# A Survey of Semantic Description Frameworks for Programming Languages*

*Yingzhou Zhang* *Baowen Xu*

Department of Computer Science and Engineering, Southeast University, Nanjing 210096, China
Jiangsu Institute of Software Quality, Nanjing 210096, China

**Abstract**

*Formal semantic description is significant for design, reasoning and standardization of programming languages, and it plays an important part in the optimization of the compiler. However, compared to the amount of effort that has been made to the research of various semantic frameworks over more than forty years, their actual applications are definitely frustrating. This survey reviews the history of developments on semantic description frame- works for programming languages. It also illustrates features and actual applications of the main frameworks (including operational, deno- tational, axiomatic and hybrid semantics). In some practical aspects, such as comprehensibility, extensibility and applicability, the qualitative comparisons of these frameworks are given distinctly. It suggests that a more popular formal semantic description should behave more elegantly in readability, modularity, abstractness, comparability, reasonability, applicability and tool-suppor*t.

**Keywords**: *axiomatic semantics, denotational semantics, formal semantics, hybrid semantics, operational semantics, semantic description frameworks*

## 1 Introduction

As we know, the definition of a programming language consists of at least two parts, the syntax and the semantics. The syntax is concerned with the form of expressions, which are allowed in the language. On the other hand the semantic definition could describe the effect of executing or evaluating any syntactically correct expression or program or it could describe how to execute or evaluate them [Hen1990]. Of course, the syntax and the semantics are closely related, the former, which decides the form and structure of programs, is the pre-condition of describing the semantics of programs. The syntax is the earlier of the two to be defined and dealt with. In the definition of ALGOL 60, P.Naur introduced the notation now known as Backus-Naur Form (BNF). Descriptions are called formal when written in a notation that already has a precise meaning. The BNF is a formal notation for describing the syntax of programming languages. It was widely recognised and has become practically universal in its use, almost totally supplanting other techniques for syntactic description.

In contrast, in over forty years of development, there is still no universally accepted formal notation for semantic description — on the contrary, a very large number of formal notations (some of them are listed in this paper) have been invented, and new formal notations are introduced regularly. The reason for this lies in the fact that program behaviour exhibits far greater complexity than program structure [Wan1997]. Nevertheless, there are a number of advantages to such a formal semantics approach. For instance, it gives a completely unambiguous definition. In comparison, programming language standards and manuals that are written using only descriptive prose are bound to be incomplete and/or ambiguous. A precise definition of the behaviour of programs is of great value to compiler writers (and to programmers if it sufficiently readable). The essence of formal semantics is the treatment of programs in a mathematically rigorous way [Cle2003].

Before we survey the main frameworks available for describing the dynamic semantics of programming languages, we should consider exactly what are the demands made on a semantic notation. There are seven areas in which a semantic description is of use, and they are summarized below [Mos1992a, Wan1997, Rin1997,

* This work was supported in part by the National Natural Science Foundation of China (60073012), National Grand Fundamental Research 973 Program of China (2002CB312000), National Research Foundation for the Doctoral Program of Higher Education of China (20020286004), Cross-Century Excellent Scholar Plan of the Education Ministry of China, Opening Foundation of State Key Laboratory of Software Engineering in Wuhan University, and Opening Foundation of Jiangsu Key Laboratory of Computer Information Processing Technology in Soochow University.

Correspondence to: Baowen Xu, Department of Computer Science and Engineering, Southeast University, Nanjing 210096, China. E-mail: bwxu@seu.edu.cn

Mos2001a, Mos2002a, Cle2003].

- The semantics is the product of the programming language *design* process, and contains and communicates the decisions of the designers. During this process, the designers need to record decisions about particular language constructs, and to be made aware of omissions and irregularities in the overall design.
- During *implementation* of the language, the semantics is of use in ensuring correct behaviour of the implementation. The semantics must communicate comprehensively and accurately to the implementers the intentions of the designers.
- *Standardization* of the language is achieved by publishing an unambiguous semantics. Programs must be able to be transported between different implementations conforming to the semantics and exhibit the same behaviors.
- A programmer's *understanding* of a language requires learning its behaviors, i.e., its semantics. The semantics must make clear the behaviors of the programming language and its various constructs in terms of familiar concepts, and must make apparent the relationship between this language and others with which the programmer is familiar.
- The semantics assists the programmer in *reasoning* about a program: verifying it does what it should, for example. This requires the semantics to allow mathematical manipulation of programs and meanings, and proof of assertions about programs and their behaviors.
- The semantics allows theoreticians to obtain new *insight* into programming concepts and open new research areas. For this, the semantics must isolate common properties of programming languages and allow investigation of these properties.
- Finally, semantic descriptions may be used to *generate* compilers and interpreters. Although some interesting prototype systems have been implemented, compilers and interpreters generated from semantic descriptions are generally not efficient enough to be practically useful—except for allowing empirical testing of the semantic description itself.

The remainder of this paper is organized as follows. Section 2 overviews the history of semantic frameworks of programming languages. Section 3 gives the syntax of a simple imperative programming language **While** which is used to illustrates the various forms of semantics in its following sections. Section 4-7 illustrate features and applications of the main semantic description frameworks: operational, denotational, axiomatic, hybrid, respectively. Section 8 gives the qualitative comparison of these frameworks with respect to practical aspects, such as readability, modularity, and applicability. The two good candidates for greater popular framework are proposed in section 9.

## 2 History of Semantic Frameworks

Since the appearance of denotational semantics in 1960s, the development of formal semantics of programming languages passes over forty years. The formal semantic frameworks are gradually theoretic-perfect, and consider practical aspects, such as readability, modularity, and extensibility, which are especially significant when describing full-scale languages. In most literatures (such as [Cle2003, NN1992, Mos1998]), the frameworks for formal semantics were classified as *operational*, *denotational*, or *axiomatic*. In operational frameworks, the semantics of a program is specified as a sequence, or execution history, of state transitions, usually as operations on some hypothetical abstract machine. A denotational semantics is given by a mathematical function which maps the syntax of the program to a semantic value, a denotation [1]. Axiomatic semantics involves rules for deducing assertions about the correctness or equivalence of programs and corresponding parts.

Each of the above three frameworks have particular properties, but the distinction between them is seldom sharp: they frequently borrow features from each other. These frameworks are however not complete substitutes for each other. They can sometimes be used in correct to describe different parts or aspects of a system. P.Mosses let this framework be a new one, called *hybrid*, since it is essentially hybrids of different kinds of frameworks [Mos2001a, Mos2001b, Mos2002a]. Make a comprehensive survey on the history of semantic frameworks, from their original study for grounded theory to now actual application with well readability and modularity, the research related to frameworks can be roughly classified into the following four phases.

*Phase 1* (1960-1980) was mainly concerned with the study on basic theory of frameworks. Denotational and axiomatic frameworks were put forwarded in this phase. The main approaches of denotational frameworks include D.Scott and C.Strachey's *Scott-Strachey Semantics* [Ten1976, Sto1977, Sch1986], D.Bjørner and C.Jones' *VDM semantics* [BJ1978], and E.Dijkstra's *predicate transformer semantics* [Dij1975, DS1990]. Scott-Strachey Semantics is the original and classic style of denotational semantics. Domains of denotations and auxiliary entities are defined by domain equations. The elements of domains are specified in typed λ-notation. VDM (the Vienna Development Method) is an approach for studying large computer software systems, developed by the Vienna workgroup in Vienna laboratory. In predicate transformer semantics, the denotation of a phrase is a predicate transformer that returns the weakest condition which ensures termination of the phrase with the argument condition holding. Since it involves

[1] This function is usually a partial function (which is made total by introducing the special member ⊥), since some syntactically correct programs may not have a defined meaning, i.e. they may contain semantic errors or be intentionally undefined.

assertions about the values of variables before and after executing statements, it is often regarded as axiomatic.

In this phase, the main axiomatic semantics includes C.Hoare's *Hoare logic* (cf. [Hoa1969, Hoa1973]) and J.Goguen's *initial algebra semantics* [GTWW1977]. Hoare logic, which is the classic style of axiomatic, gives rules for the relation between assertions about values of variables before and after execution of each construct.

*Phase 2* (1980-1990) introduced temporal feature into semantic frameworks, i.e., considered the sequence of executive process of a program. The new framework appeared in this phase was operational, whose main approaches include G.Plotkin's *structural operational semantics* (SOS) [Plo1981], G.Kahn's *natural semantics*, M.Fellesisen's *reduction semantics*, and Y.Gurevich's *abstract state machines* (ASM) [Gur1993, SSB2001, BS2003, Gle2003, AW2003]. The main aim of these approaches was to provide a simple and direct method, allowing concise and comprehensible semantic descriptions based on simple mathematics. Inspired by the idea of operational, D.Harel joined temporal feature to axiomatic, and put forward *dynamic logic* [Har1984, HKT2000], which can be described as a blend of three complementary classical ingredients: first-order predicate logic, modal logic, and the algebra of regular events.

In this phase, I.Guessarian formally put forward *algebraic semantics* [Gue1981], and then Y.Gurevich gave *algebraic operational semantics* (AOS) [Gur1987, GM1988, TZ1988], which is a hybrid of algebraic and operational semantics. Algebraic semantics can uniquely interpret meanings of programs in the way of algebra. AOS combines the notion of time with SOS in an axiomatic fashion.

In the second half of this phase, the modularity of semantic descriptions was being taken into account. Mosses directly joined denotational together with operational, and put forward a hybrid framework —*action semantics* (AS) [Mos1986, Wat1986, MW1987, Mos1989, Mos1992a, Mos1996a, Mos1996b], which posses modularity and readability simultaneously. Denotations in action semantics are so-called actions, which encapsulated some basic and fixed programming concepts, such as control and data flow, scopes of bindings, and effects on storage. Another approach to provide modularity is E.Moggi's *monadic semantics* [Mog1989, Mog1991]. He firstly introduced monads (cf. [Mog1989, Mog1991, Wad1990, LHJ1995])into denotational. Monadic semantics is based on category-theoretic *functor* concepts. It is a parameterized semantics, and can be instantiated using different underlying monads.

*Phase 3* (1990-2000) is characterized by contemplation of practical features of programs, such as reusability, expansibility, and comprehensibility. Inspired by the modularity that can be obtained in denotational semantics by the use of monads, Mosses introduced modularity into conventional operational frameworks, and brought forward *modular SOS* (MSOS) and *modular natural semantics* [Mos1998, Mos1999, Mos2002b]. In order to enhance readability of denotational, A.Blass joined interactive behaviour to denotational, and gave *game semantics* [Bla1992, AM1998, Byu2003, McC1996, HO2000, AJM2000], which models computation as interaction between a system and its environment, and models a program as a set of possible interactions.

Besides action semantics in phase 2, the main approaches of hybrid frameworks include S.Liang, P.Hudak and M.Jones' *modular monadic semantics* (MMS) (cf. [LHJ1995, Lia1998]) and K.Wansbrough's *modular monadic action semantics* (MMAS) [Wan1997, WH1997]. These approaches adopt more disciplined notations for avoiding any reference to irrelevant semantic components when defining the semantics of each construct, and all attempt to reduce the conceptual distance that must be bridged when formally specifying a high-level language (see Figure 1 (3)-(5)).

