A UNIFIED VIEW OF SEMANTICS

Mila E. Majster*
Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

Key words: Denotational semantics, assertion semantics, semantical algebra, consistency.

*Permanent address: Technische Universität München, Institut für Informatik. This work was done while the author was on a research leave at Cornell University which was supported by the Deutsche Forschungsgemeinschaft.
1. Introduction

Syntactical aspects of programming languages have been well understood for a long time already. Syntactical features are uniformly discussed in terms of grammars and have been investigated in formal language theory. In contrast to this, semantical aspects of programming languages seem harder to understand and formalize. A variety of approaches have been suggested to describe the semantics of a language, including e.g. VDL\([1,2,3,4]\) attributed grammars \([5]\), state machines \([6,7]\), \(\lambda\)-calculus and SECD-machine \([8,9]\), denotational semantics \([10-17]\) flowcharts with inductive assertions \([18]\), axioms and inference rules \([19,20]\), predicate transformers \([21]\), etc. An attempt to classify these approaches to describe semantics resulted in the classification operational-denotational-assert semantics.

Our aim is to compare the different approaches and to study their relationships. In order to do so we need a formal language, i.e. a frame in which the relationships can be investigated. We suggest to understand semantics in an algebra framework (section 2). The first ideas to use such an approach go back to \([23,24]\). Compare also \([22]\), \([25,26]\). In \([28]\) an algebraic formulation of Knuth's attribute semantics is attempted. In this paper we first show how in general operational, denotational and assertion semantics can be formulated using an algebra framework (section 3,4). Based on this presentation the main part of the paper is concerned with the investigation of the relation between the different approaches (section 4). We present a programming language PL\((C)\) and prove e.g. that the Dijkstra-style and Scott-style semantics are isomorphic. A diagram summarizing some of the results obtained here is found in section 4. In section 5 the abstraction from a particular specification technique is discussed.

2. Definitions

In the following both syntactical and semantical aspects of a programming li
Definition 1. Let $\Sigma$ be a finite set (of signatures), $S$ be a finite set (of sorts) and two mappings $r: \Sigma \to S$, $d: \Sigma \to S^*$. A $S$-sorted $\Sigma$-algebras is a pair

$$D = \{D_s | s \in S\} \cup \{O_o | o \in \Sigma\}$$

where $D_s$ is a set and $O_o$ a mapping

$$O_o : D_{s_1} \times \ldots \times D_{s_n} \to D_s$$

where $d(o) = s_1 \ldots s_n$, $r(o) = s$. A homomorphism between $S$-sorted $\Sigma$-algebras $D, D'$ is a family of mappings $\{h_s\}_{s \in S}$

$$h_s : D_s \to D'_s$$

such that for each $o \in \Sigma, x_i \in D_{s_i}$

$$h_s(O_o(x_1 \ldots x_n)) = O'_o(h_{s_1}(x_1), \ldots, h_{s_n}(x_n))$$

where $D' = \{D'_s | s \in S\} \cup \{O'_o | o \in \Sigma\}$.

If one compares the different approaches for semantical specification one finds that they have one feature in common: the semantical description of the language is heavily based on the way a program is composed of smaller components, i.e. on the syntax of the language. The above observation can be phrased differently by: the meaning of a construct is determined by the meanings of its constituents. This suggests that the meaning function has homomorphical character.

In the following we will use context-free grammars for the syntactical description of a programming language and associate a syntactical algebra with the grammar.

Definition 2. Let $G = (N,T,P,S)$ be a context-free grammar, where $N$ denotes the set of nonterminals, $T$ the set of terminals, $P$ the set of productions and $S$ the start symbol. Let for each $A \in N$

$$L_A = \{w \in T^*: A \Rightarrow^* w\}$$
For each production \( p = (A_1 u_1 A_2 u_2 \ldots u_n A_n u_{n+1}) \) with \( A_i \in N, u_i \in T^* \), let

\[
O_p : L_{A_1} \times \ldots \times L_{A_n} \to L_A
\]

\[
O_p (x_1, \ldots x_n) = u_1 x_1 \ldots u_n x_n u_{n+1}
\]

The \( N \)-sorted \( P \)-algebra

\[
A_G = \{(L_A)_{A \in N}, \{O_p\}_{p \in P}\}
\]

is called the syntactical algebra of \( G \).

A semantical algebra for \( G \) is a \( N \)-sorted \( P \)-algebra that is a homomorphic image of \( A_G \).

**Example**  Consider the grammar \( G \)

\[
\langle \text{command} \rangle ::= \langle \text{id} \rangle ::= \langle \text{exp} \rangle \ | \ \text{command; command}
\]

\[
\langle \text{exp} \rangle ::= \langle \text{id} \rangle \ | \ \langle \text{cons} \rangle \ | \ \langle \text{exp} \rangle + \langle \text{exp} \rangle
\]

\[
\langle \text{id} \rangle ::= A \ | \ B \ | \ \ldots \ Z
\]

\[
\langle \text{cons} \rangle ::= 0 \ | \ \text{succ}(\langle \text{cons} \rangle)
\]

The syntactical algebra is

\[
A_G = \{(L_{\text{co}}, L_{\text{exp}}, L_{\text{id}}, L_{\text{cons}}), \{O_i\}_{i=1}^{32}\}
\]

where e.g.

\[
O_1 : L_{\text{id}} \times L_{\text{exp}} \to L_{\text{co}}
\]

takes an identifier and an expression and produces the assignment command,

\[
O_2 : L_{\text{co}} \times L_{\text{co}} \to L_{\text{co}}
\]

takes two commands and produces the composite command.

