ideas are adopted, the central components of the definition are recursive functions passing around small state components, rather than the more monolithic state components of VDL. It should be made clear that this description is not functional in the way that the denotational descriptions of the following chapters are: this description uses functions for interpretation, whereas the denotational descriptions use functions as the denotations of language concepts.

4.2 Extent of ALGOL described

The language defined was the ECMA Subset of ALGOL, which was first described in a letter to the ACM in 1963 [Duncan, 1963] and published in April 1965 by the European Computer Manufacturers' Association [ECMA, 1965].

This is a smaller version of ALGOL 60, designed to be easier to implement across multiple computers. Many of the more contentious elements of ALGOL are removed, such as 'own' variables and recursion (See Section 1.4). 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 [Allen et al., 1972, §1]. Standard functions are included.

[55] Later, before Jones returned to Vienna in early 1973, they worked on an early 'Formal Development Support System': 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.

[56] This report is perhaps best seen as a bridge to the subsequent work in Vienna on denotational semantics (see Section 6); Mosses' reference to [Allen et al., 1972] is one of few indications of impact.

[57] This concept was to play a major part in the VDM style of denotational semantics (see Section 6), but in this 'functional' description it was presented in a rather verbose form.

Non-determinism in expression evaluation is handled, though a 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 4.5.6). The remaining two descriptions also duck non-determinism to some extent.

4.3 Syntactic issues

The 'functional' description of ALGOL uses an approach to syntax influenced by the definition work done by the Vienna group. More detail is given in the subsections below.

4.3.1 Concrete vs. abstract syntax

See [Allen et al., 1972, §5.1.1]

As with the previous VDL description, the semantic description is based on an abstract syntax. Rather than a function to turn abstract syntax into concrete, as presented in Lauer's description, a translation function which takes concrete syntax strings and translates them into abstract objects is envisaged. There are a number of comments about this translator scattered through the description but the translator itself is not completely defined. Any string of correct syntax, as defined by the ALGOL Report, will translate into an abstract object defined by *is-program*.

See [Allen et al., 1972, §3]

The same notation style for abstract syntax from the previous VDL description is maintained (see Section 3.3). 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 easy to check. Interestingly, the description includes a large section on notation, which essentially just duplicates the information in the ULD-IIIvII method and notation guide [Lucas et al., 1968b].

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

4.3.2 Inclusion of sets

See [Allen et al., 1972, §3.7–8]

The move to the exit approach requires keeping labels with their statements, rather than the use of abstract index pointers as in Lauer's description (see Section 3.5.4). 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 prompts 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 constructions is defined fully and formally, so the model depends on the assumption that they can be created.

4.3.3 Context dependencies

See [Allen et al., 1972, §2.4 & 4.3]

Unlike in the previous VDL description, where all error checking was performed dynamically, this description shows the origins of the separation of static error checking from semantics.[58] 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" [Allen et al., 1972, §2.4].

Some of this error checking is presumed to be done by the translator (see Section 4.3.1). Notes are given by some constructs for the translation process: these are typically in the form of predicates featuring implications and rule out some programs which satisfy the context-free syntax but 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.

4.4 Overall semantic style

The main semantic style is an operational, small state approach. The signature of the interpreting function is roughly $Program \times \Sigma \to \Sigma$, where each Σ contains one or more of each of the state-like components *vl*, *dn*, and *Abn*. This is discussed further in the section on state below (Section 4.5.1). In this way, it is similar to the simplicity of McCarthy's semantics. Laying aside notational differences and the specific points discussed 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-program*. This function only has an effect if there is a block which starts the program, reflecting the ALGOL procedural approach.

The physical 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 displayed alongside the ALGOL Report and a formula will align with the relevant section 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.

4.5 Specific points

This section explores some of the deeper semantic intricacies of the description.

4.5.1 Environment/state

See [Allen et al., 1972, §2.2–2.3 & 4.4–4.5]

The issue of separating the environment from the state is actually rather hidden in this description. The semantics for the phrases of the grammar (such as blocks) work in a similar way to that used by mathematicians to describe the λ-calculus' bound variables. The 'copy rule' as described in the ALGOL Report (see Section 1.4) is followed: 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

[58] This eventually led to the "context conditions" seen in the VDM Denotational description (see Section 6.3).

as appropriate using the *change-text* function (see [Allen et al., 1972, §5.4.7]). This makes a direct comparison with descriptions that use an environment difficult.

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.

4.5.2 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 point. 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.[59]

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.

4.5.3 'Own' variables

The contentious 'own' variables were not handled in this description, following the decision to model the ECMA subset of ALGOL.

4.5.4 Handling of jumps

See [Allen et al., 1972, §2.1, 4.1, & 5.4]

The exit mechanism, as first proposed in a technical report by Henhapl and Jones [Henhapl and Jones, 1970b], is used to handle jumps. The essential idea is that interpreting functions $\Sigma \to \Sigma$ become functions $\Sigma \to \Sigma \times [Abn]$, 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 and this approach can seem clumsy and long-winded at times.[60] Nevertheless, the description served as a proof of concept: the exit idea worked for a realistic 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-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-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 halted and the current *vl* and existing abnormal part are returned to the calling *int-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 means that all allowable localities can be checked.

[59] This is precisely the reaction to the problems discussed above with respect to the proofs in [Jones and Lucas, 1971].

[60] This was later resolved by the VDM 'combinators' discussed in Section 6.4.

Once the locality of the label in the abnormal part is reached, the *cue-int-st* function checks whether the current statement has the label in question; if it does, *int-st* is called and interpretation proceeds as normal. Otherwise, *cue-int-unlab-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-int-st*.

4.5.5 Procedure value handling

See [Allen et al., 1972, §4.2, 4.4, & 5.3]

As mentioned in Section 4.5.1, 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 an attempt to follow the ALGOL Report more faithfully.

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.

4.5.6 Non-determinism in expression evaluation

See [Allen et al., 1972, §4.2 & 5.3]

Although the expression evaluation order is well-defined in ALGOL for numerical operations, there are certain sub-expression 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.

4.6 Postscript on Functional Semantics

As indicated in footnote 56, the main impact of this exploration of a small state semantics and the exit mechanism was to be seen in the subsequent Vienna work on VDM. The later use in VDM of the exit combinator (see Section 6) avoids the heaviness of the many case distinctions in this description.

5 Oxford denotational description

Attention now shifts from operational semantics to the denotational method,[61] developed in Oxford in 1969 primarily by Christopher Strachey and Dana Scott,[62] which essentially involves giving meaning to programming languages by defining mappings from language constructs to denotations which are mathematical functions.

In 1974, a Doctoral student under Strachey, Peter Mosses,[63] took on the task of providing a denotational semantics of ALGOL in the Oxford style [Mosses, 1974]. It is interesting to note that Strachey's opinion of ALGOL was not high (indeed, he went so far as to co-author with Maurice Wilkes a paper [Strachey and Wilkes, 1961] 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 the description which start:

> The original inspiration for this report came from reading [the Allen-Chapman-Jones ALGOL description] and [Peter Landin's 'Correspondence' descriptions], 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 idea of using continuations to handle jumps had just been worked out, there was a desire in Oxford's Programming Research Group (PRG) to provide a full semantics of a language with jumps and ALGOL was an obvious choice. Another important driver for the decision to model ALGOL is that Mosses' thesis topic [Mosses, 1975b] was the design of a system that would enable the generation of prototype compilers from a semantic language description; this required formalising the syntax and grammar of the semantic metalanguage. This metalanguage was called the Mathematical Semantics Language, and presented in an MFCS paper [Mosses, 1975a]. The ALGOL description was accordingly written in this MSL style, and used as a proof-of-concept, though Mosses' ALGOL description was never actually run on his 'semantics implementation system'.[64] It is noted in Section 5.1 below that a denotational description of the Sal language must have been largely worked out (by Strachey and Robert Milne in their essay for the Adams Prize [Milne and Strachey, 1973][65]) before Mosses wrote his PRG monograph on ALGOL, but the latter presented much the most ambitious description task tackled by the PRG at that point in time or, in fact, thereafter.

[61] Variously also referred to as 'mathematical' or 'Scott-Strachey' semantics.

[62] b. 1932; American mathematician and logician who studied under Tarski and was awarded the Turing award in 1976 for his joint work with Michael O. Rabin on finite automata.

[63] b. 1948; D.Phil. under Strachey; spent 1976–2004 at Aarhus University in Denmark.

[64] Peter Mosses, personal communication June 2016

[65] This is another document that has been scanned and is available on-line.

5.1 Background: Brief history of the 'denotational' approach

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 Joe Stoy's book [Stoy, 1977] and LNCS paper [Stoy, 1980], Campbell-Kelly's insightful biography [Campbell-Kelly, 1985] and the 2000 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 sketching a brief history.

Both Scott [Scott, 2000] and Penrose [Penrose, 2000] record that Roger Penrose[66] suggested to Strachey in around 1958 looking at Church's λ-calculus. Penrose writes:

> I cannot clearly remember at what stage I tried to persuade Christopher Strachey of the virtues of the lambda calculus. As I recall it, my own ideas were along the lines 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.

