SCHOOL OF
COMPUTING

Title: Challenges for semantic description:
Comparing responses from the main approaches

Names: Cliff B. Jones and Troy K. Astarte

TECHNICAL REPORT SERIES

No. CS-TR- 1516 November 2017

TECHNICAL REPORT SERIES

No: CS-TR- 1516 **Date: November 2017**

Title: Challenges for semantic description:
Comparing responses from the main approaches

Authors: Cliff B. Jones and Troy K. Astarte

Abstract:

Although there are thousands of programming languages, many of them share common features. This paper reviews some key underlying language concepts and the challenges they present to the task of formal se-mantic description. Most material concerns the responses in operational, axiomatic and denotational approaches to the description of programming languages. There are interesting overlaps between the responses of these main approaches to the challenges posed by the language concepts: these similarities are exposed even where accidental details often disguise them; essential differences are also pinpointed.

Printed and published by Newcastle University,
School of Computing, Urban Sciences Building,
Newcastle upon Tyne, NE4 5TG, England.

Bibliographical details

Title and Authors

NEWCASTLE UNIVERSITY
School of Computing. Technical Report Series. CS-TR- 1516

Abstract:

Although there are thousands of programming languages, many of them share common features. This paper reviews some key underlying language concepts and the challenges they present to the task of formal se-mantic description. Most material concerns the responses in operational, axiomatic and denotational approaches to the description of programming languages. There are interesting overlaps between the responses of these main approaches to the challenges posed by the language concepts: these similarities are exposed even where accidental details often disguise them; essential differences are also pinpointed.

About the Authors:

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

and IET. He has been a member of IFIP Working Group 2.3 (Programming Methodology) since 1973.

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

Suggested keywords: Programming languages, formal semantics

Challenges for semantic description: comparing responses from the main approaches

Cliff B. Jones and Troy K. Astarte

November 30, 2017

Abstract

Although there are thousands of programming languages, many of them share common features. This paper reviews some key underlying language concepts and the challenges they present to the task of formal semantic description. Most material concerns the responses in operational, axiomatic and denotational approaches to the description of programming languages. There are interesting overlaps between the responses of these main approaches to the challenges posed by the language concepts: these similarities are exposed even where accidental details often disguise them; essential differences are also pinpointed.

1 Introduction

There are a number of different approaches to giving formal descriptions of the semantics of programming languages, but most can be placed into one of three styles: operational, denotational, or axiomatic. Any approach to describing semantics formally must find ways to tackle a set of challenges derived from common features in programming languages, such as nested blocks or concurrency. In this paper, an initially simple illustrative language is described using all three approaches, and remarks are made about how they address the particular challenges. It is interesting to note the degrees of similarity present in some responses given the apparent conceptual differences between approaches.

The paper begins by setting out some reasons for considering semantics and introducing the kernel of the example language. Simple applicative languages are considered first, and some conclusions are drawn that are relevant to imperative languages. Throughout the paper, new features for the example language are considered and the formal semantic descriptions of these features are discussed. Finally, a concurrent, object-oriented language is introduced as a vehicle to illustrate the combination of all the features covered; an operational semantics for such a language is sketched.

Note that this is not intended to be a historical paper. Readers interested in such a view of formal semantics could read [JA16] which examines four historical full semantic descriptions of ALGOL 60 and draws some conclusions. A

more complete treatment of the history of programming language semantics is forthcoming as Astarte's PhD thesis.

1.1 Why describe semantics formally

It is worth beginning by reviewing the reasons for pursuing formal semantics. Unlike natural languages, programming languages are formal objects which means they rigidly follow a fixed (and fairly small) set of rules that govern their structure and behaviour.

It is important that the many different users of the language, from programmers through standard writers to compiler creators, all understand what these rules are — and, particularly for compilers, as precisely as possible.[1] Natural language can be (and is) used for this purpose, but words are always ambiguous and can all too easily lead to contradictions or omissions. Therefore, formality is frequently utilised — and even in natural language descriptions, the careful wording used ultimately ends up in formality regardless of notation [Tur09]. Another advantage to the use of formalism is that it can help ensure completeness: if there is a form to be followed for every language construct, the chances of accidentally omitting part of a language is significantly lowered.

This is not to suggest that a formal description always defines one unique result for a program in a language: it is often necessary to leave certain parts of the description undefined in order to allow for implementation specifics and non-determinism at run time [Hoa69]. Carefully and formally delineating these areas of non-definition is, however, essential.

In addition to being formal, a useful programming language semantics must also be *tractable* — it must enable proofs to be made about the language itself, about the correctness of implementations of the language and about programs written in the language [Bur66]. Ideally, a good semantics allows the proof of deep properties, many of which are relied upon in compiler optimisation. Different approaches to semantics tend to make different properties easier to prove than others [Gor75].

Arguably an even more important use of formal semantics is in the design of programming languages [MS76]. Currently there exist thousands of programming languages, most of which are sadly lamentable. Even some of the best exhibit *feature interaction*, where features that are useful and straightforward when taken separately lead to arcane behaviour when combined. The use of a formal semantics during the creation of a language — ideally, before even any syntax is created — can contribute greatly to the simplicity and clarity of the resultant language. Unfortunately, formal semantics is typically applied *post facto* to extant languages.[2]

[1] See, for example, the work of the IBM Laboratory Vienna on producing formal definitions of PL/I for use in compiler writing, such as [BBH+74, Jon76].

[2] Encouraging exceptions include the Turing programming language [HMRC87], Standard ML [HMT87], and SPARK-Ada [CG90]. Furthermore, formal semantics played an important role in the development of Ada itself [BO80]. Formal description was also utilised in the standards for Modula-2 [Woo93] and PL/I [ANS76].

The choice of semantic description approach is often motivated by the intended use of the semantics. Received wisdom generally holds that operational semantics is most useful to compiler writers, denotational to the language designer and axiomatic semantics in program verification. However, some writers have pointed out that the distinction is not always as clear cut as this [Ame89].

Of course, the challenge of describing the semantics of a modern programming language is far greater than for, say, first-order predicate calculus. Researchers have learnt what they can from previous work by logicians and carried these lessons forward: the extensions involved are challenging and interesting.

1.2 Main approaches

The main focus in this paper is on operational, axiomatic and denotational semantics; Section 2 illustrates the differences in these approaches on a core language but it is worth briefly characterising the approaches here.

An operational semantics describes the meaning of a language in terms of an *abstract interpreter*, that takes a program and a starting state and computes allowed final states. Typically, the interpreter will be defined in terms of subfunctions for each construct in the language. The *states* of the interpreter are chosen to eschew unnecessary details.

The essence of a denotational semantics is to map a language into some space of mathematically tractable objects. For simple programming languages these objects are mathematical functions from states to states. Denotational descriptions present a series of mappings from program constructs into these functions. A key feature is the notion that the mapping should be *homomorphic*: the function denoted by a program segment should be composed from the denotations of its components.

The previous two approaches both make the notion of state explicit and can thus be viewed as model-based, In contrast, property-oriented descriptions attempt to fix semantics without an explicit state.[3] An axiomatic semantics contains axioms and rules of inference that define a set of *judgements*. In Floyd/Hoare semantics of procedural languages, the judgements are triples in which the middle component is a text in the language being described; the first and third components are predicates. The interpretation of such a triple is that if the first predicate (the pre condition) is satisfied and the text is executed to termination, then after execution the post condition will be true.

Here the notion of state is only implicit in the meta-variables used within these assertions. Axiomatic semantics is particularly concerned with proving properties of programs, and if an axiomatic specification of a language allows the proving of any true property (and no false property) of a program construct, then the construct is considered fully specified [Pag81]. If every part of the language is specified in this way, then the specification constitutes a semantics of the language. In practice, it turns out to be difficult to fully specify large-scale programming languages purely by axioms.

[3] 'Algebraic semantics' can also be viewed as property-oriented and is briefly discussed in Section 7.1.

1.3 Applicative languages

The majority of this paper is concerned with imperative programming languages (as characterised in Section 2). There are, however, some interesting semantic description techniques that can be carried over from handling applicative languages. Two common challenges are that the languages whose semantics are to be given have an unbounded number of admissible texts and that comprehensibility of the semantic description is a major objective.

One class of applicative programming language is functional programming languages and these –at least if they are *purely* functional– avoid some of the challenges that have to be faced with the semantics of languages that feature assignment-like constructs. Assignments require some model of storage, usually considered as an abstract meta-notion 'state'; avoiding assignment allows programs in functional languages to be reasoned about as though they are conventional recursive functions. There might, of course, be a performance penalty in using purely functional languages, but that discussion is beyond the scope of the present paper.

It is important to remember that all programming languages provide a repertoire of basic operators and, crucially, put in the hands of programmers ways to express things that extend this repertoire. Thus a programmer might write a program that computes factorial using only basic arithmetic operators; more ambitiously, a program for inverting matrices can be be written in a language that has no such operator.

A first-order predicate language is a simple and traditional applicative language and discussing its semantics facilitates deriving the messages that are worth taking forward to the subsequent sections of this paper. Starting with purely propositional expressions, a semantic function could be written that recurses through the structure of the expressions,[4] building up the meaning of the expression as a whole by combining the meaning of its parts. Ultimately, this function must rely on an association of the propositional identifiers with truth values. Similarly, with predicate calculus, there must be a way to determine the meaning of any predicates or functions. It is important to observe that these two sorts of associations remain fixed and can be stored in some form of static environment.

There are, of course, other ways of tackling the semantics of logical languages. In an equivalence-based strategy, some operators can be defined in terms of others (e.g. $p \Rightarrow q$ can be defined as $\neg p \vee q$). There must however be a minimum set of basic operators (e.g. the Sheffer stroke).

Classical axiomatisations (such as that in [Men64, §1.4]) are unintuitive but natural deduction rules like those presented in [Pra65] provide both a semantics and some intuition as to how to reason about expressions in logical languages.

The responses to be carried forward to the review of semantic description techniques for imperative languages are then:

[4] This task would be made easier with the use of an abstract syntax, a concept discussed later in this paper.

- Environments — what information is stored about identifiers; in what form; and how distinction is made between different denotations (e.g. identifier-value and function-definition pairs).
- Fundamental bases of meaning — saying one has, for example, 'a Boolean Algebra' doesn't fix (all of) the semantics because multiple such algebras exist.
- Understandability of description — as with deduction systems, semantic descriptions should be evaluated for intuition and usability for reasoning.

1.4 A core imperative language

A basic challenge to be faced, even before addressing the semantics of a language, is to delimit the admissible utterances of the language. Although normally presented in two dimensional layout, it is still common to think of programs as strings of characters. Some version of Backus-Naur Form notation is adequate to define the set of (context-free) strings of most programming languages: this is known as *concrete syntax*. However, following Christopher Strachey's advice to 'know what you need to say before deciding how to write it down', many semantic descriptions are based instead on an *abstract syntax*. This approach follows John McCarthy's proposal [McC62] although VDM notation is employed below. The advantages of using an abstract syntax over concrete may be less apparent for a small language like the one considered here, but for large languages, especially those with multiple syntactic forms of the same semantic construct, the benefits become more apparent. Use of abstract syntax shows concern with the *structure* of the language (rather than its form) and also allows greater flexibility in implementation. The higher level of abstraction meshes nicely with more abstract semantic approaches; however, following tradition, examples of axiomatic semantics below are built around concrete syntax.

Figure 1 contains the abstract syntax of a core of the various languages discussed in this paper. Later sections in the paper add to this core to illustrate more complex language concepts and the challenges inherent in modelling these features.

