Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Electronic Notes in Theoretical Computer Science 148 (2006) 41–73
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Formal Semantics of Programming Languages

— An Overview —

Peter D. Mosses[1]

Department of Computer Science
University of Wales Swansea
Swansea, United Kingdom

Abstract

These notes give an overview of the main frameworks that have been developed for specifying the formal semantics of programming languages. Some of the pragmatic aspects of semantic descriptions are discussed, including modularity, and potential applicability to visual and modelling languages. References to the literature provide starting points for further study.

Keywords: semantics, operational semantics, denotational semantics, SOS, MSOS, reduction semantics, abstract state machines, monadic semantics, axiomatic semantics, action semantics, programming languages, modelling languages, visual languages

1 Introduction

▷ A *semantics* for a programming language models the computational meaning of each program.

The computational meaning of a program—i.e. what actually happens when a program is executed on some real computer—is far too complex to be described in its entirety. Instead, a semantics for a programming language provides abstract entities that represent just the relevant features of all possible executions, and ignores details that have no relevance to the correctness of implementations. Usually, the only features deemed to be relevant are the relationship between input and output, and whether the execution terminates

[1] Email: p.d.mosses@swansea.ac.uk

doi:10.1016/j.entcs.2005.12.012

or not. Details specific to implementations, such as the actual machine addresses where the values of variables are stored, the possibility of 'garbage collection', and the exact running time of the program, may all be ignored in the semantics.

▷ The form and structure of programs are determined by their syntax.

Before we can even start to give a semantics to a programming language, we need a precise definition of the form and structure of the programs allowed by the language. The syntax of a language determines not only whether a program is legal or not, but also its internal grouping structure.

▷ Descriptions are called *formal* when written in a notation that already has a precise meaning.

Reference manuals and standards for programming languages usually provide formal descriptions of program syntax, written in some variant of BNF. Regarding the computational meaning of the language, however, the description in the reference manual is generally completely informal, being expressed only in natural language which, even when used very pedantically, is inherently imprecise and open to misinterpretation.

1.1 Syntax

There are several kinds of syntax of programming languages: concrete, abstract, regular, context-free, and context-sensitive.

▷ *Concrete syntax* of programming languages involves text and parsing.

Concrete syntax determines which text strings are accepted as programs, and provides a parse tree for each accepted program, indicating how the text is supposed to be grouped. Concrete syntax is typically specified by formal grammars, with productions giving sets of alternatives for each nonterminal symbol. A grammar for concrete syntax should be unambiguous, so that each accepted program has a unique parse tree; an alternative approach is to let the grammar remain ambiguous, and provide precedence rules to select unique parse trees.

Table 1 shows a fragment of a grammar that might be used to specify a concrete syntax for ML. The grammar is written in a variant of BNF, using notation for regular expressions to describe numerals and identifiers. [2]

[2] Layout characters are implicitly allowed between symbols.

Table 1
Concrete Syntax

```
exp    --> "if" exp "then" exp "else" exp | ... | infexp
infexp --> atexp ":=" atexp | ... | atexp
atexp  --> const | id | ...
const  --> bool | int | "()"
bool   --> "true" | "false"
int    --> [0-9]+
id     --> [A-Za-z][A-Za-z0-9]*
```

▷ Concrete syntax of modelling languages usually involves a mixture of text and graphical notation.

Whereas programs are normally written as text, models of software are formulated with extensive use of diagrams involving boxes, lines, and arrows [30]. Such diagrams are created interactively, in a non-serial manner. Each diagram also has a purely textual serial representation, but this is used primarily as an exchange format between tools, and not for creating diagrams. The concrete syntax of diagrams could be specified as a set of rules resembling a textual grammar, although details concerning attachment points and 2-dimensional layout have to be made explicit, which complicates matters considerably.

▷ Modularity and reuse can be useful in connection with descriptions of concrete syntax.

The Syntax Definition Formalism (SDF) [9] allows a description of concrete syntax to be divided into modules with explicit dependence. One may expect that modules specifying various low-level constructs (such as booleans, integers, and identifiers) can often be reused in the concrete syntax of different languages. However, modules specifying particular collections of constructs (e.g., expressions in ML) can in general only be reused without change when describing an extension of a language.

▷ *Abstract syntax* deals only with structure.

Whereas concrete syntax deals with the actual character strings used to write programs, abstract syntax is concerned only with the 'deep structure' of programs, which is generally represented by trees. In abstract syntax trees, each node is created by a particular constructor, and has a branch for each argument of its constructor. Apart from constructors with no arguments, the leaves of the trees may include sequences of characters corresponding to lexical symbols such as numerals and identifiers; we may also consider so-called *value-added* abstract syntax trees where abstract mathematical entities (truth-

Table 2
Abstract Syntax

$$\begin{aligned} \text{Exp} &::= \text{cond}(\text{Exp}, \text{Exp}, \text{Exp}) \mid \text{assign}(\text{Exp}, \text{Exp}) \mid \text{Const} \mid \text{Id} \mid \ldots \\ \text{Const} &::= \text{Bool} \mid \text{Int} \mid \text{null} \\ \text{Id} &::= \text{String} \end{aligned}$$

values, numbers, or even operations such as addition) are allowed as leaves.

▷ Abstract syntax can be specified in various ways.

The essential aspects of abstract syntax are the *constructors* for nodes, and *classification* of nodes into (possibly overlapping) sets. Exactly how these are specified is not so important (at least in connection with the use of abstract syntax in semantic descriptions). For example:

- as datatype definitions (as found in ML), possibly allowing also subtype definitions (as in CASL [27]);
- using object-oriented techniques and notation (as in UML [30]); or
- by simply indicating the argument and result types of each constructor.

The specification given in Table 2 uses a grammar-like notation similar to that provided in CASL for datatype definitions. The specified abstract syntax represents the deep structure of the same expressions that were used to illustrate concrete syntax in Table 1.

▷ Abstract syntax can be chosen freely.

Various choices have to be made when specifying abstract syntax, e.g.: which symbols to use as names for types and constructors; when to group similar constructs into distinguished types; and whether to leave lexical constructs represented by sequences of characters, or replace them by the corresponding mathematical entities. Usually, these choices don't significantly affect the process of describing the semantics of a particular language.

However, making the abstract syntax as *language-independent* as possible facilitates reuse of parts of the semantic description of one language in that of another. For instance, the symbols for types and constructors may be chosen to reflect the normal vocabulary used in informal discussions of programming constructs, such as expressions (abbr. 'Exp') and conditionals (abbr. 'cond'). The latter symbol also avoids suggesting any particular concrete syntax for conditional expressions: had we chosen, say, 'if_then_else' instead of 'cond', the abstract syntax would look quite out of place in a description of a language such as Java, where the concrete syntax is '_?_:_'.

Table 3
Mapping concrete syntax to abstract syntax in Prolog

```
exp(cond(E1,E2,E3)) -->
     "if", exp(E1), "then", exp(E2), "else", exp(E3).

exp(E) --> infexp(E).

infexp(assign(E1,E2)) --> atexp(E1), ":=", atexp(E2).

...
```

In Table 2, the standard mathematical sets Bool (consisting of the boolean truth-values true and false, equipped with negation, conjunction, disjunction, etc.), Int (the integers) and String are assumed to be given, together with conventional notation for their operations and relations.

▷ Abstract syntax is usually much simpler than concrete syntax.

Notice how the distinction between the concrete nonterminal symbols for expressions `exp`, infix expressions `infexp`, and atomic expressions `atexp`, has been eliminated in the illustrated abstract syntax. This entails that the structure of the abstract syntax trees is somewhat more general than that of the parse trees for the corresponding concrete syntax. Such generalization allows abstract syntax to be significantly simpler than concrete syntax, and makes it particularly suitable for use as an interface between syntax and semantics.

▷ *Complete descriptions* specify both concrete and abstract syntax.