It is widely claimed that Lisp 1.5 was based on the λ-calculus, and therefore this must have been some of the inspiration for denotational semantics. However, McCarthy frankly writes in his history of LISP paper presented to the first *History of Programming Languages* conference:

> And so, the way in which to do that was to borrow from Church's λ-calculus, to borrow the λ 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 λ-calculus, or that was the intention. The truth is that I didn't understand the λ-calculus, really.

There is a specific discussion in the same article [McCarthy, 1981, p. 180] on getting the binding rules wrong. McCarthy did use λ-notation in his semantic description of 'Micro-ALGOL', but the key distinction between this and the denotational approach is that in McCarthy's semantics, the λ function forms the semantic interpreting function, whereas in Strachey's approach the λ functions are the denotations in terms of which the programming language is defined.

Christopher Strachey had been interested for a while in formalising the foundations of programming languages and their semantics, spurred on by his experience developing the large programming language CPL. He was, however, not interested in formalising the syntax of programming languages, as the following passage reveals [Strachey, 1966a, quoted in [Peláez Valdez, 1988]]:

[66] b. 1931; renowned mathematician and physicist; Fellow of Wolfson College at the time of its formation (as was Strachey).

> Much of the theoretical work now being done in the field of programming languages is concerned with language syntax. In essence this means the research is concerned not with *what* the language says but with *how* it says it. This approach seems to put almost insuperable barriers in the way of forming new concepts—at least as far as language *meaning* is concerned.

From 1959 to 1964, Strachey employed Peter Landin to work as an assistant at his consulting business (Landin was the only other employee), as recorded in a 1971 CV [Strachey, 1971a], and urged Landin to spend part of his time working on the high-level programming language theory.[67] During this period, Landin produced his classic paper on semantics *The Mechanical Evaluation of Expressions* [Landin, 1964].[68]. He presented an early application of this approach to ALGOL at the 1964 Baden-bei-Wien conference, and an expanded version in his pair of 'correspondence' papers [Landin, 1965a,b].

Strachey's paper 'Towards a Formal Semantics' [Strachey, 1966b] at the same conference[69] discussed how an early version of the denotational approach could be applied to CPL (a language developed by Strachey and others; the main features were laid out in a 1963 article in the Computer Journal [Barron et al., 1963]). Strachey later wrote of the paper "The approach was deliberately informal, and, as subsequent events proved, gravely lacking in rigour—but, in spite of these defects, it certainly laid down the outline of our subsequent work" (quoted in Scott's reminiscences of Strachey [Scott, 2000]).

Dana Scott and Christopher Strachey first met at the April 1969 meeting of IFIP WG 2.2 in Vienna and Scott writes in his memorial paper that he was "struck by Strachey's striving to isolate clear-cut general principles" and found his approach "the most sympathetic of the members of the group" [Scott, 2000].

Scott was invited to visit the IBM Vienna lab in August of the same year. The original intention was that he would help the Vienna group understand Floyd's work on assertions [Floyd, 1967]. It turned out to be a very happy deviation that Scott actually spent most of his stay presenting the work that he had been doing with Jaco de Bakker on the theory of programs at the *Mathematisch Centrum* in Amsterdam [de Bakker and Scott, 1969].[70]

Following his interest in Strachey's work and very soon after their first meeting, Scott spent one semester in Oxford at the PRG (around October 1969–January 1970) working in collaboration with Strachey. Initially, Scott believed that it was impossible to construct a mathematical model of the type-free λ-calculus[71] and presented a typed alternative [Scott, 1969] but, after a sudden inspiration, a succession of foundational papers were written initially as PRG monographs [Scott, 1970, 1971b,a, 1973]. In an interview with the Vienna Logic Lounge [Scott and Traxler, 2015] Scott describes his moment of inspiration as follows:

[67] In this CV, Strachey notes "It is an interesting comment on the state of the subject that this work which at the time was probably the only work of its sort being carried out anywhere (certainly anywhere in England) was being financed privately by me." (also quoted by Martin Campbell-Kelly [Campbell-Kelly, 1985]).

[68] This approach is described in more detail in Section 7.6. As mentioned in Section 3.5.1, this approach was a big inspiration for the early Vienna operational semantics.

[69] 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.

[70] Both Scott and Jones were invited to give papers at the conference in 1983 that marked its renaming to *Centrum Wiskunde & Informatica*.

[71] 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 λ-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.

> 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.

Later, Scott would remark that due to familiarity with work by other logicians in related areas, he actually had all the pieces to put together the model for the *lambda*-calculus in 1957, and that he deeply regretted that it took him so long to make the connections [Scott, 2016].

Both Joe Stoy (Strachey's right-hand man at the PRG) and Peter Mosses recall that Michaelmas term 1969 was one of feverish intensity.[72] Seminars were held weekly on Wednesday afternoons and could last for many hours. Stoy particularly recalls one occasion when an exhausted Scott melodramatically clasped his forehead and exclaimed "Oh, I'm so tired!" two-thirds of the way through one of the seminars.

After the term in Oxford, Scott returned to Princeton; but in 1972 he accepted the new Professor of Mathematical Logic chair and returned to Oxford. 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, 2000]. However, Scott continued to work in the subject, refining models for the untyped λ-calculus.

Another very important technical advance in denotational semantics was the concept of 'continuations' as a way to handle abnormal termination. As recorded by John Reynolds, this concept had independent inventors. Within the PRG, Chris Wadsworth is the originator[73] and published in a joint paper with Strachey [Strachey and Wadsworth, 1974].

Campbell-Kelly reports [Campbell-Kelly, 1985] that it was one of Christopher Strachey's ambitions to be elected a Fellow of the Royal Society. To this end, Strachey wrote to a number of his contacts who tried to provide advice and guidance. One was Strachey's long-time friend and champion Lord Halsbury, to whom Strachey provided lists of people who could be relied upon to support his appointment [Strachey, 1971b], and another was James Hardy Wilkinson, already a Fellow, who advised Strachey that it was hard for computer scientists to get elected, especially if they had relatively few publications [Wilkinson, 1972]. So Wilkinson suggested that Strachey submit an essay for Cambridge University's *Adams Prize* for 1973–1974. This Strachey decided to do, co-authoring with Robert Milne, a Cambridge Ph.D. student with whom he had been working. According to Scott, he might also have been an author, were it not for the fact that all authors on submissions were required to have received a degree from Cambridge at some point [Scott, 2016].

The Milne–Strachey submission, *A Theory of Programming Language Semantics* [Milne and Strachey, 1973], was intended as a comprehensive account of the fundamental concepts in programming languages and how they may be modelled using denotational semantics. It was 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. Tragically, Strachey died suddenly of hepatitis in May 1975, shortly after hearing that the submission had not been awarded the prize.

Robert Milne rewrote the Adams essay in book form [Milne and Strachey, 1976a,b], but as Scott remarks in the introduction to the book described just below, it is vastly different to the book that might have been co-produced with Strachey owing to Milne's own extensive

[72] Talks given at the Strachey 100 centenary symposium in November 2016.

[73] Most denotational semantics publications (e.g. Stoy's textbook [Stoy, 1977]) credit Wadsworth and also Lockwood Morris independently; the story is, however, slightly more complicated: see John Reynolds [Reynolds, 1993] for a fuller history.

contributions to the theory, and the different writing styles of the two men.[74] Joe Stoy, a lecturer at the PRG who had worked closely with Strachey and was the internal examiner for Mosses' D.Phil., published a textbook on the denotational semantics style intended as an easier-to-read introduction to the subject in 1977 [Stoy, 1977], based on the lectures he had taught while on sabbatical at MIT.

5.2 Extent of ALGOL described

See [Mosses, 1974, 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. All other aspects of ALGOL are described, except where noted below (such as non-determinism in expression evaluation).

5.3 Syntactic issues

See [Mosses, 1974, p.5 & C2]

Mosses bases his semantic description on a concrete syntax of ALGOL using "annotated deduction trees", the Scott–Strachey answer to abstract syntax [Scott and Strachey, 1971], 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 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* and *makeBounds* (and their associated implicit selectors) but not for the syntactic classes.

Mosses also makes the point that for the purposes of the ALGOL description, one need not worry about parsing. It could be argued that this approach, as well as that of 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 Mosses' description; so the semantics has to trap type errors dynamically even where they could have been detected statically.

5.4 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, although the situation is somewhat complicated by the use of 'continuations' to handle abnormal exit from phrase structures (see Section 5.5.4 below).

The semantics given is, as far as possible, a homomorphic[75] 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 λ expressions.

[74] Milne remarked in his talk at the Strachey centenary conference that having spent two or three years working on the book, he then returned to the subject only very intermittently in the next 40 years [Milne, 2016].

[75] Meaning that compositionality is preserved: the denotation of $[\![S1;S2]\!]$ is equal to the denotation of $[\![S1]\!] \circ [\![S2]\!]$

The great advantage of using functions in the purely mathematical sense as the basic denotations is that functions 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 inductive reasoning has to be performed over the steps of the interpreting machine.