Some difference in approaches come from the defining context-dependent checks such as required consistency between uses and declarations of names. These can be handled within semantics (i.e. dynamically), but it is normally more fruitful to handle these issues statically. Such static checks are called *context conditions* after Aad van Wijngaarden *et al.* in the ALGOL 68 Report [vWMPK69]. Various methods for defining these kinds of checks have been developed by van Wijngaarden (two-level grammars), Knuth (attribute grammars [Knu68]), and researchers at the IBM Hursley Laboratory (dynamic syntax [HJ73]); a more thorough study would be [GP99] or [Pie02] on Type theory. Full exploration of this topic is beyond the scope of the current paper.

Context conditions in the VDM style are written as *well-formedness* predicates of the form $wf\text{-}Construct\colon Construct \times TypeMap \to \mathbb{B}$ and use an abstract *TypeMap* object of the type $Id \xrightarrow{m} ScalarType$ inherited from *Program*. These

$$
\begin{array}{l}
Program :: types : Id \xrightarrow{m} ScalarType \\
\qquad\qquad\ \ body\ : Stmt
\end{array}
$$

$$Stmt = Assign \mid If \mid While \mid Compound \mid \cdots$$

$$
\begin{array}{l}
Assign :: lhs\ : Id \\
\qquad\qquad\ \ rhs\ : Expr
\end{array}
$$

$$
\begin{array}{l}
If :: test\ : Expr \\
\qquad\ \ then\ : Stmt \\
\qquad\ \ else\ : Stmt
\end{array}
$$

$$
\begin{array}{l}
While :: test\ : Expr \\
\qquad\qquad\ \ body\ : Stmt
\end{array}
$$

$$Compound :: Stmt^*$$

Figure 1: Abstract syntax of a core language

functions generally check that the types assigned to variables match the variable declaration, and that inappropriate types are not used in expressions (for example, in an *If* statement, the *test* part must be of type $\mathbb{B}$). For constructs which contain sub-components, each such component must also be well formed.

2 Imperative (deterministic) languages

The identifying feature of an imperative programming language is that it provides statements that change things. What is affected differs between languages: changes might be updating a database or moving the position of part of a robot. Here the discussion focusses on the challenge of modelling assignments to variables, but the same principles apply to other kinds of command as long as a compatible abstract model can be created for the target of the changes.

Assignments to variables destroy *referential transparency*: the value associated with an identifier changes during execution; values previous to an assignment are destroyed. Furthermore, the order in which statements are executed becomes important. An imperative program, then, achieves its effect by executing a sequence of assignments; language features such as conditionals and loops orchestrate the execution.

As in applicative languages, programs make it possible to compute results that are not directly available as operators of the language. It therefore follows that a subsidiary challenge is to provide tractable ways of reasoning about the meaning of imperative programs whose specifications include operators that are not basic to the language and which achieve their effect using destructive assignments.

2.1 Operational Semantics

John McCarthy was one of the first to present an operational approach to defining the semantics of programming languages. In his definition of 'Micro-ALGOL' [McC66] he described the approach as "defining a function ... giving the state ... that results from applying the program ... to the [initial state]". McCarthy was also careful to point out in his earlier paper on the topic [McC60] that this is an 'abstract function', because the language in which it is expressed is more abstract than either the language being described or, say, machine assembler code. This approach to semantics is now commonly referred to as an 'abstract interpreter' because it interprets the various constructs of the language under discussion.

The core idea of operational semantics remains the same as when McCarthy first proposed it: meaning is given to a language with an abstract interpreter defined in terms of changes to abstract states. The capital Greek letter Σ is commonly used for the set of such states and, in simple cases, particular states just associate identifiers with values such as Booleans or integers:[5]

$$\Sigma = Id \xrightarrow{m} ScalarValue$$

$$ScalarValue = \mathbb{B} \mid \mathbb{Z}$$

As observed above, the key property of an imperative language is that assignments can change the state. An interpretation function for statements would take as parameters an (abstract) program and a state; its result is a final state. Historically, McCarthy [McC66] and even the Vienna operational descriptions [Lab66] did write such interpretation functions. In the current paper, the SOS style of [Plo81] is used uniformly since this notation copes with non-determinism (cf. Section 4.1) and can thus be used for all of the operational descriptions discussed.

SOS rules like the one below for assignment *can* be read like a classic interpretation function, when considered in a clockwise manner from bottom left, and this often feels more natural when looking at deterministic languages.

However, it is important to remember that SOS rules are in fact *inference rules*: above the line is a series of premises which must all be true for the rule to apply; below the line is the conclusion. Each rule indicates a relation between the state before computation and the state afterwards, given that a series of conditions holds; it records a way of judging whether a particular computation is valid. This distinction becomes important when considering non-deterministic languages, as in Section 4.1.

The basic judgements are relations (thus their signatures have powersets) between pairs of program text and pre-state, and post-computation state:

$$\xrightarrow{st} : \mathcal{P}((Stmt \times \Sigma) \times \Sigma)$$

[5] The use of the type name *ScalarValue* prepares the way for modelling compound types such as arrays below.

The precise way in which each statement in a program is interpreted obviously depends on the type of statement so one way to present a description would be to write an interpretation function by cases. However, it is more convenient to use pattern matching and this approach is used in both operational semantics and denotational semantics:

$$\frac{(rhs, \sigma) \xrightarrow{ex} v}{(mk\text{-}Assign(lhs, rhs), \sigma) \xrightarrow{st} \sigma \dagger \{lhs \mapsto v)\}}$$

(The judgements for expression evaluation ($\xrightarrow{ex}$) are described below.)

Conditional statements are interpreted by cases as follows:

$$\frac{(test, \sigma) \xrightarrow{ex} \textbf{true} \quad (then, \sigma) \xrightarrow{st} \sigma'}{(mk\text{-}If(test, then, else), \sigma) \xrightarrow{st} \sigma'}$$

$$\frac{(test, \sigma) \xrightarrow{ex} \textbf{false} \quad (else, \sigma) \xrightarrow{st} \sigma'}{(mk\text{-}If(test, then, else), \sigma) \xrightarrow{st} \sigma'}$$

Interpreting iterative statements is slightly more involved:

$$\frac{(test, \sigma) \xrightarrow{ex} \textbf{true} \quad (body, \sigma) \xrightarrow{st} \sigma' \quad (mk\text{-}While(test, body), \sigma') \xrightarrow{st} \sigma''}{(mk\text{-}While(test, body), \sigma) \xrightarrow{st} \sigma''}$$

$$\frac{(test, \sigma) \xrightarrow{ex} \textbf{false}}{(mk\text{-}While(test, body), \sigma) \xrightarrow{st} \sigma}$$

Notice that the state used in the third premise is one produced from an interpretation of the body; thus a convergence towards termination may occur. Of course, the semantics has to allow for non-terminating loops!

The basic notion of state used above plays the same role as the environment in a functional language and an evaluation function could be defined to determine the values of expressions.

$eval\colon Expr \times \Sigma \to ScalarValue$

The *eval* function above can be rewritten as a relation:

$\xrightarrow{ex}\colon \mathcal{P}((Expr \times \Sigma) \times ScVal)$

which can be split by the cases in its syntactic classes

$$\frac{e \in Id}{(e, \sigma) \xrightarrow{ex} \sigma(e)}$$

$$\frac{(e1, \sigma) \xrightarrow{ex} v1 \quad (e2, \sigma) \xrightarrow{ex} v2}{(mk\text{-}Expr(e1, \textsc{Plus}, e2), \sigma) \xrightarrow{ex} v1 + v2}$$

(Other cases should be obvious.)

This seemingly simple description actually fixes an important property of the language: the process of evaluating an expression is shown not to change the state (i.e. the values of variables) — the same σ is used throughout. Although the key feature of functions is not addressed until Section 3, it is important to note that functions with side effects would destroy this assumption.

Note that the evaluation of variables does not require a variable to be initialised and, of course, this could cause errors. In order to avoid this problem, all variables can be automatically initialised in the rule for program interpretation. These have been omitted for clarity. An alternative would be to modify the evaluation rule with a premise such as $e \in \mathbf{dom}\,\sigma$.

If a program body is a single statement, this is most usefully a *Compound* (cf. Figure 1); its interpretation is defined by the interpretation of each of the statements in (left to right) order. The rule for interpretation of *Compound* statements is as follows.

$$\frac{(s,\sigma) \xrightarrow{st} \sigma' \qquad (\mathit{mk\text{-}Compound}(\mathit{rest}),\sigma') \xrightarrow{st} \sigma''}{(\mathit{mk\text{-}Compound}([s] \frown \mathit{rest}),\sigma) \xrightarrow{st} \sigma''}$$

Here the state produced by the interpretation of the first statement, s, in the list is the state (σ') in which to interpret the rest of the statement list, *rest*. As this description is recursive, a base case is required, and here this is reached once the list of statements becomes empty. The rule is applied by pattern matching against the input and at this point simply results in an unchanged state.

$$\frac{}{(\mathit{mk\text{-}Compound}([\,]),\sigma) \xrightarrow{st} \sigma}$$

The SOS rules given so far embody the so-called *big step* operational semantics,[6] as it directly defines the final state.

The core language could be extended to consider some form of external storage such as files with the addition of *Read*/*Write* statements; this would be accomplished simply by extending Σ to include a collection of (named) files.

2.2 Denotational Semantics

For simple languages, the difference between the operational and denotational approaches is less marked than when language aspects such as non-determinacy

[6] Also referred to as *natural semantics* by Kahn [Kah87] and Nielson and Nielson [NN92]. 'Small step' operational semantics has to define the granularity at which interference can occur in concurrency and thus shows the steps between smaller portions of program text and state — the overall interpretation of a program is then the transitive closure of the step relation. Big step tends to feel more intuitive in its handling of multiple statements (and especially constructs like blocks); however, it is worth mentioning the existence of small step concepts because these are used later when concurrency comes into play in Section 4.

or the passing of functions as arguments have to be modelled. One important point is that both approaches are built around an explicit notion of state.[7]

The technical distinction between operational and denotational approaches is, however, important and the point can be made by contrasting with the earlier 'abstract interpreter' phrase: denotational semantics is more like a compiler in that it maps the source language into another language. For the simple language that is defined operationally in the preceding section, the mapping is into functions from states to states ($\Sigma \to \Sigma$). This state is the same as defined in the previous section.[8]

A language is needed to define the functional denotations and Church's Lambda notation is the standard as it provides an easy way to write un-named functions.[9]

As a simple example, the assignment statement is mapped to a function which takes a state and returns that state modified with a mapping from the identifier to the evaluation of the right hand side expression in the previous state.

$$M[\![mk\text{-}Assign(lhs, rhs)]\!] = \lambda\sigma \cdot \sigma \dagger \{lhs \mapsto eval(rhs, \sigma)\}$$

Much is made in the literature on denotational semantics about the mapping to denotations being 'homomorphic' in the sense that the structure of the commands in the object language matches the structure of the denotations. So for compound statements:[10]

$$M[\![mk\text{-}Compound([\,])]\!] = \lambda\sigma \cdot \sigma$$
$$M[\![mk\text{-}Compound([s] \frown rl)]\!] = M[\![s]\!] \circ M[\![mk\text{-}Compound(rl)]\!]$$

Here it can be seen that the sequence concatenation on the left matches the function composition on the right and thus the structure is preserved. The homomorphic property is that the denotation of the compound is built (only) from the denotations of its constituent statements.