Complete language descriptions should provide both concrete and abstract syntax, and specify the intended mapping from program texts to abstract syntax trees. When using Definite Clause Grammars (as provided by Prolog) to specify concrete syntax, the construction of the abstract syntax trees can be specified in the grammar itself, as illustrated in Table 3.

▷ Abstract syntax trees may be constructed directly.

It is possible (but relatively tedious) for a programmer to construct the abstract syntax tree of a program interactively, in the same way as concrete diagrams are constructed in modelling languages. But even then, a textual concrete syntax is still required, for displaying the program. The transformation from an abstract syntax tree to a textual form is called pretty-printing. Formal description of pretty-printing is complicated by the pragmatic issue of where to insert line breaks and indentation.

▷ Abstract syntax *graphs* may be more appropriate when the concrete syntax is visual.

Abstract syntax of programming languages is firmly based on trees. This is partly because abstract syntax is derived from parse trees of programs. However, the declaration and use of identifiers does give rise to an implicit graph structure, which could be made explicit, if desired.

For languages where the concrete syntax is largely based on diagrams, such as visual modelling languages, abstract syntax graphs appear to be more appropriate than trees [3]. This is especially the case when a diagram involves un-named components: transformation to a tree may then require the creation of unique labels for these components (e.g., when several arrows lead to an anonymous junction point [8]).

Note that trees have a considerable theoretical advantage over graphs, in that (finite) trees admit inductive definitions. This is a crucial feature for denotational semantics, and significant for proving properties of operational semantics by structural induction.

▷ *Context-free syntax* deals with grouping.

The rules for grouping in programming languages are usually fixed, so that grouping analysis is context-free. However, some languages (including ML) allow the precedence of infix and prefix operators to be specified in programs, and the grouping of expressions in such languages is clearly context-sensitive. Since abstract syntax trees are constructed only after the grouping has been determined, abstract syntax is unaffected by issues of grouping analysis.

▷ *Context-sensitive syntax* can also deal with constraints.

Well-formedness conditions such as declaration-before-use and type-checking constraints are *inherently* context-sensitive, and cannot be specified by context-free grammars. The abstract syntax of programs satisfying these well-formedness conditions may be regarded as a subset of the abstract syntax where the conditions are ignored.

1.2 Semantics

There are several levels of semantics: static semantics, dynamic semantics, and equivalences.

▷ *Static semantics* models compile-time checks.

When (abstract) syntax is restricted to be context-free, checking whether programs satisfy well-formedness constraints necessarily becomes part of seman-

tics. It is called static semantics, since it concerns only those checks that can be performed before running the program, e.g. checking that all parts of the program are type-correct. The only relevant feature of the static semantics of a program is whether the program has passed the checks or not (although error reports issued by compilers could be modelled when these are implementation-independent).

▷ *Dynamic semantics* models run-time behaviour.

Dynamic semantics concerns the observable behaviour when programs are run. Here, we may assume that well-formedness of the programs has already been checked by the static semantics: we do not need to consider the dynamic semantics of ill-formed programs.

▷ *Equivalences* between programs may abstract from details of models.

A formal semantics should give, for each program, an abstract model that represents just the relevant features of all possible executions of that program. Then two programs are regarded as semantically equivalent when their models are the same (up to isomorphism). An alternative approach is to give less abstract models, and then define a semantic equivalence relation for each model.

▷ Complete descriptions include static semantics, dynamic semantics, and semantic equivalence.

Given a program accepted by a context-free concrete syntax, a static semantics is needed in order to determine whether the program is well-formed, and thus executable. The dynamic semantics then provides a model of program executions. The semantic equivalence relation abstracts from those features that are irrelevant to implementation correctness. All together this provides the complete semantics of the given program.

▷ These notes focus on dynamic semantics, based on context-free abstract syntax.

There are several main approaches to dynamic semantics:

- *operational semantics*, where computations are modelled explicitly;
- *denotational semantics*, where only the contribution of each construct to the computational meaning of the enclosing program is modelled; and
- *axiomatic semantics*, which (in effect) models the relationship between pre- and post-conditions on program variables.

The operational framework known as *Structural Operational Semantics (SOS)* [31] is a good compromise between simplicity and practical applicability, and it has been widely taught at the undergraduate level [10,29,37,41]. A modular variant of SOS called MSOS [26] has some significant pragmatic advantages over the original SOS framework, but otherwise remains conceptually very close to it. The hybrid framework called Action Semantics [21,28,40]—not to be confused with the UML Action Semantics—combines features of denotational and operational semantics.

Let us now outline the main semantic frameworks. The aim is, for each framework, to explain its main principles, to give an impression of how semantic descriptions look in it, and to draw attention to any major drawbacks that it might have. References to the literature provide starting points for further study.

2 Structural Operational Semantics

The Structural Operational Semantics (SOS) framework was proposed by Plotkin in 1981 [31]. The main aim was to provide a simple and direct approach, allowing concise and comprehensible semantic descriptions based on elementary mathematics. The basic SOS framework has since been presented in various textbooks (e.g. [29,41]), and exploited in numerous papers on concurrency [16]; see also [1]. The big-step form of SOS (also known as Natural Semantics [15]) was used during the design of Standard ML, as well as to give the official definition of that language [17].

▷ SOS uses *rules* to specify transition relations.

SOS uses rules to give inductive specifications of transition relations on states that involve both abstract syntax trees and computed values. When describing a purely functional programming language (or a pure process calculus such as CCS [16]), SOS rules look very simple. For instance: [3]

$$\frac{E_1 \longrightarrow E_1'}{\mathrm{cond}(E_1, E_2, E_3) \longrightarrow \mathrm{cond}(E_1', E_2, E_3)} \tag{1}$$

$$\mathrm{cond}(\mathrm{true}, E_2, E_3) \longrightarrow E_2 \tag{2}$$

[3] When illustrating the various semantic frameworks, we shall use a language-independent notation for abstract syntax. In general, specifications in the literature use notation that is strongly suggestive of the concrete syntax of the particular language whose semantics is being described. The difference in style has no technical significance.

$$\mathrm{cond}(\mathrm{false}, E_2, E_3) \longrightarrow E_3 \tag{3}$$

Notice that there are no labels on the transitions in the above SOS rules: labels are normally used in SOS only in connection with communication and synchronization between concurrent processes, and don't occur at all with transitions for sequential programming constructs. In the next section, we shall see that labels are more widely exploited in the modular variant of the SOS framework, MSOS.

▷ States are not restricted to syntax trees.

In the original SOS framework, syntax is not clearly separated from auxiliary semantic entities: both syntactic and semantic entities are allowed as components of states, as illustrated in connection with bindings and stores below. In contrast, MSOS insists that states remain purely syntactic, as we shall see in Sect. 3.

2.1 Bindings

Declarations (and some other constructs) *bind* identifiers to particular values. A *bindings map* or *environment* gives the current association between identifiers and their bound values, and generally has a restricted *scope*.

▷ Bindings are usually represented by explicit components of states.

In fact the treatment of bindings in SOS is somewhat awkward. Suppose that the states for expression evaluation include bindings: State = Exp × Env. This requires the specification of transitions $(E, \rho) \longrightarrow (E', \rho)$ where the environment ρ remains *unchanged*. Clearly, it would be tedious to have to write (and read) ρ twice each time a transition is specified, and it is usual practice to introduce the notation $\rho \vdash E \longrightarrow E'$ as an abbreviation for $(E, \rho) \longrightarrow (E', \rho)$. Thus when the functional language being described involves bindings, the SOS rules given above would be reformulated as follows:

$$\frac{\rho \vdash E_1 \longrightarrow E_1'}{\rho \vdash \mathrm{cond}(E_1, E_2, E_3) \longrightarrow \mathrm{cond}(E_1', E_2, E_3)} \tag{4}$$

$$\rho \vdash \mathrm{cond}(\mathrm{true}, E_2, E_3) \longrightarrow E_2 \tag{5}$$

$$\rho \vdash \mathrm{cond}(\mathrm{false}, E_2, E_3) \longrightarrow E_3 \tag{6}$$

Alternatively, bindings can be eliminated as soon as they have been computed by *substituting* the bound values for the identifiers throughout the scope of the bindings. However, an explicit definition of the result $[\rho]T$ of substitution of ρ throughout T requires the tedious specification of a defining equation for each non-binding construct T, for instance:

$$[\rho]\mathrm{cond}(E_1, E_2, E_3) = \mathrm{cond}([\rho]E_1, [\rho]E_2, [\rho]E_3) \tag{7}$$

as well as some rather more intricate equations for the binding constructs.