There are two parts to Mosses' 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 prompted 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 (often 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: $\mathscr{A}[\![t]\!]\rho\theta$. These functions take multiple arguments (a term, an environment and a continuation) but, rather than having a tuple, the arguments are Curried. In his textbook on denotational semantics [Stoy, 1977], Joe Stoy argues that this allows varying levels of detail to be supplied for a slightly different meaning:

$\mathscr{A}[\![t]\!]$	is the meaning of a command *in vacuo*;
$\mathscr{A}[\![t]\!]\rho$	instantiates the variables by adding in an environment;
$\mathscr{A}[\![t]\!]\rho\theta$	adds a continuation, making the command 'ready to go';
$\mathscr{A}[\![t]\!]\rho\theta\sigma$	is a particular execution of the command in a particular state.

5.5 Specific points

This section addresses some specific semantic points in Mosses' description of ALGOL.

5.5.1 Environment/store

See [Mosses, 1974, 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 currently 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 presumably because finite mappings are not considered to be basic objects.

Other sorts of denotations are discussed below.

5.5.2 Shared name space

The *Area* and *Map* objects are used to model the sharing of name spaces within ALGOL. These are passed around the semantic functions as appropriate as part of the store. These stores, which are passed into each semantic function and are denoted by σ in Mosses' model, are objects from the domain S, which contains both an *Area* and a *Map*.

The use of the function *SetArea* removes references in the *Area* to variables which are no longer accessible, though as Mosses points out in the commentary on domains, this is not strictly necessary for ALGOL as there is no clean up.

5.5.3 ‘Own’ variables

As mentioned above, Mosses did not include ‘own’ variables in his description of ALGOL.

5.5.4 Handling of jumps

See [Mosses, 1974, p.13, 24, C18 & C19]

The story of continuations deserves a separate historical account. Fortunately, this has been provided by John Reynolds[76] in a historical paper [Reynolds, 1993] which was updated somewhat in his December 2004 talk at the Computer Conservation Society in London.[77] For the current purposes, the joint paper by Wadsworth and Strachey presenting continuations [Strachey and Wadsworth, 1974] 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 can be 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 Stoy’s textbook [Stoy, 1977] (any example taken from the actual ALGOL description would be somewhat larger):

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

The meaning of Γ_1 followed by Γ_2 in environment ρ with θ as the continuation to the composition is given by the meaning of Γ_1 in environment ρ with its continuation being the meaning of Γ_2 with θ as its continuation. Thus continuations preserve compositionality. Commands are supplied with continuations simply so they have the option of being ignored should an abnormal termination occur.

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.

[76] 1935–2013; a profound contributor whose nominations for the Turing Prize were never rewarded, he was Professor at Syracuse from 1970–86 and at CMU from 1986 until his untimely death.

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

5.5.5 *Procedure value handling*

See [Mosses, 1974, 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.

5.5.6 *Non-determinism in expression evaluation*

See [Mosses, 1974, p.C14 & C16]
Because denotations are functional, Mosses' description (like that from Vienna which is discussed in Section 6) 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.

5.6 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 [Bekič, 1971] before his tragic death in 1982.

Additional to Stoy's textbook [Stoy, 1977], another useful resource is the clear description of denotational semantics given in a paper by Bob Tennent [Tennent, 1976]; this also contains a formal description of Reynolds' Gedanken language.

Both the *Adams Essay* [Milne and Strachey, 1973] and the subsequent book [Milne and Strachey, 1976a,b] tackle the issue of using their formal semantic models (of SAL) to reason about compiler correctness. Had Mosses managed to process his ALGOL description on his SIS software, it would have provided a notable achievement towards the goal of using a formal semantic description for the creation of a compiler. This proved to be beyond what could be done in his time in Oxford. Peter Mosses has continued to work in the field of formal semantics throughout his career, going on to devise, for example, 'Modular Operational Semantics' [Mosses, 2004], and 'Action Semantics' [Mosses, 2005] based on Gul Agha's actor systems for concurrency. Recently, Mosses has been working on a 'Funkons' approach [Mosses and Vesely, 2014] to mechanising formal semantic descriptions based on defining a small set of 'fundamental concepts' and providing mappings from language constructs into these.

6 VDM denotational description

The technical arguments for moving away from 'grand state' semantics were clear in 1970 (see Section 4 above); the evolving 'denotational' ideas were understood; the 'exit' idea had been shown to work for a non-trivial language; what was needed was the opportunity for the Vienna crew to tackle a significant language description to try out their own combination of these ideas. This came about in 1973.

6.1 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 programs. But, in 1972, a new direction for the lab opened with IBM's 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 Radin's report on a potential machine architecture [Radin and Schneider, 1976] 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 an early task was to write a formal description of the ECMA/ANSI version of PL/I which was the language that they were to support.

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. These experiments on a 'design language' were influenced by the fact that Bekič had spent the academic year 1978–79 with Landin at Queen Mary College London and Jones had attended some of Strachey's PRG lectures in 1971–72. Jones moved back to Vienna on a 'permanent transfer' in early 1973. Much of the technical detail about the development of 'VDM'[78] is covered in Jones' paper about the scientific decisions characterising VDM [Jones, 1999] 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.[79]

Overall control of the PL/I for FS project was by Kurt Walk (by this time, Zemanek had been made an IBM Fellow and Walk was Lab director). 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. The project occupied most of the 20 or so professional members of the Lab.

The process of documenting a full VDM denotational description of the ECMA/ANSI subset of PL/I was based on an early construction of the main semantic domains constructed in a long meeting in the coffee room of the third floor of the Lab's location at Parkring, 10. These domains remained basically constant and provided a way for the au-

[78] The name 'Vienna Development Method' was actually coined rather late in the project. There is also a certain ambiguity: to many people, VDM refers to a development method for all forms of computer system (this aspect is placed in a historical context in Jones' *Annals of the History of Computing* paper on the history of program verification [Jones, 2003]); in the current paper, VDM is taken to refer specifically to the technique for language description that evolved in and from work in the Vienna Lab between 1973 and 1976.

[79] An interesting cautionary tale about formal descriptions relates to that of the FS architecture itself. As indicated, the architecture was novel and quite complicated. 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 long and detailed description was of little use to the Vienna Lab as a basis for reasoning about the run-time part of their compiler for the FS machine and Hans Bekič had to write a short formal description to guide the compiler code generation work.

thors to work somewhat independently. At the end of 1974, a Technical Report [Bekič et al., 1974] had been printed. The authors listed for TR25.139 are Hans Bekič, Dines Bjørner, Wolfgang Henhapl, Cliff Jones and Peter Lucas, although ten further colleagues are acknowledged for contributions including detailed reviews. In addition, a collection of further reports discussing aspects of developing compilers from such descriptions were written (see Section 6.6).

On St Valentine's day 1975 the FS machine project was cancelled[80] and it gradually became clear that the next mission of the Vienna Lab would be the development of conventional IBM products. After the shock of seeing several years' work apparently discarded, many of the key researchers began to leave the Lab: Bjørner back to a chair at the Technical University in his native Denmark, Henhapl to a chair in Darmstadt, Germany and finally Jones moved in 1976 to IBM's European System Research Centre in La Hulpe, Belgium.

After the cancellation of FS and thus the PL/I compiler project, Bjørner (then in Denmark) and Jones (in Belgium) agreed to try to preserve and promulgate the VDM denotational style by cajoling their former colleagues to contribute to a book [Bjørner and Jones, 1978] which includes the description of ALGOL written by Wolfgang Henhapl and Cliff Jones [Henhapl and Jones, 1978] that is the subject of this section. The table of contents of this book, printed as *Lecture Notes in Computer Science* 61, is reproduced in Figure 4.[81]

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

Just as Mosses in his introduction provides a slightly backhanded acknowledgement to Allen, Chapman, and Jones, this description 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 [his].