Note that the loss of referential transparency requires the state notion. This is now so familiar that it is taken for granted but assignments themselves complicate the denotational ideal of the homomorphic mapping

It is not difficult to see that there is a direct connection between operational and denotational descriptions:[11]

$$interpret\colon Stmt \times \Sigma \to \Sigma$$
$$M\colon Stmt \to \Sigma \to \Sigma$$

[7] The extent to which axiomatic semantics avoids making the state explicit is reviewed in Section 2.3.

[8] An Oxford denotational semantics would insist that Σ was also a function type; here the VDM mapping is assumed because this is not a significant issue in the comparison.

[9] Familiarity with this notation is assumed; a good learning resource is [AGM92].

[10] It would be more common to write a denotational description without the constructor (*mk-Compound*) but it has been made clear above that larger languages require an abstract syntax and choosing to keep the same treatment of syntactic objects in the sketched operations and denotational descriptions is useful.

[11] Here, McCarthy's original *interpret*-style description is used to make the point more clearly than can be done with the SOS rule.

They are essentially a $\lambda\sigma$ apart:

$$M[\![s]\!] = \lambda\sigma \cdot \mathit{interpret}(s, \sigma)$$

But Section 2.4 makes clear that the surface difference has a significant impact on reasoning about language descriptions.

The semantics of conditional statements is given below:

$$M[\![\mathit{mk\text{-}If}(\mathit{test}, \mathit{then}, \mathit{else})]\!] = \\ \lambda\sigma \cdot \textbf{if } M[\![\mathit{test}]\!](\sigma) = \textbf{true } \textbf{then } M[\![\mathit{then}]\!](\sigma) \textbf{ else } M[\![\mathit{else}]\!](\sigma)$$

and again is rather similar to the operational semantics given in the previous section.

However, if we look at *While*, we can see that denotational semantics requires *fixed points*.

$$M[\![\mathit{mk\text{-}While}(\mathit{test}, \mathit{body})]\!] = \\ \lambda\sigma \cdot \textbf{if } M[\![\mathit{test}]\!](\sigma) = \textbf{true} \\ \textbf{then } M[\![\mathit{body}]\!] \circ M[\![\mathit{mk\text{-}While}(\mathit{test}, \mathit{body})]\!] \\ \textbf{else } \lambda\sigma \cdot \sigma$$

Note this is not a 'normal' definition because it defines $M[\![\mathit{mk\text{-}While}(\mathit{test}, \mathit{body})]\!]$ in terms of itself. Contrast this with the 'normal' recursion in *interpret* where the function is applied to different arguments.[12]

2.3 Axiomatic Semantics

The widest use of Floyd/Hoare axioms [Flo67, Hoa69] is in the verification or development of programs. It was, however, precisely concerns about "leaving things undefined" in language semantics that led Tony Hoare to propose *Hoare triples*.[13] Perhaps the strongest case for specifying a range of permissible results is in languages that allow concurrent execution and this topic is reviewed in Section 4.2. Here, the axiomatic method is explained with the limited sequential language that has been introduced above.

In a deviation from the approach used in the paper so far, concrete syntax will be used in the sections utilising axiomatic semantics. This is purely by convention: while there is no reason *not* to use abstract syntax, doing so would be unique amongst all other works on axiomatic semantics. The reason for the lack of use of abstract syntax is probably connected to the small scale (and relative syntactic paucity) of the kinds of language to which axiomatic semantics is normally applied.

A so-called 'Hoare triple' consists of a pre condition, program text and a post condition. These are now almost universally written as $\{P\}\ S\ \{Q\}$.[14] In the

[12] In early versions of denotational semantics, Christopher Strachey used the Y combinator to denote the fixed point of a while loop [Wal67, p. 17].

[13] Some background to the Hoare approach is given in [Jon03]; since that publication, earlier drafts have been found of Hoare's attempts to build on his comment made at a conference in 1964 [Ste66, p. 142–143].

[14] In Hoare's original paper [Hoa69], he actually wrote $P\ \{S\}\ Q$ but placing the braces around the assertions emphasises their role as being non-executable.

most widely adopted style, the pre and post conditions are predicates of single states. Note that in contrast with operational and denotational semantics, these states are not explicitly defined. The triple $\{P\}\ S\ \{Q\}$ records a judgement that if S is executed in a state that satisfies the predicate P, then (providing S terminates) the resulting state will satisfy the predicate Q.

Given this interpretation, inference rules can be provided for each language construct:

$$\boxed{\textit{sequence}}\ \frac{\{P\}\ S1\ \{Q\}\quad \{Q\}\ S2\ \{S\}}{\{P1\}\ S1\ ;\ S2\ \{P3\}}$$

$$\boxed{\textit{If}}\ \frac{\{P \wedge b\}\ \mathit{Th}\ \{Q\}\quad \{P \wedge \neg\, b\}\ \mathit{El}\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ \mathit{Th}\ \textbf{else}\ \mathit{El}\ \textbf{fi}\ \{R\}}$$

$$\boxed{\textit{While}}\ \frac{\{P \wedge b\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}\ \{\neg\, b \wedge P\}}$$

The concept of *loop invariants* is a key contribution to the way users think about programs even if they are not reasoning completely formally. As noted above, programming constructs can be used to extend what can be expressed in a language. It remains true however that, if a loop is used say to compute factorial, the proof needs axioms about the operator in addition to the inference rule for while statements.

The caveat above about termination is important: the rule above does not on its own establish that the loop will terminate. This property is often (badly) termed 'partial correctness'. Dijkstra [Dij76] proposed the addition of 'variant functions' to reason about termination and these were in fact employed without that nomenclature in both [Tur49] and [Flo67]. A more pleasing approach is indicated below when the switch to relational post conditions is discussed.

In practice, users are unlikely to give a post condition in exactly the form $\neg\, b \wedge P$. Either the inference rules need to be complemented with weakening rules such as:

$$\boxed{\textit{consequence}}\ \frac{\{P\}\ S\ \{Q\}\quad P' \Rightarrow P\quad Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

or, perhaps more usefully, the other rules should be changed to reflect the potential for weakening — for example:

$$\boxed{\textit{While}'}\ \frac{\{P'\}\ S\ \{P'\}\quad P \wedge b \Rightarrow P'\quad P \wedge \neg\, b \Rightarrow Q}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}\ \{Q\}}$$

Having considered the sort of statement that controls the order in which basic statements are executed, it remains to consider the axiomatic description of assignment statements. The now standard[15] 'backwards rule' can be written

$$\boxed{\mathit{assign}}\ \frac{}{\{P^e_x\}\ x := e\ \{P\}}$$

where P^e_x means substitution of e for x (with appropriate renaming to avoid unwanted capture).

The deceptively simple –and therefore appealing– rule is not without its problems. For example Krzysztof Apt in [Apt81] discusses what needs to be done if the left-hand-side of the assignment is a reference to an element of an array. Without wishing to undervalue what might be thought of as a lucky notational success, it must be observed that the aforementioned lack of referential transparency with variables in programs should prompt care when copying their names into predicates.

Another reservation about the assignment rule arises when languages allow multiple identifiers to refer to the same location (see Section 3.3); sticking to the assignment rule above would appear to imply that 'call-by-reference' is modelled by some form of copy rule.[16]

In [Hoa69], Tony Hoare indicates that the axiomatic approach obviates the need for an explicit model of state.[17] This connects with the well-known 'frame problem' in the sense that it would be convenient if the only thing affected by an assignment to x is the value of the variable with that name. This is, of course, not the case in the presence of call-by-reference parameter passing.

It was realised early on[18] that writing relatively large collections of axioms could lead to inconsistencies. The standard way out of this danger is to provide a model for which axioms can be shown to hold. Under Tony Hoare's supervision, this is exactly what Peter Lauer undertook in his thesis [Lau71]; a later –but better-known– reference is [Don76]. Essentially, it is necessary to show that if $\{P\}\ S\ \{Q\}$ can be deduced from the axioms, then this agrees with the operational semantics as follows:

$$P(\sigma)\ \Rightarrow\ (Q(\sigma')\ \Leftrightarrow\ (S,\sigma) \xrightarrow{st} \sigma')$$

If termination is considered, it is also necessary to show:

$$P(\sigma)\ \Rightarrow\ \exists\sigma' \cdot (S,\sigma) \xrightarrow{st} \sigma'$$

[15] Floyd in [Flo67] used a forward assignment axiom that needs an existential quantifier in its post condition; having discussed the developments with several people (including Jim King whose Effigy system [Kin69] used the backward rule) it would appear to be the case that Bob Floyd spotted the simpler rule after his paper was published and that David Cooper took the information from Carnegie (where he had been for over a year) to Tony Hoare in Belfast when Cooper gave a seminar there.

[16] Various other extensions by Hoare include [CH72, Hoa72a, Hoa71a]; a useful summary is [Apt84].

[17] This idea is further developed in [HJ98].

[18] Specifically at the April 1969 IFIP WG 2.2 meeting in Vienna at which Hoare first presented his axiomatic method [Wal69]. See [JA16] for a few more comments on this meeting.

The *sequence* axiom above shows clearly why it is attractive to use post conditions that are predicates of a single state. It is, however, obvious that this is not really a good idea! What a program is intended to realise is a final state that relates in some meaningful way to its initial state. VDM has used relational post conditions since before [Jon80] — Peter Aczel showed in an unpublished note [Acz82] how to present rules for such relational specifications in a convenient way — and these rules of inference are used in subsequent VDM publications. A particular advantage of explicitly using relations is that Dijkstra's 'variant functions' are avoided simply by saying that the body of a loop should be a well-founded relation.

Hoare's 1969 paper is one of the most influential references in theoretical computer science. It can be seen as the root of developments including Edsger Dijkstra's 'weakest pre conditions' [DS90] and work on 'refinement calculus' [Mor94, BvW98]. Furthermore, this whole line of thought led, after [Hoa71b], to the use of Floyd/Hoare axioms in the development process (rather than *post facto* proof). Further discussion of these developments is available in [Jon03].

2.4 Reasoning

There are two distinct needs to reason based on a (formal) semantics. On the one hand, a programmer might want to prove that a program satisfies its specification; on the other, the designer of a compiler might want to justify the design of a compiling algorithm. (In both cases, the more refined version is to use the semantics as the basis for a stepwise development but that does not affect the distinction.)

To make things clear, both are first explained in terms of operational semantics.

In proving the correctness of a program, its specification should take the form of a pre condition and a post condition. The first of these describes any assumptions on the state before execution of the program; the second defines the judgement of the acceptability of the state produced after the program as a relation to the initial state. The post condition is a predicate of *two* states (before and after) because all but the most trivial specifications relate values in the post-state to those in the pre-state (as with defining the result of a function with respect to its arguments).

$pre: \Sigma \to \mathbb{B}$
$post: \Sigma \times \Sigma \to \mathbb{B}$

This specification is related to the implementation by formulating the related *Proof Obligation* for the program S:

$$\forall \sigma \in \Sigma \cdot pre(\sigma) \;\Rightarrow\; post(\sigma, interpret(S, \sigma))$$

Discharging this proof obligation indicates that the program S satisfies the specification given.

In the task of proving correctness of translation, the proposed algorithm might have the signature:

$translate: Stmt \rightarrow MachineCode$

and the machine code might be given semantics by:[19]

$mc\text{-}interpret: MachineCode \times \Sigma \rightarrow \Sigma$

This allows us to formulate the proof obligation as follows:

$\forall S \in Stmt, \sigma \in \Sigma \cdot mc\text{-}interpret(translate(S), \sigma) = interpret(S, \sigma)$

Although it is possible to reason about the earlier program correctness task using either an operational[20] or a denotational language description, that is exactly the task for which axiomatic semantics was envisioned.[21]

In contrast, the task of reasoning about the correctness of a language translator appears to be best handled with one of the *model oriented* (i.e. operational or denotational) description methods.