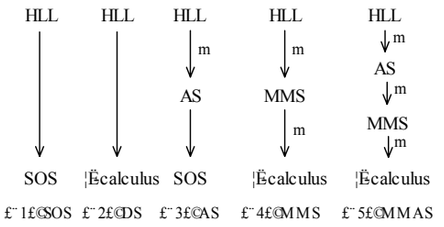


**Figure 1** Semantic Description Techniques
m : shows the transition is modular

We are now at the beginning of *Phase 4*, where user-friendly semantic frameworks with mature supported tools are developing. This phase will pay much attention to well pragmatic aspects and quality tools for semantic frameworks.

In next five sections, we shall illustrate the main semantic frameworks with fragments involving the description of a simple if-statement and an assignment expression of **While** language.

# 3 The Example Language While

This paper illustrates the various forms of semantics on a very simple imperative programming language **While** [NN1992, LS2002, KLNS2002], whose BNF syntax is as follows:

$a ::= n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2$
$b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 <= a_2 \mid \neg\ b \mid b_1 \wedge b_2$
$S ::= x := a \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid S_1 ; S_2 \mid \text{skip} \mid \text{while } b \text{ do } S$

The meta-variables (that will be used to range over constructs of categories) $a$, $b$, $S$, $n$, $x$ will range over arithmetic expressions (category **Aexp**), boolean expressions (**Bexp**), statements (**Stm**), numerals (**Num**), variables (**Var**), respectively. The meta-variables can be primed or subscripted. So, for example, $a$, $a'$, $a_1$, $a_2$ all stand for arithmetic expressions. For the limitation of paper space, we will only give semantics of if-statement and assignment

expression of **While** language.

Programming language semantics is usually divided into static and dynamic semantics. Static semantics describes properties of a program that can be reasoned about prior to the execution of the program, while dynamic semantics describe properties of the program during its execution. Static semantics especially concerns with type-checking and type-inference. For example, the semantics for an arithmetic expression is as follows [Lia1998]:

$$\begin{aligned}\boldsymbol{E}[\![ a_1 + a_2 ]\!] = \{\ & v_1 \quad \boldsymbol{E}[\![ a_1 ]\!]; v_2 \quad \boldsymbol{E}[\![ a_2 ]\!]; \\ & \textbf{if } (\ \textbf{is}_{Int}\ v_1\ \textbf{and is}_{Int}\ v_2\ )\ \textbf{then} \\ & \quad \textit{return}\ (\ \textbf{in}_{Int}\ (\textbf{out}_{Int}\ v_1 + \textbf{out}_{Int}\ v_2\ )) \\ & \textbf{else}\quad \text{err}\ \text{“type error”}\ \}\end{aligned}$$

$\textbf{in}_{Int}$ is the injection function from Int to the Value domain, whereas $\textbf{out}_{Int}$ is the projection function from the Value domain to Int. The kernel-level function err reports error conditions. In fact, domain function $\textbf{in}_{Int}$ and $\textbf{out}_{Int}$, and type-checking function $\textbf{is}_{Int}$ belong to static semantics. For clarity, in this paper, we will omit static semantics such as domain injection/projection and type-checking, and focus entirely on dynamic semantics.

# 4 Operational Frameworks

In operational frameworks, the semantics of a program is specified as an abstract machine or transition system, the computations of which represent possible executions of the program [Mos2001b, Mos2001c]. In an operational semantics we are concerned with how to execute programs and not merely what the results of execution are. More precisely, we are interested in how the states are modified during the execution of the statement. There are various approaches to operational semantics of programming languages. Here, we shall consider mainly four different approaches: SOS, natural semantics, MSOS, and ASM.

## 4.1 Conventional Operational Semantics

Generally, conventional operational semantics refers to *structural operational semantics* (SOS, also called small-step semantics, 1981). Characteristic for SOS is that the transitions for a phase generally depend only on the transitions for (one or more of) its sub-phrases, and transition relations are specified by sets of axioms and inference rules (for instance, rules $[\text{ass}_{\text{sos}}]$, $[\text{if}^{\text{tt}}_{\text{sos}}]$ and $[\text{if}^{\text{ff}}_{\text{sos}}]$ in Figure 2 from [NN1992]). In SOS, each transition modifies the syntax part of the state to reflect a step in the execution of some sub-phrase. When the execution of a sub-phrase is finished, it is replaced by its computed value.

SOS has been widely used in program analysis (eg. [Lau1968, PDG1986, Plo1983, Rep1991, BG1994, PS1995, UP2002]) and formal verification (eg. [Blo1989, WBB1993, ABV1994, WF1994, CHT2002]).

$$\begin{aligned}&\boldsymbol{A}: \quad \textbf{Aexp} \to (\textbf{State} \to \textbf{Z}) \\ &\boldsymbol{B}: \quad \textbf{Bexp} \to (\textbf{State} \to \textbf{T}) \\ &\textbf{State}: \quad \textbf{Var} \to \textbf{Z} \\ &[\text{ass}_{\text{sos}}]: \langle x := a, s\rangle \to s[x \mapsto \boldsymbol{A}[\![ a ]\!]s] \\ &[\text{if}^{\text{tt}}_{\text{sos}}]: \langle \text{if } b \text{ then } S_1 \text{ else } S_2\ , s\rangle \Rightarrow \langle S_1, s\rangle \quad \text{if } \boldsymbol{B}[\![ b ]\!]s = \textbf{tt} \\ &[\text{if}^{\text{ff}}_{\text{sos}}]: \langle \text{if } b \text{ then } S_1 \text{ else } S_2\ , s\rangle \Rightarrow \langle S_2, s\rangle \quad \text{if } \boldsymbol{B}[\![ b ]\!]s = \textbf{ff}\end{aligned}$$

**Figure 2** Structural Operational Semantics

An alternative operational semantics is called *natural semantics* (or big-step semantics, 1987) and differs from SOS by hiding even more execution details. In fact, the big-step semantics (i.e. natural semantics) is actually just a special case of the small-step semantics (i.e. SOS). The purpose of natural semantics is to describe how the overall results of executions are obtained, in contrast to SOS whose purpose is to describe how the individual steps of the computations take place. Evaluations in natural semantics are also specified by axioms and inference rules (see Figure 3 from [NN1992]). However, the rules are not suitable for semantic descriptions of concurrent languages, or even of interleaved expression evaluation, just because of the lack of intermediate states.

Natural semantics has been used extensively in the definition of programming languages, such as Standard-ML [MTH1990, MTHM1997], Eiffel [Att1996], and some object-oriented programming languages [GZ1998, Gle1999]. Natural semantics has also been used successfully to prove properties of programs (eg. [DE1999, Sym1999, vON1999]).

$$\begin{aligned}&[\text{ass}_{\text{ns}}]: \langle x := a, s\rangle \to s[x \mapsto \boldsymbol{A}[\![ a ]\!]s] \\ &[\text{if}^{\text{tt}}_{\text{ns}}]: \frac{\langle S_1, s\rangle \to s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2\ , s\rangle \to s'} \quad \text{if } \boldsymbol{B}[\![ b ]\!]s = \textbf{tt} \\ &[\text{if}^{\text{ff}}_{\text{ns}}]: \frac{\langle S_2, s\rangle \to s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2\ , s\rangle \to s'} \quad \text{if } \boldsymbol{B}[\![ b ]\!]s = \textbf{ff}\end{aligned}$$

**Figure 3** Natural Semantics

In terms of **While** language, since its programs are deterministic and expressions have no side-effects (called functional), its SOS and natural semantics are equivalent (cf. Theorem 1), for more details please see [NN1992].

**Theorem 1** For every statement $S$ of **While** we have $\boldsymbol{S}_{\text{ns}}[\![ S ]\!] = \boldsymbol{S}_{\text{sos}}[\![ S ]\!]$.

### 4.2 Modular Operational Semantics (1998)

Conventional operational semantic descriptions have rather poor modularity, since the semantic components of the transition relation (environments, stores, etc.) are made explicit in every rule, and a complete reformulation is needed when adding further components [Mos1998]. For solving this problem, Mosses put forward *modular SOS* (MSOS, [Mos1999, Mos2002b]), which is a variant of SOS where states are restricted to syntax and computed values, and all auxiliary entities[2] are incorporated in *labels* on transitions (such as labels a , ß and in Figure 4). Labels in modular SOS include, for example, environment, (pairs of) stores, and (sequences of) communication signals. The set of labels is generally infinite. It is straightforward to reduce a modular SOS to a conventional SOS, by moving the relevant components of the labels back to their usual places in the states. Similarly, modular natural semantics requires all auxiliary entities to be incorporated in labels on evaluations [Mos2001b]. The differences between modular SOS and modular natural semantics are similar to the ones between SOS and natural semantics, for more please see [Mos1998, Mos2001a, Mos2001b].

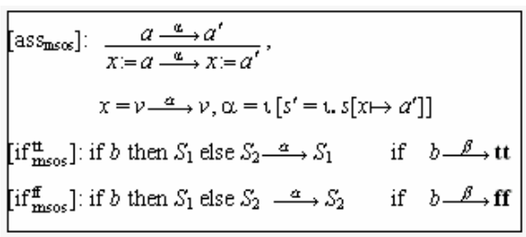


**Figure 4** Modular Operational Semantics

### 4.3 Abstract State Machines (1988)

Conventional operational semantics lacks mathematical precise notation of state. Natural semantics can only define the semantics of strictly compositional programming languages[3]. And SOS explicitly rewrites the abstract syntax trees (AST) during execution [Gle2003]. Gurevich's *abstract state machine* (ASM, 1988, defined in [Gur1993, SSB2001]) can avoid these problems. ASMs describe the semantics of programming languages operationally as state transition systems based on the AST. States are regarded as algebras over a given signature, so ASMs are called "evolving algebras". States in ASM include control-flow graphs representing the entire program. For instance, in Figure 5, the functions *fst*, and *nxt* represent normal control flow between phrases. However, flow of control need not follow the structure of the program at all: in principle, the pointer *task*, normally indicating the next part of the program to be executed, can be set arbitrarily [Mos2001b]. In ASMs, a program is regarded as an attributed AST whose attributes specify the continuations (which is an approach to define non-compositional semantics such as the semantics of **goto**-statements). In [Gle2003], S.Glesner showed that natural semantics could be transformed automatically into an equivalent ASM semantics and vice versa; and that each SOS can be transformed automatically into an equivalent ASM.

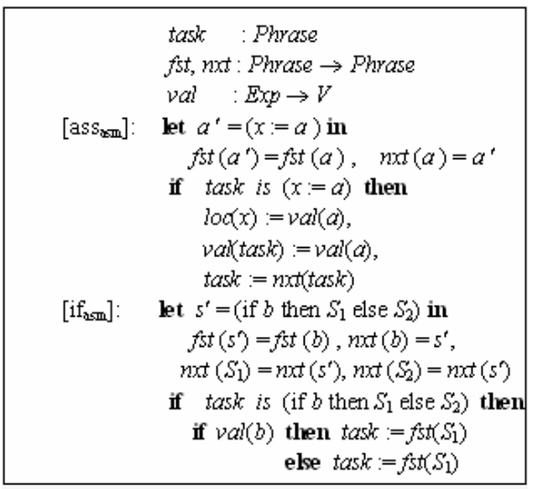


**Figure 5** Abstract State Machine

ASMs have got wide usages in semantic descriptions of programming languages, such as C [GH1993], Java [SSB2001], Prolog [BR1995], and SDL [GK1997, GGP1999, ITU2000]; and in proving the correctness of compilations (eg. [BR1994, BD1996, ZG1997, GZ1999]). ASM was adopted by ISO for standard of Prolog [BD1990], by IEEE for standard of VHDL'93 [BGM1994], and by ITU for standard of SDL-2000 [ITU2000].