6.2 Extent of ALGOL described

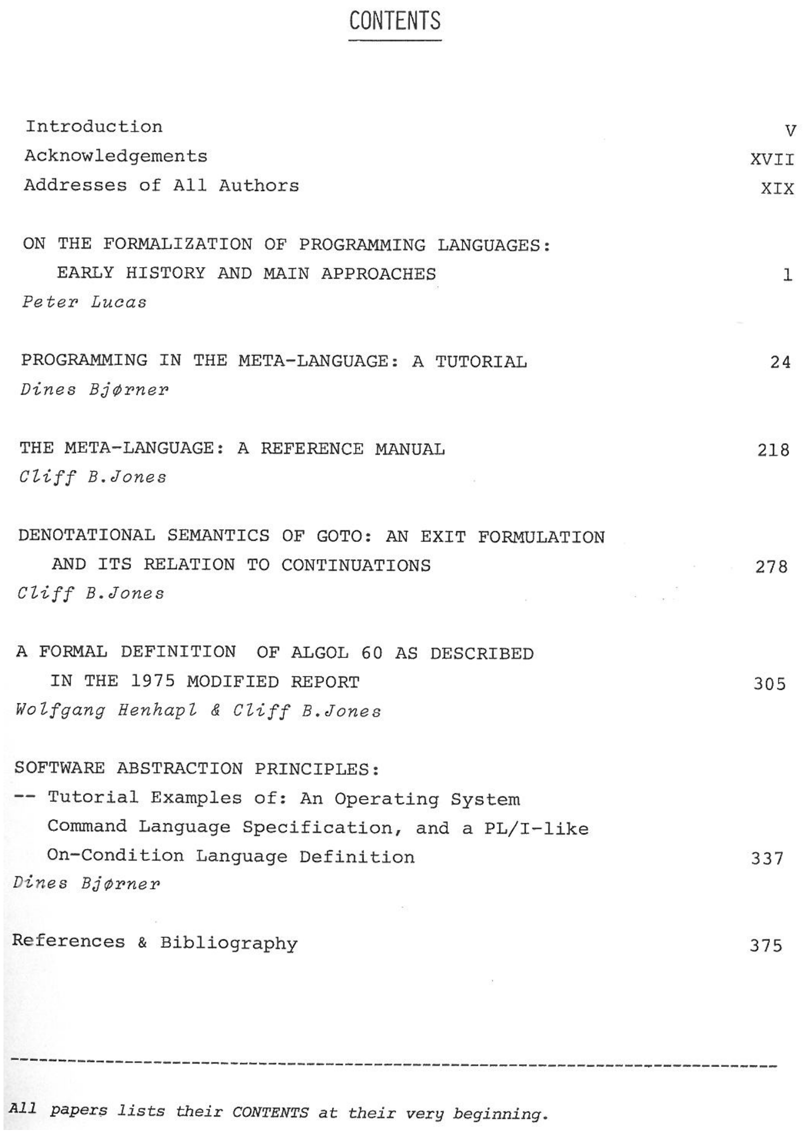
The authors of this description claim to cover all of ALGOL as given in Hill, de Morgan, and Wichmann's 1976 *Supplement to the ALGOL 60 Revised Report* [de Morgan et al., 1976b] that cleared up obscurities in the *Revised Report* [Backus et al., 1963]. In particular, the VDM description does handle 'own' variables, input/output and the so-called 'standard functions' (see Section 1.4). A few comments are offered in the introduction that suggest yet further improvements to ALGOL itself. It is also made clear in the introduction that non-determinism in expression evaluation is not described.

6.3 Syntactic issues

The shift to using VDM represents a considerable change to the VDL approach previously employed by the Vienna group. This sections explores some of the changes.

[80] According to an oral history with Dick Case [Grad, 2006].

[81] 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: [Bjørner and Jones, 1982] contains among other contributions a revised description of ALGOL by the same authors [Henhapl and Jones, 1982]. The revision differs mainly in the order of presentation.



CONTENTS

All papers lists their CONTENTS at their very beginning.

Fig. 4 Copy of the Table of Contents of *LNCS* 61 [Bjørner and Jones, 1978]

6.3.1 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 view of the world in the classic VDL style (see Section 3.3) that is seen with the inclusion of sets as first-class objects in the 'functional description' (Section 4.3.1) had by this time developed into the rich VDM notation: here there were a number of different types which were equally fundamental and linked by a series of predefined operators. VDM includes sets, sequences, maps and records as basic types, which allows sophisticated abstract constructs to be described 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 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.

6.3.2 *Context dependencies*

In common with other VDM descriptions (particularly the PL/I definition [Bekič et al., 1974][82]), 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 determine well-formedness with respect to the declared types of variables. 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. But in ALGOL, type checking cannot be totally static because of array parameter bounds and procedure parameters.

6.4 Overall semantic style

The IBM Vienna Lab had moved completely to a denotational approach to semantics but the appearance of their descriptions differs greatly 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 Vienna and Oxford denotational descriptions can be termed 'exits versus continuations'. Section 5.5.4 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 4.5.4 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 ($\Sigma \rightarrow \Sigma$) 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 loop.

In a denotational setting, functions from states to pairs of states and an optional abnormal component ($\Sigma \rightarrow \Sigma \times [\mathit{Abn}]$) are used as the basic denotations. 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 $s2$.

[82] The PL/I description appears to be the first published use of a completely formalised static error checking system.

– 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 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 (where M is the meaning function):[83] [84]

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

In fact, rather more readable names (e.g. *i-stmt*, *i-block*) were used for semantic functions 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 returns 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 6.5.4 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 a paper by Jones [Jones, 1978] in the Bjørner/Jones book and this is one of the chapters that was significantly revised in the 1982 volume [Jones, 1982]. The proofs are interesting because they tease apart distinct 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 combination of functions, it can be interpreted as an operational definition that first performs the computation before the semicolon followed by that after it.

6.5 Specific points

As mentioned above, the discussion here revolves around the earlier print of the paper [Henhapl and Jones, 1978]; the basic model in the reprint [Henhapl and Jones, 1982] is the same but the description in the latter paper is organised by language construct (as in the 'functional' description discussed in Section 4.4) 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 the later paper [Henhapl and Jones, 1982] as copies are probably easier for the reader to obtain. Another minor difference is that description in the later print [Henhapl and Jones, 1982] employs constructor functions in parameters to overloaded function names. This pattern matching idea makes the description easier to read.

[83] This is close to the style of the Prentice-Hall reprint [Henhapl and Jones, 1982] but the original description [Henhapl and Jones, 1978] used a more long-winded notation.

[84] Peter Mosses in a later paper [Mosses, 2011] points out that the use of combinators in VDM is similar to the later development of Moggi's 'monads' [Moggi, 1989]. 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.

6.5.1 Environment/state

See [Henhapl and Jones, 1982, §6.0].

As is normal in small state descriptions, there is a clear separation between environments and states. In the simplest case, an *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. In line with this requirement for inclusion of elements that can be changed by statement evaluation in the state, the overall state (Σ) has to contain the current values of every *Channel* for the model of ALGOL's input/output statements.

6.5.2 Shared name space

The sharing of name space is made considerably simpler by the separating out of typing and similarly static information into context conditions. The handling of such errors in a static context allows the semantic functions to simply make use of environments, knowing that the meta-information and use of variables will match because only 'well-formed' language parts can be given meaning with the semantics. Thus it is simply the case that an environment is given as a parameter the meaning function for every language construct, and values can thereby be shared.

6.5.3 'Own' variables

See [Henhapl and Jones, 1982, §6.0.4].

'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 for the denotations of 'own' variables. Furthermore, internal unique names are generated to avoid name clashes. As discussed in the paper itself [Henhapl and Jones, 1978, p.307], this model is given in detail because the topic of 'own' variables had been a subject of controversy.

6.5.4 Handling of jumps

See [Henhapl and Jones, 1982, §6.4.4 & 6.1.1].

The overall idea of the exit mechanism is explained above in Section 6.4. 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 any abnormal part present and uses *Mcue* functions to determine the correct place to resume giving meaning to the program.

6.5.5 Procedure value handling

See [Henhapl and Jones, 1982, §6.2.2].

As one would expect from a denotational description, procedures are denoted by functions that are ultimately of type $\Sigma \rightarrow \Sigma \times \left[Sc\text{-}val\right]$. They are Curried to require the denotations of the actual parameters (arguments) and a set of activation identifiers (see Section 6.5.4). The *Sc-val*, representing the return value, is present in the case of functions and **nil** in the case of procedures.

6.5.6 Non-determinism in expression evaluation

As in any denotational semantics, non-determinism cannot be readily handled. This means that this description, in common with Mosses' described in Section 5, fails to describe the option to evaluate expressions in arbitrary order.

6.6 Postscript on VDM

Papers are often cited more than they are thoroughly read (Floyd's 'Assigning meaning' paper [Floyd, 1967] is almost certainly an example); technical reports are perhaps more read than cited. Certainly the VDM definition of PL/I [Bekič et al., 1974] has had more influence than its relatively low citation count would suggest.

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

- PRTV (essentially SQL) [Hansal, 1976].
- The PL/I standard [ANSI, 1976] 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, wheras 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.
- Pascal [Andrews and Henhapl, 1982]: an interesting issue arises here that significantly complicates the model of Pascal and that 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.
- Database programming languages [Welsh, 1982, 1984].
- Smalltalk [Wolczko, 1988]
- The Modula-II standard [Andrews et al., 1988]: (and for once, this is really the defining document).

Although Bekič, Jones, and Lucas had been thinking and communicating about the move to a denotational description method, it has been made clear that the possibility of writing a compiler for the FS machine was the key to reconvening and extending the group. The period 1973–75 was exciting and fun for those involved. The fact that it did not result in a

PL/I compiler developed from the VDM description [Bekič et al., 1974] was caused by cancellation of the ambitious FS machine. Connected with the PL/I for FS compiler project, a number of other technical reports (e.g. [Weissenböck, 1975, Izbicki, 1975, Bekič et al., 1975, Jones, 1976]) describe aspects of compiler development from VDM language descriptions.

7 Concluding comments

Sections 7.1–7.4 offer a more explicit *post facto* comparison between distinctions made in the major Sections 3–6 above; Section 7.5 describes the (limited) tool support used in the creation of the ALGOL descriptions; and Section 7.6 lists some other significant formal language descriptions in the model-oriented camp. Finally, a brief summary of the historical passages of this paper is presented in Section 7.7.

It should be clear from the body of this paper that tackling the semantic description of languages as large as, or larger than, ALGOL requires more than just extra work. Issues such as the interaction between jump statements and block structure, or modelling the unusual lifetimes of "own" variables, all require new thought. Furthermore, the issue of "scale" necessitates taste and care in choice of notational conventions. Remembering that ALGOL posed no challenges concerning concurrency, the attempts described in this paper should offer pause to optimistic assumptions that a concept for formalising concurrency will scale to a language such as Java (certainly, the book on the formal definition of Ada [Bjørner and Oest, 1980] is not for the faint hearted).

