BRICS

Basic Research in Computer Science

BRICS NS-96-14 P. D. Mosses: A Tutorial on Action Semantics

A Tutorial on Action Semantics

Peter D. Mosses

BRICS Notes Series **NS-96-14**

ISSN 0909-3206 **December 1996**

See back inner page for a list of recent publications in the BRICS
Notes Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through World Wide
Web and anonymous FTP:

http://www.brics.dk/
ftp://ftp.brics.dk/
This document in subdirectory NS/96/14/

A Tutorial on Action Semantics

Notes for FME'96

Peter D. Mosses[1]
BRICS[2]
Dept. of Computer Science
University of Aarhus
Ny Munkegade, Bldg. 540
DK–8000 Aarhus C, Denmark

March 1996

[1]Internet: `pdmosses@brics.dk`, Home page URL: `http://www.brics.dk/~pdm`

[2]Centre for Basic Research in Computer Science, established in cooperation between the Danish National Research Foundation and the Universities of Aarhus and Aalborg

Summary

Action Semantics is a useful framework for specifying programming languages: documenting design decisions, setting standards for implementations, etc. It has unusually good pragmatics, making specifications easily accessible to programmers. Thanks to its inherent modularity, action semantics scales up smoothly to describing practical, industrially-useful languages.

In this 1/2-day tutorial, action semantics is explained, illustrated, and compared to other frameworks such as VDM and RAISE.

Foreword

This tutorial was given at FME'94 (*Formal Methods Europe*, 24–28 October 1994, Barcelona) and is being repeated at FME'96 (18–22 March 1996, Oxford). Presumably, most participants there will agree that formal methods are absolutely essential for giving precise specifications of *software systems* in general, and thus also of those systems that implement *programming languages*, i.e., compilers and interpreters. Formally specifying the entire class of valid implementations of a programming language amounts to giving a formal semantics for the language.

Frameworks for formal semantics often seem quite attractive when presented in papers and text-books, their use being illustrated by small, carefully-chosen examples. But in general, they do not scale up well to specifying the intended classes of implementations of larger, industrially-used programming languages.

The topic of this tutorial, *action semantics*, is an exception! Action semantics allows *useful* semantic descriptions of *realistic* programming languages. This has recently been demonstrated in connection with the formal specification of ANDF.[1] Another large-scale example is the action semantics of ISO Standard Pascal. Various student projects have shown that action semantics is applicable to a wide range of programming languages, including those with concurrent and object-oriented programming constructs.

Action semantics combines formality with many good pragmatic features. Regarding comprehensibility and accessibility, for instance, action semantic descriptions compete with informal language descriptions. Action semantic descriptions scale up smoothly from small example languages to full-blown practical languages. The addition of new constructs to a described language does not require reformulation of the already-given description. An action semantic description of one language can make widespread reuse of that of another, related language. All these pragmatic features are highly desirable. Action semantics is, however, the *only* semantic framework that enjoys them!

This tutorial provides a quick (half-day) introduction to action semantics. It should be accessible to graduates in Computer Science and to professional programmers. No previous knowledge of formal semantics is assumed. For further study, a full exposition of the framework and an extended example are provided in [15].

[1] Architecture-Neutral Distribution Format, a language developed by OSF

Organisation The tutorial has been allocated three hours, which is to include a short refreshment break. The presentation of the material will be organised as follows:

Lecture 1: We shall start with an 'appetiser', see Chapter 1. Then, as most of the participants at FME'96 may be already familiar with the basic notions of formal semantics, we shall run very quickly through much of Chapter 2: Section 2.1 motivates the formal description of programming languages by considering the requirements of various potential users; Section 2.2 recalls the usual notions of abstract syntax and compositional semantics. We shall take more time over Section 2.2.2, which introduces the novel concept of *actions* as used in action semantics, and over Section 2.3, which explains the intended interpretation of the meta-notation used for specifying action semantic descriptions.

Lecture 2: Chapter 3 shows action semantics in action. We shall take a walk through an illustrative description, explaining all the notation that it uses on the way. This lecture will cover the semantics of Expressions and Statements.

Lecture 3: After some refreshments, we shall continue our walk through Chapter 3, covering the semantics of Declarations and Programs. Then in Chapter 4 we shall summarise the design of action notation, and explain what makes it so special.

Lecture 4: We shall conclude by discussing, in Chapter 5, the relationship of the action semantics framework to VDM and RAISE—indicating how users of those frameworks might benefit by exploiting some of the ideas of action semantics. Finally, we shall review the current state of development of action semantics with respect to applications and tools. Any remaining time will be used for questions and discussion.

Acknowledgements Christopher Strachey developed the fundamental conceptual analysis of programming languages that underlies action semantics. David Watt (Glasgow) collaborated on the design of action notation, and on the action semantic description of realistic programming languages; his influence was crucial for the development of action semantics into a practical approach. Arie van Deursen (CWI, Amsterdam) helped with developing essential tools for checking action semantic descriptions.

This tutorial was written with the support of BRICS (Centre for Basic Research in Computer Science, established in cooperation between the Danish National Research Foundation and the Universities of Aarhus and Aalborg).

Contents

Chapter 1

Appetiser

The following very trivial example of an action semantic description is intended to arouse the reader's curiosity, and to whet the appetite for the more substantial illustrations that will be given in Chapter 3. It is *not* intended to be *understood* straight away! Some readers may find it amusing to try and guess the intended interpretation of the symbols used; but it will all be explained before long, in the course of this tutorial.

The *point* of this example is show that action semantic descriptions—henceforth abbreviated *a.s.d.s*—are quite *unlike* conventional descriptions, especially in their 'user-friendliness'. For instance:

- In the specification of abstract syntax, the symbols used are strongly suggestive of familiar concrete syntax, so there is no need for users to memorise a new nomenclature for program constructs.
- In general, ordinary English words, rather than unfamiliar Greek letters and/or mathematical signs, are used as symbols.
- The semantic equations that define the semantic functions are free of the tedious repetitive mention of contextual information (e.g., the environment of current bindings).

Further significant features of a.s.d.s, such as their inherently good modularity and extensibility, should become apparent in Chapter 3.

1.1 Abstract Syntax

grammar:

(1) Statement = ⟦ Identifier ":=" Expression ⟧ |
⟦ "if" Expression "then" Statement ⟨ "else" Statement ⟩$^?$ ⟧ |
⟦ "while" Expression "do" Statement ⟧ | ⟦ "begin" Statements "end" ⟧ .

(2) Statements = ⟨ Statement ⟨ ";" Statement ⟩* ⟩ .

(3) Expression = Numeral | Identifier | ⟦ "(" Expression ")" ⟧ | ⟦ Expression Operator Expression ⟧ .

(4) Operator = "+" | "−" | "∗" | "≠" | "and" | "or" .

(5) Numeral = ⟦ digit$^+$ ⟧ .

(6) Identifier = ⟦ letter (letter | digit)* ⟧ .

closed.

1.2 Semantic Functions

introduces: execute _ , evaluate _ , the operation-result of _ , the value of _ .

- execute _ :: Statements → action [completing | diverging | storing] .

(1) execute ⟦ I:Identifier ":=" E:Expression ⟧ =
(give the cell bound to I and evaluate E) then store the given number#2 in the given cell#1 .

(2) execute ⟦ "if" E:Expression "then" S_1:Statement ⟧ =
evaluate E then
| | check the given truth-value and then execute S_1
| or
| | check not the given truth-value and then complete .

(3) execute ⟦ "if" E:Expression "then" S_1:Statement "else" S_2:Statement ⟧ =
evaluate E then
| | check the given truth-value and then execute S_1
| or
| | check not the given truth-value and then execute S_2 .

(4) execute ⟦ "while" E:Expression "do" S:Statement ⟧ =
unfolding
| evaluate E then
| | | check the given truth-value and then execute S and then unfold
| | or
| | | check not the given truth-value and then complete .

(5) execute ⟦ "begin" S:Statements "end" ⟧ = execute S .

(6) execute ⟨ S_1:Statement ";" S_2:Statements ⟩ = execute S_1 and then execute S_2 .

- evaluate _ :: Expression → action [giving a value] .

(7) evaluate N:Numeral = give the value of N .

(8) evaluate I:Identifier = give the value bound to I or give the number stored in the cell bound to I .

(9) evaluate ⟦ "(" E:Expression ")" ⟧ = evaluate E .

(10) evaluate ⟦ E_1:Expression O:Operator E_2:Expression ⟧ =
(evaluate E_1 and evaluate E_2) then give the operation-result of O .

- the operation-result of _ :: Operator → yielder [of a value] [using the given value^2] .

(11) the operation-result of "+" = the sum of (the given number#1, the given number#2) .

(12) the operation-result of "−" = the difference of (the given number#1, the given number#2) .

(13) the operation-result of "∗" = the product of (the given number#1, the given number#2) .

(14) the operation-result of "≠" = not (the given value#1 is the given value#2) .

(15) the operation-result of "and" = both of (the given truth-value#1, the given truth-value#2) .

(16) the operation-result of "or" = either of (the given truth-value#1, the given truth-value#2) .

- the value of _ :: Numeral → number .

(17) the value of N:Numeral = number & decimal N .

1.3 Semantic Entities

includes: **Action Notation**.

introduces: value , number .

(1) datum = cell | number | truth-value | □ .

(2) token = string of (letter, (letter | digit)*) .

(3) bindable = cell | number .

(4) storable = number .

(5) value = number | truth-value .

(6) number ≤ integer .

Chapter 2

Introduction

This chapter provides some background for the topic of this tutorial, namely the action semantic description of programming languages. We start by identifying some important uses for descriptions of programming languages. Although various uses require different features of descriptions, these requirements are not necessarily conflicting. Formality is, of course, a particularly important feature, essential for most applications.

2.1 Motivation

Even though programming languages lack many of the features of natural languages, such as vagueness, it is not at all easy to give accurate, comprehensive descriptions of them. Which applications require such descriptions of programming languages—and hence motivate the study of appropriate frameworks?

applications?

design

First, there is the programming language *design* process. Designers need to record decisions about particular language constructs, especially when the design is a team effort. This amounts to giving a partial description of the language. At a later stage, the formulation of a complete language description may be useful for drawing attention to neglected details, and for revealing irregularities in the overall design.

implementation

Once a language has been designed, it usually gets *implemented*. A comprehensive description of the language is needed to convey the intentions of the language designers to the implementors. It is also needed for setting a definitive *standard* for implementations, so that programs can be transported between different implementations that conform to the standard, without modification.

understanding

reasoning

A programmer needs a description of any new language in order to relate it to previously-known ones, and to *understand* it in terms of familiar concepts. The programmer also needs a description as a basis for *reasoning* about the correctness of particular programs in relation to their specifications, and for justifying program transformations.

insight

Finally, theoreticians can obtain new *insight* into the general nature of programming languages through developing descriptions of them. This insight can then be exploited in the design of new, and perhaps more elegant, programming languages.

So we see that there are plenty of applications for descriptions of programming languages. But not all kinds of description are suitable for all purposes. The various applications mentioned above require different properties of descriptions, as we consider next.

2.1.1 Requirements

Which properties might language designers require of language descriptions? Well, design is usually a gradual process, so to start with, designers need *partial* descriptions that can easily be extended and modified. They also need descriptions that provide clear and concise documentation of individual language design decisions. For economy of effort, they might want to be able to reuse parts of descriptions of existing languages in the description of a new language. Finally, the completed language description should provide an appropriate basis for conveying the design to the implementors, and for setting standards.

partial descriptions

completeness

The implementors of a language require a *complete* and unambiguous description of it—except that certain features, such as the order of subexpression evaluation, may have been deliberately left unspecified. Explicit indication of implementation techniques in the description may be helpful, but it might discourage alternative, perhaps more efficient, implementations. Ideally, the conformance of a purported implementation to a standard imposed by a language description should be verifiable.

compiler generation

A long-term aim is to generate complete, correct, and efficient implementations automatically from language descriptions, in the same way that parsers can be generated from grammars. There have already been some encouraging experiments in this direction. This application requires that the language descriptions can be understood operationally.

practicality

What do programmers require? A language description should be easy to understand, and to relate to familiar programming concepts, without a major investment of effort in learning about the description technique itself. It should support program verification. And it shouldn't take up too much space on the shelf ...

theory

Theoreticians may require clear and elegant foundations for the exploited description technique, to support tractable reasoning about equivalence and other program features. They may take a *prescriptive* view of language description, considering only a restricted class of programming languages—those amenable to their favourite description technique—in the hope that this will prevent the design of 'big, bad, and ugly' languages. Or they may take a more liberal, *descriptive* view, requiring a universal description technique that can cope with any conceivable programming language, and hoping that poor language design will be evident from its description.

It seems highly unlikely that all these requirements can be fully satisfied simultaneously. None of the previously available frameworks appears to be suitable for use in all the applications mentioned above. This has led to the proposal of so-called *complementary* language descriptions, where several different techniques are used to give independent, but hopefully relatable, descriptions of 'the same' language. However, this is clearly impractical for larger programming languages, where a major effort is required to formulate even one description.

complementary descriptions

action semantics

The topic of this tutorial, action semantics, combines some of the best features of previous frameworks, and puts the emphasis on practicality and user-friendliness. An action semantic description is extremely modular, providing the high degree of extensibility, modifiability, and reusability required by language designers. It is also strongly suggestive of an operational understanding of the described language, and it has been found to be very well suited for generating compilers [21, 20] and interpreters [27]. Programmers should find action semantic descriptions almost as easy to read as the usual reference manuals, without much preparation. On the other hand, although the foundations of action semantics are firm enough, the *theory* for reasoning about actions (and hence about programs) is still rather weak, and needs further development. This situation is in marked contrast to that of denotational semantics [13], where the theory is strong,[1] but severe pragmatic difficulties hinder its application to realistic programming languages.

Most of these claims for the virtues of action semantic descriptions are supported by the illustrative example given in Chapter 3.

