ALGEBRAIC SPECIFICATION OF PROGRAMMING LANGUAGE SEMANTICS

(Extended Abstract)

Peter Mosses

<u>Rapporteur</u>: Mr. R.C. Millichamp

Abstract

In this talk I shall first give a summary of my views on the aims and uses of formal specification of programming languages, and indicate the main approaches. Denotational semantics will be considered in particular, and some shortcomings of this approach will be suggested. A particular sort of algebraic semantics, called "A-Semantics", will be proposed as a means of alleviating these shortcomings, and a simple example will be given. Finally, experience with using A-Semantics in teaching Denotational Semantics will be reported.

Formal Specification of Programming Languages

The following seem to be general aims, although various approaches weight them unequally and often have some special aims.

In giving a full specification of a programming language it is necessary to specify both its syntax (context-free, context-sensitive) and its so-called "semantics". This "semantics" associates with each program some convenient (for the users of the language) abstraction from reality, usually an abstract mathematical structure or function. (Unfortunately, it seems to be too late to prevent this metaphorical use of linguistic terminology with its anthropomorphic connotations).

Formality of specification is essential if one is to be able to reason about programs and languages. The aim is to capture precisely the desired "semantics" of a language, thus obtaining a firm criterion for when an implementation of the language is correct (w.r.t. the specification).

A semantic specification of a programming language aims to communicate an understanding of the language to its readers, but also important is the understanding of the language that is gained by the authors of the specification. Clearly, conciseness of expression is a (subsidiary) aim.

Formal specifications of programming languages may be useful in the following situations. During **language design** they may document design decisions - they sometimes guide those decisions in that there are differences in the comparative ease of specifying various language features. When a language design is complete, a formal specification may be used as a standard, giving the properties that must be provided by implementations - and may be relied upon by programmers. Moreover, a formal specification may be used as the starting point for the systematic (or even automatic) implementation of the specified language. For program verification, a formal specification of a programming language may be used directly, or else used to show the soundness of a program proof logic. As regards the teaching of Computer Science, apart from the study of formal semantics one might hope to use formal specifications in courses on "concepts of programming languages", but at present there are difficulties with specifying language features independently of each other. (The work on "A-Semantics" described here attempts to circumvent these difficulties).

Main Approaches

We shall not consider the specification of the concrete syntax of programming languages here, but restrict our attention to semantics, associating abstract mathematical values with abstract programs which may be considered as "trees" conforming to some abstract syntax (described by a grammar). The following comments do not attempt to give a general survey of approaches to formal semantics, but may be of use in showing readers unfamiliar with formal semantics where Denotational Semantics (and A-Semantics) differ radically from other approaces.

Operational Semantics is specified by defining an (abstract) interpreter that, when given a program and its input, goes through a sequence of states in computing the output. It can be convenient to translate abstract programs into (low-level) abstract code, rather than interpreting them directly. Although the "meaning" of programs specified in this way may be identified with the input-output function (or relation) associated with programs in this way, these I-O functions can only be investigated by considering the operation of the abstract interpreter, which is somewhat unsatisfactory. The Vienna Definition Language (VDL) is a good example of the operational approach.

Axiomatic Semantics is specified by giving a proof system whose formulae involve programs and assertions about program variables. The axioms of the proof system involve primitive commands, and for each construct forming a compound command there is some inference rule allowing assertions about the compound to be inferred from assertions about its components. The "meaning" associated with a program by this means may be regarded as a function from an assertion about the input-output relation of a program to its theoremhood: provable, the negation provable, or neither. In any case this is, as with operational semantics, a rather indirect way of getting at the program's input-output relation. (It is well-suited to program verification, though).

Denotational Semantics is specified by giving a definition of a semantic function mapping programs to (mathematical) input-output functions. The semantic function is generally defined recursively, associating abstract mathematical values (structures or functions, but not bits of program!) with each program phrase. The value thus associated with a phrase is called its denotation, and it is required that the denotations of phrases depend only on the denotations of their sub-phrases. This enables properties of programs to be proved by structural induction. Although there is no underlying abstract machine, a Denotational Semantics makes explicit how the output of a program depends on its input. Whereas Operational and Axiomatic Semantics make do with standard Set Theory and Logic, Denotational Semantics requires a theory permitting the unique solution of recursive function definitions ($f = \sim f \sim$) and value space definitions ($V \cong (V \rightarrow V)$). Such is provided by Scott's Theory of Domains; so-called λ-notation is used for specifying particular elements of value spaces (domains).

Consider a simple example: a Denotational Semantics for LOOP, given in Table 1. The Abstract Syntax of the specification uses BNF to define a domain of abstract programs on tree structures - they can be imagined as derivation trees according to the given grammar, each node being labelled by the production used. The semantic values introduce domains that are used in defining denotations of programs - here note that both the input and output of programs are natural numbers in the domain N. (LOOP programs always terminate (normally), and the richness of domain theory is not needed here at all). Then the semantic functions are specified, mutually recursively and by cases on the productions of the Abstract Syntax, giving the denotation of each program phrase by so-called Semantic Equations.