### 4.4 Other Operational Semantics

Other operational semantics include *reduce semantics* [FF1986], *enhanced operational semantics* (EOS) [DP1996, DP2001], the *SECD abstract machine* [Lan1964, Lan1966], the *VDL abstract machine* [Weg1972], and the *SMoLCS framework* [AR1987].

## 5 Denotational Frameworks

In denotational frameworks, the meaning of a program phrase is modeled by its so-called denotation (i.e. a mathematical object, generally a continuous function), which reflects the contribution of the phrase to

[2] Auxiliary entitles in SOS often include stores such as σ εS = L -> V and environments such as ρ ε Env = Var -> L, where L is some set of locations (i.e. addresses in memory). [Mos2001a]

[3] In a strictly compositional programming language, the semantics of each part of the program, which we regard in form of its abstract syntax tree, can be defined solely in terms of the semantics of its direct parts, i.e. subtrees.

overall program behaviour. For example, the meaning of a program may be given as a function from Input to Output. The focus of denotational semantics is on either the effect (which means an association between initial states and final states) or result, but not how it is obtained.

Here, we will consider three main approaches for operational: Scott-Strachey semantics, game semantics and monadic semantics.

## 5.1 Conventional Denotational Semantics

The original and classic denotational semantics is generally *Scott-Strachey semantics*. It provides the purest and most abstract way of modeling the semantics of programs. It generally avoids representing computations as sequences of steps. Instead, it determines for each phrase its denotation. Denotations are generally mathematical functions, taking current information and returning computed values and updated information. Typically, denotations are functions of environments, continuations, and stores. They are specified in $\lambda$–notation[4], and defining the domains of denotations for parameterized procedures requires reflexive Scott-domain[5]. The idea then is to define a semantic function for each syntactic category. It maps each syntactic construct to its denotation, which describes the effect of executing that construct. The inherency of denotational semantics is that semantic functions are defined compositionally [Sto1977, Sch1986, NN1992, Mit1996]. Sequencing may be represented either by composition of strict functions (illustrated in Figure 6), or by use of continuations (showed in Figure 7); the latter also called a continuation style semantics. The denotational description of nondeterminism, concurrency, and interleaving requires the use of power domains.

$$
\begin{aligned}
&\mathcal{S}_{ds}: \mathbf{Stm} \to (\mathbf{State} \to \mathbf{State}) \\
&\mathcal{S}_{ds}[\![ x := a ]\!] = \lambda s.\, s[x \mapsto \mathcal{A}[\![ a ]\!] s] \\
&\mathcal{S}_{ds}[\![ \text{if } b \text{ then } S_1 \text{ else } S_2 ]\!] = \\
&\qquad \text{cond}(\mathcal{B}[\![ b ]\!], \mathcal{S}_{ds}[\![ S_1 ]\!], \mathcal{S}_{ds}[\![ S_2 ]\!])
\end{aligned}
$$

**Figure 6** General Denotational Semantics

The applications of denotational semantics include aiding language design, establishing standards for implementation, reasoning about programs and generating compilers [e.g. Sto1977, Sch1995, KOC1991, CP1994, DDR1997, Yeu1997, YZJ1995, BZ1992, LW1995, Pol1981, BBKL1982].

[4] $\lambda$–notation is merely a notation for expressing mathematical functions by listing their arguments and results, without having to declare function names.

[5] The Scott-domain is usually $\omega$–complete partial orders (CPOs). Its equations always have "least" solutions (up to isomorphism), e.g. $D = N + [D \to D]$ defines a domain $D$ including both the natural numbers and all continuous functions on $D$. [Mos2001b]

$$
\begin{aligned}
&\mathcal{S}'_{cs}: \mathbf{Stm} \to (\mathbf{Cont} \to \mathbf{Cont}),\ \mathbf{Cont} = \mathbf{State} \to \mathbf{State} \\
&\mathcal{S}'_{cs}[\![ x := a ]\!] = \lambda c s.\, c\,(s[x \mapsto \mathcal{A}[\![ a ]\!] s]) \\
&\mathcal{S}'_{cs}[\![ \text{if } b \text{ then } S_1 \text{ else } S_2 ]\!] = \\
&\qquad \lambda c.\, \text{cond}(\mathcal{B}[\![ b ]\!], \mathcal{S}'_{cs}[\![ S_1 ]\!] c, \mathcal{S}'_{cs}[\![ S_2 ]\!] c)
\end{aligned}
$$

**Figure 7** Continuation Style Semantics

In fact, there are some relationships between denotational semantics and operational semantics. For example, the denotational semantics and operational one of **While** language are fully equivalent (showed in Theorem 2). Such denotational semantics is called *fully abstract* [Mit1996]. A fully abstract denotational semantics may be very useful, since reasoning about the denotational semantics therefore allows us to reason about the operational one. This is important since operational semantics is generally difficult to reason about directly, yet it is the most useful form of mathematical problem to construct fully-abstract denotational semantics. The game semantics introduced in next subsection can solve this problem in terms of purely functional languages.

**Theorem 2** For every statement $S$ of **While** we have $\mathcal{S}_{sos}[\![ S ]\!] = \mathcal{S}_{ds}[\![ S ]\!]$.

## 5.2 Game Semantics (1992)

Game semantics (cf. [Bla1992]) was first studied in the context of the fully abstract problem for functional programming languages. The first syntax-independent descriptions of fully abstract models for PCF (simply-typed -calculus plus arithmetic and recursion, see [Mit1996]) were achieved (in 1993) using *game semantics* [HO2000, AJM2000]. As mentioned in Set.2, game semantics is a sort of denotational semantics that retains more information about what the program does. It models computation as the playing of a certain kind of game, with two participants, called Player (P) and Opponent (O). P is to be thought of as representing the system under consideration, while O represents the environment. In the case of programming languages, the system corresponds to a term (a piece of program text) and the environment to the context in which the term is used. This is a key point at which games models differ from other process models: the distinction between the actions of the system and those of its environment is made explicit from the very beginning [McC1996, AM1998, Byu2003]. In a game semantics, a computation is modeled as interaction between P and O, and a program as a set of possible interactions. O always moves first — the environment sets the system going — and thereafter the two players make moves alternately (showed in Figure 8).

Games models have been built for higher-order programming languages with a variety of computational features: such as purely functional languages (PCF, FPC)

[HO2000, AJM2000, Nic1994, McC1996], mutable store (Idealized Algol) [AM1997a, AM1999], control operators (SPCF, exceptions) [Lai1997, MH1998], higher-order store (pointers) [AHM1998, AM1997b], nondeterminism and subtyping [HM1999, MH1999, Chr2000].

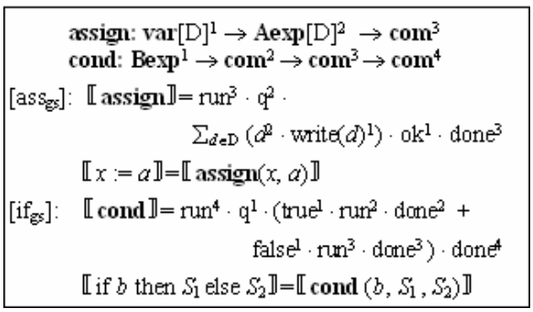


**Figure 8** Game Semantics

## 5.3 Monadic Semantics (1989)

Just because of the unrestricted use of (typed) -notation to specify semantic entities, conventional denotational descriptions have as poor modularity as conventional operational descriptions [LH1996, Mog1991, Wan1997, Mos1998, Pow2000]. When the described language is extended with unanticipated new constructs, the domains of denotations may need to be changed, and then the description of the old constructs may have to be completely reformulated to adapt it to the new domains. To avoid any reference to irrelevant semantic components when defining the semantics of each construct, a more disciplined notation should be adopted to substitute for -notation. One of such notations is *monad*[6], which is a technique for encapsulating impure features such as states, nondeterminism and I/O into a pure functional language. Monads were discovered in category theory in the 1950s and introduced to the semantics community by Moggi in [Mog1989]. Intuitively, a monad is transformation on types equipped with a composition method for transformed values. To add a new feature to a monadic semantics, we only need to add a semantic description of the new feature, and change the underlying monad, but not the semantic descriptions of the existing features. Traditional denotational semantics maps, say, a term, an environment and a continuation to an answer. In contrast, monadic semantics maps terms to computations, where the details of the environment, store, etc. are "hidden" (see Figure 9). Moggi also realized that some realistic semantics features had to be combined, and so he presented *monad constructors* that could add new notions of computation to a monad.

The monadic style in which the descriptions are written is much easier to read than a typical denotational semantic description. The applications for monadic semantics are included in those for modular monadic semantics (in Section 7.3).

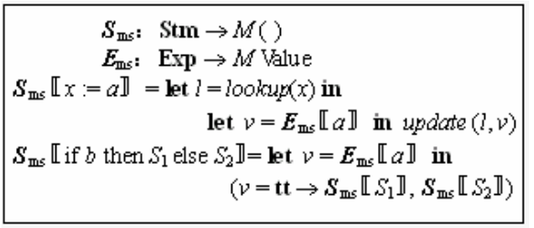


**Figure 9** Monadic Semantics

## 5.4 Other Denotational Semantics

Other denotational semantics include *VDM semantics*[BJ1978, LP1995], *predicate transformer semantics*[Dij1975, DS1990, Nau2001], the *naive denotational semantics*[BT1983], *extensible denotational semantics*[CF1994], and *partially-additive semantics* [MA1986].

# 6 Axiomatic Frameworks

Axiomatic semantics describes properties of programs as sets of constraints (or called assertions), and programs as transforming assertions. This is the most abstract of the four families of description formalisms (i.e. operational, denotational, axiomatic, hybrid). Axiomatic descriptions are not particularly well-suited to complete description of programming languages from which it is possible to automatically generate compilers. It is more suited to proving properties about programs than for automatic generation of complete language implementations.