2.2 Stores

Assignments involve (irreversible) changes to particular locations in a store. Variable identifiers are generally bound to locations, and assignments affects the values stored at locations, but not the current bindings.

▷ The separate modelling of binding and assignment allows a simple treatment of aliasing.

It is best not to confuse binding with assignment. Abstractly, a variable declaration has the effect of allocating part of the store to hold the value of the variable, and binds the identifier to some entity, traditionally called a *location*, that refers to that part of the store; it may also initialize the value of the variable. Assignment of a value to the variable affects only the store, not the binding of the variable identifier. The usefulness of this distinction can be seen most clearly in languages that allow so-called *aliasing*, where variable identifiers are bound to the same location: assigning a new value to one of them causes the value of the other(s) to change as well.

▷ Effects on storage are represented by explicit store components of states.

When the described language isn't purely functional, and expression evaluation can have side-effects, the states for expression evaluation include the current store as well as the environment: State = Exp × Env × Store. Transitions between such states are written $\rho \vdash E, \sigma \longrightarrow E', \sigma'$, so the rules given above would be reformulated as follows:

$$\frac{\rho \vdash E_1, \sigma \longrightarrow E_1', \sigma'}{\rho \vdash \mathrm{cond}(E_1, E_2, E_3), \sigma \longrightarrow \mathrm{cond}(E_1', E_2, E_3), \sigma'} \tag{8}$$

$$\rho \vdash \mathrm{cond}(\mathrm{true}, E_2, E_3), \sigma \longrightarrow E_2, \sigma \tag{9}$$

$$\rho \vdash \text{cond}(\text{false}, E_2, E_3), \sigma \longrightarrow E_3, \sigma \tag{10}$$

$$\frac{\rho \vdash E_1, \sigma \longrightarrow E_1', \sigma'}{\rho \vdash \text{assign}(E_1, E_2), \sigma \longrightarrow \text{assign}(E_1', E_2), \sigma'} \tag{11}$$

$$\frac{\rho \vdash E_2, \sigma \longrightarrow E_2', \sigma'}{\rho \vdash \text{assign}(E_1, E_2), \sigma \longrightarrow \text{assign}(E_1, E_2'), \sigma'} \tag{12}$$

$$\rho \vdash \text{assign}(L, V), \sigma \longrightarrow (\,), \sigma[L \mapsto V] \tag{13}$$

2.3 Communications

▷ Communication between concurrent processes is represented by labels on transitions.

Finally, suppose that expression evaluation can involve process creation and communication. The conventional technique in SOS is here to add labels to transitions. The SOS rules given above would be reformulated thus:

$$\frac{\rho \vdash E_1, \sigma \xrightarrow{L} E_1', \sigma'}{\rho \vdash \text{cond}(E_1, E_2, E_3), \sigma \xrightarrow{L} \text{cond}(E_1', E_2, E_3), \sigma'} \tag{14}$$

$$\rho \vdash \text{cond}(\text{true}, E_2, E_3), \sigma \xrightarrow{\tau} E_2, \sigma \tag{15}$$

$$\rho \vdash \text{cond}(\text{false}, E_2, E_3), \sigma \xrightarrow{\tau} E_3, \sigma \tag{16}$$

(τ is some fixed label that indicates a silent, uncommunicative step.)

▷ Rules require reformulation when components of states or labels on transitions are added, changed, or removed.

As illustrated above, the formulation of rules in conventional SOS has to change whenever the components of the model involved in transitions (i.e. states and labels) are changed. This is in marked contrast to the situation with MSOS, where the formulation of transitions in rules is stable, allowing the rules for each programming construct to be given definitively, once-and-for-all, as we shall see in Sect. 3.

2.4 Small-Step and Big-Step Styles

▷ In conventional SOS, the small-step and big-step styles are commonly regarded as alternatives.

Formally, the big-step style can be regarded as a special case of the small-step style: computations in the big-step style simply don't involve any intermediate states, only initial and final states. Note also that if one has defined a small-step SOS, the transitive closure of the small-step relation provides the corresponding big-step relation. In practice, authors of SOS descriptions usually choose one style or the other—and then stick to it, since changing styles involves major reformulation. In general, however, it seems better to mix the small-step and big-step styles, choosing the more appropriate style for each kind of construct by consideration of the nature of its computations:

Big-step SOS is better for constructs whose computations are *pure evaluation*, with no side-effects, no exceptions, and always terminating—e.g., for evaluating decimal numerals to numbers, for matching patterns against values, and for types;

Small-step SOS is better for all other constructs, since it makes explicit the order in which the steps of their computations are made, which is usually significant. Moreover, small-step SOS copes more easily with specifying interleaving, exception handling, and concurrency than big-step SOS does.

▷ Big-step SOS can be applicable to modelling languages.

Sometimes, modelling languages are used to specify declarative aspects of software, such as relationships between classes of objects. The focus is on the static structure of the model, not on any behavioural interpretation. Although small-step SOS is inappropriate for specifying static structure, big-step SOS can be used here to give a formal description of the intended semantics, provided that the abstract syntax is tree-structured. For instance, the big-step transition relation between a construct and its computed value may represent that the value *satisfies* the construct; an object may be regarded as satisfying any class to which it belongs. The satisfaction relationship between algebraic specifications and algebras in CASL is defined using big-step SOS [27, Part III].

▷ Small-step SOS can be applicable to behavioural semantics of modelling languages.

For example, the behavioural semantics of the visual modelling language StateFlow has been specified by transforming diagrams to abstract syntax trees, and then defining their small-step SOS in a conventional style [8].

2.5 Informal Conventions

▷ The official Definition of Standard ML is not entirely formal.

A major example of an SOS in the pure big-step style is the Definition of Standard ML [17]. The description covers the static and dynamic semantics of the entire language (both the core and module levels), and has been carefully written by a group of highly qualified authors. Nevertheless, its degree of formality still leaves something to be desired—especially in connection with two "conventions" that were adopted:

The "store convention" allows the store to be left implicit in rules where it is not being extended, updated, or inspected.

The "exception convention" allows the omission of rules that merely let unhandled exceptions preempt further sub-expression evaluation.

For instance, consider the following rule for the evaluation of conditional expressions:

$$\frac{\rho \vdash E_1 \longrightarrow \text{true} \qquad \rho \vdash E_2 \longrightarrow V}{\rho \vdash \text{cond}(E_1, E_2, E_3) \longrightarrow V} \tag{17}$$

By the above conventions, this rule abbreviates the following three rules:

$$\frac{\rho \vdash E_1, \sigma \longrightarrow \text{true}, \sigma' \qquad \rho \vdash E_2, \sigma' \longrightarrow V, \sigma''}{\rho \vdash \text{cond}(E_1, E_2, E_3), \sigma \longrightarrow V, \sigma''} \tag{18}$$

$$\frac{\rho \vdash E_1, \sigma \longrightarrow \text{raised}(EX), \sigma'}{\rho \vdash \text{cond}(E_1, E_2, E_3), \sigma \longrightarrow \text{raised}(EX), \sigma'} \tag{19}$$

$$\frac{\rho \vdash E_1, \sigma \longrightarrow \text{true}, \sigma' \qquad \rho \vdash E_2, \sigma' \longrightarrow \text{raised}(EX), \sigma''}{\rho \vdash \text{cond}(E_1, E_2, E_3), \sigma \longrightarrow \text{raised}(EX), \sigma''} \tag{20}$$

where raised(EX) indicates that the evaluation of a sub-expression has raised an exception with value EX. Such conventions are completely unnecessary when using the MSOS approach described in the next section.