A-Semantics

Over the last five years I have been developing a particular form of Denotational Semantics called A-Semantics (for want of a better name - the 'A' might stand for 'Abstract', 'Algebraic', etc.). In fact A-Semantics is formally based on Initial Algebra Semantics, an explicitly algebraic reformulation of Denotational Semantics, where Abstract Syntax is given as an (algebraic) abstract data type and the semantic function(s) for the language is(are) homomorphic. However, with A-Semantics the semantic values (as well as the Abstract Syntax) are given as abstract data types, instead of domains. These abstract data types, called abstract semantic algebras, differ from the well-known examples in the literature in that their values correspond to "actions" (potential computations) rather than data structures - their operators are combinators for actions, e.g. sequential execution of two actions.

The specific aims of A-Semantics are: to faciliate the re-use of parts (modules) of semantic descriptions; to give good modifiability and extensibility; and to exhibit operational concepts (order of execution, dataflow, etc.) in semantic descriptions. I consider standard Denotational Semantics to have some shortcomings on these points. The main reason for these shortcomings seems to be that a standard Denotational Semantics is similar in structure to a COBOL(!) program: first come some data definitions (the Semantic Values domains), then function definitions (the Semantic Functions) which depend on the precise structure of the data definitions. Unfortunately it is often necessary to use different domains to model different language constructs, which makes it difficult for different specifications to share parts; moreover, extending a Denotational Semantics to include new language constructs may require a change of domains, entailing a tedious rewriting of the Semantic Equations as well - for example, this phenomenon occurs when adding non-deterministic constructs to a deterministic language, or jumps to a procedural language.

Finally, it is quite difficult to read   -notation "operationally" - a considerable familiarity with the properties of higher-order functions on Scott-domains is required. One might claim it as a virtue of standard Denotational Semantics that it succeeds in specifying input-output functions for programs *without* suggesting any order of execution (e.g.) - but I consider that it is a matter of language design to decide on an *intended* order of execution (perhaps partial, i.e. non-deterministic), and that a semantic specification should document such decisions, thus guiding both implementors and programmers to the same operational understanding of the specified language.

A-Semantics aims to achieve re-use of parts of semantic specifications by the abstraction of abstract semantic algebras with operators corresponding to standard, language-independent concepts of computation (i.e. operational concepts). Good modifiability and extensibility is to come from each semantic equation referring *only* to operators directly expressing the concepts underlying the language construct that it specifies (e.g. in specifying a semantic equation for binary operators in arithmetic expressions, no explicit assumption is to be made about the presence or absence of side-effects, free variables, etc.). The formal specifications of the abstract semantic algebras give algebraic laws reinforcing an informal understanding of their operators (on actions) corresponding to operational concepts (e.g. using an infix semicolon for sequential execution, we have a;(a';a") = (a;a');a").

It would be inappropriate to attempt a proper explanation of an example of A-Semantics here. Hopefully, the semantic equations in Table 2, which are part of a full A-Semantics for LOOP, may give some idea of the differences from standard Denotational Semantics. The following informal explanation of the operators used in the example is *not* intended to be complete. (Note that everything inside the emphatic brackets ⟦ ⟧ is syntactic, and should not be confused with the semantic operators outside.)

Let a_1 and a_2 be *actions* - perhaps with *side-effects*, perhaps *consuming* and/or *producing* sequences of *values* v. Then $a_1;a_2$ is the compound action in which a_1 is executed (to completion) before a_2 is executed. If a_1 and a_2 produce sequences of values, these are concatenated into a single sequence, and produced by the whole action. (a_1 and a_2 may consume identical sequences of values in $a_1;a_2$, but this is not needed for LOOP.) The action $a_1!a_2$ also has a_1 executed before a_2, but here the values produced by a_1 are passed directly to a_2 for consumption, i.e. this is left-to-right functional composition. The empty action ( ) has no side-effects and produces no values.

Let x be a <u>name</u> for a value. Then x → a_1 is an action that consumes a value, and this may be referred to by @ x occurring in a_1. In general @ v is an action (with no side-effects) producing the value v .

The primitive action <u>update</u> consumes a variable and a value, and has the side-effect of assigning the value to the variable. The action <u>contents</u> consumes a variable and produces the last value assigned to that variable.