## 6.1 Traditional Axiomatic Semantics

Axiomatic semantics was developed primarily by Hoare in the late 1960s, and called *Hoare logic*, which base on predicate logic. The main aim was initially to provide a formal basis for the verification of abstract algorithms [Hoa1969]. A Hoare logic gives rules for the relation between assertions about values of variables before and after execution of each construct. Expressions are used in assertions, so their interpretation has to be purely mathematical, without effects on storage, exceptions, non-terminating function calls, etc.[Mos2001b, Mos2002a]

A general form of so-called partical correctness formula in Hoare logic is $\{P\}S\{R\}$ (sometimes $P\{S\}R$), which indicates that if the pre-condition $P$ held before executing the program $S$ then the post-condition $R$ will hold afterwards (showed in Figure 10). $P$ and $R$ are propositions and we can perform to usual logical manipulations on them. This notation allows us to move from expressions written in a programming language to expressions written in a logic. Logic is a much better

[6] Formally, a monad is a triple (*M*, **return**, **bind**), where *M* is a type constructor (a map from each type α to a corresponding type *M*α), and **return** and **bind** are functions: **return**: α -> *M*α, **bind**: *M*α -> (α *M*β) -> *M* β.

place to reason than programing languages are, and so this approach can be extended to provide a tool for reasoning about program correctness, and well as programming language semantics.

$$[\text{ass}_{ax}]: \quad \{P[x \mapsto \mathcal{A}[\![a]\!]]\}\ x := a\ \{P\}$$

$$[\text{if}_{ax}]: \quad \frac{\{t \wedge P\}\ S_1\ \{Q\},\ \{\neg t \wedge P\}\ S_2\ \{Q\}}{\{P\}\ \text{if } b \text{ then } S_1 \text{ else } S_2\ \{Q\}} \quad \text{where } t = \mathcal{B}[\![b]\!]$$

**Figure 10** Hoare Logic

Some applications of axiomatic semantics: documentation of programs and interfaces [HW1973, Sou1984, FM2001]; guidance in design and coding [HJ2000]; proving the correctness of algorithms (or finding bugs) [Hoa1969, ILL1975, Kem1982, Dav1999]; proving the correctness of hardware descriptions (or finding bugs); extended static checking, and proof-carrying code [ON1999].

## 6.2 Dynamic Logic (1984)

Traditional axiomatic descriptions of programs don't possess such feature of sequence as operational descriptions. This omission to temporal feature can bring benefits (such as concision and high abstraction) for a simple model language, but not suit for dealing with real programming language [YR1998]. So, Harel joined the temporal feature to axiomatic, and proposed *dynamic semantics* [HKT1984], which can be described as a blend of three complementary classical ingredients: first-order predicate logic, modal logic, and the algebra of regular events. Unlike classical predicate logic where truth is static, dynamic logic has explicit syntactic constructs called programs whose main role is to change the values of variables, thereby changing the truth-values of formulas. To discuss the effect of the execution of a program $\alpha$ on the truth of a formula φ, dynamic logic uses a modal construct $<\alpha>\varphi$, which intuitively states, "It is possible to execute $\alpha$ starting from the current state and halt in a state satisfying φ." There is also the dual construct $[\alpha]$ , which intuitively states, "If $\alpha$ halts when started in the current state, then it does so in a state satisfying φ."(see Figure 11) [HKT2000].

Applications of dynamic logic on formal language and program verification can be found in [HKT2000, CLR1996, EJ1996, ALP1998, Bec2001, MM2003].

$$[\text{ass}_{dl}]: \ < x := a > \varphi = \varphi\,[a/x]$$

$$[\text{if}_{dl}]: \ [\text{if } b \text{ then } S_1 \text{ else } S_2]\varphi = [b?\ ;\ S_1]\ \varphi \cup [\neg b?\ ;\ S_2]\varphi$$

**Figure 11** Dynamic Semantics

## 6.3 Algebraic Semantics (1981)

In the logic approaches mentioned above, abstract data types (ADTs) are not unique, since these approaches only provide sets of axioms and inference rules, don't explain what the designer want or not. So, it is necessary to find a new approach for describing ADTs. Algebra is a suitable tool for it [Ru1992]. *Algebraic semantics* [Gue1981, GD1992, SK1995, Zam1997], whose foundations are based on abstract algebra, involves the algebraic specification of data and language constructs. The basic idea of the algebraic approach to semantics is to name the sorts of objects and the operations on the objects, and to use algebraic axioms to describe their characteristic properties (illustrated in Figure 12). An algebraic specification contains two parts: signature[7] and equations. The methodology of algebraic semantics is customarily used to specify abstract data types. The basic principle in specifying an ADT involves describing the logical properties of data objects in terms of properties of operations (some of which may be constants) that manipulate the data.

**type** While-Program =
{ BI-Expression +
**sort** **stm**
**op** nop: → **stm**
ass: **id** × **exp** → **stm**
cond: **bexp** × **stat** × **stm** → **stm**
comp: **stm** × **stm** → **stm**
seval: **stm** × **exp** → **int**
sbval: **stm** × **bexp** → **bool**
**axiom**
comp($S$, nop) = comp(nop,$S$) = $S$
comp(comp($S_1$,$S_2$),$S_3$) = comp($S_1$, comp($S_2$,$S_3$))
seval(nop,$a$) = eval($a$)
sbval(nop,$b$) = bval($b$)
seval(comp($S$,ass($x$,$a_1$)), $a_2$) = seval($S$, subs($a_2$,$x$,$a_1$))
seval(comp($S$,cond($b$,$S_1$,$S_2$)),$a$) = **if** sbval($S$,$b$) **then**
seval(comp($S$,$S_1$),$a$) **else** seval(comp($S$,$S_2$),$a$) **fi**
defined($S$) = True }

**Figure 12** Algebraic Semantics

Algebraic semantics has frequently been used to study semantics of functional and imperative languages [eg. Wil1982, BW1982, Ait1986, BWP1987, Acz1989, CJO1994, GD1992, GH1995, Zam1997, JMK1998, Fro2003]. It is also used in the field of abstract state machines to formalize the machine model underlying an operational semantics [cf. Gog1990, GH1993].

## 6.4 Other Axiomatic Semantics

Other axiomatic semantics include *Kripke-Kleene semantics* [Fit1985a, GRS1991], *fixpoint semantics* [Fit1985b, Fit2002], and *fixpoint logic* [Var1982, AVV1997].

[7] A signature S of an algebraic specification is a pair <Sorts, Operations> where Sorts is a set containing names of sorts; Operations is a family of function symbols indexed by the functionalities of the operations represented by the function symbols.

# 7 Hybrid Frameworks

A hybrid approach to semantics involves more than one framework in the same description. The various semantic frameworks may have advantages for different levels of complete semantic descriptions [Mos2001a, Mos2001b]. Here, we will consider four approaches to axiomatic: algebraic operational semantics, action semantics, modular monadic semantics and modular monadic action semantics.

## 7.1 Algebraic Operational Semantics (1987)

*Algebraic operational semantics* (AOS, see [Gur1987, TZ1988, Ste1996]) is a hybrid framework. Like dynamic logic, it is obtained by introducing temporal feature to axiomatic. The difference is that algebraic and operational are directly bonded together in AOS. In AOS, a simple model of a clock is employed to enumerate the sequence of states that are produced by executing a program. The transition from one state in the sequence to the next is given by some atomic program, such as assigning the value of one variable to another. The idea of AOS is to decompose a program into a sequence of atomic programs (see functions *First* and *Rest* in Figure 13), such that the sequential execution of these atomic programs gives the behaviour of the whole program. The operational semantics is algebraic because the semantics of a language is defined as an algebra, and thanks to the enumeration clock, this decomposition process can be defined by using equations [Sem1997, GM1990].

$$\mathit{Comp} : \mathbf{Stm} \times \mathbf{State} \times \mathbf{Time} \to \mathbf{State}$$
$$\mathit{Act} : \quad \mathbf{AStm} \times \mathbf{State} \to \mathbf{State}$$
$$\mathit{First} : \mathbf{Stm} \times \mathbf{State} \to \mathbf{AStm}$$
$$\mathit{Rest} : \quad \mathbf{Stm} \times \mathbf{State} \to \mathbf{Stm}$$
$$\mathit{Comp}(S, s, 0) = s$$
$$\mathit{Comp}(S, s, 1) = \mathit{Act}(\mathit{First}(S, s), s)$$
$$\mathit{Comp}(S, s, t+1) = \begin{cases} * & \text{if } S \text{ is atomic} \\ \mathit{Comp}(\mathit{Rest}(S, s), \mathit{Comp}(S, s, 1), t) & \text{if } S \text{ is not atomic} \end{cases}$$
$$[\mathrm{ass}_{\mathrm{aos}}]: \mathit{First}(x := a) = x := a$$
$$\mathit{Rest}(x := a, s) = \mathrm{skip}$$
$$[\mathrm{if}_{\mathrm{aos}}]: \mathit{First}(\text{if } b \text{ then } S_1 \text{ else } S_2) = \mathrm{skip}$$
$$\mathit{Rest}(\text{if } b \text{ then } S_1 \text{ else } S_2, s) = \begin{cases} S_1 & \text{if } t = \mathbf{tt} \\ S_2 & \text{if } t = \mathbf{ff} \end{cases}$$
$$\text{where} \quad t = [\![ b ]\!] s$$

**Figure 13** Algebraic Operational Semanitcs

The basis of AOS is a function *Comp* (showed in Figure 13) such that *Comp* ($S$, $s_0$, $t$) gives the state $s_t$ that results from executing the program $S$ on the initial state $s_0$ for $t$ cycles of time. In Figure 13, the function $Act(\alpha, s)$ gives the behaviour of a basic or atomic program $a$ on a state $s$, and satisfies $s_t = Act(\alpha_{t-1}, s_{t-1}) = Comp(S, s_0, t)$.

The applications of AOS in the field of studying programming languages and correctness are declared in [GM1988, GM1990, Bor1990, TZ1988, Ste1997].

## 7.2 Action Semantics (1986)

The traditional frameworks for formal semantics, including operational (Section 4), denotational (Section 5), and axiomatic semantics (Section 6), are widely taught at university level; they are firmly based on solid theoretical foundations, and they are quite adequate for describing small-idealized fragments of programming languages. However as Mosses pointed out (in [Mos1996a]), when attempting to scale up to larger, and practical programming languages, such as Pascal and C, it turns out to be disproportionately difficult to write, read, extend, modify, and reuse descriptions in the conventional frameworks. Three attempts at solving these problems are action semantics, modular monadic semantics (Section 7.3) and modular monadic action semantics (Section 7.4) [Mos1992a, WH1997].

*Action semantics* (cf. [Mos1986], [MW1987], [Mos1996b]) improves the modularity of denotational semantics by taking denotations to be so-called actions, which are expressed using a fixed action notion consisting of various primitives and combinators. The foundations of action notation involve SOS and algebraic specifications, which avoids the use of higher-order functions expressed in lambda-notation, and are both generally regarded as more accessible than domain theory. Action notation provides direct support for specifying control flow, data flow, scopes of bindings, side-effects, procedural abstraction, and (asynchronous) communication between concurrent process [Mos1989, Mos1992a, Mos1996a, Mos1998].

In principle, the semantic description of each construct is a separate module. This facilitates reuse of complete modules in descriptions of different languages, and allows new languages to be assembled simply by combining different sets of modules [DM2003]. Action semantics is readable and intelligible without prior training (showed in Figure 14). It has an operational semantics base, from which can be derived laws or 'axioms' that can be used to prove facts about programs; and programs or fragments can be shown to be equivalent (or not) directly by using the operational semantics. Moreover it has a very flexible type system that allows new types of data to be readily defined and used.