3 Modular SOS

▷ Modular SOS allows individual constructs to be described once and for all.

As the name suggests, Modular SOS (MSOS) [22,23,25,26] is a variant of SOS that ensures a high degree of modularity: the rules specifying the MSOS of individual language constructs can be given once and for all, since their formulation is completely independent of the presence or absence of other constructs in the described language. When extending a pure functional language with concurrency primitives and/or references, the MSOS rules for the functional constructs don't need even the slightest reformulation.

In denotational semantics, the problem of obtaining good modularity has received much attention, and has to a large extent been solved by introducing so-called monad transformers [18]. MSOS provides an analogous (but significantly simpler) solution for the structural approach to operational semantics.

▷ States are purely syntactic in MSOS, and labels are exploited more than in SOS.

The crucial feature of MSOS is to insist that states are merely abstract syntax and computed values, omitting the usual auxiliary information (such as environment and stores) that they include in SOS. The only place left for auxiliary information is in the labels on transitions. This seemingly minor notational change—coupled with the use of symbolic indices to access the auxiliary information—is surprisingly beneficial. MSOS rules for many language constructs can be specified independently of whatever components labels might have; rules that require particular components can access and set those components without mentioning other components at all.

▷ Rules for constructs for control flow are particularly simple.

For instance, the MSOS rules for conditional expressions do not require labels to have any particular components, and their formulation remains valid regardless of whether expressions are purely functional, have side-effects, raise exceptions, or interact with concurrent processes:

$$\frac{E_1 \xrightarrow{X} E_1'}{\mathrm{cond}(E_1, E_2, E_3) \xrightarrow{X} \mathrm{cond}(E_1', E_2, E_3)} \tag{21}$$

$$\mathrm{cond}(\mathrm{true}, E_2, E_3) \longrightarrow E_2 \tag{22}$$

$$\mathrm{cond}(\mathrm{false}, E_2, E_3) \longrightarrow E_3 \tag{23}$$

The label X in the first rule above could include the current environment, the initial and final stores, and any emitted communication signals. When labels are omitted, as in the second and third rule, the transitions are required to be *unobservable*, with no change to the store, and no emitted communication signals. Labels on adjacent transitions in a computation are required to be *composable*: bindings must remain fixed, and the final store of the label on a transition must be the same as the initial store of the label on the following transition. [4]

▷ Rules involving auxiliary information in labels refer only to the required components.

$$\frac{E_1 \xrightarrow{X} E_1'}{\mathrm{assign}(E_1, E_2) \xrightarrow{X} \mathrm{assign}(E_1', E_2)} \tag{24}$$

$$\frac{E_2 \xrightarrow{X} E_2'}{\mathrm{assign}(E_1, E_2) \xrightarrow{X} \mathrm{assign}(E_1', E_2')} \tag{25}$$

$$\frac{\sigma' = \sigma[L \mapsto V], \quad U \in \mathrm{Unobs}}{\mathrm{assign}(L, V) \xrightarrow{\{\sigma, \sigma', U\}} E_2} \tag{26}$$

In the first two rules above, the label X is arbitrary. In the last rule, however, the relationship between the store σ at the start of the transition and the store σ' at the end of the transition is determined, and any other components are required to be unobservable.

4 Reduction Semantics

This framework was developed by Felleisen and his colleagues towards the end of the 1980's [5]. It has been used primarily in theoretical studies, where it is sometimes preferred to SOS; for instance, Reppy used Reduction Semantics to define (parts of) Concurrent ML [33].

▷ States are abstract syntax trees, corresponding to well-formed terms.

States don't involve abstract mathematical values (numbers, sets, maps, etc.) at all: they are purely syntactic. For example, numerical expressions compute

[4] In fact the labels form a *category*.

decimal numerals rather than abstract mathematical numbers. If needed, however, auxiliary constructs may be added to the abstract syntax (as in SOS).

▷ Transitions are term rewriting steps, called reductions.

Term Rewriting is an interesting and well-developed topic in its own right [4]. A rewriting step is called a *reduction* (regardless of whether the resulting term is actually smaller than the previous one or not). The sub-term that gets rewritten in a reduction is called a *redex*, and the resulting sub-term is called a *reduct*. Reductions may normally be made in any sub-term, and continue until no more are possible—perhaps never terminating.

▷ Redexes are restricted to occurrences in *evaluation contexts*.

When the term being reduced corresponds to the abstract syntax of a program, the location of the redexes of the reductions should be restricted to follow the flow of control of the computation (otherwise reductions could be prematurely made in parts of the program that were not even supposed to be executed, leading to unintended results).

An evaluation context $C : \mathrm{Ctx}$ is a term with a single *hole*. If t is a term, $C[t]$ is the result of replacing the hole in C by t. A *reduction in a context* C is written $C[t] \longrightarrow C[t']$ (written with a longer arrow), and can be made whenever there is an ordinary reduction $t \to t'$ (written with a shorter arrow). In fact $C[t]$ here is generally the entire program context of t, assuming that there are no further rules that would allow a reduction in a context to be regarded as an ordinary reduction.

Rules may be given also for rewriting the context as well as the term in that context: $C[t] \longrightarrow C'[t']$; in this case it is not required that $t \to t'$ should be an ordinary reduction.

The evaluation contexts for use in a reduction semantics are specified by a context-free grammar.

▷ Simple SOS rules correspond to reductions.

Comparing SOS with Reduction Semantics, the simple rules of an SOS generally correspond directly to rules for ordinary reductions. For example, consider the following reduction rules for continuing with the evaluation of a conditional expression after its condition has been evaluated:

$$\mathrm{cond}(\mathrm{true}, E_2, E_3) \to E_2 \tag{27}$$

$$\mathrm{cond}(\mathrm{false}, E_2, E_3) \to E_3 \tag{28}$$

▷ Conditional SOS rules correspond to productions for evaluation contexts.

Many conditional SOS rules simply express the flow of control of the computation, such as indicating which sub-expression is to be evaluated first. These SOS rules correspond not to reduction rules, but rather to productions in the grammar for evaluation contexts. For example, the rules for evaluating the condition of a conditional expression and the two sides of an assignment expression correspond to the following productions for contexts:

$$\mathrm{Ctx} ::= \mathrm{cond}(\mathrm{Ctx}, \mathrm{Exp}, \mathrm{Exp}) \mid \mathrm{assign}(\mathrm{Ctx}, \mathrm{Exp}) \mid \mathrm{assign}(\mathrm{Exp}, \mathrm{Ctx})$$

The absence of further contexts for conditional expressions prevents the premature reduction of the branches. The order of evaluation of the subexpressions in an assignment expression is left open above, allowing interleaving; sequential evaluation would be specified by using $\mathrm{assign}(\mathrm{Val}, \mathrm{Ctx})$ instead of $\mathrm{assign}(\mathrm{Exp}, \mathrm{Ctx})$.

▷ Reductions that replace the evaluation context do not correspond directly to SOS rules.

A significant advantage of Reduction Semantics is that it is straightforward to specify rules that affect the entire context of the sub-expression being evaluated. For example, the following rule specifies clearly that when 'exit' is evaluated, the remaining evaluation of the entire program is terminated.

$$C[\mathrm{exit}] \longrightarrow \mathrm{null} \tag{29}$$

Exceptions can be specified in a similar way, although to restrict exception handling to the innermost matching handler requires the introduction of many new evaluation contexts.

▷ Computed values are simply canonical terms in normal form.