Of course, \( A_G \) is also a semantical algebra. To construct a more interesting semantical algebra we consider

\[
A_{\text{co}} = \{f: S \to S\}
\]

\[
A_{\text{id}} = L_{\text{id}}
\]

\[
A_{\text{cons}} = \mathbb{N}
\]

\[
A_{\text{exp}} = \{m: S \to \mathbb{N}\}
\]
and e.g.

\[ O_1: A_{id} \times A_{exp} \rightarrow A_{co} \]

\[ O_1(x, m) = f \text{ where } f: S \rightarrow S, \]

\[ f(s) = s' \text{ with } \]

\[ s'(x) = m(s), s'(y) = s(y) \text{ if } y \neq x \]

\[ O_2: A_{co} \times A_{co} \rightarrow A_{co} \]

\[ O_2(f_1, f_2) = f_2 \circ f_1 \]

Here \( S \) is the set of states. A state is a mapping from identifiers to values. Let \( A \) be the subalgebra generated by the operations of the algebra. Then \( A \) is a semantical algebra for \( G \).

For the general case one has

**Remark 1.** Let \( G = (N, T, P, S) \) an unambiguous reduced\(^5\) context free grammar, \( B \) a \( N \)-sorted \( P \)-algebra then there is a unique homomorphism \( h: A_G \rightarrow B \). Consequently every \( N \)-sorted \( P \)-algebra gives rise to exactly one semantical algebra for \( G \).

**Proof:** as \( G \) is reduced and unambiguous it is true that \( (N, T, P, A) \) is unambiguous for every \( AcN \). Hence there is exactly one derivation tree for every \( x \in L_{A'} \). \( AcN \). There is a one-to-one correspondence between the set of derivation trees and the words built from the operation symbols \( O_p \). Hence there is a homomorphism. Uniqueness can be shown by induction.

The above mentioned homomorphism can be thought of as the "meaning function."

By the previous remark principally any \( N \)-sorted \( P \)-algebra can be taken to provide a "meaning" to a language.

In the following section 3 we show how Hoare's assertion semantics and Dijkstra's predicate transformer semantics can be viewed as semantical algebras.

\(^5\)We call a grammar reduced if for every \( AcN \) there are \( u, v \in (N \cup T)^* \) such that \( S \Rightarrow uAv \).
For a given programming language only few semantical algebras will be of interest. In section 5 we discuss how the class of semantical algebras can be restricted.

3. Algebraic Formulation of Assertion Semantics

In the following we want to present a simple programming language and show how Hoare's and Dijkstra's assertion semantics can be presented in the above framework. The programming language allows assignment, conditional, while and concatenated statements. We do not want to specify the particular primitive operation and relation symbols which are used to build expressions in our programming language. Instead we assume that the expressions are the terms of some first-order language $L_1$. An extension language $L_2$ of $L_1$ will be used as assertion language for the assertion semantics.

Let the nonlogical symbols of $L_1$ be $f$, $g$, $h$... for functions and $p$, $q$, $r$... for predicates.

The syntax of the programming language is given by

$$<command>::= \langle id \rangle::= \langle exp \rangle \mid (\langle command \rangle \cdot \langle command \rangle) \mid$$
if $<boolexp>$ then $<command>$ else $<command>$ \mid
while $<boolexp>$ do $<command>$ \mid skip

$$<exp>::= \langle id \rangle \mid \langle const \rangle \mid f(\langle exp \rangle,...,\langle exp \rangle) \mid \underbrace{n}_n$$

where $f$ is a $n$-ary function symbol in $L_1$

$$<boolexp>::= p(\langle exp \rangle,...,\langle exp \rangle) \mid (\langle boolexp \rangle \lor \langle boolexp \rangle) \mid$$
$\neg \langle boolexp \rangle \mid (\langle boolexp \rangle \rightarrow \langle boolexp \rangle)$

where $p$ is a $n$-ary predicate symbol in $L_1$.

The constants and variables (identifiers) in the programming language are the constants and variables of $L_1$.$^2$

---

$^2$We assume that variables act as input to programs and have a value before the program is executed.
We denote the programming language generated by the above grammar by PL( L_1 ). The syntactical algebra associated with the above grammar A ( L_1 ) contains the domains L_co, L_exp, L_boolexp, L_id, L_cons and an operation for each production.

Let L_2 be an extension of L_1, i.e. the set of symbols of L_1 is contained in the set of symbols of L_2. We call L_2 the assertion language. A Hoare-like specification of the programming language PL( L_1 ) is a system for the deduction of formulas of the form F_1(X)F_2 where F_1 and F_2 are formulas of the assertion language L_2 and X is a program. It usually contains the axiom scheme for the assignment and rules for composite programs. In addition a rule of consequence is contained in the specification stating

\[ F_1 \vdash F_1', F_1(X)F_2, F_2 \vdash F_2' \]

\[ F_1'(X)F_2' \]

where F_1 \vdash F_1' expresses that F_1 can be proven from F_1' in some deductive system for the assertion language L_2. Up to now, however, we did not assume the existence of any deductive system for the language L_2. We will hence proceed as follows: let \( \mathcal{S} \) be a structure for the first-order language L_2, i.e. \( \mathcal{S} \) consists of a nonempty set | \( \mathcal{S} \) | together with a n-ary relation for each n-ary function symbol in L_2 and a n-ary relation for each n-ary relation symbol in L_2.