Action semantic descriptions exist for a number of real languages [Wat1988, MW1993, Tof1993, HT1994, Mos1992b, Bun1996, Wat1999]. Various prototype compiler generators based on action semantics have been developed [Mos2001c, BMW1992, Pal1992a, Pal1992b, Mou1993, Ørb1994, Bun1996], along with some tools (cf. [Mos1992, Mos1996b, DM1996]) for assistance in developing new action semantic descriptions.

```
    execute:   Stm → action
execute⟦x := a⟧= ( evaluate a then
  store the primitive-value in the cell bound to token-of x)
execute⟦if b then S1 else S2⟧= (evaluate b then
          (check (it is true) and then evaluate S1   or
              check (it is false) and then evaluate S2 ) )
```

**Figure 14** Action Semantics

## 7.3 Modular Monadic Semantics (1996)

In section 5.7, we saw that the use of monads in denotational semantics descriptions can also improve their modularity. However, the monadic approach has the problem that, in general, it is not possible to compose two monads to obtain a new monad [LLCC2001, LCLC2002]. A proposed solution was the use of monad transformers (cf. [LHJ1995]), which transform a given monad into a new one adding new operations. Monad transformers are a form of abstraction for introducing a wide range of computational behaviors, such as state, I/O, continuations, exceptions, parsing and non-determinism. This approach was called *modular monadic semantics* (MMS, [LH1996, Lia1998]). MMS specifies the semantics of a language by a mapping from terms to computations performed within a monad. The monad hides details of semantic features such as environments and stores, and exposes operators allowing access to these features. The semantic description consists of the monad definition, defining the monad over which the language is defined, and the semantic functions relating the syntax of phrases in the language to their meaning or semantics (see Figure 15).

```
  S_mms: Stm → M( )
  E_mms: Term → M Value
    write:   (Loc, M Value) → M( )
    return: a → M a
S_mms⟦x := a⟧= ( v ← E_mms⟦a⟧; write (x, return v) )
S_mms⟦if b then S1 else S2⟧= (v ← E_mms⟦a⟧;
                    v = tt → S_mms⟦S1⟧, S_mms⟦S2⟧)
```

**Figure 15** Modular Monadic Semantics

In contrast to action semantics, MMS has the advantage of being based directly on denotational semantics, which is familiar and well-understood. Compared with traditional denotational semantics, MMS captures individual programming language features using reusable building blocks, and specifies programming languages by composing the necessary features. It achieves a high level of modularity and extensibility, in that the various notions of computation introduced are defined entirely separately and can be mixed and matched as required. Despite of this, it is still executable: there is a clear operational interpretation of the semantics [Wan1997, WH1997]. And MMS is to a large degree independent of the type system used.

Because of its truly modularity, MMS has advantages for reusing and modifying incrementally the semantic descriptions of programming languages, showed in [LCLC2001, LCLC2002, LLCC2001, Lab1998].

## 7.4 Modular Monadic Action Semantics (1997)

As mentioned above, action semantics possesses modularity and readability simultaneously. However, it is limited in scope: not all programming language concepts (such as first-class continuations) can be represented directly within action semantics—only those which Mosses has chosen to build into the system. There is no provision for the extension of action semantics. It is not really modular internally: the operational semantics deals with all facets together, and the apparent modularity exists only at the top level (see Figure 1(3)). Additionally, a number of operations do not in fact restrict their operations to a single facet. Action semantics' type system is rather unconventional. Compared to action semantics, modular monadic semantics is truly modularity (see Figure 1(4)), but it does not have the property of being intelligible without prior training. And extensible union types, the type system chosen by Liang, Hudak and Jones, are not as good as Mosses's unified algebras (which are difficult to fit within the modular monadic framework) [Wan1997, WH1997].

It is apparent that MMS is both lower-level and more general than action semantics, but that both approaches have much to commend them. So, Wansbrough naturally proposed their fusion— *modular monadic action semantics* (MMAS, [Wan1997]). MMAS is to define action semantics in terms of MMS, by using MMS to give a modular denotational definition of action notation—replacing its original SOS (which has rather poor modularity). This replacement provides the flexibility to modify or delete existing facets of action semantics or to add entirely new ones, and will allow the use of the theories of modular monadic semantics and of denotational semantics to theorise about actions. In other words, the human readability of action semantics is maintained, but the mathematical understanding of what is going on is greatly enhanced by the modular, accessible and readable description given by the MMS. Consequently, MMAS is modular and extensible, and dialects of MMAS can be created that incorporate new or modified notions of computation.

```
    execute :    Stm → action
execute⟦x := a⟧= ( aEvaluate a   cThen
      aStoreIn (yPrimitiveValue, (yDateBoundTo "x")))
execute⟦if b then S1 else S2⟧= ( aEvaluate b   cThen
       ( aCheck (yIt) 'cAndThen' aEvaluate S1   cOr
         aCheck (yNot yIt) 'cAndThen' aEvaluate S2) )
```

**Figure 16** Modular Monadic Action Semantics

The MMAS notation differs slightly from that of action semantics. All operators begin with a single lowercase letter, **a**, **c**, or **y**, identifying whether the operator is an action (such as **aEvaluate** in Figure 16), combinator (**cThen**) or yielder (**yIt**), respectively. The name of the operator follows, occasionally abbreviated, in BiCapitalised form.

Base MMAS, the unmodified form of the MMAS system [Wan1997], implements almost all of Mosses's action semantics: in most cases existing action semantic descriptions can be used with MMAS with very little modification. Such ASDs can be interpreted in an MMAS system based over a Haskell interpreter, or made into compilers using an optimising Haskell compiler.

### 7.5 Other Hybrid Semantics

Other hybrid semantics include *algebraic denotational semantics*[GP1981],*modular denotational semantics* [Esp1993, Esp1995] and *type-theoretic interpretation* [HL1994, HS1998]. As mentioned in [Mos2001b], various operational frameworks such as VDL may be considered as hybrids.

## 8 Comparisons

In sharp contrast to the popularity of formal syntax, formal semantic descriptions have seldom been exploited in practical applications concerning design and implementation of programming languages. As showed in Section 4-7, there is no shortage of semantic frameworks to choose from, nor has there been a lack of theoretical effort in establishing the foundations of the various frameworks. In [Mos1996a, Mos1998, Mos2001a, Mos2001b, Mos2001c, MW1987], Mossess pointed out that the main hindrances to greater use of formal semantics appear to be lack of user-friendliness, and lack of tool support. Ideally, formal semantic descriptions should provide a convenient way for language designers to record their decisions, and to communicate them to implementers and programmers. In these literatures, we find a list of these properties a programming language specification method should have:

- *Readability*. This property makes the description accessible to all people with interest in the language (designers, implementers and programmers).
- *Modularity*. Modularity in formal descriptions improves reusability and modifiability, also helps in breaking large descriptions into smaller and manageable components.
- *Abstractness*. The formalism should be abstract enough to free the designer from biasing towards any implementation alternative and to focus on important design issues.
- *Comparability*. It should be easy to compare different languages by looking into their formal descriptions.
- *Reasonability*. The formalism should facilitate reasoning about programs written in the defined language.
- *Applicability*. The formalism could describe nearly all programming-language concepts, such as state, I/O, (first-class) continuations, exceptions, parsing and non-determinism.
- *Tool-support*. Quality tools are badly needed to assist the writing, checking, and reading of semantic descriptions. The wider use of formal semantics depends on the availability of tools for generating (reasonably efficient) implementations from semantic descriptions.

From these aspects, we give the qualitative comparisons of the semantic description methods of current interest in Table 1. In addition, our conclusion from this table is that ASM (Section 4.3) and MMAS (Section 7.4) approaches are good candidates for such ideal frameworks. ASM approach has already made a considerable impact regarding practical applications. MMAS approach is a new approach, but it has a large and energetic following.

## 9 Summaries

The design and implementation of programming languages is an importance topic in computer science. There are two aspects to the specification of programming languages: syntax and semantics. Formal descriptions of program syntax (regular, context-free, and context-sensitive grammars) have become accepted as practically useful for documentation in reference manuals and standards, as well as for generating efficient parsers for use in compilers. In contrast, formal semantic descriptions have seldom been exploited in practical applications concerning design and implementation of programming languages. Compared to the amount of effort that has been made to the research of various semantic frameworks (main of them were listed in Section 4-7 in this paper) over more than forty years (showed in Section 2), their actual applications are definitely frustrating. Good pragmatic features, such as readability, modularity, etc. (see Table 1), are strongly demanded for efficient development and use of semantic descriptions, but are sadly lacking in most frameworks.

According to Table 1, we know that the ideal formal semantic descriptions should possess: 1) enough readability; 2) well modularity; 3) high abstractness; 4) strong comparability; 5) enough reasonability; 6) wide application; 7) more tool-support. And we find that ASM and MMAS are two good candidates for such a framework.

## Acknowledgments

Thanks to Yuan Liu and Jing Cao for their helpful comments on this paper.

**Table 1** Comparisons of the Semantic Description Methods

| Properties / Frameworks | Readability | Modularity | Abstractness | Comparability | Reasonability | Applicability | Tool-support |
|---|---|---|---|---|---|---|---|
| **SOS** | General | Poor | Low | Strong | Weak | Wide | More |
| **Modular SOS** | General | Well | Middle | General | General | Moderate | Little |
| **ASM** | Well | General | High | General | General | Wide | General |
| **Scott-Strachey** | Poor | Poor | High | General | Strong | Wide | General |
| **Game** | General | Poor | Middle | Weak | Weak | Limited | Little |
| **Monadic** | Poor | General | Middle | Weak | Strong | Moderate | Few |
| **Hoare Logic** | General | Poor | Low | Strong | General | Limited | More |
| **Dynamic Logic** | Poor | Poor | Middle | General | Weak | Limited | Little |
| **Algebra** | Poor | General | High | Weak | General | Limited | Little |
| **Algebra Operational** | General | Poor | Middle | Strong | Weak | Moderate | Little |
| **Action** | Well | General | High | General | General | Limited | General |
| **Modular Monadic** | Poor | Well | High | Weak | Strong | Moderate | Little |
| **MMAS** | Well | Well | High | General | General | Moderate | Little |

# References


[ABV1994] Aceto, L., Bloom, B. and Vaandrager, F., Turning SOS rules into equations. Information and Computation, 1994, 111(1): 1-52.

[Acz1989] Aczel, P., Algebraic semantics for intensional logics. In: Chierchia, G., Partee, B. and Turner, R. eds. Properties, Types and Meaning, Volume I: Foundational Issues, Dordrecht, 1989. 17-45.

[AHM1998] Abramsky, S., Honda, K. and McCusker, G., A fully abstract game semantics for general references. In: 13th Annual IEEE Symposium on Logic in Computer Science, New York: IEEE Computer Society Press, 1998. 334-344.

[Ait1986] Aït-Kaci, H., An algebraic semantics approach to the effective resolution of type equations. Theoretical Computer Science, 1986, 45:293-351.

[AJM2000] Abramsky, S., Jagadeesan, A.R. and Malacaria, P., Full abstraction for PCF. Information and Computation, 2000, 163(2): 409-470.

[ALP1998] Alferes, J.J., Leite, J.A. Pereira, L.M., Przymusinska, H. and Przymusinski, T.C., Dynamic logic programming. In: 6th International Conference on Principles of Knowledge Representation and Reasoning, KR'98, Morgan, 1998. 98-111.

[AM1997a] Abramsky, S. and McCusker, G., Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In: O'Heam, P.W. and Tennent, R.D. eds. Algol-like Languages, 1997, 2: 297-329.

[AM1997b] Abramsky, S. and McCusker, G., Call-by-value games. In: Nielsen, M. and Thomas, W. eds. Computer Science Logic: 11th International Workshop on Annual Conference of the EACSL, CSL'97. LNCS 1414, Berlin: Springer-Verlag, 1997. 1-17.