The computed values in a Reduction Semantics for a language like ML would include not only numerals and booleans, but also tuples, lists, and records with values as components. The syntax of values is specified by a context-free grammar—for example, by taking some of the productions for expressions Exp from the grammar for the full abstract syntax, and replacing Exp by Val (except within abstractions).

▷ Bindings are represented by substitution, which is itself tedious to specify.

Substitution replaces identifiers by the values to which they are bound, and can be specified by reduction rules (or defined equationally). In Reduction Semantics, there is unfortunately no alternative to the use of substitution to deal with the bindings that arise in the semantics of local declarations.

▷ Effects on storage are represented by rewriting a store term.

A term representing a store is a sequence of *canonical* assignments, i.e. assignments where the location and the value have already been evaluated. There is only one level of store—in contrast to the situation with local bindings—so it can be kept as a separate component of the entire program context:

$$\begin{aligned} \text{ProgCtx} &::= \text{prog-ctx}(\text{Ctx}, \text{Store}) \\ \text{Store} &::= \text{skip} \mid \text{seq}(\text{Store}, \text{update}(\text{Loc}, \text{Val})) \end{aligned}$$

When the left- and right-hand sides of an assignment expression (or statement) have been evaluated, the effect of the assignment is simply added to the store, by giving a reduction that replaces the entire context:

$$\begin{aligned} &\text{prog-ctx}(C[\text{assign}(L, V)], \sigma) \longrightarrow \\ &\qquad \text{prog-ctx}(C[\text{null}], \text{seq}(\sigma, \text{update}(L, V))) \end{aligned} \tag{30}$$

Inspecting the value stored at a particular location also involves the context, but does not change it:

$$\frac{\sigma = \text{seq}(\sigma', \text{update}(L, V))}{\text{prog-ctx}(C[\text{stored}(L)], \sigma) \longrightarrow \text{prog-ctx}(C[V], \sigma)} \tag{31}$$

$$\frac{\begin{array}{ll} \sigma = \text{seq}(\sigma', \text{update}(L', V)), & L \neq L', \\ \text{prog-ctx}(C[\text{stored}(L)], \sigma') \longrightarrow \text{prog-ctx}(C[V], \sigma') & \end{array}}{\text{prog-ctx}(C[\text{stored}(L)], \sigma) \longrightarrow \text{prog-ctx}(C[V], \sigma)} \tag{32}$$

▷ Reduction rules for communication involve separate evaluation contexts for the concurrent processes involved.

For example, suppose that a system of concurrent processes is represented as a map from thread identifiers to states of threads, then synchronous commu-

nication can be specified thus:

$$\{I_1{=}C_1[\text{send}(K,V)]\} + \{I_2{=}C_2[\text{receive}(K)]\} + TM \longrightarrow \{I_1{=}C_1[\text{null}]\} + \{I_2{=}C_2[V]\} + TM \tag{33}$$

5 Abstract State Machine Semantics

Abstract State Machines (ASM) is an operational semantics framework that was proposed by Gurevich in the late 1980's [6,7]. The main aim was to specify the individual steps of computations at the proper level of abstraction; issues such as control flow and scopes of bindings were regarded as of secondary importance. The framework has been applied to several major languages, including ML and Java. However, the details and general style of ASM specifications vary considerably between different publications; here, we shall follow [38], which appears to be competitive with SOS in its accessibility.

▷ States interpret static and dynamic function symbols.

The interpretation of a function symbol is a map from arguments to results. The function is called *static* when the map doesn't change during a computation. In contrast, the values of *dynamic* functions on particular arguments can be initialized, changed, or made undefined. Static functions of no arguments correspond to ordinary constants, whereas dynamic functions of no arguments correspond to simple updatable variables.

For example, functions corresponding to arithmetic operations are static, and so is the no-argument function *body* that gives the abstract syntax of the initial program. In contrast, the dynamic no-argument function *pos* gives the position of the phrase currently being executed in the tree representing what remains to be executed, which is itself represented by the 1-argument dynamic function $restbody : \text{Pos} \to \text{Phrase}$, where the set Phrase contains not only all possible abstract syntax trees, but also computed values, and trees where some nodes have been replaced by their computed values. [5]

▷ Transitions assign values to functions for particular arguments.

A transition may simultaneously assign values for several functions on various arguments. Each assignment may be conditional, depending on the values

[5] The idea of gradually replacing phrases by their computed values, familiar from SOS, has only recently been adopted in the ASM framework: in earlier ASM specifications, the original tree was static, and a separate dynamic function was used to associate computed values with the nodes of the tree.

Table 4
ASM semantics of conditional expressions

$$
\begin{array}{l}
execJavaExp_I = \textbf{case}\ context(pos)\ \textbf{of} \\
\quad \ldots \\
\quad \mathrm{cond}({}^{\alpha}E_1, {}^{\beta}E_1, {}^{\gamma}E_1) \quad \to pos := \alpha \\
\quad \mathrm{cond}({}^{\blacktriangleright}V_1, {}^{\beta}E_2, {}^{\gamma}E_3) \quad \to \textbf{if}\ V_1\ \textbf{then}\ pos := \beta\ \textbf{else}\ pos := \gamma \\
\quad \mathrm{cond}({}^{\alpha}\mathrm{true}, {}^{\blacktriangleright}V_2, {}^{\gamma}E_3) \to yieldUp(V_2) \\
\quad \mathrm{cond}({}^{\alpha}\mathrm{false}, {}^{\beta}E_2, {}^{\blacktriangleright}V_3) \to yieldUp(V_3) \\
\quad \ldots
\end{array}
$$

of terms formed from the function symbols. All the terms involved in a simultaneous set of assignments are evaluated before any of the assignments are actually made, so the testing of the conditions and the resulting state are independent of the order of the assignments. The values of functions on particular arguments only change due to explicit assignment: their values on other arguments remain stable.

▷ ASM specifications often introduce auxiliary notation.

The introduction of appropriate auxiliary notation allows transition rules to be specified rather concisely. However, ASM specifications of different languages tend to introduce different auxiliary notation, which leads to quite varied specifications of the same construct, and makes it difficult to reuse a transition rule from one ASM directly in another ASM. For example, the auxiliary notation introduced in the ASM specification of Java [38] includes:

- *context*(*pos*), returning either *restbody*(*pos*) or *restbody*(*up*(*pos*)); and
- *yieldUp*(*V*), abbreviating the transition *restbody* := *restbody*[*V*/*up*(*pos*)] performed simultaneously with *pos* := *up*(*pos*), thus combining the replacement of a phrase by its computed result *V* with the adjustment of *pos*.

To "streamline the notation", positions are indicated directly in patterns for phrases, the current value of *pos* being written ▶. After these preliminaries, transition rules for evaluating Java's conditional expressions can be specified as shown in Table 4.

Note that transitions are specified by assignments written with ':=', and the '→'s above are merely part of the '**case**' notation for pattern-matching (which is not itself provided by the ASM framework, but introduced *ad hoc*) in [38].

▷ Bindings are modelled by stacks of frames in ASM.

The dynamic no-argument function *locals* : (Id, Val)Map gives maps from local variable identifiers directly to the values that they are storing. To cope with redeclaration of local variables and with recursive procedural activation (both of which may require different values for the same variable identifier to coexist), a stack of frames is maintained, each frame storing the relevant *locals*. Thus the transition for an assignment expression assign(I, ▶V), where the new value V has already been computed, can be specified by $locals := locals[I \mapsto V]$, without overwriting the value of other active local variables having the same identifier I (the notation used for maps in [38] is actually slightly different).

It might seem more natural to treat *locals* as a unary dynamic function from variable identifiers to values, but the ASM framework is first-order, and doesn't allow functions themselves to be used as values.

▷ Exceptions are modelled by propagation.

In the Definition of Standard ML, an informal "exception convention" was introduced, so that a lot of tedious transition rules could be left implicit. In the ASM specification of Java, the propagation of raised exceptions is specified explicitly by introducing a predicate *propagatesAbr* on phrases, then using a special pattern *phrase*(▶A) which matches arbitrary phrases that have a raised exception A as any immediate component.