2.1.2 Features

formality

One especially significant feature of language descriptions is whether or not they are *formal*.[2] Let us distinguish between formal and informal descriptions as follows. Purely formal descriptions are expressed in well-defined, established notation, often borrowed from mathematics. Note that this notation itself may have been established either formally, using some previously-established *meta-notation*, or informally (but rigorously) as in most mathematical texts. Purely informal descriptions are expressed in natural languages, such as English.

informal reference manuals

Currently, the only comprehensive description of a programming language is usually its 'reference manual', which is mainly informal. Unfortunately, experience has shown that, even when carefully worded, such reference manuals are usually incomplete or inconsistent, or both, and open to misinterpretation. This is obviously undesirable, especially when such descriptions are used to guide implementors and to set standards. The existence of procedures for requesting clarification of international standards, which are generally based on reference manuals, confirms that misinterpretation is a problem. Moreover, informal descriptions can never provide a sound basis for reasoning about program correctness or equivalence.

validation suites

To compensate for the vaguenesses of an informal language description, a formal *validation suite* of programs is sometimes used as the final arbiter of implementation correctness. By itself, however, such a validation suite is not much use as a language description. In any case, the correct processing of a validation suite by an implementation cannot guarantee similar performance on other data.

natural language

It might be imagined that informal descriptions should be easy to read, because they are written in a natural language; but in fact the (vain) attempt to be precise in a natural language leads to a rather stilted literary style that is tedious to read on a large scale. When well-written, however, informal descriptions can provide an easily-accessible guide to the fundamental concepts underlying a programming language; this seems to be their only real strength.

formal descriptions

Formal descriptions have almost the opposite qualities to informal ones. They can be complete and consistent, and can be given a precise interpretation, appropriate for setting *definitive* standards. Questions about their consequences are answered by the theoretical foundations of the formal notation used. Formal descriptions can be used as the basis for systematic development and automatic generation of implementations. And it is one of their main strengths that they can provide a basis for sound reasoning about program correctness and equivalence.

On the other hand, it is often difficult to relate a formal description of a programming language to fundamental concepts, and to grasp the implications of the description for the implementation of programs. Poor notation, or excessively large and complex formulae can also lead to obscurity. Inferior formal descriptions can be unintentionally ambiguous or incomplete—even inconsistent! The mere use of formal notation does *not* ensure success.

compilers

One could consider the text of a *compiler*, or of an interpreter, as a formal definition of the language that it implements. The language used for writing it should already have a well-defined interpretation, of course: a so-called *meta-circular* in-

[1] At least for dealing with deterministic, sequential programs.

[2] The following considerations are presumably familiar to those attending FME'96, so the oral presentation will be very brief.

terpreter, written using the language itself being interpreted, doesn't formally define anything at all! Unfortunately, practical compilers for realistic programming languages are somewhat unwieldy objects, and demand familiarity with particular target codes. Interpreters are generally more accessible, but still tend to have many details that are incidental to the implemented language.

So much for the background of formal descriptions.

2.2 Concepts

This section examines the concepts underlying action semantic descriptions of programming languages. It considers the factorisation of language descriptions into syntax and semantics, and discusses the pragmatic issue of setting standards for programming languages.[3]

syntax v. semantics

In programming linguistics, as in the study of natural languages, it is useful to distinguish between *syntax* and *semantics*. The syntax of a programming language is concerned only with the *form* of programs: which programs are 'legal'? what are the connections and relations between the symbols and phrases that occur in them? Semantics deals with the *meaning* of legal programs.

compositional

Ideally, a comprehensive description of a programming language involves the specification of *syntactic entities*, of *semantic entities*, and of a *semantic function* that maps the former to the latter. The syntactic entities include the legal programs of the language, and the semantic entities include representations of the intended behaviours of these programs. To facilitate reasoning about parts of programs, the semantic function should give semantics not only to entire programs but also to their component phrases; and it should preferably be *compositional*, so that the semantics of a compound phrase is determined purely by the semantics of its components, independently of their other features.

2.2.1 Syntax

The syntax of a programming language determines the set of its legal programs, and the relationship between the symbols and phrases occurring in them.

concrete v. abstract

We may divide syntax into *concrete* and *abstract* syntax. Concrete syntax involves *analysis*: the recognition of legal programs from texts (i.e., sequences of characters) and their unambiguous *parsing* into phrases. Abstract syntax deals only with the compositional *structure* of phrases of programs, ignoring how that structure might have been determined. In general, it is simpler to define the semantics of programs on the basis of their abstract syntax, rather than on their concrete syntax.

For comprehensive language descriptions, both kinds of syntax are needed—together with an indication of how they are related. Here, we are mainly concerned with semantic descriptions, so we emphasise abstract syntax, after quickly recalling some familiar notions connected with concrete syntax.

2.2.1.1 Concrete Syntax

symbols

Conventionally, concrete syntax is separated into *lexical* analysis and *phrase-structure* analysis. The task of lexical analysis is to group the characters of a program text into a *sequence* of legal *symbols*, or *lexemes*. That of phrase-structure analysis is to group these symbols into phrases, thereby constructing a *parse tree*, or *derivation tree*, with the symbols as leaves.

[3] The majority of FME'96 participants will surely be familiar with most of these concepts, so the oral presentation of this section will be relatively brief, focusing on novelties of action semantics.

parse trees The parse tree produced by phrase-structure analysis of a program represents the recognised component relation between its phrases. It may also represent how the phrases have been classified.

Both lexical and phrase-structure analysis are required to be *unambiguous*: a sequence of characters making up a legal program must determine a unique sequence of symbols, which in turn must determine a unique parse tree. In the case of ambiguity, the programmer is left in doubt about the recognised structure of the program.

2.2.1.2 Abstract Syntax

Abstract syntax provides an appropriate interface between concrete syntax and semantics. It is usually obtained simply by ignoring those details of parse tree structure which have no semantic significance—leaving *abstract syntax trees* that represent just the essential compositional structure of programs.

abstract syntax trees

For instance, in concrete syntax one usually sub-classifies compound arithmetic expressions into terms and factors, in order to avoid ambiguous parsing of sequences such as `a+b*c`. But the only semantically relevant features of an arithmetic expression are its subexpressions and its operator, so for abstract syntax we can ignore whether the expression is classified as a term or as a factor.

labels The nodes in an abstract syntax tree need labelling somehow, to distinguish nodes for different constructs. For instance, while-statements and if-then-statements usually both have two components: a condition and a statement; extra symbols are required to label them distinctly, but these can be chosen arbitrarily. The lexical symbols from the concrete syntax have considerable suggestive and mnemonic value; by using them as (distributed) labels in abstract syntax trees, much of the relationship between concrete and abstract syntax can be made self-evident.

Another way of abstracting details of parse tree structure is to ignore the *order* of components, when this is semantically insignificant. For instance, the order in which the cases of a case-statement are written in the program text may be irrelevant; then one could take a *set* of cases, rather than an ordered list. Similarly, one might let declarations be (finite) maps on identifiers, instead of lists of pairs—thereby reflecting also that an identifier is not to be declared twice in the same sequence of declarations. This seems appealing, but on closer investigation turns out to have gone *too far* in abstraction, at least from a pragmatic point of view, as it significantly complicates the definition of semantic functions on abstract syntax.

going too far

It can happen that the compositional structure of programs derived from a given concrete syntax has a nesting that is inconvenient for a compositional semantics. We may then use an abstract syntax that corresponds to a rearrangement of the original structure, provided that we are prepared to specify the map from parse trees to abstract syntax trees. But when this map is complicated, the comprehensibility of the language description suffers considerably.

Some *programming environments* provide templates for constructing and editing abstract syntax trees, and for viewing them graphically, thereby allowing the use of concrete syntax to be avoided. Although this does not justify ignoring concrete syntax altogether when giving a comprehensive description of a programming language, it does underline the importance of abstract syntax, and further motivates that semantics should be defined on the basis of abstract, rather than concrete, syntax.

2.2.1.3 Context-Sensitive Syntax

Context-*free* syntax deals with those aspects of structure that can be described by context-free grammars, such as those written in the BNF formalism. Aspects which *type-checking*

fall outside context-free syntax are called context-*sensitive* and include 'declaration of identifiers before use' and 'well-typedness of expressions'.

The description of context-sensitive syntax can, in principle, be accomplished by use of so-called *attribute grammars*. Unfortunately, these grammars are not always as perspicuous as context-free ones. Moreover, context-sensitive abstract syntax makes a more complicated interface between concrete syntax and semantics than context-free abstract syntax does—which is to be expected, because the former generally captures more information than the latter. A further drawback is that attribute grammars cannot cope straightforwardly with so-called *dynamic scope rules* for declarations in programs.

An alternative way of defining context-sensitive syntax is by giving *inference rules* for well-formed phrases. Well-formedness is usually defined as a binary relation between context-dependent information and phrases, and the well-formedness of a compound phrase may depend on the well-formedness of its subphrases with modified context-dependent information. As with attribute grammars, this technique is not applicable when scope rules are dynamic.

static semantics

Here, let us keep abstract syntax *context-free*, and describe context-sensitive aspects separately. This amounts to treating context-sensitive syntax as a kind of *semantics*, called *static* semantics because it depends only on the program structure, and does not involve program input. The input-dependent behaviour of a program is referred to as its *dynamic* semantics—or simply as its semantics, when this doesn't lead to confusion. Note that static semantics is just as essential an ingredient in a comprehensive language description as dynamic semantics, and that there are significant problems with program portability due to inconsistent implementation of static semantics by compiler front-ends. In this tutorial, however, we are primarily concerned with dynamic semantics.

2.2.2 Semantics

Consider an entire program in some programming language. What is the nature of its semantics?

implementation-independent

Let us restrict our attention to programs in high-level programming languages, which generally deny the program direct control over the details of physical behaviour. The appropriate semantics of these programs is *implementation-independent*, consisting of just those features of program execution that are common to all implementations. This usually includes termination properties, but ignores efficiency considerations.

Thus the semantics of a program is an (abstract) entity that models the program's implementation-independent behaviour. The semantics of a programming language consists of the semantics of all its programs.

2.2.2.1 Semantic Functions

The semantics of a programming language can be captured by a *semantic function* that maps the abstract syntax of each program to the semantic entity representing its behaviour. How about the semantics of *parts* of programs, i.e., of phrases such as statements, declarations, expressions, etc.?

implicit semantics

Well, one could say that the semantics of a phrase is already *implicit* in the semantics of all the programs in which it occurs. Thus two phrases have the same semantics if they are *interchangeable* in any program context, i.e., when replacing the one phrase by the other never affects the behaviour of the whole program. For example, two procedures that implement different algorithms for sorting have the same implicit semantics, provided that program behaviour does not take account of efficiency.

For reasoning about phrases—their semantic equivalence, for instance—it is undesirable to have to consider all possible programs containing them, so we insist on *explicit* definition of semantics for all phrases. When the semantics of a compound phrase depends only on the semantics of its subphrases, not on other features (such as their structure) the semantics is called *compositional*. This guarantees that whenever two phrases have the same semantics, they are indeed interchangeable.

compositional

Action semantics, following denotational semantics, insists on compositionality, with the semantic function mapping not only entire programs but also all their component phrases to semantic entities. The semantics of a phrase thus entirely represents the *contribution* of the phrase to program semantics.

The semantic entities providing the semantics of parts of programs are usually more complex than those representing the behaviour of entire programs. For example, the semantics of a statement not only has to represent its direct contribution to observable program semantics (such as the relation between input and output) but also its *indirect* contribution by means of assignments to variables, etc. Note that usually, it is only the semantics of entire programs that is used to set standard for implementations.

2.2.2.2 Semantic Entities

Semantic entities are used to represent the implementation-independent behaviour of programs, as well as the contributions that parts of programs make to overall behaviour. There are three kinds of semantic entity used in action semantics: *actions*, *data*, and *yielders*. The main kind is, of course, actions; data and yielders are subsidiary. The notation used in action semantics for specifying actions and the subsidiary semantic entities is called, unsurprisingly, *action notation*.

actions

Actions are essentially dynamic, *computational* entities. The *performance* of an action directly represents information processing behaviour and reflects the gradual, step-wise nature of computation. Items of data are, in contrast, essentially static, *mathematical* entities, representing pieces of information, e.g., particular numbers. (Of course actions are 'mathematical' too, in the sense that they are abstract, formally-defined entities, analogous to abstract machines as defined in automata theory.) A yielder represents an *unevaluated* item of data, whose value depends on the *current information*, i.e., the previously-computed and input values that are available to the performance of the action in which the yielder occurs. For example, a yielder might always evaluate to the datum currently stored in a particular cell, which could change during the performance of an action.

data

yielders

Actions

performance

A performance of an action, which may be part of an enclosing action, either:

- *completes*, corresponding to normal termination (the performance of the enclosing action proceeds normally); or
- *escapes*, corresponding to exceptional termination (parts of the enclosing action are skipped until the escape is trapped); or
- *fails*, corresponding to abortion (the enclosing action performs an alternative action, if there is one, otherwise it fails too); or
- *diverges*, corresponding to nontermination (the enclosing action also diverges).

Actions can be used to represent the semantics of programs: action performances correspond to possible program behaviours. Furthermore, actions can represent the (perhaps indirect) contribution that *parts* of programs, such as statements and expressions, make to the semantics of entire programs.

nondeterminism An action may be nondeterministic, having different possible performances for the same initial information. Nondeterminism represents implementation-dependence, where the behaviour of a program (or the contribution of a part of it) may vary between different implementations—or even between different instants of time on the same implementation. Note that nondeterminism does not imply actual randomness: each implementation of a nondeterministic behaviour may be absolutely deterministic.

information classification The information processed by action performance may be classified according to how far it tends to be propagated, as follows:

- *transient*: tuples of data, corresponding to intermediate results;
- *scoped*: bindings of tokens to data, corresponding to symbol tables;
- *stable*: data stored in cells, corresponding to the values assigned to variables;
- *permanent*: data communicated between distributed actions.