[AM1998] Abramsky, S. and McCusker, G., Game semantics. In: Schwichtenberg, H. and Berger, U., eds. Logic and Computation: Proceedings of the 1997 Marktoberdorf Summer School, Berlin: Springer-Verlag, 1998.

[AM1999] Abramsky, S. and McCusker, G., Full abstraction for Idealized Algol with passive expressions. Theoretical Computer Science, 1999, 227: 3-42.

[AR1987] Astesiano, E. and Reggio, G., SMoLCS-driven concurrent calculi. In: Proceedings of International Joint Conference on Theory and Practice of Software Development, TAPSOFT'87. LNCS 249, Berlin: Springer-Verlag, 1987. 169-201.

[Att1996] Attali, I., A natural semantics for Eiffel dynamic binding. ACM Transactions on Programming Language and Systems, 1996, 18(6):711-729.

[AVV1997] Abiteboul, S., Vardi, M.Y. and Vianu, V., Fixpoint logics, relational machines, and computational complexity. Journal of ACM, 1997, 44(1):30-56.

[AW2003] ASM-Website, 2003. http://www.eecs.umich.edu/gasm/.

[BBKL1982] Bodwin, J., Bradley, L., Kanda, K., Litle, D. and Pleban, U.F., Experience with an experimental compiler generator based on denotational semantics. In: Proceeding of the SIGPLAN'82 Symposium on Compiler construction, Boston, 1982. 216-229.

[BD1990] Börger, E. and Dässler, K., Prolog: DIN papers for discussion. ISO/IEC JTCI SC22 WG17 Proglog Standardization Document 58, Middlesex, England: National Physical Laboratory, 1990.

[BD1996] Börger, E. and Durdanovic, I., Correctness of compiling Occam to transputer code. Computer Journal, 1996, 39(1): 52-92.

[Bec2001] Beckert, B., A dynamic logic for the formal verification of Jave card programs. In: 1st International Workshop of JavaCard 2000 Cannes. LNCS 2041, Berlin: Springer-Verlag, 2001. 6-25.

[BG1994] Borba, P. and Goguen, J.A., An operational semantics for FOOPS. Technical Report PRG-16-94, Programming Research Group, University of Oxford, 1994. http://web.comlab.ox.ac.uk/oucl/publications/tr/tr-16-94.html.

[BGM1994] Börger, E., Glässer, U. and Müller, W., The semantics of behavioral VHDL'93 descriptions. In: European Design Automation Conference with EURO-VHDL'94, EURO-DAC'94, New York: IEEE Computer Society Press, 1994. 500-505.

[BJ1978] Bjørner, D. and Jones, C.B., The Vinenna Development Method: the meta-language. LNCS 61, Berlin: Springer-Verlag, 1978.

[Bla1992] Blass, A., A game semantics for linear logic. Annals of

Pure and Applied Logic. 1992, 56(1-3): 183-220.

[Blo1989] Bloom, B., Ready simulation, bisimulation, and the semantics of CCS-like language [Ph.D. thesis]. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1989.

[BMW1992] Brown, D.F., Moura, H. and Watt, D.A., Actress: an action semantics directed compiler generator. In: 4th International Conference on Compiler Construction, CC'92. LNCS 641, Berlin: Springer-Verlag, 1992. 95-109.

[Bor1990] Börger, E., A logical operational semantics for full Prolog. In: 3rd Workshop on Computer Science Logic, CSL'89. LNCS 440, Berlin: Springer-Verlag, 1990. 36-64.

[BR1994] Börger, E. and Rosenzweig, D., The WAM – definition and compiler correctness. In: Logic Programming: Formal Methods and Practical Applications. North-Holland: Computer Science and Art. Int., 1994.

[BR1995] Börger, E. and Rosenzweig, D., A mathematical definition of full Prolog. Science of Computer Programming, 1995, 24: 249-286.

[BS2003] Börger, E. and Stärk, R., Abstract state machine: a method for high-level system design and analysis. Berlin: Springer-Verlag, 2003.

[BT1983] Blikle, A. and Tarlecki, A., Naïve denotational semantics. In: Proceedings of IFIP Congress on Information Processing, North-Holland, 1983.

[Bun1996] Bundgaard, J., An ANDF based Ada 95 compiler system. In: Toussaint, M. ed. 2nd International Ada- Europe Symposium on Reliable Software Technologies,. LNCS 1031, Berlin: Springer-Verlag, 1996. 81-91.

[BW1982] Broy, M. and Wirsing, M., Algebraic definition of a functional programming language. IEEE Transactions on Information Theory, 1982, 17(2):137-161.

[BWP1987] Broy, M., Wirsing, M. and Pepper, P., On the algebraic definition of programming languages. ACM Transactions on Programming Languages and Systems, 1987, 9(1):54-99.

[Byu2003] Byun, S., Introduction to linear logic and game semantics. Lecture Notes, University of Kyungsung, 2003. http://licomr.org/2003_1/.

[BZ1992] Bakker, J.W., and Zucker, J.I., Denotational semantics of concurrency. In: Proceeding of the 14th Annual ACM Symposium on Theory of Computing, San Francisco, 1982. 153-158.

[CF1994] Cartwright, R. and Felleisen, M., Extensible denotational semantics specifications. In: Proceedings of Symposium on Theoretical Aspects of Computer Software, TACS'94. LNCS 789, Berlin: Springer-Verlag, 1994. 244-272.

[Chr2000] Chroboczek, J., Game semantics and subtyping. In: 15th Annual IEEE Symposium on Logic in Computer Science, New York: IEEE Computer Society Press, 2000. 192-204.

[CHT2002] Calcagno, C., Helsen, S. and Thiemann, P., Syntactic type soundness results for the region calculus. Information and Computation, 173 (2): 199-221.

[CJO1994] Clerici, S., Jimenez, R. and Orejas, F., Semantic constructions in the specification language Glider. In: Ehrich, H.D. and Orejas, F. eds. Recent Trends in Data Type Specification. LNCS 785, Berlin: Springer-Verlag, 1994. 144-157.

[Cle2003] Clem, B., Principles of programming languages: the semantics of programming languages. Lecture Notes: COMP3610, Department of Computer Science, Australian National University, 2003. http://cs.anu.edu.au/Student/comp3610/.

[CLR1996] Carles, S., Lluis, G., Ramon, L.M. and Mara, M., Descriptive dynamic logic and its application to reflective architectures. Future Generation Computer Systems, 1996, 12(2-3): 157-171.

[CP1994] Cook, W. and Palsberg, J., A denotational semantics of inheritance and its correctness. Information and Computation, 1994, 114(2): 329-350.

[Dav1999] David von O., Hoare logic for mutual recursion and local variables. In: Rangan, C.P., Raman, V. and Ramanujam, R. eds. 19th Conference on Formal Software Theory and Theoretical Computer Science, FST&TCS'99. LNCS 1738, Berlin: Springer-Verlag, 1999. 168~180.

[DDR1997] Dong, J.S., Duke, R. and Rose, G., An object-oriented denotational semantics of a small programming language. Object-Oriented Systems journal, 1997, 4(1): 29-52.

[DE1999] Drossopoulou, S. and Eisenbach, S., Describing the semantics of Java and proving type soundness. In: Foss, A. ed. Formal Syntax and Semantics of Java. LNCS 1523, Berlin: Springer-Verlag, 1999. 41-82.

[Dij1975] Dijkstra, E.W., Guarded commands, non-determinacy, and formal derivations of programs. Communications of ACM, 1975, 18: 453-457.

[DM1996] Deursen, A. and Mosses, P.D., ASD: The action semantic description tools. In: Proceeding of 5th International Conference on Algebraic Methodology and Software Technology, AMAST'96. LNCS 1101, Berlin: Springer-Verlag, 1996. 579-582.

[DM2003] Doh, K. and Mosses, P.D., Composing programming languages by combining action semantics modules. Science of Computer Programming, 2003, 47(1): 3-36.

[DP1996] Degano, P. and Priami, C., Enhanced operational semantics. ACM Computing Survey, 1996, 28 (2): 352-354.

[DP2001] Degano, P. and Priami, C., Enhanced operational semantics: a tool for describing and analyzing concurrent systems. ACM Computing Survey, 2001, 33 (2): 135-176.

[DS1990] Dijkstra, E.W. and Scholten, C.S., Predicate calculus and program semantics. New York: Springer Press, 1990.

[EJ1996] van Eijck. J. and Jaspars, J., Ambiguity and reasoning. Technical Report CS-9616, Nertherlands: Centrum voor Wiskunde en Informatica (CWI), 1996. http://ftp.cwi.nl/CWIreports/INDEX.

[Esp1993] Espinosa, D.A., Modular Denotational semantics. Unpublished manuscript, 1993.

[Esp1995] Espinosa, D.A., Semantic Lego [Ph.D. thesis]. Graduate School of Arts and Sciences, Columbia University, 1995.

[FF1986] Fellesisen, M. and Friedman, D.P., Control operators, the SECD machine, and the λ -calculus. In: Proceedings of IFIP TC2 Working Conference, Formal Description of Programming Concepts III, North-Holland, 1986. 193-217.

[Fit1985a] Fitting, M., A Kripke-Kleene semantics for logic programs. Journal of Logic Programming, 1985, 2(4): 295-312.

[Fit1985b] Fitting, M., A deterministic Prolog fixpoint semantics. Journal of Logic Programming, 1985, 2(2): 111-118.

[Fit2002] Fitting, M., Fixpoint semantics for logic programming a survey. Theoretical Computer Science, 2002, 278(1-2):25-51.

[FM2001] Fikes, R. and McGuinness, D.L., An axiomatic semantics for RDF, RDF-Schema, and DAML+OIL. Technical

Report KSL-01-01, University of Stanford, 2001. http://www.ksl.stanford.edu/people/dlm/daml-semantics/.

[Fro2003] Fronk, A., An approach to algebraic semantics of object-oriented languages. In: 7th Brazilian Symposium on Programming language, 2003. 195-209.

[GD1992] Goguen, J.A. and Diaconescu, R., Towards an algebraic semantics for the object paradigm. In: Ehrig, H. and Orejas, F. eds. Recent Trends in Data Type Specification. LNCS 785, Berlin: Springer-Verlag, 1992. 1-59.

[GGP1999] Glässer, U., Gotzhein, R. and Prinz, A., Towards a new formal SDL semantics based on abstract state machines. In: von Bochmann, G., Dssouli, R. and Lahav, Y. eds. Proceeding of the 9th SDL Forum, SDL'99, Elsevier Science B.V., 1999. 171-190.

[GH1993] Gurevich, Y. and Huggins, J.K., The semantics of the C programming language. In: Selected papers from CSL'92 (Computer Science Logic). LNCS 702, Berlin: Springer-Verlag, 1993. 274-308.

[GH1995] Gogolla, M. and Herzig, R., An algebraic semantics for the object specification language TROLL light. In: Astesiano, E., Reggio, G. and Tarlecki, A. eds. Recent Trends in Data Type Specification. LNCS 906, Berlin: Springer-Verlag, 1995. 290-306.

[GK1997] Glässer, U. and Karges, R., Abstract state machine semantics of SDL. Journal of Universal Computer Science, 1997, 3(12): 1382-1414.