▷ Multiple threads can be modelled in various ways.

Separate ASM agents can be set up to execute threads independently, with access to the same storage. Synchronization between threads can be achieved using dynamic functions which indicate that particular threads have locked particular storage areas (since a lock can be both tested and conditionally set in a single transition).

In the cited ASM for Java, however, a different approach is used—motivated mainly by the requirement that the model should be executable using a recently-developed prototyping system called AsmGofer [34]. The idea is that at every transition during a computation for a multi-threaded Java program, an active thread is selected arbitrarily, using a so-called choice function, which is itself left unspecified. In contrast to the situation with Distributed ASMs, this modelling of threads on a single ASM allows computations that perpetually switch between threads, without making any actual progress.

6 Denotational Semantics

The framework of Denotational Semantics was developed by Scott and Strachey at Oxford in the late 1960's [20,36,39]. One of the main aims was to provide a proper mathematical foundation for reasoning about programs and for understanding the fundamental concepts of programming languages. Denotational Semantics has since been used in teaching [29,35,41] as well as in research. It has also been used to define the functional programming language Scheme [32]; attempts to give denotational semantics for larger programming languages have generally been less successful, although several major descriptions have been given using a notational variant of denotational semantics called VDM [2].

6.1 Denotations

▷ The denotation of a part of a program represents its contribution to overall behaviour.

The denotation of a construct is typically a function of arguments that represent the information available before its execution, and the result represents the information available afterwards. The intermediate states during the execution of the construct are generally of no relevance (except when interleaving is allowed) and are thus not represented, cf. big-step SOS (Natural Semantics). Usually, nontermination is represented by a special value written $\perp$ (the bottom element in a partial ordering based on information content).

▷ Denotations are defined inductively.

Semantic functions map constructs to their denotations. For example, let Exp be the abstract syntax of expressions, and let Den be the set of all (potential) denotations of expressions. A semantic function for expressions:

$$\mathcal{E} : \mathrm{Exp} \to \mathrm{Den}$$

is defined inductively by semantic equations such as:

$$\mathcal{E}[\![\mathrm{cond}(E_1, E_2, E_3)]\!] = F(\mathcal{E}[\![E_1]\!], \mathcal{E}[\![E_2]\!], \mathcal{E}[\![E_3]\!]) \tag{34}$$

▷ λ-notation is used to specify how the denotations of components are to be combined.

$F : \mathrm{Den}^3 \to \mathrm{Den}$ above is defined using so-called λ-*notation*, which is a mathematical notation for function abstraction (a function with argument x

is written $\lambda x.t$), application, and composition, extended with a case construct and a few other useful features. Note that both $\mathcal{E}$ and F above are higher-order functions, assuming that Den is a set of functions.

▷ Denotations of loops and recursive procedures are least fixed-points of continuous functions on Scott-domains.

To define the denotation d of a loop, for instance, we need to be able to provide a well-defined solution to $d = F(d)$, where $F(d)$ is a particular composition of d with the denotations of the loop condition and body. It turns out that such an equation always has a solution, and in particular it has a *least* solution—provided only that $F : \mathrm{Den} \to \mathrm{Den}$ is *continuous* in a certain sense on Den, which has to be a Scott-domain: a *cpo* (complete partially-ordered set). In fact F is *always* continuous when defined using λ-notation, so in practice, familiarity with the mathematical foundations of Denotational Semantics is not required.

6.2 Direct and Continuation-Passing Styles

▷ The denotational semantics of a purely functional language may be in *direct* or in *continuation-passing* style.

Using the direct style, let $\mathrm{Den} = \mathrm{Val}_\perp$; then the denotations of conditional expressions can be defined as follow:

$$\begin{aligned}\mathcal{E}[\![\mathrm{cond}(E_1, E_2, E_3)]\!] = \\ \text{case } \mathcal{E}[\![E_1]\!] \text{ of true} \Rightarrow \mathcal{E}[\![E_2]\!] \mid \\ \text{false} \Rightarrow \mathcal{E}[\![E_3]\!]\end{aligned} \tag{35}$$

Note that if the denotation of E_1 is $\perp$, then so is that of the whole conditional expression; this reflects that if the evaluation of E_1 never terminates, then neither does that of the enclosing expression. If the evaluation of E_1 does terminate, it should give either tt or ff, which are here the denotations of the corresponding boolean constants:

$$\mathcal{E}[\![\mathrm{true}]\!] = \mathrm{tt} \qquad \mathcal{E}[\![\mathrm{false}]\!] = \mathrm{ff} \tag{36}$$

(In Denotational Semantics, one generally avoids use of syntactic phrases such as true and false in the set of denotations.)

The so-called continuation style of denotational semantics looks rather different. Here one would take $\mathrm{Den} = \mathrm{K} \to \mathrm{A}$, where $\mathrm{K} = \mathrm{Val} \to \mathrm{A}$ and A

is some set of values representing the possible results of executing complete programs (e.g. for ML, A would be Val together with some values representing unhandled exceptions). The idea is that the continuation given as argument to $\mathcal{E}[\![E_1]\!]$ is supposed to be applied to the value computed by E_1; if E_1 never terminates, or terminates exceptionally, the continuation is simply ignored. The continuation for E_1 involves the denotations of E_2 and E_3, which are both given the continuation k provided for the entire conditional expression.

$$\begin{aligned}\mathcal{E}[\![\text{cond}(E_1, E_2, E_3)]\!] = \\ \lambda k.\mathcal{E}[\![E_1]\!](\lambda t.\text{case } t \text{ of tt} \Rightarrow \mathcal{E}[\![E_2]\!]k \mid \\ \text{ff} \Rightarrow \mathcal{E}[\![E_3]\!]k)\end{aligned} \tag{37}$$

If E_1 is simply true, its denotation applies the continuation k to the corresponding value:

$$\mathcal{E}[\![\text{true}]\!] = \lambda k.k(\text{tt}) \qquad \textit{etc.} \tag{38}$$

▷ Bindings are represented by explicit arguments of denotations.

Regardless of whether denotations are in the direct style or using continuations, the dependency of the values of identifiers on the bindings provided by their context is represented by letting denotations be functions of environments. For instance, let Den = Env → $\text{Val}_\perp$; then the direct semantics for conditional expressions would be formulated as follows:

$$\begin{aligned}\mathcal{E}[\![\text{cond}(E_1, E_2, E_3)]\!] = \\ \lambda\rho.\text{case } \mathcal{E}[\![E_1]\!]\rho \text{ of tt} \Rightarrow \mathcal{E}[\![E_2]\!]\rho \mid \\ \text{ff} \Rightarrow \mathcal{E}[\![E_3]\!]\rho\end{aligned} \tag{39}$$

The denotation of an identifier simply applies the environment to the identifier itself:

$$\mathcal{E}[\![I]\!] = \lambda\rho.\rho(I) \tag{40}$$

▷ Effects on storage are represented by letting denotations be functions from stores to stores.