Transient information is made available to an action for immediate use. Scoped information, in contrast, may generally be referred to throughout an entire action, although it may also be hidden temporarily. Stable information can be changed, but not hidden, in the action, and it persists until explicitly destroyed. Permanent information cannot even be changed, merely augmented.

When an action is performed, transient information is given only on completion or escape, and scoped information is produced only on completion. In contrast, changes to stable information and extensions to permanent information are made *during* action performance, and are unaffected by subsequent divergence, failure, or escape.

facets The different kinds of information give rise to so-called *facets* of actions, focusing on the processing of at most one kind of information at a time:

- the *basic* facet, processing independently of information (control flows);
- the *functional* facet, processing transient information (actions are *given* and *give* data);
- the *declarative* facet, processing scoped information (actions *receive* and *produce* bindings);
- the *imperative* facet, processing stable information (actions *reserve* and *unreserve* cells of storage, and *change* the data stored in cells); and
- the *communicative* facet, processing permanent information (actions *send* messages, *receive* messages in buffers, and offer *contracts* to *agents*).

These facets of actions are independent. For instance, changing the data stored in a cell—or even unreserving the cell—does not affect any bindings. There are, however, some *directive* actions, which process a mixture of scoped and stable information, so as to provide finite representations of self-referential bindings. There are also some *hybrid* primitive actions and combinators, which involve more than one kind of information at once, such as an action that both reserves a cell of storage and gives it as transient data.

combinators The standard notation for specifying actions consists of action *primitives*, which may involve yielders, and action *combinators*, which operate on *subactions*.

Data

mathematical *computational*

The information processed by actions consists of items of *data*, organised in structures that give access to the individual items. Data can include various familiar mathematical entities, such as truth-values, numbers, characters, strings, lists, sets, and maps. It can also include entities with computational origins, such as tokens, cells, and agents, used for accessing other items, and some compound entities with data components, such as messages and contracts. Actions themselves are not data, but they can be incorporated in so-called *abstractions*, which are data, and subsequently *enacted* back into actions. (Abstraction and enaction are a special case of so-called *reification* and *reflection*.) New kinds of data can be introduced *ad hoc*, for representing special pieces of information.

Yielders

evaluation

Yielders are entities that can be *evaluated* to yield data during action performance. The data yielded may depend on the current information, i.e., the given transients, the received bindings, and the current state of the storage and buffer. In fact action notation provides primitive yielders that evaluate to compound data (tuples, maps, lists) representing entire slices of the current information, such as the current state of storage. Evaluation cannot affect the current information.

data operations

Compound yielders can be formed by the application of data operations to yielders. The data yielded by evaluating a compound yielder are the result of applying the operation to the data yielded by evaluating the operands. For instance, one can form the sum of two number yielders. Items of data are a special case of data yielders, and always yield themselves when evaluated.

2.3 Meta-Notation

modules

A semantic description consists of three main parts, concerned with specifying abstract syntax, semantic functions, and semantic entities. We specify these parts as separate *modules*, which, in larger specifications, may themselves be divided into *submodules*, just as we normally divide technical reports and textbooks into sections and subsections.

reference *inheritance*

Let us adopt the following discipline in our modular specifications: each module has to have a *title*, and it has to be *self-contained*, in the sense that all the notation used in a module must be specified there too. Of course, when some module M uses notation that is already specified in another module M', there is no point in repeating all of M' literally in M: it is sufficient to *refer* to M' from M, using its title. Similarly, when the submodules M_i of a module M use some common notation, we may as well specify that notation just once, at the level of M, letting it be *inherited* by each M_i. A reference to a module M thus provides not only the notation specified directly in M and its submodules, but also that which is specified in modules referenced by M and in supermodules enclosing M.

The modular structure of an action semantic description is formally specified as follows.

Abstract Syntax .

Semantic Functions

needs: **Abstract Syntax**, **Semantic Entities**.

Semantic Entities .

Our notation for indicating modular structure is intended to be unobtrusive, and most of the time we shall disregard the modularisation and focus on what is specified in the bodies of the modules.

2.3.1 Abstract Syntax

Now let us consider how to specify abstract syntax. Reference manuals for programming languages generally use formal context-free *grammars*, augmented by some form of regular expressions, to specify *concrete* syntax. A formal grammar consists of a set of *productions*, involving *terminal* symbols, which may be characters or strings, as well as auxiliary *nonterminal* symbols. Formal grammars have excellent pragmatic properties, such as readability and modifiability; let us adapt them for specifying abstract syntax.

SIMPLE

As an illustration, we shall consider the abstract syntax for a SIMPLE, impractical language, involving some basic statements and expressions. The main virtue of this little example is that it nicely demonstrates the use of formal notation in action semantic descriptions, without the distracting (and sometimes tedious) details that arise when describing the semantics of realistic programming languages. Moreover, the SIMPLE language is a familiar example to readers who have studied previous frameworks for semantic description, such as denotational semantics, and most programmers will be able to recognise it as a much-restricted version of some high-level programming language that they have used. Hence we need not bother here to give an informal explanation of the intended semantics of SIMPLE, but focus our attention entirely on the *form* of its action semantic description.

Consider the grammar-like specification given in Section 1.1. The module title is **Abstract Syntax**; the module body consists mainly of a set of (numbered) equations. Ignoring the double brackets ⟦ ... ⟧ for now, equations (3) and (4) have the same form as *productions* in a particular variant of BNF grammar—one that is commonly used for specifying concrete syntax in reference manuals, such as the ISO standard for PASCAL. Terminal symbols are written as quoted strings of characters, such as "(" and "or". Nonterminal symbols are written as unquoted words, such as Expression, and we adopt the convention that they generally start with a capital letter, to avoid confusing them with symbols for semantic functions and entities, which we write using lower case letters. The alternatives for each nonterminal symbol are started by =, separated by |, and terminated by a period.

grammar

symbols

regular expressions

The other equations involve so-called *regular expressions*. In our notation, a regular expression is either a single symbol, or it consists of a *sequence* $\langle R_1 \ldots R_n \rangle$, a grouped set of *alternatives* $(R_1 \mid \ldots \mid R_n)$, an *optional* part $R^?$, an *optional repeatable* part R^*, or an *obligatory repeatable* part R^+.

The standard nonterminals digit and letter are always implicitly specified in our grammars, for convenience when specifying the lexical syntax of numerals and identifiers. The terminal symbols that they generate are single characters, rather than strings of characters. (We make a distinction between a character, such as '0', and the string consisting of just that character, "0".)

In equation (1) in Section 1.1, the second alternative ⟦ "if" Expression "then" Statement ⟨ "else" Statement ⟩$^?$ ⟧ is equivalent to the following two alternatives:

⟦ "if" Expression "then" Statement ⟧ |
⟦ "if" Expression "then" Statement "else" Statement ⟧.

The use of such optional parts can often make grammars much more concise.

We could specify the abstract syntax of Statements in various ways. That shown in equation (2) earlier, namely:

Statements = ⟨ Statement ⟨ ";" Statement ⟩* ⟩

corresponds closely to the way that the *concrete* syntax of statements is usually specified in reference manuals. The usual style in texts on semantics corresponds to what we could write as:

$$\mathsf{Statements} = \mathsf{Statement} \mid [\![\ \mathsf{Statements}\ \text{“;”}\ \mathsf{Statements}\ ]\!] \ .$$

This is similar to the way we have specified the abstract syntax of expressions. We then say that the nonterminal symbol Statements is *recursive*, both to the left and right. The choice of style doesn't matter much in the case of statements, because grouping here is semantically irrelevant. It would matter if we were using grammars to specify *concrete* syntax: grammars with left and right recursive nonterminals are always *ambiguous*, which makes them inappropriate as a basis for parsing—unless disambiguation rules are specified as well.

ambiguity

We have considered the form of the grammars that we use to specify abstract syntax. Let us omit the precise formal interpretation of a grammar as an *algebraic specification of sorts of trees*; it is enough to know that occurrences of $[\![\ \ldots \ ]\!]$ indicate the construction of nodes of trees, and that strings are simply nodes whose branches are all single characters.

sort equations

Notice that numerals and identifiers are *special cases* of expressions, rather than merely occurring as components of expressions. We do not indicate formally which nonterminal corresponds to entire *programs*, because we give the semantics of all phrases anyway. In the SIMPLE language, we may assume that a program consists of a single statement.

That concludes the explanation of how to specify abstract syntax for use in action semantic descriptions.

2.3.2 Semantic Functions

semantic equations

In action semantics, we specify semantic functions by *semantic equations*. Each equation defines the semantics of a particular sort of phrase in terms of the semantics of its components, if any, using constants and operations for constructing semantic entities. The required compositionality of semantic functions is generally apparent from the semantic equations.

inductive definition

Mathematically, a set of semantic equations is simply an inductive definition of maps from syntax to semantics. Those familiar with algebraic semantics may understand the equations as a presentation of a target algebra; then the unique homomorphism to the target algebra from the initial algebra of abstract syntax corresponds to the semantic functions. Programmers may prefer to regard semantic equations as a definition of mutually-recursive functions by cases, as allowed, for instance, in the functional programming language HASKELL.

translation

It is also possible to view a set of semantic equations as merely specifying a *translation* from programming language syntax to *notation* for semantic entities. But it is the semantic entities themselves that count, not the notation in which they are expressed: two phrases may get translated differently yet still have the same semantics.

As an illustration, consider the action semantics of SIMPLE. The semantic equations that define the required semantic functions are shown in Section 1.2 in their entirety. The various symbols used in the right hand sides of the equations will be explained in Chapter 3. Most of them are rather suggestive of their intended operational interpretation, except perhaps for the combination unfolding...unfold..., which represents iteration. But for now, we are concerned only with the *form* of semantic equations, not with the semantics that they specify.

A semantic function always takes a single, syntactic argument and gives a semantic entity as result. The symbols used to denote semantic functions may be chosen freely; in this tutorial we try to maximise their suggestiveness, at the expense of conciseness, but this is not obligatory. Each symbol generally consists of several words, e.g., the operation-result of _ , and the place-holder _ indicates where the argument goes.

functionality

It is usual to specify the *functionality* of each semantic function. For instance, evaluate _ :: Expression $\rightarrow$ action [giving a value] asserts that for every abstract syntax tree E for an expression, the semantic entity evaluate E is an action which, when performed, gives a value. The actual definition of evaluate E by the semantic equations is then required to be consistent with this.

formality

The right hand sides of the semantic equations involve the standard notation for actions and data provided by action semantics. It must be emphasised that the notation is *absolutely formal*! The fact that it is possible to read it informally—and reasonably fluently—does not preclude reading it formally as well. The grouping of the symbols might not be completely obvious to those who have not seen action notation before, but it is in fact unambiguous, taking into account a uniform precedence rule whereby infix operations have weaker precedence than prefixes, which in turn are weaker than postfixes.

precedence

Each semantic equation defines the result of applying a particular semantic function to any abstract syntax tree whose root node has the indicated form, in terms of applying (perhaps different) semantic functions to the branches of the node. For instance, equation (2) in Section 1.2 defines the application of the semantic function execute _ to nodes with four branches, where the first branch is the string "if", the second an expression E, the third "then", and the last branch is a statement S, in terms of evaluate E and execute S. Even though the presence of "if" as the first branch of a valid abstract syntax tree implies that the second branch is an expression and the fourth a statement, we prefer to specify this relation explicitly, as shown in equation (2). Readers familiar with denotational semantics should observe that we do *not* write $[\![\ E\ ]\!]$ and $[\![\ S\ ]\!]$ on the right of the equation, since E and S by themselves already stand for trees. Nevertheless, we can still regard $[\![\ \ldots\ ]\!]$ informally as separating syntactic symbols from semantic symbols in the left hand side of a semantic equation.

$[\![\ldots]\!]$

Since we allow regular expressions in our grammars, we have to be able to define semantic functions on nodes with an unbounded number of branches. Our technique for this is illustrated in equations (5) and (6) in Section 1.2. The use of the angle brackets $\langle \ldots \rangle$ instead of $[\![\ \ldots]\!]$ in equation (6) indicates that we are defining a function on a *tuple* of nodes, rather than on a single node. Tupling here is associative, i.e., $\langle\ x\ \langle\ y\ z\ \rangle\ \rangle$ is the same as $\langle\ \langle\ x\ y\ \rangle\ z\ \rangle$, so when tuple notation is used in the left hand side of a semantic equation, we have to ensure it cannot be matched in more than one way to a tuple argument. In equation (6) we insist that S_1 should match a *single* statement, whereas S_2 may match any sequence of statements; we could just as well interchange the restrictions. Incidentally, $\langle\ x\ \rangle$ is the same as x, regardless of whether x is itself a tuple or a node, and the empty tuple may be written as $\langle\ \rangle$.

$\langle \ldots \rangle$

short cuts

Sometimes, the semantics of a particular sort of node is uniform, i.e., independent of its immediate structure. Then the argument of the semantic function may be a single variable, restricted to the corresponding sort of node, and the right hand side of the semantic equation may involve the application of a different semantic function to the variable. See equations (7) and (8) for examples. When the semantics of a syntactic construct is that construct itself, considered as a semantic entity, as is the case for identifiers in the example, we do not need to write a semantic function. In fact that is the case for numerals too: decimal _ is a *standard* operation mapping strings[4] to natural numbers, and N is its own semantics! Of course, we

[4]Strings are regarded as both syntactic *and* semantic entities.

could define a semantic function corresponding to decimal _ rather easily, but here we keep the example as short as possible.

consistency, completeness

Each semantic function must be defined *consistently* and *completely* on the sort of abstract syntax tree for which it is required. Thus any tree of that sort must match the pattern in the left hand side of precisely one semantic equation for that function. When the right hand sides of equations involve applications of semantic functions only to branches of the tree matching the left hand side, well-definedness is ensured; otherwise, one has to check that no direct circularities are involved.