[Gle1999] Glesner, S., Natural semantics for imperative and object-oriented programming language. GI Jahrestagung, 1999. 370-379.

[Gle2003] Glesner, S., ASMs versus natural semantics: a comparison with new insights. Abstract State Machines – Advances in Theory and Apllications, ASM 2003, Berlin: Springer-Verlag, 2003. 293-308.

[GM1988] Gurevich, Y. and Moss, L.S., Algebraic operational semantics and Modula-2. 1st Workshop on Computer Science Logic, CSL'87. LNCS 329, Berlin: Springer-Verlag, 1988. 81-101.

[GM1990] Gurevich, Y. and Moss, L.S., Algebraic operational semantics and Occam. In: 3rd Workshop on Computer Science Logic, CSL'89. LNCS 440, Berlin: Springer-Verlag, 1990. 176-192.

[GM1996] Goguen, J.A. and Malcolm, G., Algebraic semantics of imperative programs. USA: MIT Press, 1996.

[Gog1990] Goguen, J., A algebraic approach to refinement. In: Bjorner, D., Hoare, C.A.R. and Langmaack, H. eds. Proceedings of VDM and Z: Formal Methods in Software Development, VDM'90. LNCS 428, Berlin: Springer-Verlag, 1990. 12-28.

[GP1981] Goguen, J.A. and Parsaye, K.G., Algebraic denotational semantics using parameterized abstract modules. In: Diaz, J. and Ramos, I. Eds. Proceedings of International Colloquium on Formalization of Programming Concepts. LNCS 107, Berlin: Springer-Verlag, 1981. 292-309.

[GRS1991] Gelder A.V., Ross, K.A. and Schlipf, J.S., The well-founded semantics for general logic programs. Journal of ACM, 1991, 38: 620-650.

[GTWW1977] Goguen, J.A., Thatcher, J.W., Wagner, E.G., and Wright, J.B. Initial algebra semantics and continuous algebras. Journal of the ACM, 1977, 24:68-95.

[Gue1981] Guessarian, I., Algebraic semantics. LNCS 99, Berlin: Springer Press, 1981.

[Gur1987] Gurevich, Y., Algebraic operational semantics. In: Nori, K.V. ed. Foundations of Software Technology and Theoretical Computer Science, FSTTCS'87. LNCS 287, Berlin: Springer-Verlag, 1987.

[Gur1993] Gurevich, Y., Evolving algebras 1993: Lipari guide. In: Börger, E., ed. Specification and Validation Methods. USA: Oxford University Press, 1995. 9-36.

[GZ1998] Glesner, S. and Zimmermann, W., Using many-sorted natural semantics to specify and generate semantics analysis. In: Proceeding of Systems Implementation Conference, IFIP WG 2.4, Chapman & Hall, 1998.

[GZ1999] Goos, G. and Zimmermann, W., Verification of Compilers. In: Correct System Design. LNCS 1710, Berlin: Springer-Verlag, 1999. 201-230.

[Har1984] Harel, D., Dynamic logic. In: Gabbay, D. and Guenther, F. eds. Handbook of Philosophical Logic Volume II - Extensions of Classical Logic. Netherlands: Reidel Publishing Company, 1984. 497-604.

[Hen1990] Hennessy, M., The semantics of programming languages: an elementary introduction using structural operational semantics. New York: John Wiley & Sons, 1990.

[HJ2000] Huisman, M. and Jacobs, B., Java program verification via a Hoare logic with abrupt termination. In: Maibaum, T. ed. 3rd International Conference on Fundamental Approaches to Software Engineering, FASE'00, LNCS 1783. Springer-Verlag, 2000. 284-303.

[HKT2000] Harel, D., Kozen, D. and Tiuryn, J., Dynamic logic. USA: The MIT Press, 2000.

[HL1994] Harper, R. and Lillibridge, M., A type-theoretic approach to higer-order modules with sharing. In: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'94, Portland, 1994. 123-137.

[HM1999] Harmer, R. and McCusker, G., A fully abstract game semantics for finite nondeterminism. In: 14th Annual IEEE Symposium on Logic in Computer Science, New York: IEEE Computer Society Press, 1999. 422-430.

[HO2000] Hyland, M. and Ong, L., On full abstraction for PCF: I, II and III. Information and Computation, 2000, 163(2): 285-408.

[Hoa1969] Hoare, C.A.R., An axiomatic basis for computer programming. Communications of the ACM, 1969, 12(10):576-583.

[HS1998] Harper, R. and Stone, C., A type-theoretic interpretation of Standard ML. In: Plotkin, G., Stirling, C. and Tofte, M. eds. Proof, Language and Interaction: Essays in Honour of Robin Milner. USA: MIT Press, 1998.

[HT1994] Hansen, B.S. and Toft, T.U., The formal specification of ANDF, An application of action semantics. In: Mosses, P.D. ed. 1st International Workshop on Action Semantics, Edinburgh: University of Aarhus, 1994. 34-42.

[HW1973] Hoare, C.A.R. and Wirth, N., An axiomatic definition of the programming language Pascal. Acta Informatica, 1973, 2:335-355.

[ILL1975] Igarashi, S., London, R.L. and Luckham, D.C., Automatic program verification I: a logic basis and its implementation. Acta Information, 1975, 4:145-182.

[ITU2000] ITU-T., SDL formal semantics definition. ITU-T Recommendation Z.100 Annex F, International Telecommunication Union, 2000.

[JMK1998] Joxan, J., Michael, M., Kim, M. and Peter, S., The semantics of constraint logic programs. The Journal of Logic Programming, 1998, 37(1-3): 1-46.

[Kem1982] Kemmerer, R.A., Formal verification of an operating system security kernel. Michigan: UMI Research Press, 1982.

[KLNS2002] Klein, G., Loetzbeyer, H., Nipkow, T. and Sandner, R., A WHILE-language and its semantics, Technical Report, 2002. http://isabelle.in.tum.de/PSV2000/library/HOL/IMP/README.html

[KOC1991] Kurtz, B.L., Oliver, R.L. and Collins, M., The design, implementation, and use of DSTutor: a tutoring system for denotational semantics. ACM SIGCSE Bulletin, 1991, 23(1): 169-177.

[Lab1998] Labra, J., An implementation of modular monadic semantics using folds and monadic folds. In: 3rd International Summer School on Advanced Functional Programming, Portugal, 1998.

[Lai1997] Laird, J., Full abstraction for functional languages with control. In: 12th Annual IEEE Symposium on Logic in Computer Science, New York: IEEE Computer Society Press, 1997. 58-67.

[Lan1964] Landin, P.J., The mechanical evaluation of expressions. Computer Journal, 1964, 6(4):308-320.

[Lan1966] Landin, P.J., A formal description of Algol60. In: Proceedings of IFIP TC2 Working Conference, Formal Language Description Languages for Computer Programming, North-Holland, 1966. 266-294.

[Lau1968] Lauer, L., Formal definition of Algol 60, Technical Report TR. 25.088, IBM Lab. Vienna, 1968.

[LCLC2001] Labra Gayo, J.E., Cueva Lovelle, J.M., Luengo Diez, M.C. and Cernuda del Rio, A., Reusable monadic semantics of logic programs with arithmetic predicates. In: Proceedings 2001 APPIA-GULP-PRODE Joint Conference on Declarative Programming, AGP'01, Portugal: University of Evora Publisher, 2001. 31-45.

[LCLC2002] Labra Gayo, J.E., Cueva Lovelle, J.M., Luengo Diez, M.C. and Cernuda del Rio, A., Reusable monadic semantics of object oriented programming languages. In: 6th Brazilian Symposium on Programming Languages, SBLP'02, Brazil: PUC-Rio University, 2002.

[LH1996] Liang, L. and Hudak, P., Modular denotational semantics for compiler construction. In: 6th European Synposium on Programming Languages and Systems, ESOP'96. LNCS 1058, Berlin: Springer-Verlag, 1996. 219-234.

[LHJ1995] Liang, S., Hudak, P. and Jones, M., Monad transformers and modular interpreters. In: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95, New York: ACM Press,1995. 333-343.

[Lia1998] Liang, S., Modular monadic semantics and compilation [Ph.D. thesis]. Yale: University of Yale, 1998.

[LLCC2001] Labra Gayo, J.E., Luengo Diez, M.C., Cueva Lovelle, J.M. and Cernuda del Rio, A., A language prototyping system using modular monadic semantics. In: Workshop on Language Definitions, Tools and Applications, LDTA'01. ENTCS 44, Netherlands: Elesvier, 2001.

[LP1995] Larsen, P.G. and Pawlowski, W., The formal semantics of ISO VDM-SL. Computer Standards and Interfaces, 1995, 17(5-6): 585-602.

[LS2002] Loetzbeyer, H. and Sandner, R., A WHILE-language and two Semantics, Technical Report, 2002. http://wwwbroy.informatik.tu-muenchen.de/~isabelle/library/ZF/IMP/README.html.

[LW1995] Li, Z. and Wang, B., On denotational semantics of Smalltalk-80. Chinese Science Abstracts Series Part A, 1995, A14(5): 50-50.

[MA1986] Manes, E.G. and Arbib, M.A., Algebraic Approaches to Program Semantics. Berlin: Springer Press, 1986.

[McC1996] McCusker, G., Games and full abstraction for FPC. In: 11th Annual IEEE Symposium on Logic in Computer Science, LICS'96, New York: IEEE Computer Society Press, 1996. 174-183.

[MH1998] Malacaria, P. and Hankin, C., A new approach to control flow analysis. In: 7th International Conference on Compiler Construction, CC'98, Lisbon, 1998. 95-108.

[MH1999] Malacaria, P. and Hankin, C., Non-deterministic games and program analysis: an application to security. In: 14th Annual IEEE Symposium on Logic in Computer Science, New York: IEEE Computer Society Press, 1999. 443-452.

[Mit1996] Mitchell, J.C., Foundations for programming languages. USA: MIT Press, 1996.

[MM2003] Maarten, M. and Michael, M., Regular equivalence and dynamic logic. Social Networks, 2003, 25(1):51-65.

[Mog1989] Moggi, E., An abstract view of programming languages. LFCS Report, ECS-LFCS-90-113, University of Edinburgh, 1989. http://www.lfcs.informatics.ed.ac.uk/reports/90/ECS-LFCS-90-113/.

[Mog1991] Moggi, E., Notions of computation and monads. Information and Computation, 1991, 93:55-92.

[Mos1986] Mosses, P.D., Action semantics. In: 4th Workshop on Abstract Data Type, ADT'86, University of Braunschweig, 1986.

[Mos1989] Mosses, P.D., Unified algebras and action semantics. In: 6th Annual Symposium on Theoretical Aspects of Computer Science, STACS'89, 1989. 17-35.

[Mos1992a] Mosses, P.D., Action semantics. Cambridge, UK:Cambridge University Press, 1992.

[Mos1992b] Mosses, P.D., On the action semantics of concurrent programming languages. In: Semantics: Foundations and Applications, Proceeding of REX Workshop. LNCS 666, Berlin: Springer-Verlag, 1992. 398-424.

[Mos1996a] Mosses, P.D., Theory and practice of action semantics. In: Penczek, W., Szalas, A. and Wierzbicki, T. eds. 21th International Symposium on Mathematical Foundations of Computer Science, MFCS'96. LNCS 1113, Berlin: Springer-Verlag, 1996. 37-61.