The action <u>while</u> a_1 <u>do</u> a_2, with a_1 producing a truth-value and both a_1 and a_2 consuming values produced by the previous iteration of a_2, is a standard operator. However, <u>zero-vars</u>, having the (side-)effect of assigning 0 to "every" variable, is not available in the standard abstract semantic algebras, and has to be specified in an auxiliary abstract semantic algebra. (Because of the restricted use of <u>zero-vars</u> in LOOP, the axiom

<u>zero-vars</u> ; (@ var ! <u>contents</u>) = <u>zero-vars</u>; @ 0

is sufficient to specify it.)

Please do not worry about the appropriateness of the concrete symbols and notation used for the above operators, they are only intended as a rudimentary vehicle for the underlying (operational) concepts.

Teaching Denotational Semantics

Let us now consider the topic of our Seminar explicitly: The Teaching of Computer Science. Perhaps a few words about my background are appropriate here. I read Mathematics at Oxford, and then studied for an M.Sc. and a D.Phil. at the late Professor Strachey's Programming Research Group. My doctoral dissertation was on "Mathematical Semantics and Compiler Generation", and treated the theory and implementation of the Semantics Implementation System SIS, which I have described earlier at this meeting. After a couple of years as a post-doctoral Research Assistant at Oxford, I went to Aarhus University, Denmark, where I have since been teaching courses on formal semantics, concepts of programming languages and (recently) algebraic methods for computer science. My research work has been devoted to Denotational (and now A-) Semantics.

At Aarhus, the students have already had at least 3 years of the combined course on Mathematics and Computer Science (min. $5\frac{1}{2}$ years duration) before they may choose to follow my one-semester (15 week) course on Denotational Semantics. They spend about 10 hours per week on this course, including 3 contact-hours.

I used to teach Denotational Semantics using Mike Gordon's book "The Denotational Description of Programming Languages", supplemented by some (unpublished) lecture notes on Scott Domain Theory by Chris Wadsworth. I would start by considering higher-order functions and λ-notation, then cover Gordon's book with its extensive examples of modelling programming language constructs on functions. The course would finish by treating Domain Theory (rather superficially), with the students using SIS to implement a small subset of Pascal at the same time. This course seemed to work fairly well as an introduction to standard Denotational Semantics, although a more leisurely two-semester course could certainly be more thorough.

Now, I prefer to teach Denotational Semantics via A-Semantics, using my own (unpublished) lecture notes. The course, which has been run twice, starts with an introduction to the algebraic specification of abstract data types; the initial algebra approach is used, which has the advantage that Initial Algebra Semantics can then be explained without further preparation. Then various standard abstract semantic algebras are specified, and used in describing most of the constructs covered by Gordon. The course concludes with an introduction to λ-notation for higher-order functions, and a review of the standard modelling techniques used in Denotational Semantics. There is unfortunately no room in this one-semester course for more than a mention of Scott Domain Theory. Neither is it feasible to use SIS in connection with A-Semantics: SIS cannot interpret the specifications of abstract semantic algebras directly.

Table 1 : Standard Denotational Semantics for LOOP

Abstract Syntax (Domains)

($\underline{\text{Prog}}$)	Prog	::=	$\underline{\text{read}}$ var;cmd; $\underline{\text{write}}$ exp
($\underline{\text{Cmd}}$)	cmd	::=	cmd1; cmd2 $\mid$ var := exp
			$\underline{\text{to}}$ exp $\underline{\text{do}}$ cmd $\mid$ (cmd)
($\underline{\text{Exp}}$)	exp	::=	0 $\mid$ var $\mid$ $\underline{\text{succ}}$ exp
($\underline{\text{Var}}$)	var	---	$\underline{\text{standard}}$

Semantic Values (Domains)

$s \in \underline{S} = \underline{var} \rightarrow \underline{N}$ -- states

$n \in \underline{N}$ - - $\underline{\text{standard natural numbers}}$

Semantic Functions

$\mathcal{P} : \underline{Prog} \rightarrow (\underline{N} \rightarrow \underline{N})$

$\mathcal{P}[\![\underline{\text{read}}\ var;\ cmd;\ \underline{\text{write}}\ exp]\!](n) =$

$$\mathcal{E}[\![exp]\!](\ \mathcal{C}[\![cmd]\!](\ (\lambda var!\ 0)[n/var]))$$

$\mathcal{C} : \underline{cmd} \rightarrow (\underline{S} \rightarrow \underline{S})$

$\mathcal{C}[\![cmd1;\ cmd2]\!](S) = \mathcal{C}[\![cmd2]\!](\ \mathcal{C}[\![cmd1]\!](S))$

$\mathcal{C}[\![var{:=}exp]\!](S) = S\ [\ \mathcal{E}[\![exp]\!](S)\ /\ var\ ]$

$\mathcal{C}[\![\underline{\text{to}}\ exp\ \underline{\text{do}}\ cmd]\!](S) = (\mathcal{C}[\![cmd]\!])^{\mathcal{E}[\![exp]\!](S)}\ (S)$