A formula F in L_2 is said to be true in the structure \( \mathcal{S} \) if its universal closure is true in \( \mathcal{S} \). The universal closure of a formula is obtained by binding all free occurrences of variables in F by universal quantifiers. Instead of the previous rule of consequence we use the modified rule of consequence

( \( \mathcal{S} \)-consequence): F_1 \vdash F_1', F_1(X)F_2, F_2 \vdash F_2' \]

\[ F_1'(X)F_2' \]

where F_1 \vdash F_1', F_2 \vdash F_2' are formulas in L_2 that are true in \( \mathcal{S} \).
Given a structure $S$ for the assertion language $L_2$ we construct a semantical algebra for $PL(L_1)$ from the usual rules of inferences [19] and the above rule of $S$-consequence. Let $F(L)$ be the set of all formulas in the first-order language $L$.

**Definition 3.** Let $\phi$ be a binary relation on $F(L)$, i.e. $\phi \subseteq F(L) \times F(L)$. The closure $CL(\phi, S)$ of $\phi$ with respect to the structure $S$ is a binary relation on $F(L)$ defined as follows:

1) $\phi \subseteq CL(\phi, S)$

2) If $(F_1, F_2) \in CL(\phi, S)$ and if $G_1 \rightarrow F_1$, $F_2 \rightarrow G_2$ are true in $S$ then $(G_1, G_2) \in CL(\phi, S)$

3) No other elements are in $CL(\phi, S)$

**Remark 2.** If $\phi_1 = CL(\phi_1, S)$, $\phi_2 = CL(\phi_2, S)$ then

a) $\phi_1 \circ \phi_2 = CL(\phi_1 \circ \phi_2, S)$

b) $\phi = CL(\phi, S)$ where

$$\phi = \{(F_1, F_2): (F_1 \land F_2) \in \phi_1 \text{ and } (F_1 \lor \neg F, F_2) \in \phi_2\}$$

for any quantifier-free formula $F$ in $F(L)$.

The above closure operation will serve us as a substitute for the rule of $S$-consequence.

Let $L_1$, an extension $L_2$ of $L_1$ and a structure $S$ for $L_2$ be given. We define an algebra with domains

$$A_{co} = \{\phi: \phi \text{ is a binary relation on } F(L_2)\}$$

$$A_{exp} = L_{exp}$$

$$A_{boolexp} = L_{boolexp}$$

$$A_{cons} = L_{cons}$$

$$A_{id} = L_{id}$$

Obviously, $S$ can also be viewed as providing a structure for the language $L_1$. 

\footnote{Obviously, $S$ can also be viewed as providing a structure for the language $L_1$.}
Let $\Sigma$ be a structure for $\mathcal{L}$. The algebra associated with $\Sigma$ is built from

function from programs and sets of states to sets of states. Each set of state can be thought of as being described by a predicate. We consider the function $\nu^\mathcal{L}$ as a predicate on the state space $\mathcal{S}$.

Let $\mathcal{P}(\mathcal{L})$ be the set of all programs for $\mathcal{L}$.

\[ \nu^\mathcal{L}(\mathcal{P}(\mathcal{L})) \]

We should be clear by now, how any other semantic specification of $\mathcal{L}$ in $\mathcal{S}$.

\[ \text{Proof: } \] by structural induction on $\mathcal{L}$.

(\text{resp.} (\Sigma, \mathcal{P}(\mathcal{L})) \)\text{H} \nu \text{ to } (\Sigma, \mathcal{P}(\mathcal{L})) \nu \text{ as the unique homomorphism from } (\Sigma, \mathcal{P}(\mathcal{L})) \text{ to } (\Sigma, \mathcal{P}(\mathcal{L})) \nu \text{.}

\[ x \in x \] \text{ or } (x^\mathcal{L} \in x^\mathcal{L}) \text{ or } (x \in x^\mathcal{L}) \text{ or } (x = x^\mathcal{L}) \text{ or } (x \neq x^\mathcal{L}) \text{ or } (x = x^\mathcal{L})

where $x$ are variables in $\Sigma$ then for any program $x \in \mathcal{P}(\mathcal{L})$.

Let $\mathcal{S}$ be a set for $\mathcal{L}$ and the rules of $\mathcal{L}$.

\[ \text{Remark: } \] Let $\mathcal{S}$ be a deductive system for $\mathcal{L}$. If $\mathcal{S}$ is a set for $\mathcal{L}$ and the rules of $\mathcal{L}$.

\[ \text{Let } \mathcal{V} \text{ denote the subalgebra generated by the operations.} \]

\[ \text{As before there} \]

\[ \text{and operations are before the closure with respect to } \mathcal{V} \text{.} \]

The closure $\mathcal{C}(\mathcal{D})$ with respect to $D, E$.

\[ \text{As before we build an algebra with} \]

\[ \text{The closure with respect to } \mathcal{L}, \mathcal{V}, \mathcal{P}(\mathcal{L}) \text{ is denoted by} \]

\[ \text{As before we build an algebra with} \]

\[ \text{where $F_L$ means that $F_L$ can be proven in $\mathcal{L}$ under hypotheses $P_L$.} \]
Let us now consider the case where we have a deductive system $D$ for our assert-
where $P(S)$ is the power set of the set of states $S$. A state $\text{true}$ is a mapping from the variables to $|S|$. 