The choice between operational and denotational semantics as a basis for such proofs depends on a number of factors. The higher level of abstraction in noting that denotations are functions (for now, from states to states) certainly makes it easy to establish some properties of a language (e.g. the equivalence of a while loop to its unwrapping with a conditional around the original loop).

For translation algorithms that closely follow the phrase structure of the source language, denotational semantics is probably most appropriate because it is easy to reason about the functional semantic objects. Robert Milne and Christopher Strachey tackle implementation correctness in both the 'Adams Essay' [MS74] and the book [MS74] published after Strachey's death; members of the IBM Lab Vienna addressed compiler correctness but as they concerned the large (and Baroque) language PL/I the publications are mainly technical reports such as [BBH+74, Wei75, Izb75, BIJW75, Jon76].

Unfortunately, many compiling techniques are not obviously algebraic in form: optimisations such as register allocation or *strength reduction* cut right across the phrase structure of the language. In such cases, it might well be easier to base the argument on an operational description. Relevant publications on using operational descriptions to reason about compiling include [MP67, Pai67, Luc68, Jon69, JL71].

One point of comparison that is worth clarifying is that operational semantics can be made as 'compositional' as denotational semantics. It is true that denotational semantics has an obvious check of homomorphic mapping (to functional denotations[22]) but, providing the 'grand state' mistakes of early operational descriptions are avoided, the shape of an operational semantics can

[19] This has been deliberately simplified by ignoring the fact that the abstract states (Σ) of the language description need to be reified to representations on the object-time storage organisation.

[20] This approach is explored in John Hughes' thesis [Hug11] and [HJ08].

[21] As observed in Section 2.3, such proofs also require axioms of any new operators.

[22] Finding neat functional denotations is not always possible. The topic of abnormal exits such as 'goto' statements is postponed to Section 6 but forces considerable contortions of the space of denotations.

closely follow the phrase structure of the language being described. The main penalty for using, for example, an SOS description is that proofs have to use induction over the steps of computation rather than, say, Scott induction.

One significant point in the comparison of denotational and operational descriptions concerns termination. A big step (or 'natural') operational semantics applied to

while $x \neq 0$ **do** $x := x - 1$ **od**

will, for a negative initial value of x, simply iterate indefinitely. In contrast, the least fixed point of the denotation of this program is exactly the appropriate partial function (from states to states).

The greatest payoff for the level of abstraction in denotational semantics is almost certainly in proving deeper properties of a defined language.

2.5 Response to the challenge of assignment

The main challenge presented by simple imperative languages is the need to store and update values associated with variables when assignments are made. The response given by both operational and denotational semantics is to model the storage of the computer with an abstract state. There is little fundamental difference between the states used in denotational and operational semantics. Axiomatic semantics avoids an overt state by using value replacement, but the collection of meta-variables used in assertions does essentially imply a state.

3 ALGOL-like blocks, functions, procedures

For the simple language presented above, the differences between the semantic description styles seem minor. But that language lacks features that make real languages convenient for programmers. The challenge of describing language features like named procedures and environmentally-separated blocks adds significant complexity to the task of language description and begins to show interesting differences in the response by each semantic school.

The need to model the local entities of different blocks and sharing of locations presents particular challenges, especially in the presence of more complicated data structures such as arrays. Procedures add additional problems when different parameter mechanisms are considered and so-called *higher-order* procedures (whose parameters or results are procedures themselves) are particularly problematic in some approaches. This section discusses these challenges and the solutions in the different approaches.

It is interesting to observe how similar the treatment is in denotational and operational approaches — and to note the key difference on procedure denotations. Axiomatic semantics ends up taking a different tack by avoiding the question of environments and instead using name substitution.

3.1 Local naming

In first-order predicate calculus:

$$\forall x \in X \cdot (\ldots \forall x \in Y \cdot (\ldots) \ldots \exists x \in Z \cdot (\ldots))$$

the three bindings of x are distinct. The need for separate name spaces in programming languages is even stronger because program texts are likely to be rather long.

Most programming languages offer ways of localising a name space so that the same identifier can denote a different variable in nested *blocks*.

$$Stmt = \cdots \mid Block$$

$$\begin{array}{lll} Block :: & types & : Id \xrightarrow{m} ScalarType \\ & body & : Stmt \\ & \ldots & \end{array}$$

With the simple storage model of Section 2, identifiers are mapped to denotations (so far only values) and there is no need to change the underlying state notion. The only delicate point is that –at block exit– the semantics must recover the denotations of those identifiers that denoted a different variable in the inner block.

As with the interpretation of programs, the declaration of variables without initialisation could cause problems with evaluation and it is assumed that there is an automatic initialisation performed.

Context conditions must also be reconsidered now that the same identifier may denote different values and types throughout computation. This can be achieved by requiring that usage of names in a well-formed block matches the closest embracing declaration. A well-formed program now need only require that every constituent block is well-formed.

3.2 Functions, procedures and (simple) parameters

The pragmatics of functions and procedures is that they can be used to factor out portions of program text that can be called from many places.[23] From a user point of view, procedure calls are statements that get executed in the order dictated by their position in a list of statements whereas functions occur in expressions.[24] Functions and procedures require similar modelling techniques

[23] Although compiling techniques are not the main topic of this paper, it is worth observing that implementing general recursion and parameter passing required some interesting techniques — see [RR62]; there is a very detailed reconstruction of the development of the idea of the *Display* mechanism in [vdH17].

[24] It is worth noting that functions which can cause side effects considerably complicate expression evaluation. At a minimum, they remove the possibility of saying that $eval$: $Expr \times \Sigma \rightarrow Value$ because of the potential state change inherent if functions with side effects are allowed as part of $Expr$. Something that causes language descriptions more trouble is that unless the order of evaluation of expressions is strictly defined (which is rare because languages tend to leave compilers the freedom to optimise register use), evaluating expressions containing functions with side effects results in non-determinism. This general topic is resumed in Section 4.

in terms of the semantic objects required and are therefore treated together in the remainder of this section.

$$\begin{array}{l} \mathit{Block} :: \cdots \\ \qquad\quad \mathit{body} : \mathit{Stmt} \end{array}$$

$$\mathit{Stmt} = \cdots \mid \mathit{ProcCall}$$

Context conditions of procedures are similar to those for blocks, but additionally require that the evaluated types of parameters in a procedure call match those declared in the procedure definition. This means that the *TypeMap* object must also store information on procedure definitions.

Functions and procedures have fixed denotations so they do not belong in the store which contains values that can be changed (by assignment) within statements. This can be handled by introducing an 'environment' to contain the denotations:

$$\mathit{Env} = \mathit{Id} \xrightarrow{m} \mathit{Den}$$

$$\mathit{Den} = \mathit{FunDen} \mid \mathit{ProcDen} \mid \cdots$$

The basic model is not difficult; that having been said, the features that have been devised in various languages to make procedures more useful are myriad and necessitate extension of the role of the environment.

The passing of parameters of simple values (e.g. $\mathbb{N}, \mathbb{B}$) is straightforward: these are simply given new identifiers within the local environment of the function or procedure. However, more complex parameter passing mechanisms require more consideration.

3.3 Sharing

Thus far, it has been assumed that identifiers denote simple values such as numbers or Booleans. However, for reasons of efficiency, it is sometimes useful to have more than one identifier referring to the same entity. Because of potential name clashes, making precise the semantics of such sharing is non-trivial. Classically, logicians (e.g. in describing the Lambda calculus) have used a *copy rule* with 'suitable changes of names to avoid clashes' to describe such concepts. For programming languages, the text of the procedure is modified to copy in the names, references or values of arguments, with appropriate renaming to avoid name clashing. This is the how the ALGOL report [BBG$^+$60] attempts to fix the semantics; it can be formalised, as in the operational description of ALGOL 60 [ACJ72].

Many programming tasks require composite entities such as arrays which gives rise to the notion of *left hand values* for elements of arrays. These considerations are the main reasons for allowing different ways of passing arguments to functions or procedures. Surprisingly many alternative parameter passing mechanisms have been devised and each has its use:

- Call *by value* is the most obvious and is appropriate for simple types — the argument (which might be an expression) is evaluated and this value is copied into the body of the function or procedure. Typically this is achieved by creating a new memory allocation for the value and therefore modifications to this variable are not seen in the calling scope.
- Copying of data can be reduced by using *by location* (aka *by reference*) parameter passing, in which a pointer to the storage location of the argument is passed instead. This enables the function to modify the value of the argument variable in a way that will affect the calling context.
- The full *by name* parameter mechanism of ALGOL 60 is even more challenging semantically: the denotations of arguments are evaluated anew each time the respective parameter name is encountered within the body of the function. (This specialises to 'by location' mode when the argument (or 'actual parameter' in ALGOL speak) is a simple identifier.)
- Call by *value/return* offers a useful compromise; by copying the value of each argument into a new location and then returning the (potentially modified) values to their original locations; it facilitates the return of multiple values from procedures/functions but avoids punning of names.[25]

In model-oriented methods, all of the above can be modelled with:[26]

$Env = Id \xrightarrow{m} Den$

$Den = \cdots \mid Loc$

$Loc = ScalarLoc \mid ArrayLoc$

$ArrayLoc = \mathbb{N}^* \xrightarrow{m} ScalarLoc$

$\Sigma = ScalarLoc \xrightarrow{m} ScalarValue$

In SOS it is clear that the environment is not changed by simple statements such as assignments as *env* is not in the range of the $\xrightarrow{st}$ relation.

$$\frac{(rhs, \sigma) \xrightarrow{ex} v}{(mk\text{-}Assign(lhs, rhs), env, \sigma) \xrightarrow{st} \sigma \dagger \{env(lhs) \mapsto v\}}$$

The task of creating and passing locations is handled in the semantics of blocks and calling.

Similarly, in denotational semantics, the fact that environments are not changed by simple statements is apparent from the 'Curried':

$M\colon Stmt \to Env \to \Sigma \to \Sigma$

[25] Unless the same argument is passed to different parameters — but this is an easy static check.

[26] Records are similar to arrays but have fields that are not necessarily of the same type; modelling records and combinations of arrays/records is straightforward.

It is interesting the extent to which the description of semantic objects and a few type definitions (i.e. no actual rules or formulae) can suggest (to an experienced reader) the main points about a language. The rest of this paper is written at this level of abstraction.

The passing of parameters in environment-based semantics is not difficult — the semantic function, relation or mapping is extended to include an environment as a parameter and this environment is modified at evaluation time. The parameter passing mechanism chosen affects the level at which the environment or its sub-contents are modified.

It is, however, important to clarify how the context of a procedure or function is captured in model-oriented approaches. In an operational approach, one part of *ProcDen*/*FunDen* is its text. But this is not enough: if functions/procedures can be declared in any block and called from any deeper block, then there must be a way of fixing the 'environment' in which they are to be executed, so that there is a proper evaluation of any parameter identifier that is passed in, and no clashes with local names used within the text of the procedure. To address this, an environment is usually part of the interpreting function or relation for procedures and function. This approach is essentially identical to the *static chain* method for address resolution, in which each scope contains some meta-information linking it to its direct lexical parent.

In denotational approaches, *FunDen*/*ProcDen* are functions in the standard mathematical sense, with the appropriate environment bound in forming a *closure*. [27]

Environments are therefore also parameters to the meaning function, as seen above.

3.4 Handling parameters and sharing in the axiomatic approach

Using 'by location' parameter passing means that multiple identifiers refer to the same *location* and, at a minimum, this undermines the axiom of assignment in Hoare triples. So the axiomatic approach, tending to ignore the concepts of both state and environment, uses quite a different strategy to model-oriented techniques: a form of repeated name substitution is used, essentially a modification of the copy rule described above.