To see *how* our semantic equations define the semantics of statements, the reader might like to use them to work out the action corresponding to the SIMPLE statement if x $\neq$ 0 then x := x + 1. Starting from the application of execute _ to the abstract syntax tree of the statement, each application of a semantic function to a tree is replaced by the right hand side of the matching equation, with the variables E, S, etc., replaced by the corresponding subtrees, until no semantic functions remain. Clearly, the final result is independent of the order in which replacements are made.

2.3.3 Semantic Entities

To complete the action semantic description of SIMPLE programs, the notation that is used in the semantic equations for expressing semantic entities is specified. In fact the standard notation provided by action semantics already includes notation for all the actions we need, so we refer to the module that specifies the standard notation.[5]

actions

symbols

The standard symbols used in action notation are ordinary English *words*. In fact action notation mimics natural language: terms standing for actions form imperative verb phrases involving conjunctions and adverbs, e.g., check it and then escape, whereas terms standing for data and yielders form noun phrases, e.g., the items of the given list. Definite and indefinite articles can be exploited appropriately, e.g., choose a cell then reserve the given cell. (This feature of action notation is reminiscent of Apple's HYPERCARD scripting language HYPERTALK [6], and of COBOL.)

grouping

These simple principles for choice of symbols provide a surprisingly grammatical fragment of English, allowing specifications of actions to be made fluently readable—without sacrificing formality at all! To specify grouping unambiguously, we may use parentheses, but for large-scale grouping it is less obtrusive to use indentation, which we emphasise by vertical rules, as illustrated in the semantic equations for statements given in Section 1.2.

conciseness

Compared to other formalisms, such as the so-called λ-*notation*, action notation may appear to lack conciseness: each symbol generally consists of several letters, rather than a single sign. But the comparison should also take into account that each action combinator usually corresponds to a complex pattern of applications and abstractions in λ-notation. For instance, (under the simplifying assumption of determinism!) the action term 'A_1 then A_2' might correspond to something like $\lambda\varepsilon_1.\lambda\rho.\lambda\kappa.A_1\varepsilon_1\rho(\lambda\varepsilon_2.A_2\varepsilon_2\rho\kappa)$. In any case, the increased length of each symbol seems to be far outweighed by its increased perspicuity. It would also be rather misleading to use familiar mathematical signs to express actions, whose essence is unashamedly computational. For some applications, however, such as formal reasoning about program equivalence on the basis of their action semantics, optimal conciseness may be highly desirable, and it would be appropriate to use abbrevia-

[5]The use of **includes** makes the notation specified in the module **Action Notation** available not only in the current module, **Semantic Entities**, but also in the module that refers to it, **Semantic Functions**. Had we used **needs** instead of **includes**, we would have to refer to **Action Notation** explicitly from **Semantic Functions** as well.

tions for our verbose symbols. The choice of abbreviations is left to the discretion of the user. Such changes of symbols do not affect the *essence* of action notation, which lies in the standard primitives and combinators, rather than in the standard verbose symbols.

casual reading

The informal appearance and suggestive words of action notation should encourage programmers to read it, at first, rather casually, in the same way that they might read reference manuals. Having thus gained a broad impression of the intended actions, they may go on to read the specification more carefully, paying attention to the details. A more cryptic notation might discourage programmers from reading it altogether.

information

The intended interpretation of the standard notation for actions is specified operationally, once and for all, in [15]. All that one has to do before using action notation is to specify the information that is to be processed by actions—this may vary significantly according to the programming language being described. This may involve *extending* data notation with further sorts of data, and *specialising* standard sorts, using sort equations. Furthermore, it may be convenient to introduce formal *abbreviations* for commonly-occurring, conceptually-significant patterns of notation. Extensions, specialisations, and abbreviations are all specified *algebraically*. The specification in Section 1.3, illustrates the use of sort equations to specialise some standard sorts of data, and to specify two nonstandard sorts of data for use in the semantic equations, namely value and number.

sort equations

We use the same symbol | for *sort union* as we used for combining alternatives in grammars. Thinking of sorts of data as *sets* (which isn't quite right, but close enough for now) we may regard | as ordinary set union; it is associative, commutative, idempotent,[6] and its unit is the empty sort, written nothing. Although sort equations look a bit like the so-called domain equations used in denotational semantics, their formal interpretation is quite different. As well as equations we may specify sort inclusions, as in equation (6) in Section 1.3; in fact the use of $\Box$ in equation (1) also expresses an inclusion, as it leaves open what other sorts might be included in datum. Note that we can also express sort *intersection*, using the operation _&_. Also, individual values can be used as if they are the corresponding singleton sorts.

Let us not dwell on the intended interpretation of the semantic entities specified in Section 1.3, since the purpose of this section is merely to explain the form of action semantic descriptions, and the details of our simple example are not particularly interesting in themselves.

[6] Idempotency of _ | _ means $X \mid X = X$.

Chapter 3

Illustration

Now that we have considered the main concepts underlying action semantics, let us take a walk through an illustrative description, explaining all the notation that it uses as we go along.

The language used here to illustrate action semantic descriptions is essentially a sublanguage of ADA, which should give it some flavour of reality, even though it is far from an industrial-scale example. A somewhat larger sublanguage of ADA is described in [15, Appendix A], including: compound values, variables, and types; pointers ('accesses' in ADA terminology); packages; named actual parameters; separate subprogram body definitions; entry selection statements; and function definitions.

The modular structure of our action semantic description is formally specified as follows.

Abstract Syntax

Expressions	.
Statements	**needs: Expressions, Declarations.**
Declarations	**needs: Expressions, Statements.**
Programs	**needs: Expressions, Declarations.**

Semantic Functions needs: Abstract Syntax, Semantic Entities.

Expressions	.
Statements	**needs: Expressions, Declarations.**
Declarations	**needs: Expressions, Statements.**
Programs	**needs: Expressions, Declarations.**

Semantic Entities .

Note that the specification of Semantic Entities has been moved to the appendix, for convenience.

3.1 Abstract Syntax

The abstract syntax of the language to be described is specified as follows. The strings used as labels in nodes are sufficiently suggestive of ADA concrete syntax that informal explanation of the various constructs should be superfluous.

grammar:

3.1.1 Expressions

(1) Identifier = ⟦ letter (letter | digit)* ⟧ .

(2) Literal = ⟦ digit$^+$ ⟧ | ⟦ digit$^+$ '.' digit$^+$ ⟧ .

(3) Expression = Literal | Identifier | ⟦ "(" Expression ")" ⟧ |
⟦ Unary-Operator Expression ⟧ |
⟦ Expression Binary-Operator Expression ⟧ |
⟦ Expression Control-Operator Expression ⟧ .

Note that literals and identifiers are *special cases* of expressions, rather than merely occurring as components of expressions.

We make no attempt to distinguish syntactically between expressions according to the sort of entity to which they evaluate (truth-values or numbers). Such distinctions between expressions would not simplify the semantic description at all, and they would in any case be context-dependent.

(4) Expressions = ⟨ Expression ⟨ "," Expression ⟩* ⟩ .

As we do not consider function calls or array component selections, expression lists are only used in procedure calls.

(5) Unary-Operator = "+" | "−" | "abs" | "not" .

(6) Binary-Operator = "+" | "−" | "*" | "/" | "mod" | "rem" | "=" | "/=" |
"<" | "<=" | ">" | ">=" | "and" | "or" | "xor" .

(7) Control-Operator = ⟨ "and" "then" ⟩ | ⟨ "or" "else" ⟩ .

The distinction between control operators and binary operators is semantically relevant, since the intended order of evaluation of their operands is different.

3.1.2 Statements

(1) Statement = ⟦ "null" ";" ⟧ | ⟦ Identifier ":=" Expression ";" ⟧ |
⟦ "if" Expression "then" Statement$^+$ ⟨ "else" Statement$^+$ ⟩$^?$ "end" "if" ";" ⟧ |
⟦ ⟨ "while" Expression ⟩$^?$ "loop" Statement$^+$ "end" "loop" ";" ⟧ | ⟦ "exit" ";" ⟧ |
⟦ ⟨ "declare" Declaration$^+$ ⟩$^?$ "begin" Statement$^+$ "end" ";" ⟧ |
⟦ Identifier ⟨ "(" Expressions ")" ⟩$^?$ ";" ⟧ | ⟦ "return" ";" ⟧ |
⟦ Identifier "." Identifier ";" ⟧ |
⟦ "accept" Identifier ⟨ "do" Statement$^+$ "end" ⟩$^?$ ";" ⟧ .

(2) Block = ⟨ Declaration* "begin" Statement$^+$ "end" ⟩ .

A block is essentially a statement with some local declarations. Following ADA, blocks can occur directly in ordinary statement sequences, whereas in PASCAL, for example, they can only occur in subprogram declarations.

3.1.3 Declarations

(1) Declaration = [[Identifier ":" "constant" Identifier$^?$ ":=" Expression ";"]] |
[[Identifier ":" Identifier ⟨ ":=" Expression ⟩$^?$ ";"]] |
[["procedure" Identifier ⟨ "(" Formals ")" ⟩$^?$ "is" Block ";"]] |
[["task" Identifier "is" Entry$^+$ "end" ";"]] |
[["task" "body" Identifier "is" Block ";"]] .

(2) Entry = [["entry" Identifier ";"]] .

Task heads are supposed to be declared before the corresponding bodies, although we don't bother to insist on this in our grammar above. We retain the entries of a task head only for the sake of familiarity, as they are irrelevant to our dynamic semantics.

(3) Formal = [[Identifier ":" Mode$^?$ Identifier]] .

(4) Formals = ⟨ Formal ⟨ ";" Formal ⟩* ⟩ .

(5) Mode = "in" | ⟨ "in" "out" ⟩ | "out" .

Following ADA, parameterless procedures omit the parentheses as well as the formals, and a missing formal parameter mode is equivalent to the mode "in".

3.1.4 Programs

(1) Program = [[Declaration$^+$ Identifier]] .

Let us assume that the identifier of a program indicates a main procedure, without parameters, to be called when the program is run.

closed.

That concludes the illustration of how to specify abstract syntax for use in action semantic descriptions.

3.2 Semantic Functions

Now let us begin to define the semantic functions for our illustrative language. We declare the symbols used for the semantic functions at the start of each main module.

3.2.1 Expressions

introduces: the token of _ , the value of _ , evaluate _ ,
the unary-operation-result of _ , the binary-operation-result of _ , moderate _ .

3.2.1.1 Identifiers

- the token of _ :: Identifier → token .

Tokens have a standard usage in action notation: they get bound to data. The sort token is specified in Section A.1 to be a subsort of strings.

(1) the token of I:Identifier = uppercase I .

Following ADA, let us canonicalise identifiers by converting all letters to one case. For languages where case differences are taken seriously, the semantic function the token of _ would simply be the identity function on identifiers, and then we could omit it altogether. The operation uppercase _ on strings comes from our standard data notation.

3.2.1.2 Literals

- the value of _ :: Literal $\longrightarrow$ number .

The sort number is specified in Section A.5 to be a sort including both integer and (approximate) real numbers.

(1) the value of $[\![$ d:digit$^+$ $]\!]$ = integer-number of decimal $[\![$ d $]\!]$.

(2) the value of $[\![$ d_1:digit$^+$ '.' d_2:digit$^+$ $]\!]$ =
real-number of the sum of (decimal $[\![$ d_1 $]\!]$,
the product of (decimal $[\![$ d_2 $]\!]$,
the exponent of (decimal "10", the negation of the count of d_2))) .

The operation decimal is a standard data operation on strings; similarly count is the standard data operation that returns the number of components in any sort of tuple. We could define these operations as semantic functions, but it wouldn't be very exciting, so we take this short-cut. Formally, we are regarding a digit sequence as its own semantics, i.e., a string! The use of $[\![$ ... $]\!]$ in the right hand sides of the semantic equations above is atypical. It is needed because decimal expects its argument to be a string, not a tuple of characters.

So much for literals. Now for some action!

3.2.1.3 Evaluating Expressions

- evaluate _ :: Expression $\longrightarrow$ action [giving a value] [using current bindings | current storage] .

Let us be content here with an informal reading of such indications of the sorts of actions, as a thorough explanation involves considering operations that can be applied to entire sorts.

(1) evaluate L:Literal = give the value of L .

The primitive action give Y completes, giving the data yielded by evaluating the yielder Y.

(2) evaluate I:Identifier = give the entity bound to the token of I then
| give the given value or
| give the value assigned to the given variable .

The functional action combination A_1 then A_2 represents ordinary functional composition of A_1 and A_2: the transients given to the whole action are propagated only to A_1, the transients given by A_1 on completion are given only to A_2, and only the transients given by A_2 are given by the whole action. Regarding control flow, A_1 then A_2 specifies normal left-to-right sequencing.

The primitive action give Y fails when Y yields nothing. The yielder the entity bound to T refers to the current binding for the particular token T, provided that there is one, otherwise it yields nothing, causing the giving action to fail.

The yielder given Y yields all the data given to its evaluation, provided that this is of the data sort Y. For instance the given value (where 'the' is optional) yields a single individual of sort value, if such is given. Otherwise it yields nothing, and give the given value thus fails. This causes the alternative currently being performed to be abandoned and, if possible, some other alternative to be performed instead, i.e., *back-tracking*.

The action A_1 or A_2 represents implementation-dependent choice between alternative actions, although here A_1, A_2 are such that one or the other of them is always bound to fail, so the choice is deterministic. The yielder the value assigned to Y refers to the current storage for the particular variable yielded by Y, analogously to the entity bound to T. If I is currently bound to an entity that is neither a value nor a variable (e.g., a procedure) both alternatives fail, causing their combination to fail as well.

(3) evaluate ⟦ "(" E:Expression ")" ⟧ = evaluate E .

(4) evaluate ⟦ O:Unary-Operator E:Expression ⟧ =
evaluate E then give the unary-operation-result of O .

(5) evaluate ⟦ E_1:Expression O:Binary-Operator E_2:Expression ⟧ =
(evaluate E_1 and evaluate E_2) then give the binary-operation-result of O .