$$A_{\text{exp}} = \{ m : S \to |S| \}$$

$$A_{\text{boolexp}} = \{ b : S \to \{ \text{true}, \text{false} \} \}$$

$$A_{\text{Id}} = L_{\text{Id}}$$

$$A_{\text{Cons}} = \text{constants of } S$$

$$O_{\text{skip}} = \lambda U$$

$$O_{\text{assign}}(x, m) = \lambda U_{x, m}$$

where $U_{x, m} = \{ s \in \text{Def}(m) : s' \in U \text{ with } s'(y) = s(y) \text{ for } y \neq x, s'(x) = m(s) \}$

$$O_{\text{concat}}(d_1, d_2) = \lambda U d_1(d_2(U))$$

$$O_{\text{if}}(b, d_1, d_2) = \lambda U \{ s \in \text{Def}(b) : b(s) = \text{true} \implies scd_1(U) \}$$

$$\cap \{ s \in \text{Def}(b) : b(s) = \text{false} \implies scd_2(U) \}$$

$$O_{\text{while}}(b, d) = \lambda U \{ s : \exists k : s \in H(k, U, b, d) \}$$

where $H(0, U, b, d) = \{ s \in \text{Def}(b) : b(s) = \text{false} \} \cap U$ and $H(k, U, b, d) =$

$$= \{ (s \in \text{Def}(b) : b(s) = \text{true} \implies \text{scd}(k \cdot k, U, b, d)) \} \cap \{ s \in \text{Def}(b) : b(s) = \text{false} \} \cup H(0, U, b, d) \}$$

Then for any $n$-ary function symbol $f$ in $L_1$ we take

$$O_f : A_{\text{exp}} \times \cdots A_{\text{exp}} \to A_{\text{exp}}$$

$$O_f(m_1, \ldots, m_n) = m'$$

where $m'$ is defined at $s$ if $m_i$ is defined at $s$ for all $i$ and $f_S$ (the interpretation of $f$ in $S$) is defined at $(m_1(s), \ldots, m_n(s))$ and then $m'(s) = f_S(m_1(s), \ldots, m_n(s))$. The remaining operations are defined similarly.

The subalgebra generated by the operations is a semantical algebra for $A(L_1)$. It will be denoted by $A_D(L_1, S)$.  

\text{Remember that it is assumed that variables act as an input to programs and have a value before a program is executed.}
4. Comparing Semantics

A denotational semantic description of the language PL($L_1$) is given by a structure $S$ for $L_1$ and by

$$A_{co} = [S_{\perp T} \rightarrow S_{\perp T}]$$

where $S_{\perp T} = S \cup \{\perp, \top\}$ is the complete flat lattice of states and $[\rightarrow]$ denotes the domain of continuous functions.

$$A_{exp} = [S_{\perp T} \rightarrow [S_{\perp T}]]$$

where $[S_{\perp T}]$ is the flat complete lattice constructed from the set $|S_{\perp T}|$. 

$$A_{boolean} = [S_{\perp T} \rightarrow \{true, false, \perp, \top\}]$$

$$A_{id} = L_{id}$$

$$A_{cons} = \text{constants of } S$$

The operations are

$$0_{skip} = \lambda s s$$

$$0_{assign}(x, m) = \lambda s s'$$

where

- $s' = \perp$ if $m(s) = \perp$
- $s' = \top$ if $m(s) = \top$
- $s'(y) = s(y)$ if $y \neq x$
- $s'(x) = m(s)$

$$0_{concat}(f_1, f_2) = \lambda s f_2(f_1(s))$$

$$0_{if}(b, f_1, f_2) = \lambda s s'$$

where

- $s' = \perp$ if $b(s) = \perp$
- $s' = \top$ if $b(s) = \top$
- $s' = f_1(s)$ if $b(s) = true$
- $s' = f_2(s)$ if $b(s) = false$

$$0_{while}(b, f)(s) = \begin{cases} s & \text{if } b(s) = \perp \\ t & \text{if } b(s) = \top \\ 0_{while}(b, f)(f(s)) & \text{if } b(s) = true \\ s & \text{if } b(s) = false \end{cases}$$
Some of the remaining operations are

\[ O_1: A_{id} \rightarrow A_{exp} \]

\[ O_1(x) = \lambda su \quad \text{where} \quad \begin{cases} \ u = 1 & \text{if } s = 1 \\ \ u = \tau & \text{if } s = \tau \\ \ u = s(x) & \text{else} \end{cases} \]

\[ O_2: A_{const} \rightarrow A_{exp} \]

\[ O_2(a) = \lambda s \quad \text{if } s = 1 \quad \text{then } 1 \\
\quad \text{if } s = \tau \quad \text{then } \tau \\
\quad \text{else } a \]

The subalgebra generated from the operations well be denoted by \( A_S( L_1, S ) \).

In order to be able to compare different semantical algebras we introduce

**Definition 4.** Let \( G \) be a context-free grammar, \( A, B \) semantical algebras for \( G \).

\( A \) is **strong-consistent** with \( B \) if there is a homomorphism from \( A \) to \( B \).