It might seem that the easiest would be to add stores as arguments and results to the direct-style denotations given above, taking Den = Env → (Store → $(\text{Val} \times \text{Store})_\perp$). However, that would lead to the following semantic equation

for conditional expressions, which is not as perspicuous as one might wish:

$$\begin{aligned}\mathcal{E}[\![\mathrm{cond}(E_1, E_2, E_3)]\!] = \\ \lambda\rho.\lambda\sigma.(\lambda(t,\sigma').\mathrm{case}\ t\ \mathrm{of}\ \mathrm{tt} \Rightarrow \mathcal{E}[\![E_2]\!]\rho\,\sigma'\ \mid \\ \mathrm{ff} \Rightarrow \mathcal{E}[\![E_3]\!]\rho\,\sigma') \\ (\mathcal{E}[\![E_1]\!]\rho\,\sigma)\end{aligned} \tag{41}$$

So let us instead try adding stores to the denotations used with the continuation-style semantics. The appropriate set of denotations is then $\mathrm{Den} = \mathrm{Env} \to \mathrm{K} \to \mathrm{C}$, where $\mathrm{K} = \mathrm{Val} \to \mathrm{C}$, and $\mathrm{C} = \mathrm{Store} \to \mathrm{A}$, and we may give a relatively straightforward-looking semantic equation—not even mentioning the stores, which automatically follow the flow of control in continuation semantics:

$$\begin{aligned}\mathcal{E}[\![\mathrm{cond}(E_1, E_2, E_3)]\!] = \\ \lambda\rho.\lambda k.\mathcal{E}[\![E_1]\!]\rho(\lambda t.\mathrm{case}\ t\ \mathrm{of}\ \mathrm{tt} \Rightarrow \mathcal{E}[\![E_2]\!]\rho\,k\ \mid \\ \mathrm{ff} \Rightarrow \mathcal{E}[\![E_3]\!]\rho\,k)\end{aligned} \tag{42}$$

▷ Nondeterminism and interleaving can be represented by letting denotations be set-valued functions.

In operational frameworks based on transition relations, the possibility of nondeterministic computations doesn't make any difference to how rules are formulated. In denotational semantics, however, the use of functions as denotations means that the ranges of the functions have to be changed to allow them to return sets of possible results; moreover, other functions that are to be composed with these set-valued functions have to be extended to functions that take sets as arguments. As one may imagine, the extra notation required leads to further complication of the specifications of denotations.

Since interleaving generally entails nondeterminism, its denotational description obviously requires the use of set-valued functions. However, a further problem arises: it simply isn't possible to compose functions that map initial states to (sets of) final states so as to obtain a function corresponding to their interleaving at intermediate states. So-called *resumptions* are needed: these are functions representing the points at which interleaving can take place, and correspond closely to the computations used in operational semantics.

▷ Denotational semantics is applicable to modelling languages.

The denotation of a model can be taken to be the set of entities that the model specifies. As with big-step SOS, the abstract syntax of the modelling language has to be tree-structured.

6.3 Monadic Semantics

▷ Use of monadic notation gives good modularity.

The straightforward use of λ-notation to specify how denotations are combined requires awareness of the exact structure of the denotations: whether they are functions of environments, stores, continuations, etc. When new constructs are added to a language, it may be necessary to change the structure of the denotations, and reformulate all the semantic equations that involve those denotations. Thus it appears that use of λ-notation is a major hindrance to obtaining modularity in denotational descriptions.

However, suppose that we define some *auxiliary notation* for combining denotations, corresponding to fundamental concepts such as sequencing. We may then be able to specify denotations using the auxiliary notation, without any dependence on the structure of denotations. If we later change that structure, we shall also have to change the definition of the auxiliary notation—but the use of that notation in the semantic equations may remain the same.

Monadic Semantics provides a particular auxiliary notation for use in Denotational Semantics. It was developed by Moggi at the end of the 1980's [18,19], and inspired by category-theoretic concepts. The basic idea is that denotations compute values of particular types; when two such denotations are sequenced, the value computed by the first one is made available to the second one, written 'let $x = d_1$ in d_2' (where d_2 usually depends on x). The only other bit of essential notation is for forming a denotation that simply computes a particular value v, which is written '$[v]$'. A set of denotations equipped with this notation may be regarded as a mathematical structure called a *monad*. Here is how the semantic equation for conditional expressions looks in the monadic variant of Denotational Semantics:

$$\begin{aligned} \mathcal{E}[\![\mathrm{cond}(E_1, E_2, E_3)]\!] = \\ \quad \text{let } t = \mathcal{E}[\![E_1]\!] \text{ in} \\ \quad \text{case } t \text{ of } \mathrm{tt} \Rightarrow \mathcal{E}[\![E_2]\!] \mid \mathrm{ff} \Rightarrow \mathcal{E}[\![E_3]\!] \end{aligned} \tag{43}$$

Note that as well as being independent of the structure of denotations, the monadic semantic equation is also more perspicuous and suggestive of the intended semantics than our previous semantic equations were.

▷ Monad transformers add support for further features.

How about further ways of combining denotations that might be needed, but which are not based on sequencing? Some *monad transformers* are available: fundamental ways of adding features to denotations (bindings, effects on storage, exceptions, nondeterministic choice, interleaving, etc.), together with appropriate notation. Unfortunately, it isn't always so straightforward to combine different monad transformers, and difficulties can arise when trying to redefine auxiliary notation in connection with applying a monad transformer. (We shall consider Action Semantics, a hybrid framework that doesn't suffer from such problems, in Sect. 8.)

7 Axiomatic Semantics

Axiomatic Semantics was developed primarily by Hoare in the late 1960's [11]. The main aim was initially to provide a formal basis for the verification of abstract algorithms; later, the framework was applied to the definition of programming languages, and consideration of Axiomatic Semantics influenced the design of Pascal [12].

▷ A Hoare Logic gives rules for the relation between assertions about values of variables before and after execution of each construct

Usually, the constructs concerned are only statements S. Suppose that P and Q are assertions about the values of particular variables; then the so-called *partial correctness* formula $P\{S\}Q$ states that if P holds at the beginning of an execution of S and the execution terminates, Q will always hold at the end of that execution. Notice that $P\{S\}Q$ does not require S to terminate, nor does it require Q to hold after an execution of S when P didn't hold at the beginning of the execution.

A Hoare Logic specifies the relation $P\{S\}Q$ inductively by rules, in the same way that as an SOS specifies a transition relation.

▷ Expressions are assumed to have no side-effects.

Expressions are used in assertions, so their interpretation has to be purely mathematical, without effects on storage, exceptions, non-terminating function calls, etc. For example, consider conditional statements with the following abstract syntax, where the conditions are restricted to pure boolean-valued expressions:

$$\text{Stm} ::= \text{cond}(\text{Exp}, \text{Stm}, \text{Stm})$$

A typical rule given for this construct is:

$$\frac{(P \wedge E)\{S_1\}R, \qquad (P \wedge \neg E)\{S_2\}R}{P\{\mathrm{cond}(E, S_1, S_2)\}R} \tag{44}$$

Notice the use of E as a sub-formula in the assertions, holding when the expression evaluates to true.

Similarly, the usual rule for assignment is:

$$P[E/I]\{\mathrm{update}(I, E)\}P \tag{45}$$

This involves the substitution $P[E/I]$ of an expression E for an identifier I in an assertion P.

▷ Bindings can be represented by explicit environments.

In many presentations of Hoare Logic, bindings are left implicit: the relation $P\{S\}Q$ is defined on the basis of a fixed set of bindings. To reflect local declarations, it is necessary to use more elaborate formulae such as $\langle \rho \mid P\{S\}Q \rangle$, where the current bindings ρ are made explicit.

▷ Hoare Logic for concurrent processes involves rules for interleaving.

Hoare Logic is exploited in connection with the development and verification of concurrent processes. The rules can get rather complicated.

▷ Predicate transformer semantics is essentially denotational.

In connection with a methodology for developing programs from specifications, Dijkstra defined, for each statement S and postcondition Q, the weakest precondition P guaranteeing total correctness: if P holds at the beginning of the execution of S, then S always terminates, and Q holds at the end of the execution. Although the assertions used here are similar to those in Hoare Logic, the definition of the weakest precondition P is actually inductive in the structure of the statement S, and Dijkstra's semantics is better considered as denotational (with the denotations being predicate transformers, i.e. functions on the interpretation of assertions) rather than axiomatic.

8 Action Semantics

The Action Semantics framework was developed by the present author, in collaboration with Watt, in the second half of the 1980's [21,28,40]. (The

UML Action Semantics is to some extent similar in spirit to the original Action Semantics framework, although there are major technical differences.)

▷ Action Semantics is a hybrid of denotational and operational semantics.