For simple functions without side effects the rule

function $f(L);\ S$

applies where L is the list of parameters and S the compound statement making up the body of the function. Then in a triple $\{P\}\ S\ \{Q\}$ the effect of applying the function can be deduced by substituting the call to the function f within Q with the results of applying the function. Note that the parameters of f are not only its explicit parameters but also all free variables called within S (referred

[27] As is the case with axiomatic Semantics 2.3, strictly, the function itself is not produced: the semantics maps to a Lambda expression that could be proved equivalent to the mathematical function using properties about the function.

to as implicit parameters). This is the approach presented in the axiomatic 'definition' of Pascal [HW73].

This name substitution approach essentially treats procedures as consisting of two functions, one of which maps the initial values of the arguments into their final values and another which maps the initial values of all global variables used in the procedure into their final values.

A clarification of the procedure rules is given by Apt in [Apt81]. The text of a program containing procedures is modified to have the procedure call replaced by the text of the body of the procedure, with a careful substitution of parameter names for argument names; a similar substitution process occurs with the assertion decorations of the text. However, Apt notes that the rules only apply if the parameter and argument name sets are entirely disjoint and there are no free occurrences of the parameters in the assertions. Appropriate name changes can always be made to achieve this but significant care must be taken.

Arrays (even without sharing) need careful handling in axiomatic semantics, as also discussed by [Apt81]. Allowing expressions as the subscripts in subscripted variables can lead to problems, particularly when these expressions reference the same array. One way to address this is to replace the whole array with a new one modified at the index to which assignment has been made, but this is not a particularly elegant solution.

3.5 Higher-order functions and procedures

The pragmatics of allowing parameters to be procedures and functions is to facilitate higher-order programs. Not only is this concept beloved by functional language users, it is also a prime tool for abstraction in programming. The simple *map list* idea

$$\textit{map-list}: (A \to B) \times A^* \to B^*$$

is a small example of how high levels of re-use and abstraction can be achieved. There are, of course, far more exotic cases that introduce new ways of achieving recursion: see, for example, Knuth's 'man and boy' example [Knu64] that was written as a challenge for ALGOL 60 compilers.

This topic is placed in a separate sub-section because it causes one of the most telling differences between operational and denotational approaches. The clue to the source of the problem is that, once functions can take functions as arguments, the possibility arises that a function can be applied to itself. (This does also introduce a minor issue around types that is reviewed at the end of this sub-section.)

The fact that, in operational semantics approaches, the denotation of a procedure is its text and statically containing environment means that no new concepts are needed to model the passing of procedures or functions.

In denotational approaches, however, the denotation of procedures are actual functions (as indicated in Section 3.3). During the development of denotational semantics, this brought Strachey to a serious mathematical problem: since the cardinality of the function space $X \to X$ must be greater than that of X, there

is a paradox with functions that can take themselves as arguments. There was thus a point in time where the idea of 'denotational' (or at that time 'mathematical') semantics claimed that semantics could be given by mapping programs to mathematical functions (expressed in the Lambda calculus) but the approach was built on sand in the sense that no one could offer a model of the untyped Lambda calculus.

This problem was resolved with Dana Scott's 1969 invention of domains with suitably restricted functions. This was a major intellectual achievement and has been widely described; perhaps the most accessible text remains [Sto77] but Scott's own [Sco80] provides a clear description of the context of his models of the untyped Lambda calculus.

The challenge of modelling self-applying functions thus gives rise to the largest diversion so far between operational and denotational approaches. It is interesting to look more carefully at what is going on here. The 'homomorphic' rule says that the denotation of a construct should be built up from the denotations of its constituent parts. But the name of a procedure can only be given a denotation by storing it in an environment.

There is, in fact, another issue to be resolved for functions that can take themselves as arguments; that issue concerns defining their type. Consider first a binary tree structure built up with records:

$$\begin{array}{lll} \mathit{BinTree} :: & \mathit{left} & : \left[\mathit{BinTree}\right] \\ & \mathit{value} & : \mathbb{N} \\ & \mathit{right} & : \left[\mathit{BinTree}\right] \end{array}$$

The name of the type *BinTree* is used to express the recursive embeddings and the marking of the fields as optional lets the instances be finite.

In order to declare a function type that can take itself as argument, there must be a way of naming a function type. In fact, ALGOL 60 ducked this problem: the language is almost strongly typed except for function and array types. Both PL/I and Pascal offer such separate naming of function (entry) types. It is worth noting that the ability to name function types is required for mutually recursive procedures because they cannot be given in an order such that definition precedes use.

3.6 Responses to modelling procedures and functions

Blocks and procedures bring new challenges to semantic descriptions, particularly with the concerns of name sharing and local entities. Denotational and operational semantics solve this problem by separating out an environment from the state, but very cautious name substitution is needed in axiomatic semantics, particularly when certain parameter mechanisms are used. Procedures become another kind of denotable value in model based semantics, but this requires careful foundation for denotational semantics when higher-order functions are allowed.

4 Modelling non-deterministic languages

There are two essentially different reasons that non-determinism figures in programming languages:[28]

- the originator of a language might wish to allow freedom to the designers of implementations to make optimisations such as common sub-expression elimination or 'strength reduction'
- a language might encompass features that result in non-deterministic execution — the most telling example is concurrency where differing progress of threads can yield differing results of a program

It is clear that the specification (or description) of a language must fix the full –and exact– range of acceptable outcomes. This matters both to programmers writing programs in L and implementers of L. The challenge is leaving some aspects of the language undefined, but properly constrained. This problem is further complicated by questions of *granularity* of interleaving: a semantic description must be capable of describing granularity at least as fine as that handled by the language. The difficulty of these points is a significant challenge for the semantic description: having a sufficiently rich notation to allow communication of these aspects while remaining readable. These challenges existed as soon as languages such as PL/I were faced; the various responses are interestingly different in appearance but can be argued to have a common core.

4.1 Operational semantics

The pragmatics of concurrent programming languages should be obvious: both low-level systems programming and high-level applications need to express algorithms that accommodate differing run-time progress. In model-oriented semantic approaches,[29] there appears to be no alternative to recording the text of the threads that remain to be executed and adjoining it to the shared state (Σ) that is being updated. Such pairings of states and remaining thread texts are referred to as *configurations*.

In order to capture the possible mergings of the threads, an operational semantics must show the non-deterministic choice between the threads. Precisely how this is done fixes the granularity of merging.[30]

A first thought might be to record a function that maps a configuration to the set of its possible successor configurations but this becomes notationally messy.

[28] A separate need to have a formal treatment of non-deterministic specifications arises when considering *program development* — see Section 4.2.

[29] Remember that this paper focusses on procedural languages and the discussion here is on shared-variable concurrency; no attempt is made to address process algebras such as ACP [BK84], CSP [Hoa85] or CCS [Mil89].

[30] Many attempts to provide ways of reasoning about concurrent programs (see Section 4.2) make the unreasonable assumption that assignment statements are atomic; this level of granularity is totally unrealistic for real implementations of languages.

It is, of course, equivalent to think of this as a relation between configurations and it transpires that this is notationally much cleaner.

There are many ways to define such a relation. Looking back at the early operational semantics VDL descriptions [Lab66, LW69], it is clear that their operational description offered a way of describing such non-determinacy: control trees contained a structured version of the program text that still had to be executed but these control trees were made part of the (grand) state.[31] Plotkin's SOS [Plo81] provides much clearer descriptions because the non-determinacy is factored out of the rules themselves; it moves to the selection of a semantic rule (the remaining text and state are kept separate).

Consider an extension to the language of Fig. 1 that introduces a simple kind of parallel statement, in which multiple threads can execute concurrently. Each thread is simply a sequence of assignment statements:

$$\mathit{Stmt} = \cdots \mid \mathit{Parallel}$$

$$\mathit{Parallel} = \mathit{ThreadId} \xrightarrow{m} \mathit{Thread}$$

$$\mathit{Thread} = \mathit{Assign}^*$$

A large-step approach is inappropriate here: an interpreting rule like $\xrightarrow{st}$ from Section 2.1 would interpret an entire sequence of assignments as one. This limits the language to executing the *Parallel* as though each *Thread* were atomic. What is needed is a set of rules which each peel off and execute one of the remaining statements in any non-empty thread. For this we use the relation for parallel interpretation, $\xrightarrow{par}$.

$$\xrightarrow{par} : \mathcal{P}((\mathit{Parallel} \times \Sigma) \times (\mathit{Parallel} \times \Sigma))$$

Any non-empty thread is a candidate for execution and –after a step of an active thread is executed– the thread map is updated to remove the executed statement.

$$\frac{P(i) \neq [\,] \qquad (\mathbf{hd}\ P(i), \sigma) \xrightarrow{st} \sigma'}{(P, \sigma) \xrightarrow{par} (P \dagger \{i \mapsto \mathbf{tl}\ S(i)\}, \sigma')}$$

A program is complete when all of its branches have terminated (become empty).

$$\frac{\sigma_0 = \{v \mapsto 0 \mid v \in \mathbf{dom}\ vm\} \qquad (P, \sigma_0) \xrightarrow{par} *(P', \sigma') \qquad \forall t \in \mathbf{dom}\ P' \cdot P'(t) = [\,]}{\mathit{mk\text{-}Program}(vm, P) \xrightarrow{pr} \sigma'}$$

In this way, the interpretation of the program as a whole is the transitive closure of $\xrightarrow{par}$.

Using this 'small step' approach, assignments from either thread may be interleaved in any order, as the choice of which thread to interpret next is lifted to the choice of rule instantiation.

[31] For a fuller discussion see [JA16, §3].

As an example extension to the language, consider the interpretation of an *if* statement. The 'big step' approach from Section 2.1 is no longer appropriate, as we want to allow the possibility of interleaving the test of the condition as well as the statements within the body of the branches of the conditional:

$$\frac{(test, \sigma) \xrightarrow{ex} \mathbf{true}}{([mk\text{-}If(test, th, el)] \curvearrowright rest, \sigma) \xrightarrow{st} (th \curvearrowright rest, \sigma)}$$

$$\frac{(test, \sigma) \xrightarrow{ex} \mathbf{false}}{([mk\text{-}If(test, th, el)] \curvearrowright rest, \sigma) \xrightarrow{st} (el \curvearrowright rest, \sigma)}$$

These rules perform only the test, simply prepending the appropriate body of statements to the remaining thread text. Interpretation then proceeds as normal on this sequence.

Extensions for other language features can be made in a similar style to this; for example, a small step model of a *while* loop unwraps the loop with a conditional surrounding it.

Note that so far the assumption is that assignment statements represent the level of atomicity in the language. Allowing interference to take place at the expression evaluation level is possible and makes two things clear:

- The way that SOS factors out the non-deterministic choice of rules that match the current configuration is extremely helpful in preventing the issue of concurrency from polluting a whole definition. But there is a sense in which the configurations are just a way of presenting the 'control trees' that were much criticised in VDL operational descriptions. (The danger with these control trees in a grand state semantics was that it was hard to determine where they could or could not be updated.)

- A further observation is that, in SOS descriptions, the non-determinacy with expressions looks different from that with statements: with expressions, the non-determinacy is resolved when a variable is accessed (or a function returns a value) and the effect is to place a value in the evaluation tree; with statements, the effect is reflected in a state change and the executed statement is discarded from the resulting configuration.