**Lemma 6.** Let \( G \) be an unambiguous reduced context-free grammar, \( A_G \) the syntactical algebra of \( G \), \( A, B \) semantical algebras. If there is a homomorphism \( h \) from \( A \) to \( B \) then \( h_A(z) = h_B(A(h^{-1}_A, A(z))) \) where \( h_A : A_G \rightarrow A \), \( h_B : A_G \rightarrow B \) are the unique homomorphisms existing according to Remark 1.

**Proof:** from the uniqueness of the homomorphism from \( A_G \) to \( B \) we get

\[
h_B(A(z)) = h_A(h^{-1}_A, A(z)) \quad \text{for } h_A \text{ is onto and } h_B(A(x_1)) = h_A(A(x_2))
\]

implies \( h_B(A(x_1)) = h_A(h^{-1}_A, A(x_1)) = h_A(A(x_2)) = h_B(A(x_2)) \)

hence, \( h_A(z) = h_A(h^{-1}_A, A(z)) = h_B(A(z)) \) for any \( x \in A_G \).

**Lemma 7.** Let \( G, A_G, A, B \) be as in Lemma 6. If for every nonterminal \( A \) of \( G \) the relation \( ((x_A, h_B(A(h^{-1}_A, A(x_A)))) : x_A \in L_1) \) is a mapping then \( h_A : A_A \rightarrow N \) with \( h_A(x) = h_B(A(z)) \) for some \( z \) with \( h_A(z) = x \) is a homomorphism.

**Proof:** it follows from the conditions that \( h_A(x) \) is a mapping. The homomorphism property is a consequence of the homomorphism property of \( h_A \) and \( h_B \).

\[ \Theta \text{For any function } f: A \rightarrow B \quad f^{-1}(b) = \{ a: f(a) = b \}. \]
By Lemma 6 and 7 we know what to do in order to establish that the semantic algebra $A$ is strong-consistent with $B$. We have to check if

$$h^A_A(x) = h^A_A(y)$$

implies $h^B_B(x) = h^B_B(y)$ for $x, y \in L_A$, $A \in N$. If this is the case then by Lemma 7 $A$ is strong-consistent with $B$. If it is not the case then $A$ is not strong-consistent with $B$ by Lemma 6.

Let us now consider the language $PL(L_1)$. The grammar of $PL(L_1)$ is unambiguous and reduced hence Lemma 6 and 7 can be applied. We compare first the Dijkstra-like semantics with the Scott-like semantics.

**Lemma 8.** Let $S$ be a structure for $L_1$. Let $h: A(L_1) + A_S(L_1, S)$

$$g: A(L_1) + A_D(L_1, S)$$

be the unique homomorphisms. Then for all $X \in L_C$

$$f = h^h(X) \text{ implies } g^X(X) = \lambda U^{-1}(U)$$

**Proof:** this can be proved by structural induction. The statement is trivial for $X = 0$ Skip. For $X = 0 \text{ assign }(x, E)$ we get with $h^{E}(E) = \#$ and $g^{E}(E) = \#$ that $f = h^h(X) = 0 \text{ assign }(x, \#)$. Given $U \in S$

$$f^{-1}(U) = \{s \in \text{Def}(m) \mid s' : s \in U \text{ where } s'(y) = s(y), y \neq x, s'(x) = m(s)\}$$

$$= \{s \in \text{Def}(m) : s' : s \in U \text{ where } s'(y) = s(y), y \neq x, s'(x) = \#(s)\}$$

$$= U_x, \#$$

Hence $\lambda U^{-1}(U) = g^X(X)$.

It remains to show that for $X_i \in L_C$, $i = 1, 2$ for which $f_i = h^h(X_i)$ implies $g^X_i(X_i) = \lambda U^{-1}(U)$ we may conclude that

$$0 \text{ concat}(X_1, X_2), 0 \text{ if}(B, X_1, X_2), 0 \text{ while}(B, X_1)$$

fulfill the condition. This is shown in a straightforward way.

**Lemma 9.** $A_S(L_1, S)$ is strong consistent with $A_D(L_1, S)$.

---

$\oplus^{-1}(U) = \{x \in S : f(x) \in U\}$.

$\ominus \text{Def}(m) = \{s : m(s) \neq 1 \text{ and } m(s) \neq 1\}$.
Proof: by Lemma 8.

\[ h_{\mathcal{C}_0}(X) = h_{\mathcal{C}_0}(Y) \] implies \( g_{\mathcal{C}_0}(X) = g_{\mathcal{C}_0}(Y) \). Hence by Lemma 7 there is a homomorphism from \( \mathcal{A}_S(\mathcal{L}_{\mathcal{L}_1}, \mathcal{S}) \) to \( \mathcal{A}_D(\mathcal{L}_{\mathcal{L}_1}, \mathcal{S}) \).

Lemma 10. \( \mathcal{A}_S(\mathcal{L}_{\mathcal{L}_1}, \mathcal{S}) \) and \( \mathcal{A}_D(\mathcal{L}_{\mathcal{L}_1}, \mathcal{S}) \) are isomorphic

Proof: by Lemma 8 we know that for each \( g_{\mathcal{C}_0}(X) \) there is an \( f = h_{\mathcal{C}_0}(X) \) such that \( g_{\mathcal{C}_0}(X) = \lambda U f^{-1}(U) \). We show \( f \) is unique. Assume \( g_{\mathcal{C}_0}(X) = \lambda U f_1^{-1}(U) = \lambda U f_2^{-1}(U) \), let \( s \in S \) then \( f_1^{-1}(\{s\}) = f_2^{-1}(\{s\}) \). Assume there is \( x : f_1(x) \neq f_2(x) \). Let w.l.o.g. assume that \( f_1(x) \in S \). Then \( x \in f_1^{-1}([f_1(x)]) = f_2^{-1}([f_1(x)]) \) hence we conclude that \( f_2 \) is also defined at \( x \), hence \( f_2(x) = f_2 f_2^{-1}([f_1(x)]) = [f_1(x)] \) hence \( f_2(x) = f_1(x) \).