7.1 Operational vs. denotational

The general inclusion of the discussions in the proceedings of the *Formal Language Description Languages* conference [Steel, 1966] has been praised above and it is worth drawing attention to one specific figure: Peter Landin prepared a 'categorisation 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 3) and Allen, Chapman, and Jones (Section 4) use an operational semantics approach whereas the descriptions from Mosses (Section 5) and Henhapl and Jones (Section 6) 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. In particular, almost anything which could affect the computation was placed in a 'grand state'; as a consequence, it is unclear when such items can be changed. In contrast, 'small state' descriptions attempt to show things such as environments (mapping identifiers to locations) as arguments to the semantic description and make the major transitions from stores (mapping locations to values) to stores. The 'functional semantics' outlined in Section 4 is a small state description. This observation is important if one considers how a 'Structural Operational Semantics' (SOS, as presented in [Plotkin, 1981] and reprinted as [Plotkin, 2004a]) of ALGOL would be presented.

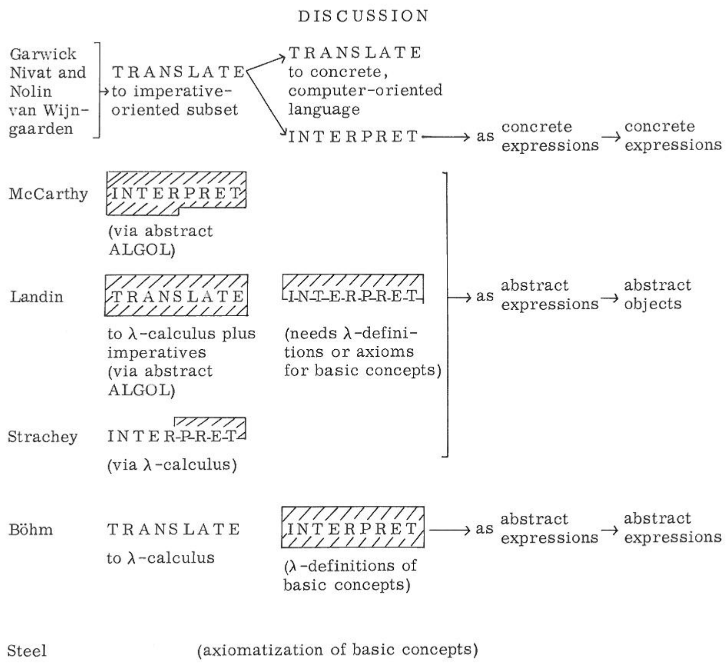

DISCUSSION
Garwick
Nivat and Nolin
van Wijngaarden
TRANSLATE to imperative-oriented subset
TRANSLATE to concrete, computer-oriented language
INTERPRET
as concrete expressions → concrete expressions
McCarthy
INTERPRET (via abstract ALGOL)
Landin
TRANSLATE to λ-calculus plus imperatives (via abstract ALGOL)
INTERPRET (needs λ-definitions or axioms for basic concepts)
as abstract expressions → abstract objects
Strachey
INTERPRET (via λ-calculus)
Böhm
TRANSLATE to λ-calculus
INTERPRET (λ-definitions of basic concepts)
as abstract expressions → abstract expressions
Steel
(axiomatization of basic concepts)

Fig. 5 Landin's "categorisation chart" [Steel, 1966, p.290] of different semantic approaches.

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 onerous for languages that allow concurrency. Plotkin had published the fundamental contribution that proposed power domains as a model for concurrency [Plotkin, 1976] but made the decision to teach an operational approach in his Aarhus course in 1981. His reflections [Plotkin, 2004b] that accompany the republication [Plotkin, 2004a] of his Aarhus lecture notes offer a useful perspective. He writes:[85]

> 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 com-

[85] Interestingly, also in [Plotkin, 2004b] he adds: "I recall Dana [Scott] . . . 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."

posite 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 statements. It is also worth observing that a guideline suggesting that there should be one SOS rule per composite object has a similar effect.[86] 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 be seen when the hypotheses of multiple rules match a given configuration. Despite these caveats, much of the structural clarity of the homomorphic rule can be preserved in SOS description.

7.2 Modelling decisions

All three of the descriptions in Sections 3, 4 and 6 base the descriptions on an abstract syntax; the description in Section 5 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 natural numbers ($\mathbb{N}$); perhaps more questionably, floating point numbers might be shown as reals ($\mathbb{R}$). In either 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 Section 4 and fully exploited in the description in Section 6, provides a significant advantage in the task of giving semantics 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 compilers can 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 Reynolds’ history [Reynolds, 1993]) have an Oxford connection; the ‘exit’ approach originated in a VDL context and is deployed in the VDM model [Henhapl and Jones, 1978]. This can be viewed as a modelling decision because either route fits with the overall denotational approach. Indeed, a paper by Jones in the same book as the ALGOL description [Jones, 1978] establishes the equivalence of the approaches on a language fragment that encompasses the essential challenges of ALGOL.

‘Own’ variables were a contentious subject in ALGOL and two of the descriptions decline to cover them at all. It is quite clear that doing so removes some of complication from the modelling process: in Lauer’s description a pass of the entire program is required and in Henhapl and Jones’ an entirely separate environment is created.

[86] Again from [Plotkin, 2004b]: “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.”

7.3 Fundamental objects

An interesting dimension for comparison of semantic approaches is in their choice of fundamental abstract objects. The growth in the richness in Vienna semantics can be seen starting with their inspiration from McCarthy, through the use of pure objects in VDL, the addition of sets in Allen, Chapman, and Jones, and finally the extensive 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.

McCarthy's approach to abstract syntax uses explicitly-defined constructor and selector functions (see Section 2 for some examples), with predicates describing language constructs as the basic types of the metalanguage. In classic VDL, as seen in Section 3, 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.

The 1972 'functional' semantics maintains the object 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 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 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 [Jones, 1969, §1]. In contrast to McCarthy's explicit relation between constructor functions, predicates and selectors, the Vienna group took a definition such as:

$$\begin{array}{lllll} X & :: & a & : & \mathit{TypeA} \\ & & b & : & \mathit{TypeB} \end{array}$$

to define implicitly the constructor and selector functions:

$$\begin{array}{l} \mathit{mk\text{-}X}\colon \mathit{TypeA} \times \mathit{TypeB} \to X \\ a\colon X \to \mathit{TypeA} \\ b\colon X \to \mathit{TypeB} \end{array}$$

and $x \in X$ could only be true if x was built with *mk-X*.

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:

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

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 [Lucas et al., 1968b]. In fact, rather heavy weather is made of the concept of 'abstract objects' in papers by Zemanek himself and Ollongren [Zemanek, 1968, Ollongren, 1971]. 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 ‘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 a function $Map = N \to V$ (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**.