The action A_1 and A_2 represents implementation-dependent order of performance of the indivisible subactions of A_1, A_2. When these subactions cannot 'interfere' with each other, as here, it indicates that their order of performance is simply irrelevant. Left-to-right order of evaluation can be specified by using the combinator A_1 and then A_2 instead of A_1 and A_2 above. In both cases, the values given by the subactions get tupled, and subsequently passed on by the combinator A_1 then A_2.

The evaluation of an expression may give any individual of sort value. We leave it to the semantics of operators, specified below, to insist on individuals of particular sorts—numbers, for instance. For simplicity, we do not bother with precise error messages in case the given operands are *not* of the right sort for a particular operator: we merely let the application of the corresponding operation yield nothing, so that the action which gives it must fail. In any case, errors arising due to wrong sorts of operands are statically detectable in most languages, and should therefore be the concern of a static semantic description, not of the dynamic semantics that we are developing here.

(6) evaluate ⟦ E_1:Expression "or" "else" E_2:Expression ⟧ =
evaluate E_1 then
| | check the given truth-value then give true
| or
| | check not the given truth-value then evaluate E_2 .

(7) evaluate ⟦ E_1:Expression "and" "then" E_2:Expression ⟧ =
evaluate E_1 then
| | check the given truth-value then evaluate E_2
| or
| | check not the given truth-value then give false .

The action check Y requires Y to yield a truth-value; it completes when the value is true, otherwise it fails. It is used for guarding alternatives. For instance, (check Y then A_1) or (check not Y then A_2) expresses a deterministic choice between A_1 and A_2, depending on the condition Y.

3.2.1.4 Operating Unary Operators

- the unary-operation-result of _ :: Unary-Operator → yielder [of value] [using given value] .

Assuming that applications of operators to operands should never diverge or escape, we may represent the semantics of an operator as a yielder. Otherwise, we could use actions here too. But note that we cannot let the semantics of an operator be simply an algebraic operation, since our meta-notation is first-order.

(1) the unary-operation-result of "+" = the given number .

(2) the unary-operation-result of "−" = the negation of the given number .

(3) the unary-operation-result of "abs" = the absolute of the given number .

(4) the unary-operation-result of "not" = not the given truth-value .

Numerical operations such as negation and absolute are specified (loosely) in Section A.5. The truth-values are the standard ones from data notation, equipped with the usual logical operations, such as not.

3.2.1.5 Operating Binary Operators

- the binary-operation-result of _ :: Binary-Operator → yielder [of value] [using given value^2] .

(1) the binary-operation-result of "+" = the sum of (the given number#1, the given number#2) .

The yielder given $Y\#n$ yields the n'th individual component of a given tuple, for $n > 0$, provided that this component is of sort Y.

(2) the binary-operation-result of "−" = the difference of (the given number#1, the given number#2) .

(3) ...

(4) the binary-operation-result of "xor" = not (the given truth-value#1 is the given truth-value#2) .

3.2.1.6 Moderating Expressions

- moderate _ :: Expressions → action [giving argument$^+$]
  [using given mode$^+$ | current bindings | current storage] .

The evaluation of actual parameter expressions in procedure calls is dependent on the modes of the corresponding formals in procedure declarations. The semantic entities corresponding to syntactic modes are specified in Section A.6.

(1) moderate I:Identifier = give the first of the given mode$^+$ then
| | check either (it is the reference-mode, it is the copy-mode) then
| | give the variable bound to the token of I
| or
| | check (it is the constant-mode) then evaluate I .

The tuple selector operation first is standard in data notation, as is rest, used below. The yielder 'it' formally abbreviates the given datum; here, we could write the given mode instead.

(2) E & Identifier = nothing ⇒
moderate E:Expression = give the first of the given mode$^+$ then
check (it is the constant-mode) then evaluate E .

Here we collapse the uniform semantic equations for the remaining Expression constructs into a single conditional equation whose condition holds when E is abstract syntax *not* of sort Identifier. (The meta-notation doesn't provide sort difference, so we have to use intersection instead.)

(3) moderate ⟨ E_1:Expression "," E_2:Expressions ⟩ =
moderate E_1 and (give the rest of the given mode$^+$ then moderate E_2) .

Whereas the data flow in A_1 then A_2 is analogous to that in ordinary function composition $g \circ f$ (at least when the functions are strict) the data flow in A_1 and A_2 is analogous to so-called *target-tupling* of functions, sometimes written $[f, g]$ and defined by $[f, g](x) = (f(x), g(x))$. That is, both A_1 and A_2 are given the same transient data as the combination. Note that the tupling of the data given by A_1 and A_2 is associative.

3.2.2 Statements

introduces: execute _ .

3.2.2.1 Executing Statements

- execute _ :: Statement$^+$ $\rightarrow$ action
  [completing | escaping with an escape-reason | diverging | storing | communicating]
  [using current bindings | current storage | current buffer] .

(1) execute $\langle$ S_1:Statement S_2:Statement$^+$ $\rangle$ = execute S_1 and then execute S_2 .

The basic action combination A_1 and then A_2 combines the actions A_1, A_2 into a compound action that represents their normal, left-to-right sequencing, performing A_2 only when A_1 completes.

Note that the semantics of a statement is well-defined, because the above semantic equation can only match a statement sequence in one way.

(2) execute $[\![$ "null" ";" $]\!]$ = complete .

The primitive action complete is the unit for A_1 and then A_2.

(3) execute $[\![$ I:Identifier ":=" E:Expression ";" $]\!]$ =
| give the variable bound to the token of I and evaluate E
then assign the given value#2 to the given variable#1 .

The action assign Y_1 to Y_2 is specified in Section A.3.

(4) execute $[\![$ "if" E:Expression "then" S:Statement$^+$ "end" "if" ";" $]\!]$ =
evaluate E then | | check the given truth-value and then execute S
| or
| | check not the given truth-value .

Since check D doesn't give any data, and execute S doesn't refer to given data, it doesn't make any difference whether we use A_1 and then A_2 or A_1 then A_2 to combine these actions above.

It is important not to omit 'or check not the given truth-value' above, for then the execution of an if-then statement with a false condition would fail, rather than simply completing.

(5) execute $[\![$ "if" E:Expression "then" S_1:Statement$^+$ "else" S_2:Statement$^+$ "end" "if" ";" $]\!]$ =
evaluate E then | | check the given truth-value and then execute S_1
| or
| | check not the given truth-value and then execute S_2 .

(6) execute $[\![$ "loop" S:Statement$^+$ "end" "loop" ";" $]\!]$ =
unfolding (execute S and then unfold)
trap
| | check there is given an exit
| or
| | check there is given a procedure-return and then escape with it .

The action combination unfolding A performs A but whenever it reaches the dummy action unfold, it performs A instead. It is mostly used in the semantics of iterative constructs, with unfold occurring exactly once in A, but it can also be used with several occurrences of unfold.

The action A_1 trap A_2 sets A_2 as the trap action to be performed when A_1 escapes. Once an escape has been trapped, normal sequencing is resumed.

The primitive action escape with Y terminates abnormally, giving the data yielded by Y to the action that traps the escape (if any). An exit is simply a data item, as is a procedure-return. The data operation there is d results in true when d is a data item other than nothing.

(7) execute ⟦ "while" E:Expression "loop" S:Statement$^+$ "end" "loop" ";" ⟧ =
unfolding | evaluate E then | | check the given truth-value and then execute S and then unfold
or
| check not the given truth-value
trap
| | check there is given an exit
or
| check there is given a procedure-return and then escape with it .

(8) execute ⟦ "exit" ";" ⟧ = escape with an exit .

(9) execute ⟦ "begin" S:Statement$^+$ "end" ";" ⟧ = execute S .

(10) execute ⟦ "declare" B:⟨ Declaration$^+$ "begin" Statement$^+$ "end" ⟩ ";" ⟧ = execute B .

A declare statement is essentially a block. Some languages, Pascal for instance, don't allow such anonymous blocks to occur in the middle of a statement sequence, only directly in other declarations, but the extra generality here doesn't complicate the semantic description noticeably.

(11) execute ⟦ I:Identifier ";" ⟧ =
enact the procedure-abstraction of the parameterless procedure bound to the token of I .

The action enact Y performs the action incorporated in the abstraction yielded by Y. The performance of the incorporated action is not given any transient data, nor does it receive any bindings. However, transients and/or bindings may have already been supplied to the incorporated action, using the notation for yielders explained later.

The notation for constructing procedure entities from abstractions is specified in Section A.6.

(12) execute ⟦ I:Identifier "(" E:Expressions ")" ";" ⟧ =
give the procedure bound to the token of I then
| give the procedure-abstraction of the given procedure and
| | give the formal-modes of the given parameterized procedure then
| | moderate E
then enact the application of the given abstraction#1
to the argument$^+$ yielded by the rest of the given data .

Suppose that Y_1 yields abstraction of A, and that Y_2 yields data d. Then the yielder application Y_1 to Y_2 evaluates to abstraction of (give d then A).

(13) execute ⟦ "return" ";" ⟧ = escape with a procedure-return .

(14) execute ⟦ I_1:Identifier "." I_2:Identifier ";" ⟧ =
give the task-agent bound to the token of I_1 then
| send a message [to the given task-agent] [containing entry of the token of I_2] and then
| receive a message [from the given task-agent] [containing the done-signal] .

The primitive action send Y, where Y yields a *sort* of message, initiates the transmission of a message. The usual form of Y is a message [to Y_1] [containing Y_2], where Y_1 and Y_2 are individuals. The sort yielded by Y is implicitly restricted to messages from the performing agent, with the next local serial number, and this should determine an individual message.

The action receive Y waits indefinitely for a message of the sort specified by Y to arrive, removes it from the buffer, and gives it.

The notation for entries and signals that are contained in the messages is specified in Section A.7.

(15) execute ⟦ "accept" I:Identifier ";" ⟧ =
receive a message [from any task-agent] [containing entry of the token of I] then
send a message [to the sender of the given message] [containing the done-signal] .

(16) execute ⟦ "accept" I:Identifier "do" S:Statement$^+$ "end" ";" ⟧ =
receive a message [from any task-agent] [containing entry of the token of I] then
| execute S and then
| send a message [to the sender of the given message] [containing the done-signal] .

3.2.2.2 Executing Blocks

- execute _ :: Block → action [escaping with an escape-reason | diverging | storing | communicating]
[using current bindings | current storage | current buffer] .

(1) execute ⟨ "begin" S:Statement$^+$ "end" ⟩ = execute S .

(2) execute ⟨ D:Declaration$^+$ "begin" S:Statement$^+$ "end" ⟩ =
furthermore elaborate D hence
| | synchronize D and then execute S and then relinquish D
| trap
| | relinquish D and then escape with the given escape-reason .

The action furthermore A produces the same bindings as A, together with any received bindings that A doesn't override. In other words, it overlays the received bindings with those produced by A.

The combination A_1 hence A_2 lets the bindings produced by A_1 be received by A_2, which limits their scope—unless they get reproduced by A_2. It is analogous to functional composition. The compound combination furthermore A_1 hence A_2 (recall that prefixes have higher precedence than infixes!) corresponds to ordinary block structure, with A_1 being the block head and A_2 the block body: nonlocal bindings, received by the combination, are also received by A_2 unless they are overridden by the local bindings produced by A_1.

The use of synchronize D here is concerned with task initialisation, considered later.

Whereas bindings produced by declarations automatically disappear at the end of their scope, locally-declared variables are not thereby automatically *relinquished*; it has to be specified explicitly. Notice that the trap is needed to ensure that exits and procedure returns do not evade relinquish D.

3.2.3 Declarations

introduces: elaborate _ , relinquish _ , synchronize _ , the mode of _ , the modes of _ , actualize _ .

3.2.3.1 Elaborating Declarations

- elaborate _ :: Declaration* → action [binding | diverging | storing | communicating]
[using current bindings | current storage | current buffer] .

(1) elaborate $\langle$ D_1:Declaration D_2:Declaration$^+$ $\rangle$ = elaborate D_1 before elaborate D_2 .

The action A_1 before A_2 represents sequencing of declarations. Like furthermore A_1 hence A_2, it lets A_2 receive bindings from A_1, together with any bindings received by the whole action that are not thereby overridden. The combination produces all the bindings produced by A_2, as well as any produced by A_1 that are not overridden by A_2. Thus A_2 may rebind a token that was bound, or hidden, by A_1. Note that the bindings received by the combination are not reproduced at all, unless one of A_1, A_2 explicitly reproduces them.

The use of the combinator A_1 before A_2 in the semantics of declaration sequences allows later declarations to refer to the bindings produced by earlier declarations—but not the other way round.

(2) elaborate $\langle$ $\rangle$ = complete .

(3) elaborate $[\![$ I_1:Identifier ":" "constant" I_2:Identifier$^?$ ":=" E:Expression ";" $]\!]$ =
evaluate E then bind the token of I_1 to the given value .

The declarative action bind T to Y produces the binding of the token T to the bindable data yielded by Y. It does *not* reproduce any of the received bindings!

(4) elaborate $[\![$ I_1:Identifier ":" I_2:Identifier ";" $]\!]$ =
allocate a variable for the type bound to the token of I_2
then bind the token of I_1 to the given variable .

The action allocate d for Y is *ad hoc*, specified in Section A.3. As we only deal with simple variables in this tutorial, allocate a variable for Y merely chooses, reserves, and gives a single storage cell.

(5) elaborate $[\![$ I_1:Identifier ":" I_2:Identifier ":=" E:Expression ";" $]\!]$ =
| allocate a variable for the type bound to the token of I_2 and evaluate E
then
| bind the token of I_1 to the given variable#1 and
| assign the given value#2 to the given variable#1 .

The basic and functional combinators, such as A_1 and A_2, all pass the *received* bindings to their subactions without further ado—analogously to the way A_1 and A_2 passes all the given data to both A_1 and A_2. They are similarly unbiased when it comes to combining the bindings produced by their subactions: they produce the *disjoint union* of the bindings, providing this is defined, otherwise they simply fail.