Hence \( f \) is unique, hence the homomorphism from \( \mathcal{A}_S(\mathcal{L}_{\mathcal{L}_1}, \mathcal{S}) \) to \( \mathcal{A}_D(\mathcal{L}_{\mathcal{L}_1}, \mathcal{S}) \) is one-to-one; it is onto by definition.

Remark 11. Let \( d = \lambda U f^{-1}(U) \) and \( s \in S \). Let \( \{U_i\}_{i \in I} \) be the family of subsets of \( S \) such that \( s \in d(U_i) \). Then \( \bigcap_{i \in I} U_i \) contains exactly one element or \( I = \emptyset \).

By Lemma 10 and Remark 11 we can give the homomorphism from \( \mathcal{A}_D(\mathcal{L}_{\mathcal{L}_1}, \mathcal{S}) \) to \( \mathcal{A}_S(\mathcal{L}_{\mathcal{L}_1}, \mathcal{S}) \) by associating with each \( d = g_{\mathcal{C}_0}(X) \) the following mapping \( f : S \xrightarrow{} S \)

\[
 f(1) = 1 \\
 f(\tau) = \tau \\
 f(s) = \begin{cases} 
 nU & \text{if } \exists U : s \in F(U) \\
 d(U) & \text{if } s \in \bigcap_{i \in I} U_i \\
 1 & \text{else}
\end{cases}
\]

The above shows that basically the Scott-like semantics for \( \mathcal{L}_{\mathcal{L}} \) and the Dijkstra-like semantics are the same; moreover it is shown how the Scott-meaning of a program is calculated from the Dijkstra-meaning and vice versa.

Let us now compare Scott-like and Hoare-like semantics.

Definition 5. Let \( \mathcal{L}_{\mathcal{L}_1}, \mathcal{L}_2 \) be first-order languages, \( S \) a structure for \( \mathcal{L}_2 \).
We say \( (F_1, F_2) \) holds for \( f, f : S_{\mathcal{L}_1} \xrightarrow{} S_{\mathcal{L}_1} \), iff whenever for \( s \in S \)

\[
 F_1(s(y_1), \ldots, s(y_n)) \quad \text{is true in } S \text{ then } f(s) = s' \in S \text{ implies that}
\]

\[
 F_2(y_1, \ldots, y_n)
\]
(s'(z_1),...,s'(z_m)) is true in S. y_1,...,y_n (resp. z_1,...,z_m) are the free variables of F_1 (resp. F_2).

Lemma 12. Let X be a program, i.e. X \in L_{co}. h: A(L_1) + A_S(L_1, S)  
i: A(L_1) + a_H(L_2, S)

the unique homomorphism. Then (F_1,F_2) \in i_{co}(X) implies that (F_1,F_2) holds for h_{co}(X).

Proof: by structural induction in A(L_1). The proof is analogous to the one given in [20], [30] and is hence omitted.

Corollary 13. Let j: A(L_1) + A_H(L_2,D) where D is a deductive system such that j is a model. Then (F_1,F_2) \in i_{co}(X) implies (F_1,F_2) holds for h_{co}(X).

Proof: obvious by Remark 5.

The relation between A_S(L_1, S) and A_H(L_2,D) expressed in Corollary 13 is often phrased the "consistency of the axiomatic definition" [30] or the "consistency of axiomatic and denotational semantics"[20].

As the following results show this relation between semantical algebras is weaker than our notion of strong-consistency.

Lemma 14. Let L_1 \subseteq L_2 be first-order languages, D a deductive system for L_2 and S a structure for L_2 that is a model of D. Let \{X_i\}_{i \in N} be an effective enumeration of all programs X_i for which there is exactly one variable x_i acting as parameter\(\oplus\). If there are variable-free legal\(\boxtimes\) expressions E_i \in L_{exp}, i \in N, such that

\{ i: x_i does not halt for x_i = E_i \}

is not recursively enumerable then A_S(L_1, S) is not strong-consistent with A_H(L_2, D).

---

\(\oplus\)A variable acts as a parameter if it is used in the right-hand side of an assignment before it has been assigned a value.

\(\boxtimes\)A variable-free expression E is legal if h_{exp}(E)(s) \neq 1 for some s.
Proof: in the following we construct two programs \( X, Y \) which compute the same function, i.e. \( h_{\text{co}}(X) = h_{\text{co}}(Y) \), but for which the Hoare-meaning is different. Then by Lemma 6 no homomorphism from \( A_S(L_{1}, S) \) to \( A_H(L_{2}, D) \) can exist. Let

\[
X = \text{while } \text{true} \text{ do } \text{skip}
\]

\[
Y_n = x_n := E_n; x_n := E_n
\]

Clearly \((\text{true, false}) \in j_{\text{co}}(X)\), where \( j: A(L_{1}) \rightarrow A_H(L_{2}, D) \) is the unique homomorphism. Moreover \((\text{true, false}) \in j_{\text{co}}(Y_n)\) implies that \( X_n \) does not halt on input \( E_n \). Hence, there must be some \( n \) for which \( X_n \) does not halt on \( E_n \) but \((\text{true, false}) \notin j_{\text{co}}(Y_n)\). Let us assume the contrary. Then we enumerate effectively all formulas \( P(st)G \) that can be proved in Hoare's system using the deductive system \( D \) and for which \( F = \text{true, G = false} \) and \( st \) has the form \( Y_n \) for some \( n \). By this we enumerate all \( n \) for which \( X_n \) does not halt at input \( E_n \); hence there must be an \( n_0 \) such that \( j_{\text{co}}(Y_{n_0}) \) does not contain \( \text{true, false} \).