Moving to a level of granularity larger than assignments, a programmer may wish to make *multiple* statements executable only as an atomic block.

$$Stmt = \cdots \mid Atomic$$

$$Atomic :: Assign^*$$

$$\frac{(sl, \sigma) \xrightarrow{st} \sigma'}{([mk\text{-}Atomic(sl)] \curvearrowright rest, \sigma) \xrightarrow{st} (rest, \sigma')}$$

Atomicity is, of course, a key issue in the database world and it is interesting to note the similarities and differences from the programming language universe. It would not be difficult to add data types to a programming language that

provided ways to declare and manipulate relations similar to those in the standard relational model (see [Dat82]). As discussed at a Schloss Dagstuhl event on atomicity [JLRW05, §2.4.2], this then highlights the point that database systems strive to prevent data races, where possible, by system induced locking (and, where pre-planning fails, to detect races and handle the recovery) whereas programmers using typical programming languages are held responsible to plan and control locking.

4.2 Axiomatic response

As indicated in Section 2.3, the axiomatic approach copes with general non-determinism naturally. This observation that it is important to leave aspects of a language undefined was made by Tony Hoare in [Ste66, p.142–143] and –via multiple drafts– led him to his famous 'axiomatic basis' paper [Hoa69].[32] Moreover, it became clear in using methods such as VDM that specifications that allow a range of implementations are a powerful way of structuring design decisions (see for example [Abr10, Jon90]).

Unfortunately the specific case of non-determinacy being caused by concurrent execution presents severe challenges for the axiomatic approach. The source of the difficulty is precisely the *interference* that has to be modelled explicitly in the operational descriptions of the previous sub-section. Before facing the fact that post conditions alone are insufficient to specify components that suffer interference, it is interesting to trace an early attempt to finesse that difficulty and its more recent manifestation in (Concurrent) Separation Logic.

Hoare singled out the case of disjoint concurrency in [Hoa72b] and made the observation that the post conditions of two parallel threads could be conjoined providing there were no shared variables. Hoare's 1972 paper covered normal (stack) variables in which case the disjointness is simply a check of the 'alphabets' of the threads. John Reynolds introduced *Separation Logic* [Rey78, Rey89] to support reasoning about *heap variables* (i.e. data structures that contain pointers and whose topology can be changed by updating said pointers). Reasoning about parallel threads that share a heap can be very delicate. An interesting collaborative attack (see [BO16]) led to *Concurrent Separation Logic* [O'H07] which has spawned many variants — see [Par10]. The essential idea is akin to Hoare's observation: what one wants to do is to conjoin the post conditions of parallel threads but this is only valid if the interference is eliminated. What separation logics facilitate is concise statements of the disjoint ownership of heap addresses.[33] More recently, [JY15] notes that certain cases of heap separation can be viewed as reifications of abstract descriptions of separate entities.

In [O'H07], it is suggested that separation logic should be used to reason about race-free programs and *Rely/Guarantee* (R/G) conditions should be used

[32] Of course, the soundness notion at the end of Section 2.3 needs to be enriched but this is straightforward.

[33] This led Jones to make a suggestion at the MFPS meeting in 2005 where O'Hearn presented concurrent separation logic that it might better be thought of as *ownership logic*.

for 'racey' programs.[34] The initial publications on R/G go back to [Jon81] — more recently the same underlying concept has been expressed in a refinement calculus [Mor94, BvW98] style in [HJC14, JHC15]. This, in particular, makes algebraic properties such as the distribution of rely and guarantee conditions over sequential and parallel program operators much clearer.

The basic R/G idea is that acceptable interference should be documented with rely conditions in the same way that sequential Floyd/Hoare logic records acceptable starting states with pre conditions. Also, just as post conditions express obligations on the running code, guarantee conditions record the upper limit of interference that a component can inflict on its environment.

Specifications of components using R/G conditions can then be used as a basis for design justification. In a step where the sub-components are also specified using R/G conditions, clear proof obligations exist to justify development steps for parallel operators. Unsurprisingly, these Proof Obligations are more complicated than those for sequential Floyd/Hoare logic but the essential property of *compositionality* is preserved.

Just as at the end of Section 2.3 the soundness proofs of these proof obligations needs to be established. It is possible to extend the operational semantics to carry an interference relation and then to interpose it at points appropriate to the granularity of the language; this approach is used in [CJ06, Col08]. Alternatively, *Aczel traces* (see [Acz83] or the more accessible [dR01]) can provide a space of denotations and [CHM16] does this in a way that conducts proofs at a significantly higher level of abstraction.

4.3 Denotational response

The key to the utility of a denotational semantic description is the choice of a space of denotations which admit tractable reasoning. Denotations for the language of threads above could be either relations over states or functions from states to sets of states. In either case, there is a need to mark (potential) nontermination. It is important to note that the problem of interference remains: just as an operational semantics must indicate the granularity of thread switching by the way in which configurations are changed and rematched, the relations must be composed appropriately.

Thus far, there is a lot of similarity between denotational and operational presentations of the semantics for non-determinacy resulting from concurrent threads. The combination of non-determinism with higher order functions (cf. Section 3.5) however poses extra difficulties for the denotational approach. Here *Power Domains* [Plo76, Smy76] are required to preserve the mathematical properties that overcome the cardinality paradoxes related to higher-order language constructs. Again operational semantics is inherently simpler because procedures and functions are modelled simply by their texts.

[34] Although this seemingly simple dichotomy ignores the way in which non-interference at an abstract level can be used to establish race freedom in a representation — a nice example is Simpson's 'Four-Slot' implementation of Asynchronous Communication Mechanisms in [JP11].

4.4 Responses to the challenges of non-determinism and concurrency

The challenges of parallelism brings some variance in the response from the various semantics. In operational semantics, the non-determinism is lifted to the rule level and the real power of SOS to merely constrain acceptable solutions (rather than generate a unique solution) is displayed. In some ways this is similar to certain axiomatic responses, where interference and interaction is constrained by logical propositions. Denotational semantics runs into foundational trouble since the traditional function can no longer be used as a base for denotations. Instead, contortions of the semantic domains such as power domains are required.

5 Applying the ideas to a concurrent object-based language

This section outlines the semantics of COOL,[35] which is a concurrent object-orient language, designed to be small enough to model in a small space but realistic in its handling of the issues identified above.

SIMULA 67 [DMN68] was designed as a language in which simulation programs could be constructed; this provides a wonderful intuition for *Object-Oriented* (OO) programming languages: objects are blocks that can be instantiated as required,[36] block descriptions are the class definitions, local variables are the instance variables and procedures are methods. The scope of method names is of course external to the class to enable objects to call methods on other objects.[37]

Key issues in the design of a concurrent language are how to generate and synchronise concurrent threads. Although it gives an unconventional OO language, the aims of this section can be achieved by limiting (instances of) objects to running one method at a time and generating concurrency by arranging that many objects can be active. This ensures that instance variables are free from data races and, crucially, that the level of interference is in the hands of the programmer because only by sharing references (to objects) is interference possible.

The move from the unconstrained concurrency of multiple threads of Section 4 to a simple OO-language can be summarised as follows:

- The language in Section 4 has dangerous data races because of the single shared state.

[35] COOL was inspired by –and is similar to– POOL [AR92]. COOL is used in teaching a course on language semantics at Newcastle University.

[36] When Ole-Johan Dahl made this comment to Jones, the whole OO area became clearer.

[37] The desire to add some notion of object orientation to languages such as C did not necessarily result in languages with clear semantics. SmallTalk [GR83] is a principled OO language and Bertrand Meyer's Eiffel language [Mey88] adopts the pre/post specification idea to provide 'contracts'.

- In the language of this section each object (instance) has a local state and can run as a thread.
- Such extreme separation needs to be tempered by providing some communication between the threads. This is easy to achieve by allowing methods to be called in objects. Parameter evaluation is by value; object references can be passed thus opening up both (controlled) sharing and passing of the ability to invoke methods.
- Any object can create an instance of another class and receives the unique *Reference* of the new object. The relevant statement might be called *New*.
- The only way in which objects can begin execution is by having their methods called by other objects (the exception is for the initial object which begins execution at program start). Objects retain references to their client objects and should eventually stop execution and return values.
- Thus far, there is no obvious source of the claimed concurrency but there many ways to facilitate this:
  - A class could have a designated initial method that begins to execute in any newly created object of that class: instantiating multiple objects results in concurrent execution. Similarly, a program could have a set of designated objects which all begin execution when the program starts (this latter approach is presented in the language description below).
  - ABC/L [Yon90] included a *FutureCall* statement that essentially forks the called method — the join occurs when the client object executes a *Wait* statement.
  - An alternative explored in [San99] is to have a *Release* statement that prematurely releases the client object before the server method is complete. Using this strategy, the client can resume execution while the server continues to execute. This can be further enriched by a *Delegate* statement, which passes responsibility to another object for executing and returning to the client when complete.
- A language built around objects that lacks inheritance is sometimes referred to as *object-based* but inheritance can be added to the features above by viewing it as a way of calling nested blocks.

An operational semantics for such a language can be built around the following semantic objects.

The basic threads per object are keyed to their *Reference*s:

$ObjectStore = Reference \xrightarrow{m} ObjectInformation$

This keeps a record of the states of all the objects that exist at a given time in the execution of the program.

Each *ObjectInformation* contains the information needed to determine the current state and activity of the object:[38]

$$\begin{array}{lll} \mathit{ObjectInformation} :: & \mathit{class} & : \mathit{Id} \\ & \sigma & : \mathit{Store} \\ & \mathit{mode} & : \textsc{Ready} \mid \mathit{Run} \mid \mathit{Wait} \end{array}$$

The local *Store* of an object simply contains the current values of its variables:

$$\mathit{Store} = \mathit{Id} \xrightarrow{m} \mathit{Value}$$

$$\mathit{Value} = [\mathit{Reference}] \mid \mathbb{Z} \mid \mathbb{B}$$

where the set *Reference* is infinite and $\mathbf{nil} \notin \mathit{Reference}$.

Modes of objects indicate their current activity status. Objects which are Ready are not currently doing anything; method calls may be made to such objects. The other modes indicate some form of activity.

$$\begin{array}{lll} \mathit{Run} :: & \mathit{remainder} & : \mathit{Statement}^* \\ & \mathit{client} & : \mathit{Reference} \end{array}$$

Objects in *Run* mode are currently executing. It is important to retain the list of statements which they have yet to execute (compare with the configurations of Section 4.1) and the reference of the object which initiated their execution, which will be awaiting the eventual return of a value (or a special token indicating there is no return value).

$$\begin{array}{lll} \mathit{Wait} :: & \mathit{lhs} & : \mathit{Id} \\ & \mathit{remainder} & : \mathit{Statement}^* \\ & \mathit{client} & : \mathit{Reference} \end{array}$$

Objects waiting for a value to be returned must keep track of the (local) variable to which this value should be saved, the list of statements to which they will resume executing and the client by which they were originally called.

Programs are defined as a specification of objects and some initialisation.

$$\begin{array}{lll} \mathit{Program} :: & \mathit{cs} & : \mathit{ClassStore} \\ & \mathit{startingclasses} & : \mathit{Id}^* \\ & \mathit{startingmethods} & : \mathit{Id}^* \end{array}$$

The *startingclasses* sequence indicates which classes within the *ClassStore* are to be initialised at program commencement and *startingmethods* indicates which methods within these classes should be executed.