(6) elaborate $[\![$ "procedure" I:Identifier "is" B:Block ";" $]\!]$ =
bind the token of I to parameterless procedure of the closure of abstraction of
(execute B trap check there is given a procedure-return) .

When current bindings evaluates to b, closure Y_1 yields abstraction of (produce b hence A).

The use of closure above ensures static bindings: the execution of the block B when the function is called receives the same bindings as the declaration. These bindings, however, do not include that for I itself, so *self-referential*, or *recursive*, calls of the function are not possible. In fact it is easy to allow self-reference: just change 'bind' to 'recursively bind' in the semantics equations for elaborate D. But it is not quite so straightforward to allow *mutual* reference. [15, Appendix A] shows how this can be done, by redirecting so-called *indirect* bindings.

Notice that an enaction of an abstraction bound to a procedure identifier can only complete when the execution of the block escapes, giving a return. If the execution of the block escapes for any other reason, or completes, the enaction fails.

(7) elaborate ⟦ "procedure" I:Identifier "(" F:Formals ")" "is" B:Block ";" ⟧ =
bind the token of I to parameterized modalized (the modes of F,
procedure of the closure of abstraction of
| furthermore actualize F thence
| | | execute B trap check there is given a procedure-return
| | and then copy-back the given map [token to variable]) .

The performance of actualize F above not only *produces* bindings for the formal parameters, it also *gives* a map corresponding to the bindings for copy-mode parameters. This map is exploited to copy back the final values of the local formal parameter variables to the actual parameter variables. The action copy-back Y is *ad hoc*, specified in Section A.6. The combination A_1 thence A_2 passes transients *as well as* bindings from A_1 to A_2.

The operation modalized $(m,\ p)$ attaches the modes m to a procedure p, so that formal-modes of p can obtain them when procedure call statements are executed. This is specified in Section A.6.

(8) elaborate ⟦ "task" I:Identifier "is" E:Entry$^+$ "end" ";" ⟧ =
offer a contract [to any task-agent] [containing abstraction of the initial task-action] and then
| receive a message [containing a task-agent] then
| bind the token of I to the task-agent yielded by the contents of the given message .

The primitive action offer Y, where Y yields a sort of contract, initiates the arrangement of a contract with another, perhaps only partially specified, agent. The usual form of Y is a contract [to an agent] [containing abstraction of A], where A is the action to be performed according to the contract.

The action initial task-action is defined in Section A.7.

(9) elaborate ⟦ "task" "body" I:Identifier "is" B:Block ";" ⟧ =
send a message [to the task-agent bound to the token of I]
[containing task of the closure of abstraction of execute B] .

Executions of task blocks receive all the bindings that were current where their body was declared. These may include bindings to other tasks: a system of communicating tasks can be set up by first declaring all the heads, then all the bodies. They may also include bindings to variables; but attempts to assign to these variables, or to inspect their values, always fail, because the cells referred to are not local to the agent performing the action.

3.2.3.2 Relinquishing Variable Declarations

- relinquish _ :: Declaration$^+$ → action [completing | storing]
[using current bindings | current storage] .

Whereas bindings produced by declarations automatically disappear at the end of their scope, locally-declared variables are not thereby automatically *relinquished*. Here we introduce an extra semantic function on declarations for this purpose.

(1) relinquish ⟨ D_1:Declaration D_2:Declaration$^+$ ⟩ = relinquish D_1 and relinquish D_2 .

(2) relinquish ⟦ I:Identifier ":" I:Identifier X:⟨ ":=" Expression ⟩$^?$ ";" ⟧ =
dispose of the variable bound to the token of I .

(3) D: ⟦ Identifier ":" "constant" Identifier$^?$ ":=" Expression ";" ⟧ |
⟦ "task" Identifier "is" Entry$^+$ "end" ";" ⟧ |
⟦ "function" Identifier ⟨ "(" Formals ")" ⟩$^?$ "return" Identifier "is" Block ";" ⟧ |
⟦ "procedure" Identifier ⟨ "(" Formals ")" ⟩$^?$ "is" Block ";" ⟧ |
⟦ "task" "body" Identifier "is" Block ";" ⟧ ⇒
relinquish D = complete .

3.2.3.3 Synchronizing Task Declarations

- synchronize _ :: Declaration$^+$ → action [completing | diverging | communicating]
  [using current bindings | current buffer] .

The action synchronize D is used to delay the execution of the statements of a block until all the tasks declared in the block have been started.

(1) synchronize ⟨ D_1:Declaration D_2:Declaration$^+$ ⟩ = synchronize D_1 and synchronize D_2 .

(2) synchronize ⟦ "task" "body" I:Identifier "is" B:Block ";" ⟧ =
receive a message [from the task-agent bound to the token of I] [containing the begin-signal] .

(3) D: ⟦ Identifier ":" "constant" Identifier$^?$ ":=" Expression ";" ⟧ |
⟦ Identifier ":" Identifier ⟨ ":=" Expression ⟩$^?$ ";" ⟧ |
⟦ "task" Identifier "is" Entry$^+$ "end" ";" ⟧ |
⟦ "function" Identifier ⟨ "(" Formals ")" ⟩$^?$ "return" Identifier "is" Block ";" ⟧ |
⟦ "procedure" Identifier ⟨ "(" Formals ")" ⟩$^?$ "is" Block ";" ⟧ ⇒
synchronize D = complete .

3.2.3.4 Modes

- the mode of _ :: Mode$^?$ → mode .

(1) the mode of ⟨ ⟩ = the constant-mode .

(2) the mode of "in" = the constant-mode .

(3) the mode of ⟨ "in" "out" ⟩ = the copy-mode .

(4) the mode of "out" = the reference-mode .

3.2.3.5 Modes of Formal Parameters

- the modes of _ :: Formals → mode$^+$.

(1) the modes of ⟦ I_1:Identifier ":" M:Mode$^?$ I_2:Identifier ⟧ = the mode of M .

(2) the modes of ⟨ F_1:Formal ";" F_2:Formals ⟩ = (the modes of F_1, the modes of F_2) .

3.2.3.6 Actualizing Formal Parameters

- actualize _ :: Formals → action [binding | giving a map [token to variable] | storing]
  [using given argument$^+$ | current bindings | current storage] .

The map of tokens to variables given by actualization is used for copying-back the values of copy-mode parameters on procedure return. Maps, together with operations for creating and combining them, are provided by the standard data notation used in action semantics.

(1) actualize ⟦ I_1:identifier ":" M:"in"$^?$ I_2:Identifier ⟧ =
bind the token of I_1 to the value yielded by the first of the given argument$^+$
and give the empty-map.

(2) actualize [[I_1:identifier ":" "out" I_2:Identifier]] =
bind the token of I_1 to the variable yielded by the first of the given argument$^+$
and give the empty-map .

(3) actualize [[I_1:identifier ":" "in" "out" I_2:Identifier]] =
| give the variable yielded by the first of the given argument$^+$
| and allocate a variable for the type bound to the token of I_2
then
| bind the token of I_1 to the given variable#2 and
| give map of the token of I_1 to the given variable#1 and
| assign (the value assigned to the given variable#1) to the given variable#2 .

(4) actualize ⟨ F_1:Formal ";" F_2:Formals ⟩ =
| actualize F_1 and (give the rest of the given argument$^+$ then actualize F_2)
then give the disjoint-union of (the given map#1, the given map#2) .

3.2.4 Programs

introduces: run _ .

- run _ :: Program → action [completing | diverging | storing | communicating]
  [using current storage | current buffer] .

(1) run [[D:Declaration$^+$ I:Identifier]] =
produce required-bindings hence
furthermore elaborate D hence
| synchronize D and then
| | give the procedure bound to the token of I then
| | enact the procedure-abstraction of the given parameterless procedure
| and then send a message [to the user-agent] [containing the terminated-signal] .

The primitive action produce Y produces a binding for each token mapped to a bindable value by the map yielded by Y. See Section A.9 for the definition of the bindings of required identifiers in our illustrative language. The analogous definition for full Ada would be substantially larger!

The termination message sent above insists that the user should be able to notice when the program has terminated; this might be useful when the user runs the program on a remote agent.

To complete our semantic description of the illustrative language, we have to specify the notation that is used in the semantic equations for expressing semantic entities. This is done in Appendix A, which also refers to the standard action notation and data notation used in action semantics [15].

Chapter 4

Action Notation

We started this tutorial by glancing at the action semantic description of a very simple sequential language, without trying to explain it. Then we ran through the motivation for formal semantic descriptions, and presented the basic concepts of action semantics; we also explained the meta-notation that is used in a.s.d.s.

In a full course on action semantics, aiming at giving the reader some proficiency not only in reading a.s.d.s but also in *writing* them, we would then have made a systematic study of action notation; instead, we plunged into a (by text-book standards) medium-scale illustration of the the use of that notation, giving a few hints about the precise intended interpretation of each symbol the first time it was used. This has probably left the reader with a somewhat poor impression of the design of action notation, which is in fact quite systematic. It is high time to explain that design, and to indicate what is so special about it.

check Before proceeding, the reader might like to check two things: that the simple example presented in Chapter 1 is by now very easy to understand; and that in fact the semantic equations of those simple language constructs were largely reused in Chapter 3 without any major modifications at all!

Recall that the information processed by actions is classified according to how it gets propagated: transient (data), scoped (bindings), stable (storage), and permanent (communications). An action may process all these kinds of information at once. By focusing on the processing of (at most) one kind of information at once, we obtain the so-called *facets* of actions. The overall design of action notation is based on keeping these facets independent.

facets

Let us now review all the action combinators, primitives, and yielders that were used in our illustrative example in Chapter 3, grouping them according to the facet for which they are most significant. Then we shall point out the regularity of action notation, and explain why it ensures extensibility of a.s.d.s.

Let A_1, A_2 range over actions, Y over yielders, and D over sorts of data.

Basic The following notation is primarily for specifying control flow:

combinators	*primitives*	*yielders*	*sorts*
A_1 or A_2	fail	the D yielded by Y	action
A_1 and A_2	check Y	$op(Y_1, \ldots, Y_n)$	yielder
A_1 and then A_2	complete		truth-value
A_1 trap A_2	escape		
unfolding A_1	unfold		
indivisibly A_1	enact Y	abstraction of A	abstraction

The binary combinators are all associative, and have units: fail is the unit for _ or _ , complete is the unit not only for _ and _ but also for _ and then _ , and escape is the unit for _ trap _ .

Functional The following notation is primarily for specifying data flow:

combinators	*primitives*	*yielders*	*sorts*
A_1 then A_2	give Y choose Y escape with Y	given $D\#n$ it application Y_1 to Y_2	data

neutrality It is used together with the basic notation, most of which is *neutral* with respect to data flow, i.e.: the combinators let data flow into and out of their subactions (concatenating data when both subactions give some); the primitives give no data. However, A_1 trap A_2 is an exception, its dataflow being analogous to that of A_1 then A_2: when A_1 escapes, any data it gives is passed to A_2; and escape gives all the data given to it. The primitive escape with Y is merely an abbreviation for give Y then escape.

dependency The general principle is that control flow follows data dependency. Thus both A_1 then A_2 and A_1 trap A_2 necessarily have left-to-right performance. When there is no data (or other) dependency between subactions, the control flow depends on the combinator used, e.g., both A_1 and A_2 and A_1 and then A_2 have the same neutral dataflow, but different control flow.

Declarative The following notation is primarily for specifying flow (scopes) of bindings:

combinators	*primitives*	*yielders*	*sorts*
A_1 hence A_2 A_1 before A_2 furthermore A_1	bind Y_1 to Y_2 produce Y	the D bound to Y closure Y	token bindable

It is used together with the basic and functional notation, most of which is neutral with respect to bindings flow, i.e.: the combinators let bindings flow into and out of their subactions (taking disjoint union when both subactions produce some); the primitives produce no bindings. Note that escapes never produce any bindings. Thus the bindings produced by A_1 trap A_2 are those produced by A_1 when it completes, and those produced by A_2 when A_1 escapes.

Control flow follows binding dependency. Thus both A_1 hence A_2 and A_1 before A_2 necessarily have left-to-right performance.

Imperative The following notation is primarily for specifying storage allocation and assignment:

primitives	*yielders*	*sorts*
store Y_1 in Y_2 reserve Y unreserve Y	the D stored in Y	cell storable

The imperative primitive actions neither give data nor produce bindings.

Communicative The following notation is for specifying process activation and communication:

combinators	*primitives*	*yielders*	*sorts*
patiently A_1	offer Y send Y remove Y	contracting-agent performing-agent current buffer	agent sendable message

The communicative primitives neither give data nor produce bindings.

Hybrid The following notation provides abbreviations for useful compound actions:

combinators	*primitives*
A_1 thence A_2	allocate D receive D

The hybrid combinator A_1 thence A_2 has non-neutral flow in more than one facet, it is like A_1 then A_2 regarding data flow, and like A_1 hence A_2 regarding flow of bindings. The primitive hybrid actions above both give data and have some side-effect.

That completes the list of the action combinators, primitives, and yielders that were used in Chapter 3. Incidentally, we have used about 2/3 of the entire action notation in our illustrative example; many programming languages can be described using the same subset, or with the addition of only very few extra primitives. See [15] for a description of the entire standard action notation.

So, what is so good about the design of action notation? Why has does it work so well in practice, seemingly ensuring extensibility (and modifiability)?

crucial feature

The crucial feature of the design is the way that combinators and primitives which are concerned primarily with one facet generally get extended *neutrally* to the other facets. For instance, suppose one has used a combination A_1 `then` A_2 because there is a data dependency of A_2 on A_1; if it happens that some bindings flow to the combination, the neutral extension to the declarative facet means that these bindings automatically get propagated to the subactions. Thus one may insert a reference to a binding in one of the subactions without changing the specified combinator! Another example: suppose one has used A_1 `and` A_2 where, say, both subactions are purely functional and one merely wants to collect the given data, without overspecifying the order of performance; if one modifies the subactions so that they now make changes to storage, or even activate processes and communicate, there is no *need* at all to change the combinator—and if one wishes to keep determinism, by enforcing sequential performance, all that one has to do is to replace '`and`' by '`and then`'!