Corollary 15. If \( L_{1} = L_{N} \) with nonlogical symbols \((0, 1, +, \cdot, =, <)\) and the restriction of \( S \) to \( L_{1} \) is the standard model for the natural numbers then \( A_S(L_{1}, S) \) is not strong-consistent with \( A_H(L_{2}, D) \).

Proof: \( E_{1} \overset{1+1+...1}{\underset{i}{\longrightarrow}} \) \( 1 \)-times

We investigate now the relationship between the semantic algebras \( A_S(L_{1}, S) \) and \( A_H(L_{2}, S) \). Similarly to [29], we define when the language \( L_{2} \) is expressive. It will then be the case that the expressiveness of \( L_{2} \) guarantees that

\[
A_H(L_{2}, S) \text{ is strong-consistent with } A_S(L_{1}, S) \text{ and } A_S(L_{1}, S) \text{ is strong-consistent with a factor algebra of } A_H(L_{2}, S).
\]

Let fixed \( L_{2} \supset L_{1} \) and a structure \( S \) for \( L_{2} \) be given. For every formula
For $F \in F(L_2)$ and any program $X$ in $PL(L_1)$ we define

$$G(F,X) = h_{co}(X)(F)$$

with $F = \{ scS \mid F \text{ is true in } S \}$

where $x_1 \ldots x_n$ are the free variables of $F$. Let $x_{n+1} \ldots x_{n+m}$ be the variables occurring in $X$. We say the formula $G \in F(L_2)$ expresses $G(F,X)$ iff

1) $G$ has the free variables $x_1 \ldots x_{n+m}$

2) for $scS$

$$s(x_1) \ldots s(x_n) \text{ is true in } S$$

We say $L_2$ is expressive relative to $L_1$ and $S$ iff

1) "=" is in $L_2$ and receives its standard meaning in $S$

2) $L_1(\emptyset) \subset L_2$, where $L_1(\emptyset)$ is obtained from $L_1$ by adding a name for each element in $| S |$ to $L_1$.

3) For every formula $F \in F(L_2)$ and any program $X$ in $PL(L_1)$ there is a formula $G$ in $F(L_2)$ that expresses $G(F,X)$

Lemma 16. Let $L_2$ be expressive relative to $L_1$ and $S$. If $(F,G)$ holds for $h_{co}(X)$ then $(F,G) \in i_{co}(X)$, where

$$i: A(L_1) \rightarrow A_H(L_2, S)$$

Proof: by structural induction.

Lemma 17. Let $L_2$ be expressive relative to $L_1$ and $S$. Then $A_H(L_2, S)$ is strong-consistent with $A_S(L_1, S)$.

Proof: we show that $i_{co}(X) = i_{co}(Y)$ implies $h_{co}(X) = h_{co}(Y)$ and invoke

Lemma 7. Assume $h_{co}(X) \neq h_{co}(Y)$. Then there is $scS$ such that w.l.o.g.

$$h_{co}(X)(s) = s' \in S$$

and $h_{co}(Y)(s)$ is either not an element in $S$ or $h_{co}(Y)(s) \neq s'$. Let
\(x_1, \ldots x_n\) be the variables occurring in \(X\) and let \(s(x_i) = a_i, i = 1, \ldots , n,\)
\(s'(x_i) = b_i, i = 1, \ldots n, a_i,b_i \in \mathcal{S} n.\)

Consider the formulas:

\[ F = x_1 = a_1 \land x_2 = a_2 \land \cdots x_n = a_n \]

\[ G = (x_1 = b_1 \land x_2 = b_2 \land \cdots x_n = b_n) \]

\(F\) and \(G\) are formulas in \(L_2\) as \(L_2\) is expressive. The pair

\((F,G)\) holds for \(h_{co}(X)\), hence by Lemma 16 \((F,G) \in i_{co}(X)\) yielding

a contradiction.

**Corollary 18.** There is a congruence relation \(E = \{ E_{co}, E_{exp}, E_{boolexp} \ldots \}\)

on \(A_{H}(L_2,\mathcal{S})\) such that \(A_{S}(L_1,\mathcal{S})\) is isomorphic and hence strong-consistent

with \(A_{H}(L_2,\mathcal{S})\). Moreover, \(\pi_1 \cong_{co} \pi_2 \iff \pi_1 = \pi_2.\)

**Proof:** the homomorphism \(\xi\) from \(A_{H}(L_2,\mathcal{S})\) to \(A_{S}(L_1,\mathcal{S})\) is onto, which

proves the first part of the statement using standard algebraic argumens. For the second part we show that \(\xi_{co}\) is one-to-one. Assume

\[ \xi_{co}(\pi_1) = \xi_{co}(\pi_2) \Rightarrow \pi_1 = \pi_2 \]

We show \(\pi_1 = \pi_2\)

\[ \xi_{co}(\pi_1) = h_{co}(X) \text{ for some } X \in L_{co} \text{ with } i_{co}(X) = \pi_1 \]

\[ \xi_{co}(\pi_2) = h_{co}(Y) \text{ for some } Y \in L_{co} \text{ with } i_{co}(Y) = \pi_2 \]

We show w.l.o.g. that \(\pi_1 \prec \pi_2\). Let \((F,G) \in \pi_1\), hence \((F,G)\) holds

for \(h_{co}(X)\) by Lemma 12. Thus \((F,G)\) holds for \(h_{co}(Y)\) and by Lemma 16

we conclude \((F,G) \in i_{co}(Y) = \pi_2\).