7.4 Superficial differences

An obvious superficial difference between the descriptions in Sections 5 and 6 is the notation style. In the former, the semantics of ALGOL assignments begins:

```
case"Var := AssL":
    let χ = Main(𝒯_var⟦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.

7.5 Tool support

Each of the ALGOL descriptions contains 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[87] used a system called 'Formula/360' [Schwarzenberger and Zemanek, 1966] 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. Kurt Walk[88] credits Werner Koch with this idea; however, the technical report in which it was published (cited above) acknowledges technical contributions from Peter Lucas and Erich Neuhold, but not from Koch.

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 [Mosses, 1975b]. SIS would have not only type checked the description but could also have provided a prototype implementation. Mosses however informed the current authors[89] that his description of ALGOL was never processed by SIS.

7.6 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 Formal Language Description Languages paper 'A formal description of ALGOL 60' [Landin, 1966], presented in 1964, is an introduction to his later pair of papers [Landin, 1965a,b] which present a correspondence between ALGOL and λ notation. This is achieved by way of an abstract object language called 'imperative applicative expressions' into which both a λ-based model and ALGOL are mapped. The interpretation of these AEs is given by a machine referred to as the SECD (*Stack-Environment-Control-Dump*) machine. This machine is cited in an introduction to the VDL definition of PL/I by Lucas and Walk [Lucas and Walk, 1969] as an inspiration for the state in VDL.
– Rod Burstall's 'set of sentences in first order logic' [Burstall, 1970] describes a major part[90] 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.
– Tony Hoare and Niklaus Wirth in 1973 attempted to write an axiomatic definition of Wirth's Pascal language (in many ways a spiritual successor of ALGOL). This was prompted by Wirth's admiration of the axiomatic technique, but the authors found that there were some rather tricky aspects, and Wirth, in conversation with Daylight [2012], explains "we didn't really finish the job." In particular, problems were caused by some of the more esoteric aspects of the language such as aliases, and also the ever tricky goto.

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

[88] Personal communication, 2016

[89] Personal communication, 2016.

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

Space concerns prevent a full section on other semantic descriptions, but some key references can be found in the conclusions sections of the technical report version of the current paper [Jones and Astarte, 2016].

7.7 Historical summary

The year 1958 saw the publication of the International Algorithmic Language, or ALGOL 58 as it came to be known, which was the first language to use syntax formalism in its definition [Backus, 1959]. This was, however, placed at the end of the paper, and it was not until the definition of ALGOL 60 that formalised syntax was used throughout the document as the main way to define syntax [Backus et al., 1960]. Modifications to the method were made by Peter Naur in this and the Revised Report when it was published a few years later [Backus et al., 1963]. However, despite the formalism used in the syntax, the semantics of the language was still presented in natural language. ALGOL was an extremely influential language, gaining huge traction quickly as the primary computing language for academia and instigating a paradigm shift in the way people thought about programming [Priestley, 2011, Chap. 9].

Around the same time as the rise of ALGOL, the start of the 1960s, many computer scientists were beginning to think about ways of increasing the rigour of programming languages by introducing formalism. One of these was John McCarthy, whose 1962 paper 'Towards a mathematical science of computation' proposed the use of abstract syntax and abstract interpretation machines [McCarthy, 1962]. Another was Peter Landin, who used aspects of λ notation and operational interpreting machines in his 1964 paper 'The mechanical evaluation of expressions' [Landin, 1964], and some follow-ups refining the idea and applying it to ALGOL [Landin, 1965a,b]. In September 1964 a conference was held in Baden-bei-Wien, organised under the auspices of IFIP TC-2 but practically by Heinz Zemanek and his team at IBM. Here, some pioneering ideas were presented, discussed, and argued over by some of the greatest minds in computing [Steel, 1966].

Following this, the IBM Lab in Vienna switched its focus to be primarily on formal semantics of programming languages, and they developed a method known as the Vienna Definition Language. It was first used in a series of reports to define the IBM language PL/I, and later applied to ALGOL as a proof-of-concept [Lauer, 1968a], and as ammunition against those who criticised the method as weighty and unwieldy, which was partly due to its having been applied to the monolithic PL/I.

Around this time, the mid- to late-1960s, Christopher Strachey was refining his ideas for a strongly mathematical approach to the semantics of programming languages, spurred on by his experience of the difficulty of designing the large programming language CPL. The central idea was to think of programs as functions mapping previous storage states into new storage states, but while he had hit on the idea of using λ notation for the functions, he had no sound mathematical model on which to base this idea [quoted in Scott, 2000]. The breakthrough came with a visit by Dana Scott, who visited Strachey at Oxford for a term in autumn 1969 and developed the requisite model using domain theory.

Concurrently, Cliff Jones, who was on secondment to the IBM Lab in Vienna, was working with Peter Lucas on problems with proving correctness of programs defined using VDL [Jones and Lucas, 1971]. The large state approach used in VDL meant that a lot of the proofs were extremely difficult. Another problem with VDL was the control tree's handling of jumps, which tended to "take the machine by surprise" [Allen et al., 1972]. Together with his colleague in Vienna Wolfgang Henhapl, Jones proposed an alternative idea [Henhapl and Jones, 1970b]. Jones soon moved back to IBM Hursley and there was able to try out these new ideas idea by writing, together with colleagues Dave Allen and Dave Chapman, a description of ALGOL 60 [Allen et al., 1972].

Back in Oxford, and now in the early 1970s, the ‘mathematical semantics’ approach by Scott and Strachey was progressing strongly, with the invention by the doctoral student Chris Wadsworth of continuations as a way of handling jumps [Strachey and Wadsworth, 1974]. Another addition came from another student, Peter Mosses, who formalised the syntax of mathematical semantics and worked on developing an implementation system for the semantics [Mosses, 1975b,a]. These ideas were then all put together in a description of ALGOL [Mosses, 1974].

This semantics approach, now more commonly known as ‘denotational semantics’, was heard by Jones and Hans Bekič (who was, at the time, spending time with Peter Landin in London) of IBM with interest. When the task of developing a new PL/I compiler came to the Vienna Lab, they (now including Jones again) eagerly took the opportunity to use a formal definition of the language to aid them. They were keen to try out the new denotational ideas, but were unconvinced by the complexity of continuations and the terseness of the Oxford-style notation. From a fusion of the core denotational concept and the earlier “exit” approach to jumps, combined with a desire to create an intelligible notation system, came the VDM denotational definition of PL/I [Bekič et al., 1974]. When the PL/I project was cancelled, some of those involved managed to rescue the VDM idea and publish some aspects of it in a book; this included another proof-of-concept ALGOL description [Henhapl and Jones, 1978].

Acknowledgements

The authors have benefited from detailed discussions with Peter Mosses, David Beech, Erich Neuhold, Kurt Walk, Gerhard Chroust, John Tucker and Joe Stoy; they are also grateful to Martin Campbell-Kelly for detailed comments on an early draft of the precursor technical report [Jones and Astarte, 2016]. An anonymous referee provided useful comments and Mark Priestley wrote a detailed and constructive review that has prompted significant changes to our initial submission.

Astarte 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.

Jones 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 to Juliane Mikoletzky at the Technical University of Vienna for information concerning the Zemanek *Nachlass*.

References

K. Alber and P. Oliva. Translation of PL/I into abstract syntax. Technical Report 25.086, IBM Laboratory Vienna, ULD-IIIvII, June 1968.

K. Alber, P. Oliva, and G. Urschler. Concrete syntax of PL/I. Technical Report 25.084, IBM Laboratory Vienna, ULD-IIIvII, June 1968.

K. Alber, H. Goldmann, P. E. Lauer, P. Lucas, P. Oliva, H. Stigleitner, K. Walk, and G. Zeisel. Informal introduction to the abstract syntax and interpretation of PL/I. Technical Report 25.099, IBM Laboratory Vienna, ULD-IIIvIII, June 1969.

C. D. Allen, D. Beech, J. E. Nicholls, and R. Rowe. An abstract interpreter of PL/I. Technical Report TN 3004, IBM Laboratory Hursley, ULD-II, November 1966.

C. D. Allen, D. N. Chapman, and C. B. Jones. A formal definition of ALGOL 60. Technical Report 12.105, IBM Laboratory Hursley, August 1972. URL `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/TR12.105.pdf`.

D. Andrews and W. Henhapl. Pascal. In Bjørner and Jones [1982], chapter 6, pages 175–252. ISBN 0-13-329003-4. URL `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/BjornerJones1982/Chapter-7.pdf`.

D. Andrews, A. Garg, S. Lau, and J. Pitchers. The formal definition of Modula-2 and its associated interpreter. In R. E. Bloomfield, L. S. Marshall, and R. B. Jones, editors, *VDM '88 VDM — The Way Ahead*, volume 328 of *Lecture Notes in Computer Science*, pages 167–177. Springer, 1988.

ANSI. Programming language PL/I. Technical Report X3.53-1976, American National Standard, 1976.

B. Arbab and D. M. Berry. Operational and denotational semantics of Prolog. *The Journal of Logic Programming*, 4(4):309–329, 1987.