This feature, which seems to be unique to action notation, appears to be essential for obtaining a high degree of extensibility and modifiability in semantic descriptions, which in turn is essential for scaling up smoothly to industrial-scale applications.

assumptions

There are, however, a couple of underlying assumptions. One is that all the information to be processed by actions can be classified as transient, scoped, stable, or permanent. For instance, if one were to propagate bindings by storing them in cells of storage, one would probably lose some extensibility. Another assumption, in the same vein, is that the desired information processing can be expressed straightforwardly using the standard combinators and primitives, without resorting to the 'simulation' of one facet by another one.

Anyway, the semantics of a wide range of programming languages (functional, imperative, object-oriented, concurrent) can be given without violating these assumptions, so let's not worry about them further in this tutorial.

polymorphism

The extension of action combinators and primitives to all facets gives a kind of *polymorphism* in action notation. It is worth emphasising that it is this polymorphism, rather than any modular superstructure on semantic descriptions, which ensures the good pragmatics of action semantics. Further beneficial aspects of action semantic descriptions include the use of algebraic specifications for abstract data types, and of a flexible modular specification language that allows mutually-dependent modules and inheritance.

Chapter 5

Conclusion

To conclude, we shall consider the relationship between action semantics and other frameworks such as VDM and RAISE—indicating how users of those frameworks might benefit by exploiting some of the ideas of action semantics. Finally, we shall review the current state of development of action semantics with respect to applications and tools.

5.1 Relationships

Action semantics is not the only available framework for giving formal semantic descriptions of programming languages: other frameworks include denotational semantics, VDM, RAISE, and operational semantics. Why should one prefer action semantics—at least for describing full-scale, realistic programming languages?

Let us look briefly at the other frameworks in turn, pointing out their major drawbacks in comparison to action semantics. To make the discussion accessible to those not familiar with the various frameworks, we indicate their main features, and cite primary references. However, a full appreciation of the comparisons requires in-depth familiarity with both action semantics and the other frameworks, as well as a more thorough discussion than can be provided here.

It is necessary to take account of the fact that many potential users of action semantics may be reluctant to leave their present frameworks, because of such factors as the availability of useful tools, investment in education, constraints imposed by collaboration in ESPRIT projects, etc. We shall indicate how one might get at least some of the pragmatic benefits of action semantics by change of specification *style* within each framework.

syntax

Almost all frameworks for formal semantics are based on context-free abstract syntax, although the notations that they use to specify abstract syntax are quite varied. The grammars that we use in action semantic descriptions perhaps have some significant advantages over other ways of specifying abstract syntax. But our main interest here is in the essential differences concerning the specification of *semantics*, once abstract syntax has been specified, so let us not pursue this point further.

formality

Concerning *formality*, the inherent disadvantages of informal semantic descriptions were discussed in Section 2.1. We have seen in Chapter 3 the extent to which action semantics combines the strengths of informal descriptions—readability, comprehensibility, direct relation to programming concepts—with the essential feature of formality. Hence we do not discuss informal descriptions any further.

5.1.1 Denotational Semantics

Denotational semantics was the starting point for the development of action semantics, so it is hardly surprising that there are substantial similarities between the two frameworks, as well as some significant differences.

Both involve semantic functions that map abstract syntax, compositionally, to semantic entities. Semantic functions can also be regarded as syntax-directed translations, but it is better to focus on the semantic entities themselves, rather than on how they are expressed. The semantic entity corresponding to a particular phrase in denotational semantics is called its *denotation*. Both frameworks use semantic equations to define the semantic functions. Even the notation $[\![\ldots]\!]$ for abstract syntax trees in action semantics corresponds closely to the use of these so-called emphatic brackets for separating syntactic from semantic notation in denotational descriptions.

The essential difference between denotational and action semantics concerns the nature of semantic *entities*, and the notation used to express them. Denotational semantics uses *higher-order functions* on so-called Scott-domains as semantic entities, and it uses a rich, typed *λ-notation* to express particular functions, together with the values on which the functions are defined. For a summary of denotational semantics and λ-notation, see for example the chapter in the *Handbook of Theoretical Computer Science* [13]; for a fuller account, Schmidt's book [25] is recommended.

functions

Higher-order functions on Scott-domains have a rich and elegant mathematical theory, which can be useful when one has to prove that particular phrases enjoy certain semantical properties. Unfortunately, however, the functions required to represent the semantics of the usual constructs of programming languages tend to be rather complex. This stems from the fact that the basic operations on functions provided by the λ-notation—composition, tupling, application, abstraction, etc.—do *not* correspond directly to fundamental concepts of programming languages. These operations may be appropriate for representing the semantics of mathematical formalisms, such as Church's λ-calculus, and that of pure, lazy functional programming languages, but they are so far removed from the constructs of conventional imperative and concurrent programming languages that their use to represent the semantics of these constructs requires elaborate *coding* techniques, which can be quite obscure to the lay reader.

complexity

For instance, there are two main techniques for representing the basic concept of sequential execution: strict composition of functions on states, and reverse composition of functions on continuations. In fact the functions composed typically have further arguments, representing context-dependent information, such as the current bindings; this makes the pattern of composition really quite complex, even for representing something so simple as sequencing.

The complexity of purely functional representations makes it difficult to *read* the semantic equations that describe the semantics of a conventional programming language. It is even more difficult to comprehend their operational *implications*—primarily because λ-abstraction corresponds to call-by-name parameter passing, not to call-by-value. But the really serious pragmatic problem with denotational descriptions of practical programming languages concerns their poor *modifiability* and *extensibility*. Not only are concepts such as sequencing represented by *complex* patterns of function composition and application, but also it may be necessary to replace these by radically different (although systematically related) patterns when one modifies or extends the features of the described language. For instance, in all textbooks on denotational semantics so far, the simple semantic equations for expression evaluation that are first presented have to be completely reformulated when declarations and statements are added. In a student text, this may be tolerable, although difficult to motivate. In the middle of developing a semantic description

pragmatic problems

of a realistic programming language, it would certainly be discouraging, to say the least, to have to make large-scale reformulations of previous work.

A conventional denotational semantic description is essentially a largish functional program, written in a rather poor programming style where almost all parts of the program are sensitive to the exact structure of the 'data types' involved. It is possible to alleviate some of the pragmatic problems of denotational semantics by introducing *auxiliary notation* that corresponds to action primitives and combinators, defining the interpretation of this action notation as higher-order functions, using the usual λ-notation, instead of defining it operationally as in [15]. It is instructive to compare this hybrid of denotational and action semantics [14] with the conventional denotational approach [13]. But note that one can go *too far* in this direction: introducing an *ad hoc* auxiliary operation for *each* language construct merely shifts the content of the semantic equations to the definition of the auxiliary operations, without achieving any improvements in pragmatics.

auxiliary notation

The use of an adequate auxiliary notation essentially hides the detailed structure of the underlying data types from the semantic equations—turning them almost into *abstract* data types. This greatly improves the comprehensibility, modifiability, and extensibility of the semantic equations, but not that of the definitions of the auxiliary operations: they may still have to be changed considerably when adding new operations, as may be needed to describe an extended language.

An alternative approach to improving the techniques of denotational semantics has been developed by Moggi [12], exploiting the category-theoretic notion of a *monad*. He provides a number of monad constructors that correspond to the main techniques used in denotational semantics for representing the semantics of programming concepts as higher-order functions, and these constructors can be combined in various ways. However, it remains to be seen to what extent these elegant theoretical ideas can form the basis for a practical framework for semantic description, and whether they can be made accessible to programmers, and others, who are not so well acquainted with category theory.

monads

5.1.2 VDM

VDM, The Vienna Development Method [3], has an elaborate notation called META-IV, which can be used to give denotational descriptions of programming languages, as well as for specifying more general software. Although there are quite a few variants of META-IV, these share a substantial, partly-standardised notation that provides declarative and imperative syntax (e.g., let-constructions, storage allocation, sequencing, exception-handling) as well as a variety of familiar data types, such as trees, sets, and mappings. The treatment of exceptions in action semantics was directly inspired by that in META-IV, although our trap combinator is a bit simpler than META-IV's.

META-IV

META-IV, like denotational semantics, suffers from notational sensitivity to the structure of data types. To quote a bit of advice from one of the originators of META-IV [4]:

> ... before embarking on the ... [semantic] function definitions, make sure to get the underlying Domains straightened out first. ... Failure to do so ... usually proves disastrous – ultimately requiring the complete rewrite of masses of formulae.

It might appear that META-IV supports something similar to action notation, as functions with side-effects can indeed be combined (albeit only by sequencing). However, the META-IV notation is what we might call 'facet-conscious': if one has used the purely functional part of the notation to specify something, but then discovers that part of it should have a side-effect, a major reformulation is required.

facets

For example, consider static semantics: one might not realise that the generation of unique tags for type expressions involves a side-effect until 'too late'.

auxiliary notation

What can action semantics offer META-IV users, to ameliorate their situation? Essentially the same as proposed above for denotational semantics: to define all the action combinators and primitives as auxiliary notation, and eschew the direct use of the META-IV constructs for sequencing, etc.! Presumably the definitions of the combinators could exploit the imperative META-IV constructs, so the definitions would be more perspicuous than the analogous ones in a pure conventional denotational framework.

5.1.3 RAISE

RSL

RAISE is an acronym for 'Rigorous Approach to Industrial Software Engineering'. The RAISE specification language, RSL [24], is a wide-spectrum language in the algebraic tradition. It has a considerable number of of predefined types; users may introduce new types, and define them either axiomatically or by relating them to previously-defined ones. RSL may be regarded as an extension of the VDM notation META-IV with facilities for axiomatic specification, concurrency, and modular structure. RSL has been given a formal (but somewhat intricate) semantics, and the practical use of the notation is supported by the sophisticated RSL Tools.[1]

facets

The semantic entities provided by RSL include not only data and pure functions, but also 'operations' (accessing or updating storage) and concurrent communicating processes. The syntactic expression of these entities is much less 'facet-conscious' than in META-IV: RSL has essentially a single kind of value-denoting expression, and the same notation can be used for application, regardless of what kind of values the arguments denote. This eliminates some of the pragmatic (extensibility) problems experienced with META-IV; however, some new ones crop up, in connection with the different styles of *axiomatic* definition available for functions, operations, and processes.

RSL and action semantics have quite a lot in common! In particular, the way that the functional, imperative, and concurrent facets can be combined in RSL is very much in the spirit of action notation. One might even say that RSL goes *further* than action notation, by not distinguishing syntactically between values corresponding to actions and those corresponding to yielders and data. So why should one still prefer action notation? There are three main reasons:

sequential evaluation

First, the combination of the functional and imperative facets in RSL involves sequential (left to right) evaluation of all expressions. It seems impossible to specify that the order of evaluation between two expressions should be implementation-dependent, with potential nondeterminism from the interleaving of side-effects. (Such implementation-dependent evaluation order is needed for the semantics of PASCAL, for example.) Note that evaluation of the parallel combination of RSL expressions involves a (somewhat peculiar) state-copying, which makes it inappropriate for ordinary interleaving of side-effects. In fact much of the basic facet, used in action notation for specifying flow of control, is missing in RSL: apart from the lack of interleaving, there is nothing corresponding to traps and escapes. This latter omission is quite strange, as exception-handling is a prominent (albeit somewhat elaborate) feature of META-IV. Traps and escapes can, of course, be simulated in RSL by 'flagging' values, or by passing continuations—at the cost of extensive reformulation when introducing them.

Second, RSL does not provide any declarative facet: it has no combinators for specifying the flow (i.e., scopes) of bindings. It seems that one would have to resort

[1] The licensing cost of the **RSL** Tools confirms that they are intended for serious use in industry, rather than for casual use in academia!

environments to the standard denotational (and META-IV) technique of having 'environments' as explicit parameters, which is not only tedious, but also undermines extensibility.

synchronous communication Third—and somewhat more debatable—the RSL treatment of concurrency is based on synchronous communication and CSP-like combinators, in contrast to action notation's asynchronous communication primitives and process forking. Here, action notation seems operationally more realistic, and at least as expressive, compared to RSL.

Perhaps some of the above drawbacks of RSL could be eliminated by defining new functions or operations in RSL? Or perhaps one could define action notation axiomatically, using new types in RSL? It seems unlikely.

alternative approach Fortunately, an alternative approach to using action notation within RSL has been developed, and used in connection with the formal semantics of ANDF [19]. The idea is to define the action combinators and primitives as *abstract syntax* constructors in RSL. The standard (structural) operational semantics of action notation [15] can also be specified straightforwardly in RSL, using sets of configurations to represent nondeterministic choices. Then for each programming language, one may define 'semantic' functions as syntactic translations into action notation—each program gets mapped to an action abstract syntax tree. To get the intended action *semantics* of programs, one applies the (iterated) transition function for action notation to the tree, which (at least for terminating programs) returns the final state(s).

Thus everything needed for action semantics is defined within RSL, and the RSL Tools can be used to check, maintain, and even reason about action semantic descriptions.