Some results of this section can be summarized in the following diagram:

\[ \begin{array}{c}
\xi_{co}(\pi_1) = h_{co}(X) \\
\xi_{co}(\pi_2) = h_{co}(Y)
\end{array} \]

where a solid arrow expresses strong-consistency, whereas a dotted arrow means strong-
consistency if \(L_2\) is expressive.
It should not be difficult for the reader to verify that a suitable operational
semantic description of PL($L_1$), similar to the computational model given in [20],
gives rise to a semantic algebra $A_{op}(L_1, S)$ which is strong-consistent with
$A_S(L_1, S)$.

5. Abstraction

In the previous sections we have seen how different semantic specifications of
a programming language can be presented in the same framework. The various semant
algebras which have been described before differ mainly in the formalism chosen to
describe the meaning of a command. In one case the formalism involves functions on
state set, in another relations over formulas are used, etc. The question can be
raised, if these formalisms are necessary at all. We will briefly discuss this
issue in the following.

Let us consider the programming language PL($L_1$). Every semantic algebra $A_1$
by definition a homomorphic image of the syntactical algebra $A(L_1)$. By standard
algebraic arguments we know then that $A$ is isomorphic to some factor algebra of
$A(L_1)$, i.e.

$$
\begin{array}{c}
A = A(L_1) \\
E
\end{array}
$$

for a uniquely determined congruence relation $E$. Hence, we may say that
"semantics can be considered as a congruence relation on syntax". As isomorphic
algebras can be considered as the "same" as far as their mathematical properties
are concerned one could think of working with factor algebras of the syntactical
algebras instead of one of the algebras $A_H(L_2, S), A_S(L_1, S)$, etc.

Isomorphism of $N$-sorted P-algebras (where $G = (N,T,P,S)$) induces an equivalence
relation on the class of $N$-sorted P-algebras. We call an equivalence class with
respect to this equivalence relation abstract semantics. As every equivalence
class contains exactly one factor algebra $A(L_1)/E$ we may choose $A(L_1)/E$
as a representative for the whole class.
If we want to give abstract semantics for a programming language the question arises: how do we specify the congruence relation $E$, i.e. how do we tell in a particular case what $E$ looks like.

This situation is very similar to the one arising for formal specifications of abstract data types [31,32]. For abstract data types one tries to establish a set of (equational) axioms to characterize the congruence relation $E$. Some of the obvious axioms that are true for every semantical algebra given in sections 3,4 are e.g.

$$0_{\text{concat}}(X, 0_{\text{skip}}) = X$$
$$0_{\text{concat}}(0_{\text{skip}}, X) = X$$
$$0_{\text{concat}}(0_{\text{concat}}(X,Y),Z) = 0_{\text{concat}}(X,0_{\text{concat}}(Y,Z))$$
$$0_{\text{if}}(B,X,Y) = 0_{\text{if}}(\neg B,Y,X)$$

where $X,Y$ are variables for programs (or more precisely for meanings of programs).

In [33] we investigated how classes of semantic algebras can be characterized using an axiomatic theory of programs. The question behind this attempt is: what are the axioms and theorems about programs and program constructors that hold for all the different semantical descriptions given for one particular language? In studying this question we found an interesting relationship between these axioms and monotonous operators on complete lattices that have the meaning of a program construct as fixpoint.

Another problem is, how the context-sensitiveness of real-life programming languages is dealt with. Probably this problem could be solved by admitting that the operations of the syntactical algebra are partial.

6. **Summary**

In contrast to syntactical aspects of programming languages semantical aspects seem harder to understand and formalize. Many different attempts have been made so far to specify the semantics of a programming language. In this paper we suggest a uniform framework for presenting, understanding and comparing different
forms of semantical descriptions. Basically, we derive from each semantical description a certain semantical algebra. In section 3 we construct for a particular programming language $PL(L_1)$ the semantical algebras $A_S(L_1, S)$ for Scott-like semantics, $A_H(L_2, S)$ and $A_H(L_2, D)$ for Hoare-like semantics and $A_D(L_1, S)$ for Dijkstra's semantics. In section 4 the notion of strong-consistency is used to compare the different semantic algebras.

It is shown e.g. that $A_S(L_1, S)$, i.e. Scott-like semantics, and $A_D(L_1, !)$, i.e. Dijkstra-like semantics, are isomorphic. In section 5 we consider the abstraction from a particular formalism for expressing the meaning of programs. We introduce the notion of abstract semantics by which we mean the factor algebras of the syntactical algebras, i.e. $A(L_1)/E$. We briefly discuss, how abstract semantics could be characterized in an axiomatic theory of programs and program constructors.
References