J. W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *Proceedings of the International Conference on Information Processing*, pages 125–132. UNESCO, 1959.

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, et al. Report on the algorithmic language ALGOL 60. *Numerische Mathematik*, 2(1):106–136, 1960.

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithm language ALGOL 60. *Communications of the ACM*, 6(1):1–17, Jan. 1963. URL `http://homepages.cs.ncl.ac.uk/cliff.jones/publications/OCRd/BBG63.pdf`.

J. W. de Bakker and D. Scott. A theory of programs. Manuscript notes for IBM Seminar, Vienna, August 1969.

K. Bandat. Tentative steps towards a formal description of PL/I. Technical Report 25.056, IBM Laboratory Vienna, July 1965.

K. Bandat, E. F. Codd, R. A. Larner, P. Lucas, J. E. Nicholls, and K. Walk. Unambiguous definition of PL/I. IBM internal memo, October 1965. Technical University of Vienna NL. 14. 072/2 Zemanek. PL/I History Documents.

D. W. Barron, J. N. Buxton, D. F. Hartley, and C. Strachey. The main features of CPL. *Computer Journal*, 6: 134–143, 1963.

D. Beech, J. E. Nicholls, and R. Rowe. A PL/I translator. Technical Report TN 3003, IBM Laboratory Hursley, ULD-II, October 1966a.

D. Beech, R. Rowe, R. A. Larner, and J. E. Nicholls. Concrete syntax of PL/I. Technical Report TN 3001, IBM Laboratory Hursley, ULD-II, November 1966b.

D. Beech, R. Rowe, R. A. Larner, and J. E. Nicholls. Abstract syntax of PL/I. Technical Report TN 3002, IBM Laboratory Hursley, ULD-II, May 1967.

H. Bekič. Defining a language in its own terms. Technical Report 25.3.016, IBM Laboratory Vienna, December 1964.

H. Bekič. Towards a mathematical theory of processes. Technical Report TR 25.125, IBM Laboratory Vienna, 1971.

H. Bekič and K. Walk. Formalization of storage properties. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 28–61. Springer-Verlag, 1971.

H. Bekič, D. Bjørner, W. Henhapl, C. B. Jones, and P. Lucas. A formal definition of a PL/I subset. Technical Report 25.139, IBM Laboratory Vienna, December 1974. URL `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/TR25139/`.

H. Bekič, H. Izbicki, C. B. Jones, and F. Weissenböck. Some experiments with using a formal language definition in compiler development. Technical Report LN 25.3.107, IBM Laboratory Vienna, December 1975.

T. J. Bergin and R. G. Gibson, editors. *History of programming languages—II*. ACM Press, New York, NY, USA, 1996.

K. W. Beyer. *Grace Hopper and the Invention of the Information Age*. The MIT Press, 2009. ISBN 026201310X, 9780262013109.

D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer-Verlag, 1978.

D. Bjørner and C. B. Jones, editors. *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.

D. Bjørner and O. N. Oest. *Towards a formal description of Ada*, volume 98 of *LNCS*. Springer, 1980.

R. M. Burstall. Formal description of program structure and semantics in first-order logic. *Machine Intelligence*, 5:79–98, 1970.

M. Campbell-Kelly. Christopher Strachey, 1916–1975: A biographical note. *Annals of the History of Computing*, 7:19–42, January 1985.

E. G. Daylight. *The Dawn of Software Engineering: From Turing to Dijkstra*. Lonely Scholar, Belgium, 2012. ISBN 9491386026, 9789491386022.

E. W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Communications of the ACM*, 11 (3):147–148, 1968.

F. G. Duncan. ECMA subset of ALGOL 60. *Communications of the ACM*, 6(10):595–599, Oct. 1963.

F. G. Duncan. Our ultimate metalanguage: an afterdinner talk. In *Formal Language Description Languages for Computer Programming*, pages 295–295. North-Holland, 1966.

ECMA. Standard ECMA-2 for a subset of ALGOL. Technical report, European Computer Manufacturers' Association, 1965.

A. Endres. Early language and compiler developments at IBM Europe: A personal retrospection. *Annals of the History of Computing, IEEE*, 35(4):18–30, 2013.

M. Fleck. Formal definition of the PL/I compile time facilities. Technical Report 25.095, IBM Laboratory Vienna, ULD-IIIvIII, June 1969.

M. Fleck and E. Neuhold. Formal definition of the PL/I compile time facilities. Technical Report 25.080, IBM Laboratory Vienna, ULD-IIIvII, June 1968.

R. W. Floyd. On the nonexistence of a phrase structure grammar for ALGOL 60. *Communications of the ACM*, 5(9):483–484, Sept. 1962.

R. W. Floyd. Assigning meanings to programs. In *Proc. Symp. in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

K. A. Fröschl, G. Chroust, and J. Stockinger, editors. *In memoriam Heinz Zemanek*, volume Band-311. OCG, 2015. ISBN 978-3-903035-00-3.

B. 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.

A. Hansal. A formal definition of a relational data base system. Technical Report UKSC 0080, IBM UK Scientific Centre, Peterlee, Co. Durham, June 1976.

W. Henhapl and C. B. Jones. The block concept and some possible implementations, with proofs of equivalence. Technical Report 25.104, IBM Laboratory Vienna, April 1970a.

W. Henhapl and C. B. Jones. On the interpretation of GOTO statements in the ULD. Technical Report 25.3.065, IBM Laboratory Vienna, March 1970b.

W. Henhapl and C. B. Jones. A run-time mechanism for referencing variables. *Information Processing Letters*, 1(1):14–16, 1971.

W. Henhapl and C. B. Jones. A formal definition of ALGOL 60 as described in the 1975 modified report. In Bjørner and Jones [1978], pages 305–336. URL http://homepages.cs.ncl.ac.uk/cliff.jones/publications/OCRd/HJ82.pdf.

W. Henhapl and C. B. Jones. ALGOL 60. In Bjørner and Jones [1982], chapter 6, pages 141–174. ISBN 0-13-329003-4. URL http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/BjornerJones1982.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10): 576–580, 1969.

C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

C. A. R. Hoare. Hints on programming language design. Technical report, Stanford University, Stanford, CA, USA, 1973.

C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2(4):335–355, 1973.

G. van den Hove. On the origin of recursive procedures. *The Computer Journal*, 2014.

IFIP. Working Conference Vienna 1964. Formal Language Description Languages. Program. Christopher Strachey Collection, Bodleian Library, Oxford. Box 287, E.39, February 1964.

H. Izbicki. On a consistency proof of a chapter of a formal definition of a PL/I subset. Technical Report 25.142, IBM Laboratory Vienna, February 1975.

C. B. Jones. A comparison of two approaches to language definition as bases for the construction of proofs. Technical Report 25.3.050, IBM Laboratory Vienna, February 1969.

C. B. Jones. A technique for showing that two functions preserve a relation between their domains. Technical Report LR 25.3.067, IBM Laboratory Vienna, April 1970.

C. B. Jones. Formal definition in compiler development. Technical Report 25.145, IBM Laboratory Vienna, February 1976.

C. B. Jones. Denotational semantics of goto: An exit formulation and its relation to continuations. In Bjørner and Jones [1978], pages 278–304.

C. B. Jones. More on exception mechanisms. In Bjørner and Jones [1982], chapter 5, pages 125–140. ISBN 0-13-329003-4. URL `http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/BjornerJones1982`.

C. B. Jones. Scientific decisions which characterize VDM. In *FM'99 – Formal Methods*, volume 1708 of *LNCS*, pages 28–47. Springer-Verlag, 1999.

C. B. Jones. The early search for tractable ways of reasoning about programs. *IEEE, Annals of the History of Computing*, 25(2):26–49, 2003.

C. B. Jones and T. K. Astarte. An Exegesis of Four Formal Descriptions of ALGOL 60. Technical Report 1498, Newcastle University School of Computer Science, September 2016.

C. B. Jones and P. Lucas. Proving correctness of implementation techniques. In E. Engeler, editor, *A Symposium on Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 178–211. Springer-Verlag, 1971.

D. E. Knuth. Backus Normal Form vs. Backus Naur Form. *Communications of the ACM*, 7(12):735–736, Dec. 1964.

D. E. Knuth and R. W. Floyd. Notes on avoiding "go to" statements. *Information processing letters*, 1(1): 23–31, 1971.

D. E. Knuth and L. T. Pardo. The early development of programming languages. Technical Report STAN-CS-76-562, Stanford University, 1976.

P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

P. J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Communications of the ACM*, 8(2):89–101, Feb. 1965a.

P. J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part II. *Communications of the ACM*, 8(3):158–167, Mar. 1965b.

P. J. Landin. A formal description of ALGOL 60. In *Formal Language Description Languages for Computer Programming*, pages 266–290. North Holland, 1966.

R. A. Larner and J. E. Nicholls. Plan for development of formal definition of PL/I. IBM internal memo, September 1965. Technical University of Vienna NL. 14. 072/2 Zemanek. PL/I History Documents.

P. 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. Technical Report 25.072, IBM Laboratory Vienna, 1967.

P. E. Lauer. Formal definition of ALGOL 60. Technical Report 25.088, IBM Laboratory Vienna, December 1968a. URL `http://homepages.cs.ncl.ac.uk/cliff.jones/publications/OCRd/Lau68.pdf`.

P. E. Lauer. An introduction to H. Thiele's notions of algorithm, algorithmic process, and graph schemata calculus. Technical Report TR 25.079, IBM Laboratory Vienna, January 1968b.

P. E. Lauer. *Consistent Formal Theories of the Semantics of Programming Languages*. PhD thesis, Queen's University of Belfast, 1971. Printed as TR 25.121, IBM Laboratory Vienna.

J. A. N. Lee. The formal definition of the BASIC language. *The Computer Journal*, 15(1):37–41, 1972.

J. 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.

P. Lucas. Two constructive realisations of the block concept and their equivalence. Technical Report 25.085, IBM Laboratory Vienna, June 1968.

P. Lucas. Formal semantics of programming languages: VDL. *IBM Journal of Research and Development*, 25(5):549–561, 1981.

P. Lucas and K. Walk. On the formal description of PL/I. *Annual Review in Automatic Programming*, 6: 105–182, 1969.

P. Lucas, K. Alber, K. Bandat, H. Bekič, P. Oliva, K. Walk, and G. Zeisel. Informal introduction to the abstract syntax and interpretation of PL/I. Technical Report 25.083, IBM Laboratory Vienna, ULD-IIIvII, June 1968a.