$\mathcal{C}[\![\ (and)\ ]\!](S) = \mathcal{C}[\![cmd]\!](S)$

$\mathcal{E} : \underline{Exp} \rightarrow \underline{(S} \rightarrow \underline{N)}$

$\mathcal{E}[\![\ 0\ ]\!](S) = 0$

$\mathcal{E}[\![\ var\ ]\!](S) = S(var)$

$\mathcal{E}[\![\ \underline{\text{succ}}\ exp\ ]\!](S) = \mathcal{E}[\![exp]\!](S) + 1$

Table 2 : A-Semantics of LOOP

Abstract Syntax (Initial Algebra) - - as in Table 1

Semantic Values (Abstract Semantic Algebras)

- - references to standard modules defining the sort $\underline{A}$ of actions, and the required operators.

Semantic Functions

$\mathcal{P} : \underline{\text{Prog}} \rightarrow \underline{A}$

$\mathcal{P}\llbracket \underline{\text{read}}\ \text{var};\ \text{cmd};\ \underline{\text{write}}\ \text{exp} \rrbracket =$

$\qquad \underline{n} \rightarrow (\ \underline{\text{zero-vars}};\ (@\ \text{var};\ @\ \text{n})\ !\ \underline{\text{update}};$

$\qquad\qquad \mathcal{E}[\text{cmd}];\ \mathcal{E}[\text{exp}])$

$\mathcal{C} : \underline{\text{Cmd}} \rightarrow \underline{A}$

$\mathcal{C}\llbracket \text{cmd1};\ \text{cmd2} \rrbracket = \mathcal{C}\llbracket \text{cmd1} \rrbracket;\ \mathcal{C}\llbracket \text{cmd2} \rrbracket$

$\mathcal{C}\llbracket \text{var}{:=}\text{exp} \rrbracket = (@\ \text{var};\ \mathcal{E}\llbracket \text{exp} \rrbracket)\ !\ \underline{\text{update}}$

$\mathcal{C}\llbracket \underline{\text{to}}\ \text{exp}\ \underline{\text{do}}\ \text{cmd} \rrbracket =$

$\qquad \epsilon\llbracket \text{exp} \rrbracket\ !\ (\underline{\text{while}}\ \underline{n} \mapsto @\ \underline{\text{positive}}\ \underline{n}$

$\qquad\qquad \underline{\text{do}}\ \underline{n'} \mapsto (e\llbracket \text{cmd} \rrbracket; @\ \underline{\text{pred}}\ \underline{n'}))\ !\ (\ )$

$\mathcal{C}\llbracket (\text{cmd}) \rrbracket = \mathcal{C}\llbracket \text{cmd} \rrbracket$

$\mathcal{E} : \underline{\text{Exp}} \rightarrow \underline{A}$

$\mathcal{E}\llbracket 0 \rrbracket = @\ 0$

$\mathcal{E}\llbracket \text{var} \rrbracket = @\ \text{var}\ !\ \underline{\text{contents}}$

$\mathcal{E}\llbracket \underline{\text{succ}}\ \text{exp} \rrbracket = \mathcal{E}\llbracket \text{exp} \rrbracket\ !\ \underline{n} \mapsto @\ \underline{n} + 1$

Literature

D.S. Scott:

"Data Types as Lattices",
SIAM Journal on Computing 5 (1976) 522-587

"Domains for Denotational Semantics",
in Proc. ICALP 82, Aarhus, July 1982, LNCS 140 (Springer),
pp. 577-613.

R.D. Tennent:

"The Denotational Semantics of Programming Languages",
Comm. ACM 19 (1976) 437-453.

M.J.C. Gordon:

"The Denotational Description of Programming Languages",
(Springer, 1979).

ADJ (J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright):

"Initial Algebra Semantics and Continuous Algebras",
Journal ACM 24 (1977) 68-95.

"An Initial Algebra Approach to the Specification, Correctness
and Implementation of Abstract Data Types",
in : R. Yeh (ed.), Current Trends in Programming Methodology IV
(Prentice-Hall, 1979).

P.D. Mosses:

"Abstract Semantic Algebras!", to appear in: Proc. IFIP TC2
Working Conf. on Formal Description of Programming Concepts II,
Garmisch-Partenkirchen, June 1982 (North-Holland).

DISCUSSION

Professor Nakajima asked the speaker how long the course he taught was. **Professor Mosses** replied that it was given to students who had alreadv had four years teaching in Computer Science, and that the courses was fourteen to fifteen weeks long with ten hours per week, of which three were content hours. The course was originally based on Stoy's book but subsequentlv on the text by Gordon, with supplementary notes by Wadsworth.

Professor Burstall enquired why category theory was necessary in the course. The speaker felt that if you were prepared to rely on the existence of such things as the least fixed point then it wasn't necessary, but if the students were not willing to accept this, then one must introduce category theory.