ClassStore is the global directory of all classes in the program; whereas the *ObjectStore* is the store of dynamic information on the extant objects, the *ClassStore* holds the static information on all possible objects.

$$\mathit{ClassStore} = \mathit{Id} \xrightarrow{m} \mathit{ClassInformation}$$

$$\begin{array}{lll} \mathit{ClassInformation} :: & \mathit{variables} & : \mathit{Id} \xrightarrow{m} \mathit{Type} \\ & \mathit{methods} & : \mathit{Id} \xrightarrow{m} \mathit{MethodInfo} \end{array}$$

[38] The texts of object classes are stored in a separate *ClassStore*, discussion of which is postponed to the consideration of the *Program* type.

The information here defines the variables declared in the class and their types (there are no dynamic declarations in this language) and the methods available to be called in the language. More detail need not be given on *MethodInfo* but it contains parameter information and statements to be executed for each method.

Thus the main semantic relation has the type:

$\xrightarrow{st}: \mathcal{P}((\mathit{ClassStore} \times \mathit{ObjectStore}) \times \mathit{ObjectStore})$

Once the program has commenced, the *ClassStore* and *ObjectStore* maps are globally available to the semantics during execution. However, individual objects have access to only the *ClassStore* object — to enable them to call methods in other objects — and of course their own internal store.

6 Abnormal ordering

Many programming languages contain features that bring about non-sequential order of execution of statements. The most obvious example is the 'goto' statement (attacked by Dijkstra in [Dij68] and defended by Knuth in [Knu74]) but it is certainly not the sole source of difficulty: (loop) breaks, exception mechanisms and even returns from functions or procedures present similar challenges. Expressed in denotational terms, the difficulty is that the 'homomorphic rule' cannot directly apply when the meaning of a construct depends on something that is not present in the construct. Put another way, the obvious idea that the semantics of the sequential composition of two statements should be the composition of the semantics of those two statements cannot apply when the first statement appoints as its successor a statement elsewhere.

One response from operational semantics that shows rather clearly what has to happen can be seen in VDL descriptions. In early Vienna Lab operational semantics, an explicit 'Control Tree' recorded the text that was still to be executed; abnormal sequencing was modelled by surgery on this control tree.[39]

Within the denotational camp, there are two rather different responses to the challenges of abnormal ordering. Most researchers (and certainly those strongly connected to Oxford) use *Continuations*. The core idea is to recover some semblance of the homomorphic rule by making the denotation of a label represent the effect of starting execution at that label. In order to develop such denotations it is necessary to pass to every semantic function a denotation that corresponds to the execution of the remainder of the program. This makes the semantics higher order than one might expect and arguably more complicated than these specific constructs require.

In contrast, VDM denotational descriptions (and the Isabelle formulations of semantics in [NK13]) effectively extend the denotations from $\Sigma \to \Sigma$ to have

[39] It is interesting to note that [McC66] had an explicit program counter that could be seen as a hint towards what had to be done with control trees when a massive language like PL/I (complete with concurrency) had to be described.

ranges that can represent abnormal results. The potential messiness that something more complicated than functional composition is now needed for sequential composition can be hidden by 'combinators'.[40]

Incorporating the exit ideas into SOS descriptions is something that has not been published. It would be easy to do this explicitly with extra cases for all language constructs but this would result in the messiness visible in [ACJ72]. Frustratingly, this is much heavier than what VDM achieves with combinators. Since the latter could be read operationally, it should be possible to find a way of adding something like the combinators to SOS rules.

An axiomatic approach to jumps is proposed in [CH72], although the authors do acknowledge that jumps may be better avoided where possible and indeed most axiomatic semantic descriptions skip the topic entirely. The essential idea is adapted from earlier (operational) work by Landin [Lan65a, Lan65b], which treated jumps like procedures whose body is the sequence of statements following the label up until the end of its enclosing block. Rather than returning control to the calling context, however, it is resumed from the end of the block enclosing the label. Clint and Hoare's approach is largely the same, although they prefer to restrict the declaration of labels (and their 'bodies') to the beginning of blocks. The rules do allow for labels to be declared anywhere within the block, with some slight added complexity. However, only one label may be declared per block, and further restrictions prevent jumping into compound and conditional statements.

It is interesting to note that this approach bears some obvious similarities to the continuations used in denotational semantics. Although notationally very different, the idea of a label representing computation left to be performed is at the core of both ideas.[41]

There is also a clear comparison to the configurations used in the operational semantics of Section 4.1 in which the *text* of the computation yet to be executed is stored.

7 Closing remarks

This section mentions some current research (Section 7.1), related references (Section 7.2) and offers some general conclusions.

[40] In [Mos11], Peter Mosses makes the interesting link between VDM's use of such combinators and Eugenio Moggi's 'monads' [Mog89]. The differences between the VDM exit scheme and continuations are teased apart by proofs of equivalence in [Jon78, Jon82].

[41] Indeed, in de Bakker's book *Mathematical Theory of Program Correctness*, a book showing the use of all kinds of semantics in program proof, de Bruin gives a similar axiomatic rule but notes that it is hard to clearly see the correctness of this rule or use the rule in proofs [dBDBZ80]. Instead, a denotational-style continuations semantics is presented and proofs are built around that.

7.1 Algebraic semantics

Work on this topic is too recent to present a full evaluation; here only some pointers and superficial comments are offered.

- Hoare [HvS12, HMSW11] and others [Hay16, HCM+16] are building on Dexter Kozen's 'Kleene algebra with tests' [Koz97] that looks for algebraic laws that abstract from any detailed model of a programming language.
- As with 'Boolean algebras', the algebraic laws normally admit more than one model: saying, for example, that the sequence operator of semicolon is associative but non-commutative does not preclude a semantics in which statements are executed right to left.
- The clear advantage of recording algebraic laws about programming constructs is the same as in classical algebra: if proofs can be conducted at that level of abstraction they are likely to be much easier and more general than any attempt to reason about a model-oriented language description.
- a specific example is the use made in [Hay16] of an 'interchange law' to justify the equivalent of the most important Rely/Guarantee introduction rule. Furthermore, Hayes and colleagues have gone on to present a 'Synchronous Kleene Algebra' that also covers synchronous event-based concurrent languages [HCM+16].
- It is interesting that there are echoes here of the 'program schema' research [LPP70, Pat67] that was one of the earliest avenues of programming language research.

7.2 Related references

Frank de Boer has provided a proof system for POOL [dB91] which he shows to be consistent and complete with respect to an operational semantics. The assertion language works on three levels and is not first order (although it is not a higher order logic in the sense that, say, HOL is). There are also some restrictions of the POOL language.

Another paper by the current authors [JA16] looks at four complete formal descriptions of ALGOL 60, making technical comparisons as well as providing a historical context for the development of the semantic styles in general and the creation of the descriptions in particular.

7.3 Conclusions

A number of the most important challenges presented by programming languages to formal description are discussed in this paper.

- The challenge of associating identifiers with variable values is solved in operational and denotational semantics with a notion of state that is essentially the same in both cases. In axiomatic semantics an explicit state

is apparently avoided, but the meta-variables used in assertions essentially form an implicit state.

- In axiomatic semantics, phrase structuring in programming languages, such as that used in blocks and procedures, is handled by copying text and careful name substitution to avoid clashes. In model-oriented approaches, an abstract environment associates identifiers with locations. This is once again similar in both denotational and operational semantics.

- One area in which the various approaches differ significantly is handling non-determinism and concurrency: in SOS, a relation is defined economically by factoring out the non-determinism into the way in which rules match configurations; in axiomatic approaches a number of options have been explored including separation logic, temporal logic and rely/guarantee; and denotational semantics requires complex refactoring of its domain spaces.

- The description of an illustrative concurrent object-oriented language indicates that it may be easiest to use an SOS approach to bring all these aspects together in a readable form.

Thus it has been shown that despite apparently fundamental differences and notation divergence, the practical experience of modelling programming languages often remains remarkably similar across approaches.

The complexity of formally recording semantics for practical programming languages — larger and more feature-rich than the one demonstrated in this paper — seems unavoidable. Writing and understanding formal descriptions of languages always presents a steep learning curve and this investment of time feels to be in competition with the 'intuitive' understanding that most language designers and programmers assume they possess. Due to this, most programming languages are not even described formally *post facto*, let alone during the design process. Sadly, most programming languages are also not very good: they are hard to learn, too packed with features whose interactions prove awkward, or their behaviour is difficult to predict. With more careful use of formalism at an appropriate point in the design phase, some of these unfortunate problems could be avoided. Although working out a formal semantics is a non-trivial task, it takes significantly less time than building a compiler and the former provides a better basis for thought experiments than the latter.

References

[Abr10] J.-R. Abrial. *The Event-B Book*. Cambridge University Press, Cambridge, UK, 2010.

[ACJ72] 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.

[Acz82] P. Aczel. A note on program verification. (private communication) Manuscript, Manchester, January 1982.

[Acz83] P. H. G. Aczel. On an inference rule for parallel composition. private communication, 1983.

[AGM92] S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors. *Handbook of logic in computer science (vol. 2): background: computational structures*. Oxford University Press, Inc., New York, NY, USA, 1992.

[Ame89] P.H.M. America. The practical importance of formal semantics. In *J.W. de Bakker, 25 jaar semantiek*. CWI, 1989.

[ANS76] ANSI. Programming language PL/I. Technical Report X3.53-1976, American National Standard, 1976.

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: a survey—part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[Apt84] K. R. Apt. Ten years of Hoare's logic: A survey – part II: Nondeterminism. *Theoretical Computer Science*, 28:83–109, 1984.

[AR92] Pierre America and Jan Rutten. A layered semantics for a parallel object-oriented language. *Formal Aspects of Computing*, 4(4):376–408, 1992.

[BBG+60] John W Backus, Friedrich L Bauer, Julien Green, C Katz, John McCarthy, P Naur, AJ Perlis, H Rutishauser, K Samelson, B Vauquois, et al. Report on the algorithmic language ALGOL 60. *Numerische Mathematik*, 2(1):106–136, 1960.

[BBH+74] Hans Bekič, Dines Bjørner, Wolfgang Henhapl, Cliff B. Jones, and Peter Lucas. A formal definition of a PL/I subset. Technical Report 25.139, IBM Laboratory Vienna, December 1974.

[BIJW75] Hans Bekič, H. Izbicki, Cliff B. Jones, and F. Weissenböck. Some experiments with using a formal language definition in compiler development. Laboratory note LN 25.3.107, IBM Laboratory Vienna, December 1975.

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

[BJ82] Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development*. Prentice Hall International, 1982.

[BK84] Jan A Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and control*, 60(1-3):109–137, 1984.

[BO80] Dines Bjørner and Ole Nybye Oest, editors. *Towards a formal description of Ada*. Number 98 in LNCS. Springer-Verlag, 1980.

[BO16] Stephen Brookes and Peter W O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, 2016.

[Bur66] Rod M. Burstall. Semantics of assignment. *Machine Intelligence*, 2:3–20, 1966.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A systematic Introduction*. Springer Verlag, 1998.

[CG90] Bernard Carré and Jonathan Garnsworthy. Spark—an annotated Ada subset for safety-critical programming. In *Proceedings of the Conference on TRI-ADA '90*, TRI-Ada '90, pages 392–402. ACM, 1990.

[CH72] Maurice Clint and C. A. R. Hoare. Program proving: Jumps and functions. *Acta informatica*, 1(3):214–224, 1972.

[CHM16] Robert J. Colvin, Ian J. Hayes, and Larissa A. Meinicke. Designing a semantic model for a wide-spectrum language with concurrency. *Formal Aspects of Computing*, pages 1–22, 2016.

[CJ06] J. W. Coleman and C. B. Jones. Guaranteeing the soundness of rely/guarantee rules. Technical Report CS-TR-955, School of Computing Science, University of Newcastle, March 2006.