P. Lucas, P. E. Lauer, and H. Stigleitner. Method and notation for the formal definition of programming languages. Technical Report 25.087, IBM Laboratory Vienna, ULD-IIIvII, June 1968b. URL `http://homepages.cs.ncl.ac.uk/cliff.jones/publications/VDL-TRs/TR25.087.pdf`.

J. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

J. McCarthy. A formal description of a subset of ALGOL. In *Formal Language Description Languages for Computer Programming*, pages 1–12. North-Holland, 1966.

J. McCarthy. History of LISP. In R. L. Wexelblat, editor, *History of programming languages*, chapter 4, pages 173–183. Academic Press, 1981.

J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. Technical Report CS38, Computer Science Department, Stanford University, April 1966. See also pages 33–41 Proc. Symp. in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science, American Mathematical Society, 1967.

R. E. Milne. Semantic relationships: Reducing the separation between practice and theory. Unpublished, November 2016. Talk given at Strachey 100 centenary conference.

R. E. Milne and C. Strachey. A theory of programming language semantics. Privately circulated, 1973. An essay submitted for the Adams Prize 1973–74.

R. E. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, 1976a. Part A: Indices and Appendices, Fundamental Concepts and Mathematical Foundations.

R. E. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, 1976b. Part B: Standard Semantics, Store Semantics and Stack Semantics.

R. Milner. An algebraic definition of simulation between programs. Technical Report CS-205, Computer Science Dept, Stanford University, February 1971.

E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University Laboratory for the Foundation of Computer Science, 1989.

R. M. de Morgan, I. D. Hill, and B. A. Wichmann. Modified report on the algorithmic language ALGOL 60. *The Computer Journal*, 19(4):364–379, 1976a.

R. M. de Morgan, I. D. Hill, and B. A. Wichmann. A supplement to the ALGOL 60 Revised Report. *The Computer Journal*, 19(3):276–288, 1976b.

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

P. D. Mosses. The semantics of semantic equations. In A. Blikle, editor, *Mathematical Foundations of Computer Science: 3rd Symposium at Jadwisin near Warsaw, June 17–22, 1974*, pages 409–422, Berlin, Heidelberg, 1975a. Springer Berlin Heidelberg.

P. D. Mosses. *Mathematical Semantics and Compiler Generation*. PhD thesis, University of Oxford, April 1975b.

P. D. Mosses. Modular structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60:195–228, 2004.

P. D. Mosses. *Action semantics*, volume 26. Cambridge University Press, 2005.

P. D. Mosses. VDM semantics of programming languages: combinators and monads. *Formal Aspects of Computing*, 23(2):221–238, 2011.

P. D. Mosses and F. Vesely. Funkons: Component-based semantics in K. In *Rewriting Logic and Its Applications*, pages 213–229. Springer, 2014.

P. Naur. The European side of the last phase of the development of ALGOL 60. In R. L. Wexelblat, editor, *History of programming languages*, chapter 3, pages 92–137. Academic Press, 1981a.

P. Naur. Formalization in program development. *BIT Numerical Mathematics*, 22(4):437–453, 1981b.

P. Naur and B. Randell. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969.

A. Ollongren. A theory for the objects of the Vienna Definition Language. Technical Report 25.123, IBM Laboratory Vienna, September 1971.

M. E. Peláez Valdez. *A gift from Pandora's box: The software crisis*. PhD thesis, University of Edinburgh, 1988.

R. Penrose. Reminiscences of Christopher Strachey. *Higher-Order and Symbolic Computation*, 13(1):83–84, 2000.

A. J. Perlis. The American side of the development of ALGOL. In R. L. Wexelblat, editor, *History of programming languages*, chapter 3, pages 75–91. Academic Press, 1981.

PL/I Definition Group of the Vienna Laboratory. Formal definition of PL/I (Universal Language Document No. 3). Technical Report 25.071, IBM Laboratory Vienna, ULD-IIIvI, December 1966.

G. D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5:452–487, September 1976.

G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, July–December 2004a.

G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, July–December 2004b.

M. Priestley. *A science of operations: machines, logic and the invention of programming*. Springer Science & Business Media, 2011.

G. Radin. The early history and characteristics of PL/I. In R. L. Wexelblat, editor, *History of programming languages*, pages 551–589. Academic Press, 1981.

G. Radin and H. P. Rogoway. NPL: highlights of a new programming language. *Communications of the ACM*, 8(1):9–17, 1965.

G. Radin and P. Schneider. An architecture for an extended machine with protected addressing. Technical Report 00.2757, IBM Poughkeepsie Lab, May 1976.

B. Randell. *The origins of digital computers: selected papers*. Springer, 2013.

J. C. Reynolds. The discoveries of continuations. *Lisp and symbolic computation*, 6(3-4):233–247, 1993.

F. Schwarzenberger and H. Zemanek. Editing algorithms for texts over formal grammars. Technical Report 25.066, IBM Laboratory Vienna, July 1966.

D. Scott. A type-theoretical alternative to CUCH, ISWIM, OWHY. Typed script – Oxford, October 1969.

D. Scott. The lattice of flow diagrams. Technical Report PRG-3, Oxford University Computing Laboratory, Programming Research Group, November 1970.

D. Scott. Continuous lattices. Technical Report PRG-7, Oxford University Computing Laboratory, Programming Research Group, August 1971a.

D. Scott. The lattice of flow diagrams. In *Symposium on Semantics of Algorithmic Languages*, pages 311–366. Springer-Verlag, 1971b.

D. Scott. Models for various type-free calculi. In P. Suppes, L. Henkin, A. Joja, and G. Moisil, editors, *Studies in Logic and Foundations of Mathematics Vol. 74 (Proc. of the 4th International Congress for Logic, Methodology and Philosophy of Science, Bucharest, 1971)*, pages 158–187. North Holland Publishing Company, 1973.

D. Scott. Some reflections on Strachey and his work. *Higher-Order and Symbolic Computation*, 13(1): 103–114, 2000.

D. Scott. Greetings to the participants at "Strachey 100". A talk read out at the Strachey 100 centenary conference, November 2016.

D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. Technical Monograph PRG-6, Oxford University Computing Laboratory, Programming Research Group, 1971.

D. Scott and T. Traxler. Logic Lounge with Dana Scott. Online, June 2015. URL `https://www.youtube.com/watch?v=nhc94A829qI`. Video interview.

L. Shustek. An interview with Fred Brooks. *Communications of the ACM*, 58(11):36–40, Oct. 2015.

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

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

J. E. Stoy. Foundations of denotational semantics. In D. Bjørner, editor, *Abstract Software Specifications: 1979 Copenhagen Winter School January 22 – February 2, 1979 Proceedings*, pages 43–99. Springer Berlin Heidelberg, 1980.

C. Strachey. System analysis and programming. *Scientific American*, 215:112–124, 1966a.

C. Strachey. Towards a formal semantics. In *Formal Language Description Languages for Computer Programming*. North Holland, 1966b.

C. Strachey. Jumping into and out of expressions, August 1970. Unpublished note. Strachey Papers, Bodleian Library, Oxford. Folder C229.

C. Strachey. Curriculum vitae, December 1971a. Written by Strachey to send to the Times newspaper in case of the need for obitual information. Strachey Papers, Bodleian Library, Oxford. Folder A3.

C. Strachey. Letter to Lord Halsbury, October 1971b. Strachey Papers, Bodleian Library, Oxford. Folder A3.

C. Strachey. The varieties of programming language. Technical Monograph PRG-10, Oxford University Computing Lab, March 1973.

C. Strachey and C. P. Wadsworth. Continuations – a mathematical semantics for handling jumps. Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, January 1974.

C. Strachey and M. V. Wilkes. Some proposals for improving the efficiency of ALGOL 60. *Communications of the ACM*, 4(11):488–491, Nov. 1961.

R. D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19: 437–453, 1976.

G. Urschler. Concrete syntax of PL/I. Technical Report 25.096, IBM Laboratory Vienna, ULD-IIIvIII, June 1969a.

G. Urschler. Translation of PL/I into abstract syntax. Technical Report 25.097, IBM Laboratory Vienna, ULD-IIIvIII, June 1969b.

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

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

M. de Vere Roberts. Radiogram to Kurt Bandat. IBM internal memo, September 1965. Technical University of Vienna NL. 14. 072/2 Zemanek. PL/I History Documents.

C. P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, Programming Research Group, University of Oxford, September 1971.

K. Walk, K. Alber, K. Bandat, H. Bekič, G. Chroust, V. Kudielka, P. Oliva, and G. Zeisel. Abstract syntax and interpretation of PL/I. Technical Report 25.082, IBM Laboratory Vienna, ULD-IIIvII, June 1968.

K. Walk, K. Alber, M. Fleck, H. Goldmann, P. E. Lauer, E. Moser, P. Oliva, H. Stigleitner, and G. Zeisel. Abstract syntax and interpretation of PL/I. Technical Report 25.098, IBM Laboratory Vienna, ULD-IIIvIII, April 1969.

P. Wegner. The Vienna definition language. *ACM Computing Surveys (CSUR)*, 4(1):5–63, 1972.

F. Weissenböck. A formal interface specification. Technical Report 25.141, IBM Laboratory Vienna, February 1975.

A. Welsh. The specification, design and implementation of NDB. Master's thesis, Department of Computer Science, University of Manchester, October 1982. Also published as tech. report UMCS-82-10-1.

A. Welsh. *A Database Programming Language: Definition, Implementation and Correctness Proofs*. PhD thesis, Department of Computer Science, University of Manchester, October 1984. Also published as tech. report UMCS-84-10-1.

R. L. Wexelblat, editor. *History of programming languages*. Academic Press, 1981.

B. Wichmann. The ALGOL bulletin, February 2004. URL `http://archive.computerhistory.org/resources/text/algol/algol_bulletin/`.

J. H. Wilkinson. Letter to Christopher Strachey, December 1972. Strachey Papers, Bodleian Library, Oxford. Folder A3.

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

H. Zemanek. Abstrakte objekte. *Elektron. Rechenanl.*, 5:208–217, 1968.

Y. Zhang and B. Xu. A survey of semantic description frameworks for programming languages. *ACM Sigplan Notices*, 39(3):14–30, 2004.

K. Zimmermann. Outline of a formal definition of FORTRAN. Technical Report 25.3.053, IBM Laboratory Vienna, 1969.