As in denotational semantics, inductively-defined semantic functions map phrases to their denotations, only here, the denotations are so-called *actions*. The notation for actions is itself defined operationally [21,24].

▷ Action Semantics avoids the use of higher-order functions expressed in λ-notation.

The universe of pure mathematical functions is so distant from that of (most) programming languages that the representation of programming concepts in it is often excessively complex. The foundations of reflexive Scott-domains and higher-order functions are unfamiliar and inaccessible to many programmers (although the idea of functions that take other functions as arguments, and perhaps also return functions as results, is not difficult in itself).

▷ Action semantics provides a rich action notation with a direct operational interpretation

The universe of actions involves not only control and data flow, but also scopes of bindings, effects on storage, and interactive processes, allowing a simple and direct representation of many programming concepts.

Computed values are given by actions, and the action combination 'A_1 *then* A_2' passes all the values given by A_1 to A_2. For example, assuming evaluate : Exp $\rightarrow$ *Action*, the value computed by evaluate E_1 is the one tested by the action '*given true*' below:

$$\begin{aligned} &\text{evaluate cond}(E_1, E_2, E_3) = \\ &\quad \text{evaluate } E_1 \textit{ then} \\ &\quad (\textit{ given true then } \text{evaluate } E_2 \\ &\qquad \textit{otherwise } \text{evaluate } E_3 \ ) \end{aligned} \tag{46}$$

Bindings are implicitly propagated to the sub-actions of most actions, and can always be referred to, as illustrated below:

$$\text{evaluate } I = \textit{give the val bound to } I \tag{47}$$

Effects on storage implicitly follow the flow of control:

$$\begin{aligned} &\text{evaluate assign}(E_1, E_2) = \\ &\qquad \text{evaluate } E_1 \textit{ and } \text{evaluate } E_2 \\ &\qquad \textit{then update}(\textit{the loc\#1}, \textit{the val\#2}) \end{aligned} \tag{48}$$

Concurrent processes are represented by agents that perform separate actions, with asynchronous message-passing.

9 Conclusion

In these notes, we have considered the main frameworks for semantic description, giving fragments to illustrate the different styles of specification that are used there.

Most of the frameworks have significant disadvantages regarding the modularity of semantic descriptions, severely limiting the possibility of reusing parts of the specification of one language in the specification of another:

- The original SOS framework may require reformulation of transition rules when the described language is extended.
- Reduction Semantics has reasonable modularity, but uses substitution to deal with scopes of bindings, which is tedious to specify.
- Semantic descriptions using Abstract State Machines tend to introduce *ad hoc* auxiliary notation to abbreviate the many details (e.g. concerning stacks of frames), which hinders reuse of transition rules in specifications of different languages.
- The direct use of λ-notation in Denotational Semantics requires reformulation of semantic equations when the described language is extended. Use of monadic notation eliminates this problem, but the foundations of the framework may be regarded as too abstract for use by programmers (despite the recent popularity of monadic techniques in functional programming).
- Axiomatic Semantics becomes quite complicated when used to describe real programming languages such as Java, and in any case is a somewhat indirect way of defining models for programming languages.

MSOS appears to have significant advantages over all the alternative frameworks, at least regarding the actual *specification* of semantics. Action Semantics is as modular as MSOS, but it may be regarded as a disadvantage that it requires familiarity with Action Notation, which has somewhat more constructs than the notations underlying the other frameworks. A major

advantage of Action Semantics is that it supports automatic generation of (reasonably efficient) compilers from semantic descriptions [13,14].

Proving properties of programs and languages is quite demanding in all the frameworks (apart from in Axiomatic Semantics, which is closely related to the specification of *invariants* in programs); the modularity of MSOS and Action Semantics should allow reuse of proofs of properties between descriptions of different languages, giving them some advantages over other frameworks in this respect.

Acknowledgement

The initial version of this paper was written while the author was employed at the University of Aarhus, Denmark, and supported by BRICS (Basic Research in Computer Science [www.brics.dk], funded by the Danish National Research Foundation).

References

[1] L. Aceto, W. Fokkink, and C. Verhoef. Structural operational semantics. In *Handbook of Process Algebra*, chapter 3, pages 197–291. Elsevier Science, 2001.

[2] D. Bjørner and C. B. Jones, editors. *Formal Specification & Software Development*. Prentice-Hall, 1982.

[3] S. Cook and S. Kent. Language anatomy. In *Software Factories*, chapter 8. Wiley, 2004.

[4] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*, volume B, chapter 6. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[5] M. Felleisen and D. P. Friedman. Control operators, the SECD machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1987.

[6] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[7] Y. Gurevich. May 1997 draft of the ASM Guide. Technical Report CSE-TR-336-97, University of Michigan EECS Department, 1997.

[8] G. Hamon and J. Rushby. An operational semantics for Stateflow. In *FASE 2004*, LNCS 2984, pages 229–243. Springer, 2004.

[9] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF: Reference manual. *ACM SIGPLAN Notices*, 24(11):43–75, 1989.

[10] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. Wiley, New York, 1990.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, 1969.

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

[13] J. Iversen. An action compiler targeting Standard ML. In *LDTA 2005, Proc. 5th Workshop on Language Descriptions, Tools and Applications*, ENTCS. Elsevier, 2005. To appear.

[14] J. Iversen. *Formalisms and Tools Supporting Constructive Action Semantics*. PhD thesis, University of Aarhus, 2005.

[15] G. Kahn. Natural semantics. In *STACS'87*, LNCS 247, pages 22–39. Springer, 1987.

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

[17] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[18] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh, 1990.

[19] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[20] P. D. Mosses. Denotational semantics. In *Handbook of Theoretical Computer Science*, volume B, chapter 11. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[21] P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[22] P. D. Mosses. Foundations of modular SOS. BRICS RS-99-54, Dept. of Computer Science, Univ. of Aarhus, 1999.

[23] P. D. Mosses. Foundations of Modular SOS (extended abstract). In *MFCS'99*, LNCS 1672, pages 70–80. Springer, 1999.

[24] P. D. Mosses. A modular SOS for Action Notation (extended abstract). In *AS'99, 2nd Intl. Workshop on Action Semantics, Amsterdam, The Netherlands, Proceedings*, BRICS NS-99-3, pages 131–142. Dept. of Computer Science, Univ. of Aarhus, 1999. Full version available at http://www.brics.dk/RS/99/56/.

[25] P. D. Mosses. Pragmatics of modular SOS. In *AMAST'02*, LNCS 2422, pages 21–40. Springer, 2002.

[26] P. D. Mosses. Modular structural operational semantics. *J. Logic and Algebraic Programming*, 60–61:195–228, 2004. Special issue on SOS.

[27] P. D. Mosses, editor. CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.

[28] P. D. Mosses and D. A. Watt. The use of action semantics. In *Formal Description of Programming Concepts III*, pages 135–166. North-Holland, 1987.

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

[30] OMG. *Unified Modeling Language Specification*, v1.5 edition, 2001. OMG Document formal/03-03-01, http://www.omg.org.

[31] G. D. Plotkin. A structural approach to operational semantics. *J. Logic and Algebraic Programming*, 60–61:17–139, 2004. Originally published as DAIMI FN-19, Dept. of Computer Science, Univ. of Aarhus, 1981.

[32] J. Rees, W. Clinger, et al. The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, 1986.

[33] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[34] J. Schmid. *Introduction to AsmGofer*, 2001. http://www.tydo.de/AsmGofer.

[35] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn & Bacon, 1986.

[36] D. S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In *Proc. Symp. on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series*. Polytechnic Institute of Brooklyn, 1971.

[37] K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1995.

[38] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer, 2001.

[39] R. D. Tennent. The denotational semantics of programming languages. *Commun. ACM*, 19:437–453, 1976.

[40] D. A. Watt. *Programming Language Syntax and Semantics*. Prentice-Hall, 1991.

[41] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.