[Col08] Joseph William Coleman. *Constructing a Tractable Reasoning Framework upon a Fine-Grained Structural Operational Semantics*. PhD thesis, Newcastle University, January 2008.

[Dat82] CJ Date. A formal definition of the relational model. *ACM Sigmod record*, 13(1):18–29, 1982.

[dB91] Frank de Boer. A proof system for the language POOL. *Foundations of Object-Oriented Languages*, pages 124–150, 1991.

[dBDBZ80] Jacobus Willem de Bakker, Arie De Bruin, and Jeffery Zucker. *Mathematical theory of program correctness*, volume 980. Prentice-Hall International London, 1980.

[Dij68] E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., USA, 1976.

[DMN68] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. SIMULA 67 common base language. Technical Report S-2, Norwegian Computing Center, Oslo, 1968.

[Don76] J. E. Donahue. *Complementary Definitions of Programming Language Semantics*, volume 42 of *LNCS*. Springer-Verlag, 1976.

[dR01] W. P. de Roever. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, New York, NY, USA, 1990.

[Flo67] 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.

[Gor75] M. Gordon. Operational reasoning and denotational semantics. Technical Report STAN-CS-75-506, Stanford University, Computer Science Department, August 1975.

[GP99] Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax involving binders. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99. IEEE Computer Society, 1999.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Hay16] I. J. Hayes. Generalised rely-guarantee concurrency: An algebraic foundation. *Formal Aspects of Computing*, 28(6):1057–1078, November 2016.

[HCM+16] I.J. Hayes, R.J. Colvin, L.A. Meinicke, K. Winter, and A. Velykis. An algebra of synchronous atomic steps. In J. Fitzgerald, C. Heitmeyer, S. Gnesi, and A. Philippou, editors, *FM 2016: Formal Methods: 21st International Symposium, Proceedings*, volume 9995 of *LNCS*, pages 352–369, Cham, November 2016. Springer International Publishing.

[HJ73] K. V. Hanford and C. B. Jones. Dynamic syntax: A concept for the definition of the syntax of programming languages. In *Annual Review in Automatic Programming*, volume 7, pages 115–140. Pergamon, 1973.

[HJ98] C.A.R. Hoare and H. Jifeng. *Unifying theories of programming*. Prentice Hall, 1998.

[HJ08] John R. D. Hughes and C. B. Jones. Reasoning about programs via operational semantics: Requirements for a support system. *Automated Software Engineering*, 15(3–4):299–312, 2008.

[HJC14] Ian J. Hayes, Cliff B. Jones, and Robert J. Colvin. Laws and semantics for rely-guarantee refinement. Technical Report CS-TR-1425, Newcastle University, July 2014.

[HMRC87] Richard C. Holt, Philip A. Matthews, J. Alan Rosselet, and James R. Cordy. *The Turing Programming Language: Design and Definition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[HMSW11] C. A. R. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene Algebra and its foundations. *J. Log. Algebr. Program.*, 80(6):266–296, 2011.

[HMT87] Robert Harper, Robin Milner, and Mads Tofte. *The Semantics of Standard ML: Version 1.* University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1987. Hard copy.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa71a] C.A.R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium On Semantics of Algorithmic Languages — Lecture Notes in Mathematics 188*, pages 102–116. Springer-Verlag, 1971.

[Hoa71b] Charles Anthony Richard Hoare. Proof of a program: FIND. *Communications of the ACM*, 14(1):39–45, January 1971.

[Hoa72a] C.A.R. Hoare. A Note on the FOR Statement. *BIT*, 12(3):334–341, 1972.

[Hoa72b] C.A.R. Hoare. Towards a theory of parallel programming. In *Operating System Techniques*, pages 61–71. Academic Press, 1972.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hug11] John Robert Derek Hughes. *Reasoning about programs using operational semantics and the role of a proof support tool*. PhD thesis, Newcastle University, 2011.

[HvS12] Tony Hoare and Stephan van Staden. In praise of algebra. *Formal Aspects of Computing*, 24(4-6):423–431, 2012.

[HW73] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

[Izb75] H. Izbicki. On a consistency proof of a chapter of a formal definition of a PL/I subset. Technical Report TR 25.142, IBM Laboratory Vienna, February 1975.

[JA16] Cliff B. Jones and Troy K. Astarte. An Exegesis of Four Formal Descriptions of ALGOL 60. Technical Report CS-TR-1498, Newcastle University School of Computer Science, September 2016. Forthcoming as a paper in the HaPoP 2016 proceedings.

[JHC15] Cliff B. Jones, Ian J. Hayes, and Robert J. Colvin. Balancing expressiveness in formal approaches to concurrency. *Formal Aspects of Computing*, 27(3):465–497, 2015.

[JL71] 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.

[JLRW05] C. B. Jones, D. Lomet, A. Romanovsky, and G. Weikum. The atomic manifesto: a story in four quarks. *ACM SIGMOD Record*, 34(1):63–69, 2005.

[Jon69] C. B. Jones. A proof of the correctness of some optimising techniques. Technical Report LN 25.3.051, IBM Laboratory, Vienna, June 1969.

[Jon76] C. B. Jones. Formal definition in compiler development. Technical Report 25.145, IBM Laboratory Vienna, February 1976.

[Jon78] Cliff B. Jones. Denotational semantics of goto: An exit formulation and its relation to continuations. In Bjørner and Jones [BJ78], pages 278–304.

[Jon80] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International, Englewood Cliffs, N.J., USA, 1980.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[Jon82] Cliff B. Jones. More on exception mechanisms. In Bjørner and Jones [BJ82], chapter 5, pages 125–140.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[Jon03] Cliff B. Jones. The early search for tractable ways of reasoning about programs. *IEEE, Annals of the History of Computing*, 25(2):26–49, 2003.

[JP11] Cliff B. Jones and Ken G. Pierce. Elucidating concurrent algorithms via layers of abstraction and reification. *Formal Aspects of Computing*, 23(3):289–306, 2011.

[JY15] Cliff B. Jones and Nisansala Yatapanage. Reasoning about separation using abstraction and reification. In Radu Calinescu and Bernhard Rumpe, editors, *Software Engineering and Formal Methods*, volume 9276 of *LNCS*, pages 3–19. Springer, 2015.

[Kah87] Gilles Kahn. Natural semantics. In *STACS '87: Proc. Fourth Annual Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag, 1987.

[Kin69] J. C. King. *A Program Verifier*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1969.

[Knu64] Donald E. Knuth. Man or boy. *ALGOL Bulletin*, 17(7), 1964.

[Knu68] Donald E Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, 1968.

[Knu74] D. E. Knuth. Structured programming with GO TO statements. Technical Report STAN-CS-74-416, Computer Science Dept, Stanford University, May 1974.

[Koz97] Dexter Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, May 1997.

[Lab66] Vienna Laboratory. Formal definition of PL/I (Universal Language Document No. 3). Technical Report 25.071, IBM Laboratory Vienna, December 1966.

[Lan65a] Peter J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Communications of the ACM*, 8(2):89–101, February 1965.

[Lan65b] Peter J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part II. *Communications of the ACM*, 8(3):158–167, March 1965.

[Lau71] 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 Lab. Vienna.

[LPP70] David C Luckham, David Michael Ritchie Park, and Michael S Paterson. On formalised computer programs. *Journal of Computer and System Sciences*, 4(3):220–249, 1970.

[Luc68] Peter Lucas. Two constructive realisations of the block concept and their equivalence. Technical Report TR 25.085, IBM Laboratory Vienna, June 1968.

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

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[McC62] John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, volume 62, pages 21–28, 1962.

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

[Men64] Elliott Mendelson. *Introduction to Mathematical Logic*. van Norstrand, 1964.

[Mey88] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mog89] Eugenio Moggi. *An Abstract View of Programming Languages*. PhD thesis, Edinburgh University Laboratory for the Foundation of Computer Science, 1989.

[Mor94] C. C. Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.

[Mos11] Peter D. Mosses. VDM semantics of programming languages: combinators and monads. *Formal Aspects of Computing*, 23(2):221–238, 2011.

[MP67] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Mathematical aspects of computer science*, 1, 1967.

[MS74] Robert Milne and Christopher Strachey. A theory of programming language semantics. Privately circulated, 1974. Submitted for the Adams Prize.

[MS76] R. Milne and C. Strachey. *A Theory of Programming Language Semantics (Parts A and B)*. Chapman and Hall, 1976.

[NK13] Tobias Nipkow and Gerwin Klein. *Concrete Semantics. A Proof Assistant Approach*. Springer, 2013.

[NN92] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.

[O'H07] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, May 2007.

[Pag81] Frank G. Pagan. *Formal Specification of Programming Languages*. Prentice-Hall, 1981.

[Pai67] J. A. Painter. Semantic correctness of a compiler for an Algol-like language. Technical Report AI Memo 44, Computer Science Department, Stanford University, March 1967.

[Par10] Matthew Parkinson. The next 700 separation logics. In Gary Leavens, Peter O'Hearn, and Sriram Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *LNCS*, pages 169–182. Springer Berlin / Heidelberg, 2010.

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

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Plo76] G. D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5:452–487, September 1976.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[Pra65] Dag Prawitz. *Natural Deduction: a Proof-Theoretical Study*. Dover publications, 1965.

[Rey78] John C. Reynolds. Syntactic control of interference. In *Proceedings of fifth POPL*, pages 39–46. ACM, 1978.

[Rey89] John C. Reynolds. Syntactic control of interference: Part 2. In *Proceedings of 16th ICALP*, volume 372 of *LNCS*, pages 704–722. Springer-Verlag, 1989.

[RR62] B Randell and LJ Russell. Discussions on ALGOL translation at Mathematisch Centrum. *English Electric Report W/AT*, 841, 1962.

[San99] Davide Sangiorgi. Typed pi-calculus at work: A correctness proof of Jones's parallelisation transformation on concurrent objects. *TAPOS*, 5(1):25–33, 1999.

[Sco80] Dana Scott. Lambda calculus: some models, some philosophy. *Studies in Logic and the Foundations of Mathematics*, 101:223–265, 1980.

[Smy76] M. B. Smyth. Powerdomains. Technical report, Department of Computer Science, University of Warwick, May 1976.

[Ste66] T. B. Steel. *Formal Language Description Languages for Computer Programming*. North-Holland, 1966.

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

[Tur49] A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University Mathematical Laboratory, Cambridge, June 1949.

[Tur09] Raymond Turner. The meaning of programming languages. *American Philosophical Association Newsletter on Philosophy and Computers*, 9(1):2–6, 2009.

[vdH17] Gauthier van den Hove. *New Insights from Old Programs: The Structure of the First ALGOL 60 System*. PhD thesis, Iniversity of Amsterdam, 2017.

[vWMPK69] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster. *Report on the Algorithmic Language ALGOL 68*. Mathematisch Centrum, Amsterdam, October 1969. Second printing , MR 101.

[Wal67] Kurt Walk. Minutes of the 1st meeting of IFIP WG 2.2 on Formal Language Description Languages. Kept in the van Wijngaarden archive, September 1967. Held in Porto Conte, Alghero, Sardinia.

[Wal69] Kurt Walk. Minutes of the 3rd meeting of IFIP WG 2.2 on Formal Language Description Languages, April 1969. Held in Vienna, Austria.

[Wei75] F. Weissenböck. A formal interface specification. Technical Report TR 25.141, IBM Laboratory Vienna, February 1975.

[Woo93] Mark Woodman. A taste of the Modula-2 standard. *ACM SIGPLAN Notices*, 28(9):15–24, September 1993.

[Yon90] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990. ISBN 0-262-24029-7.