[Mos1996b] Mosses, P.D., A tutorial on action semantics. Tutorial notes for FME'96: Formal Methods Europe, Oxford, 1996.

[Mos1998] Mosses, P.D., Semantics, modularity, and rewriting logic. In: Kirchner, C. and Kirchner, H., eds. 2nd International Workshop on Rewriting Logic and its Applications, ENTCS 15. Netherlands: Elesvier, 1998.

[Mos1999] Mosses, P.D., Foundations of modular SOS (extended abstract). In: Kutylowski, M., Pacholski, L. and Wierzbicki, T. eds. 24th International Symposium on Mathematical Foundations of Computer Science, MFCS'99. LNCS 1672, Berlin: Springer-Verlag, 1999. 70-80.

[Mos2001a] Mosses, P.D., What use is formal semantics. PSI Report, Perspective of System Informatics 2001, Russia, 2001. http://www.iis.nsk.su/psi01/reports/mosses_e.shtml.

[Mos2001b] Mosses, P.D., The varieties of programming language semantics and their uses. In: Bjørner, D., Broy, M. and

Zamulin, A.V, eds. 4th International Andrei Ershov Memorial Conference on Perspective of System Informatics, PSI'01, LNCS 2244. Berlin: Springer-Verlag, 2001. 165-190.

[Mos2001c] Mosses, P.D., Action semantics and compiler generation. Course Lecture, Department of Computer Science, University of Aarhus, 2001. http://wiki.daimi.au.dk:8000/ascg-01/.

[Mos2002a] Mosses, P.D., Fundamental concepts and formal semantics of programming language. Lecture Notes, BRICS & Department of Computer Science, University of Aarhus, 2002. http://wiki.daimi.au.dk:8000/dSprogSem-02/.

[Mos2002b] Mosses, P.D., Pragmatics of Modular SOS. In: Kirchner, H. and Ringeissen, C. eds. 9th International Conference on Algebraic Methodology and Software Technology, AMAST'02. LNCS 2422, Berlin: Springer-Verlag, 2002. 21-40.

[Mou1993] Moura, H.P., Action notation transformations [Ph.D. thesis]. Department of Computer Science, University of Glasgow, 1993.

[MTH1990] Milner, R., Tofte, M. and Harper, R., The definition of standard ML. USA: MIT Press, 1990.

[MTHM1997] Milner, R., Tofte, M., Harper, R. and MacQueen, D., The Definition of standard ML (revised). USA: MIT Press, 1997.

[MW1987] Mosses, P.D. and Watt, D.A., The use of action semantics. In: Formal Description of Programming Concepts III, Proceedings of IFIP TC2 Working Conference, North-Holland, 1987. 135-166.

[MW1993] Mosses, P.D. and Watt, D.A., Pascal action semantics. Technical Report 17.08, University of Aarhus, 1993. http://www.brics.dk/Projects/AS/action-semantics/93/.

[Nau2001] Naumann, D.A., Predicate transformer semantics of a higher-order imperative language with record subtyping. Science of Computer Programming, 2001, 41(1): 1-51.

[Nic1994] Nickau, H., Hereditarily sequential functionals. In: Proceedings of the Symposium on Logical Foundations of Computer Science, LFCS'94, Berlin: Springer, 1994. 240-252.

[NN1992] Nielson, H.R. and Nielson, F., Semantics with applications: a formal introduction. Wiley Professional Computing Series, Chichester, England: John Wiley & Sons, 1992.

[ON1999] von Oheimb, D. and Nipkow, T., Machine-cheking and Jave specification: proving type-safety. In: Alves-Foss, J. ed. Formal Syntax and Semantics of Java. LNCS 1523, Berlin: Springer-Verlag, 1999.

[Ørb1994] Ørbæk, P., OASIS: an optimizing action-based compiler generator. In: 5th International Conference on Compiler Construction, CC'94. LNCS 786, Berlin: Springer-Verlag, 1994. 1-15.

[Pal1992a] Palsberg, J., An automatically generated and provably correct compiler for a subset of ada. In: 4th IEEE International Conference on Computer Languages, ICCL'92, New York: IEEE Press, 1992. 117-126.

[Pal1992b] Palsberg, J., A provably correct compiler generator. In: Proceedings of European Symposium on Programming, ESOP'92. LNCS 582, Berlin: Springer-Verlag, 1992. 418-434.

[PDG1986] PL/I Definition Group. Formal definition of PL/I version 1, Report TR 25.071, American: Nat. Standards Institute, 1986.

[Plo1981] Plotkin, G.D., A structural approach to operational semantics. Technical Report, DAIMI FN–19, Department of Computer Science, University of Aarhus, 1981.

[Plo1983] Plotkin, G.D., An operational semantics for CSP. In: Bjørner, D. ed. Proceeding IFIP TC2 Working Conference on Formal Description of Programming Concepts- II, North-Holland, 1983. 199-225.

[Pol1981] Polak, W., Program verification based on denotation semantics. In: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language, Williamsburg, 1981. 149-158.

[Pow2000] Power, J., Modularity in denotational semantics. In: 13th Annual Conference on Mathematical Foundations of Programming Semantics, MFPS XIII (6), New York: Elsevier Press, 2000.

[PS1995] Palsberg, J. and Schwartzbach, M.I., Safety analysis versus type inference. Information and Computation, 1995, 118 (1): 128-141.

[Rep1991] Reppy, J., CML: a higher-order concurrent language. In: Programming Language Design and Implementation, ACM SIGPLAN, 1991. 293-259.

[Rin1997] Ringström, J., Compiler generation for data-parallel programming languages from two-level semantics specifications [Ph.D. Thesis]. Sweden: University of Linköping, 1997.

[Ru1992] Ru-zhan, L., Formal semantics of programming languages. Beijing: Science Press, 1992 (in Chinese).

[Sch1986] Schmidt, D.A., Denotational semantics: a methodology for language development. Boston, Massachusetts: Allyn and Bacon, 1986.

[Sch1995] Schellekens, M., The Smyth completion: a common foundation for denotational semantics and complexity analysis. In: Proceeding of Electronic Notes in Theoretical Computer Science, MFPS 11, 1995, 1: 211-232.

[SK1995] Slonneger, K. and Kurtz, B., Formal syntax and semantics of programming languages: a laboratory based approach. Iowa, USA: Addison & Wesley, 1995.

[Sou1984] Soundararajan, N., Axiomatic semantics of Communicating sequential processes. ACM Transaction on Programming Language and System, 1984, 6(4): 647-662.

[SSB2001] Stärk, R., Schmid, J., and Börger, E., Java and the Java virtual machine: definition, verification, validation. Berlin: Springer-Verlag, 2001.

[Ste1996] Stephenson, K., An algebraic approach to syntax, semantics and compilation [Ph.D. thesis]. Department of Computer Science, University of Wales Swansea, 1996.

[Ste1997] Stephenson, K., Compiler correctness using algebraic operational semantics. Technical Report CSR 1-97, University of Wales Swansea, 1997. http://www-compsci.swan.ac.uk/reports/yr1997/CSR1-97.pdf.

[Sto1977] Stoy, J., Denotational semantics: the Scott-Strachey approach to programming language theory. USA: MIT Press, 1977.

[Sym1999] Syme, D., Proving Java type soundness. In: Foss, A. ed. Formal Syntax and Semantics of Java. LNCS 1523, Berlin: Springer-Verlag, 1999. 83-118.

[Ten1976] Tennent, R.D., The denotational semantics of programming languages. Communications of the ACM, 1976, 19 (8): 437-453.

[Tof1993] Toft, J.U., Feasibility of using RSL as the specification language for the ANDF formal specification. Technical Report 202104-RPT-12, Denmark, 1993.

[TZ1988] Tucker, J.V. and Zucker, J.I., Program correctness over

abstract data types, with error-state semantics. New York: Elsevier Science Inc., 1988.

[UP2002] Ulidowski, L. and Phillips, L., Ordered SOS process languages for branching and eager bisimulations. Information and Computation, 2002, 178 (1): 180-213.

[Var1982] Vardi, M.Y., The complexity of relational query languages. In: 14th ACM Symposium on Theory of Computing, 1982. 137-146.

[vON1999] von Oheimb, D. and Nipkov, T., Machine-checking the Java specification: proving type-safety. In: Foss, A. ed. Formal Syntax and Semantics of Java. LNCS 1523, Berlin: Springer-Verlag, 1999. 83-118.

[Wad1990] Wadler, P.L., Comprehending monads. In: proceedings of the 1990 ACM Conference on Lisp and Functional Programming. New York: ACM Press, 1990. 61-78.

[Wan1997] Wansbrough, K., A modular monadic action semantics [MS Thesis]. Department of Computer Science, University of Auckland, 1997.

[Wat1986] Watt, D.A., Executable semantic descriptions. Software: Practice and Experience, 1986, 16: 13-43.

[Wat1988] Watt, D.A., An action semantics of Standard ML. In: Proceeding of 3rd Workshop on Mathematics Foundations of Programming Language Semantics. LNCS 298, Berlin: Springer-Verlag, 1988. 572-598.

[Wat1999] Watt, D.A., JAS: a Java action semantics. In: Mosses, P.D. and Watt, D.A. eds. 2nd International Workshop on Action Semantics, University of Aarhus, Denmark: BRICS NS, 1999,3: 43-56.

[WBB1993] Weber, S., Bloom, B. and Brown, G., Compiling Joy into silicon: an exercise in applied structural operational semantics. In: Bakker, J., Roever, W. and Rozenberg, G. eds. Proceedings REX Workshop on Semantics: Foundations and Applications. LNCS 666, Berlin: Springer-Verlag, 1993. 639-659.

[Weg1972] Wegner, P., The Vienna definition language. ACM Computer Survey, 1972, 4(1): 5-63.

[WF1994] Wright, A. and Felleisen, M., A syntactic approach to type soundness. Information and Computation, 1994, 115 (1): 38-94.

[WH1997] Wansbrough, K. and Hamer, J., A modular monadic action semantics. In: Proceeding of the Conference on Domain-Specific Language, Santa Barbara, California: The USENIX Association, 1997. 157-170.

[Wil1982] Williams, J.H., On the development of the algebra of funtcional programs. ACM Transactions on Programming Languages and Systems, 1982, 4(4):733-757.

[Yeu1997] Yeung, W.L., Denotational semantics for JSD. In: 4th Asia-Pacific Software Engineering and International Computer Science Conference, APSEC'97, Hong Kong, 1997.72-82.

[YR1998] Yan-bing, W. and Ru-zhan, L., Semantics of program base on trace part I: trace and semantics objects. Journal of Software, 1998, 9 (5): 366-370 (in Chinese).

[YZJ1995] Yuzhong, Q., Zhijian, W. and Jiafu, X., Denotational Semanitcs of a simple model Eiffel. Journal of Computer Science of Technology, 1995, 10(3): 214-226.

[Zam1997] Zamulin, A.V., Algebraic semantics of object-oriented data models. In: Technology of Object-oriented languages and System-Tools-24, Beijing, 1997. 43-53.

[ZG1997] Zimmermann, W. and Gaul, T., On the construction of correct compiler backends: an ASM approach. Journal of Universal Computer Science, 1997, 3(5): 504-567.