5.1.4 Structural Operational Semantics

Structural operational semantics, SOS, was developed for use in describing programming languages by Plotkin [22, 23]. The framework of so-called natural semantics developed by Kahn [10] is a particular discipline of SOS, related to inference rules for natural deduction in logic. Astesiano [1] provides a newer exposition, see also the SMoLCS framework [2]. The main feature is that although configurations involve program syntax, transitions are specified in a structural fashion that keeps track of control *implicitly*, avoiding the need for a control pointer into the entire program tree. The possible transitions for a compound phrase depend on the possible transitions for some or all of the subphrases, which gives a compositional flavour to SOS specifications—without achieving a truly compositional semantics, however. An SOS is usually presented as a logic with axioms and inference rules for the possibility of transitions between configurations. Alternatively, one can use Horn clause axioms together with the standard inference rules of Horn clause logic.

control implicit

The definition of action notation in [15] is based on structural operational semantics. An action semantic description of a programming language can therefore be regarded as an easy way of defining an SOS for it, by *reducing* the language to action notation—although of course the syntactic components of configurations now involve action primitives and combinators, rather than the original program phrases. Essentially, the action combinators correspond to particular patterns in SOS rules. For example, A_1 `and then` A_2 corresponds to the pattern where only transitions involving A_1 are considered until a terminal configuration is reached, whereupon A_2 is considered—as specified, once and for all, in [15]. By reducing the described programming language to action notation, one avoids repetitious specification of such patterns, as well as having standard symbols by which to represent them.

action notation

The pragmatic aspects of SOS are acceptable, as witnessed by the current popularity of the framework. However, this is partly due to the widespread exploitation *formality* of some elaborate *conventions* concerning propagation (of exceptions, for instance); their formal status is rather dubious. Without these conventions, SOS descriptions are much more cumbersome, and can be quite tedious to read. Note that the modifiability and extensibility of SOS descriptions are not nearly as good as for action semantics, although somewhat better than for denotational semantics.

5.1.5 Evolving Algebras

This framework [7] employs algebras—more precisely, first-order structures—to represent the configurations of operational transition systems. It provides a particularly concise and expressive notation for updates on complex structures, whereby there is no need to mention parts of structures that are to remain unchanged; this gives some useful modularity to specifications. Transition rules are usually conditional on details of the current configuration, and several rules can be applied at once.

configurations All the structure resides in the configurations, including the program syntax and a pointer that indicates the current place of execution. There is no semantic decomposition or analysis of programs: the reader has to imagine how the presence of the program in the initial configuration will affect the computations, as determined by the transition rules. Evolving algebras can be regarded as the antithesis of action semantics, as they emphasise configurations at the expense of actions.

pragmatics Experiments have shown that evolving algebras can be used to specify particular implementations[2] of practical programming languages, such as C, Prolog, and occam2. However, the algebras involved are quite large, and despite the succinct notation for structure updates, the rules get rather large and complicated. It would be interesting to see whether evolving algebras would benefit from exploiting action notation when specifying rules. In the other direction, action semantics might benefit from using evolving algebras to specify the operational semantics of action notation, making it easier to add new facets, etc.

5.2 Applications and Tools

The development of action semantics has involved rather modest resources, and there is still some way to go before we have a satisfactory collection of examples and tools. The Proceedings of the first international workshop on action semantics [16] give a good impression of the level of activity in the area. In this section we point out what we consider to be the major recent achievements.

5.2.1 The Pascal Action Semantics

This version of the Pascal Action Semantics [17] provides an almost complete formal specification of the dynamic semantics of Standard Pascal (Level 0). It reflects the authors' understanding of the Pascal Standard. However, it is intended primarily as a 'showcase' example of the use of action semantics, rather than as a contribution to the understanding of Pascal.

This version consists of modules specifying Abstract Syntax, Semantic Functions, and Semantic Entities, together with a Symbol Index. It is intended only for readers who are already familiar with the action semantics framework, as described in the recent books by each author. More work is needed before it is ready for a more general readership, including completion of an automated checking of the internal consistency of the specification, using the ASD Tools.

[2] Which isn't quite the same as defining semantics!

5.2.2 The ANDF-FS

The ANDF-FS [19] is the first example of 'industrial' use of action semantics. The technique exploited has already been described here, see Section 5.1.3. Note that action semantics is used not only for the dynamic semantics, but also for the *static* semantics of ANDF!

The preliminary studies that led to the adoption of action semantics for the ANDF-FS make interesting reading [8, 26], see also [9].

Particularly encouraging is the fact that the ANDF-FS hasn't simply been put on the shelf to gather dust: it is being actively used, for reference purposes, in connection with a project to construct an interpreter for ANDF-FS. The participants of that project have had little or no previous training in formal semantics, yet they found the document quite accessible—thanks to the verbose, user-friendly, action notation.

5.2.3 The ASD Tools

The ASD Tools[3] include facilities for parsing, syntax-directed (and textual) editing, checking, and interpretation of action semantic descriptions. Such facilities significantly enhance accuracy and productivity when writing and maintaining large specifications, and are also particularly useful for those learning how to write a.s.d.s. The notation supported by the ASD Tools is a direct ASCII representation of the standard notation used for a.s.d.s in the literature, as defined in [15] and illustrated in this tutorial.

The ASD Tools are implemented using the ASF+SDF system [11], which is itself based on the Centaur system (developed by INRIA, amongst others). A licence for Centaur is required; this is available to academic institutions free of charge, and the entire ASD Tools implementation can be freely obtained by FTP.

5.2.4 Other Tools

David Watt's group at Glasgow has developed prototype tools [5, 18] for interpreting action notation, and for compiling it into C. The interpreter deals with most of the standard action notation except for the communicative facet; the compiler insists on static scopes for binding.

Peter Ørbæk at Aarhus has developed a compiler from action notation into RISC code, employing extensive static analysis [20]. The subset of action notation accepted is adequate to describe PASCAL-like and functional programming languages. The code generated is generally no worse than four times slower than that produced by optimising C compilers.

5.2.5 Mailing List

To obtain up-to-date information about action semantics and the tools being developed, one may subscribe to the action semantics mailing list by sending a message to `action-request@brics.dk`.

[3]To be demonstrated at FME'96

Appendix A

Semantic Entities

Most of the notation used here for specifying semantic entities has a fairly obvious interpretation, so few comments are provided.

Sorts **needs: Values, Variables, Subprograms, Tasks, Escapes.**
Values **needs: Numbers.**
Variables **needs: Values, Types.**
Types .
Numbers .
Subprograms

- **Modes** .
- **Arguments** **needs: Values, Variables.**
- **Procedures** **needs: Modes, Arguments.**

Tasks .
Escapes .
Required Bindings **needs: Types, Numbers.**

includes: [15]/**Action Notation**.

A.1 Sorts

introduces: entity .

- entity = value | variable | type | procedure | task-agent (*disjoint*) .
- datum = entity | escape-reason | mode | message | entry | □ .
- token = string of (uppercase letter, (uppercase letter | digit)*) .
- bindable = entity .
- storable = value .
- sendable = agent | task | entry | signal | □ .

A.2 Values

introduces: value .

includes: **[15]/Data Notation/Instant/Distinction** (value *for* s , _ is _).

- value = truth-value | number (*disjoint*) .

A.3 Variables

introduces: variable , assign _ to _ , the _ assigned to _ , allocate _ for _ , dispose of _ .

- assign _ to _ :: yielder [of value], yielder [of variable] → action [storing] .
- the _ assigned to _ :: value, yielder [of variable] → yielder [of value] .
- allocate _ for _ :: variable, yielder [of type] → action [giving a variable | storing] .
- dispose of _ :: yielder [of variable] → action [storing] .

(1) variable = cell .

(2) assign (Y_1:yielder [of value]) to (Y_2:yielder [of variable]) =
  store the storable yielded by Y_1 in the cell yielded by Y_2 .

(3) the ($v \leq$value) assigned to (Y:yielder [of variable]) =
  the (v & storable) stored in the cell yielded by Y .

(4) allocate ($v \leq$variable) for (Y:yielder [of type]) = allocate a cell .

(5) dispose of (Y:yielder [of variable]) = unreserve the cell yielded by Y .

For simplicity here, we do not bother to distinguish between cells for storing different sorts of values, so the type entities are quite redundant. In a more realistic example, the specification of variable allocation and assignment can become quite complex.

The standard action store Y_1 in Y_2 changes the data stored in the cell yielded by Y_2 to the storable data yielded by Y_1. The cell concerned must have been previously reserved, using reserve Y, otherwise the storing action fails. Here, Y has to yield a particular, individual cell.

allocate a cell abbreviates the following hybrid action:

```
indivisibly
| choose a cell [not in the mapped-set of the current storage] then
| | reserve the given cell and give it .
```

Reserved cells are made available for reuse by unreserve Y, where Y yields an individual cell.

A.4 Types

introduces: type , boolean-type , integer-type , real-type .

- type = boolean-type | integer-type | real-type (*individual*) .

A.5 Numbers

introduces: number , integer-number , real-number , min-integer , max-integer , integer-number of _ , real-number of _ , negation _ , absolute _ , sum _ , difference _ , product _ , quotient _ , modulo _ , remainder _ .

includes: **[15]/Data Notation/Instant/Total Order**
(value *for* s , _ is _ , _ is less than _ , _ is greater than _).

- number = integer-number | real-number .
- min-integer , max-integer : integer .
- integer-number of _ :: integer → integer-number ($partial$) .
- real-number of _ :: rational → real-number ($partial$) .
- negation _ , absolute _ :: number → number ($partial$) .
- sum _ , difference _ , product _ , quotient _ ::
  number^2 → number ($partial$) .
- modulo _ , remainder _ :: integer-number^2 → integer-number ($partial$) .
- _ is _ , _ is less than _ , _ is greater than _ ::
  integer-number, integer-number → truth-value ($total$) ,
  real-number, real-number → truth-value ($total$) .

(1) i : integer [min min-integer] [max max-integer] ⇒ integer-number of i : integer-number .

(2) i : integer [min successor max-integer] ⇒ integer-number of i = nothing .

(3) i : integer [max predecessor min-integer] ⇒ integer-number of i = nothing .

(4) real-number of (r:approximation) : real-number .

(5) real-number of (r:interval approximation) : real-number of (approximately r) .

(6) integer-number of i : integer-number ⇒

  (1) negation integer-number of i = integer-number of negation i ;

  (2) absolute integer-number of i = integer-number of absolute i .

  (3) ...

(7) integer-number of i_1 : integer-number ; integer-number of i_2 : integer-number ⇒

  (1) sum (integer-number of i_1, integer-number of i_2) = integer-number of sum (i_1, i_2) ;

  (2) difference (integer-number of i_1, integer-number of i_2) = integer-number of difference (i_1, i_2) ;

(8) real-number of r : real-number ⇒

  (1) negation real-number of r = real-number of negation r ;

  (2) absolute real-number of r = real-number of absolute r .

(9) real-number of r_1 : real-number ; real-number of r_2 : real-number ⇒

  (1) sum (real-number of r_1, real-number of r_2) : real-number of sum (r_1, r_2) ;

  (2) ...

(10) integer-number of i_1 : integer-number ; integer-number of i_2 : integer-number ⇒

  (1) integer-number of i_1 is integer-number of i_2 = i_1 is i_2 ;

  (2) integer-number of i_1 is less than integer-number of i_2 = i_1 is less than i_2 ;

  (3) integer-number of i_1 is greater than integer-number of i_2 = i_1 is greater than i_2 .

(11) real-number of r_1 : real-number ; real-number of r_2 : real-number ⇒

  (1) real-number of r_1 is real-number of r_2 = r_1 is r_2 ;

  (2) real-number of r_1 is less than real-number of r_2 = r_1 is less than r_2 ;

  (3) real-number of r_1 is greater than real-number of r_2 = r_1 is greater than r_2 .

The specification of real arithmetic uses a loosely-specified sort of rational approximations and intervals, to avoid insisting that implementations should be exact. Similarly, there are loosely-specified bounds on integers.

A.6 Subprograms

A.6.1 Modes

introduces: mode , constant-mode , reference-mode , copy-mode .

- mode = constant-mode | copy-mode | reference-mode (*individual*) .

includes: **Data Notation/Instant/Distinction** (mode *for* s , _ is _).

(1) distinct (constant-mode, reference-mode, copy-mode) = true .

A.6.2 Arguments

introduces: argument , copy-back _ .

- argument = value | variable .
- copy-back _ :: yielder [of map [token to variable]] → action [completing | storing] .

privately introduces: copy-back _ from _ .

(1) copy-back (Y:yielder [of map [token to variable]]) =
copy-back Y from the elements of the mapped-set of Y .

(2) copy-back (Y_1:yielder [of map [token to variable]]) from (Y_2:yielder [of token*]) =
check (Y_2 is ( )) or
| assign (the value assigned to the variable bound to the first of Y_2)
| to the variable yielded by (Y_1 at the first of Y_2) and then
| dispose of the variable yielded by (Y_1 at the first of Y_2) and then
| copy-back Y_1 from the rest of Y_2 .

A.6.3 Procedures

introduces: procedure , procedure of _ , procedure-abstraction _ ,
parameterless _ , parameterized _ , modalized _ , formal-modes _ .

- procedure of _ :: abstraction → procedure (*partial*) .
- procedure-abstraction _ :: procedure → abstraction (*total*) .
- parameterless _ , parameterized _ :: procedure → procedure (*partial*) .
- modalized _ :: (modes, procedure) → procedure (*partial*) .
- formal-modes _ :: procedure → modes (*partial*) .

(1) a : action [completing | diverging | storing | communicating]
[using given arguments | current storage | current buffer] ⇒
procedure of abstraction of a : procedure .

(2) p = procedure of a ⇒ the procedure-abstraction of p:procedure = a .

(3) p = procedure of a ⇒
parameterless p:procedure : procedure ; parameterized p:procedure : procedure .

(4) p = parameterless procedure of a ⇒ the procedure-abstraction of p:procedure = a .

(5) p = parameterized procedure of a ⇒ the procedure-abstraction of p:procedure = a .

(6) p = parameterized modalized (m:modes, procedure of a) ⇒

(1) the procedure-abstraction of p:procedure = a ;

(2) the formal-modes of p:procedure = m .

A.7 Tasks

introduces: task-agent , task , task of _ , task-abstraction _ , initial task-action , signal , begin-signal , done-signal , terminated-signal , entry , entry of _ , _ is entered in _ .

- task-agent $\leq$ agent .
- task of _ :: abstraction $\rightarrow$ task $(total)$.
- task-abstraction _ :: task $\rightarrow$ abstraction $(total)$.
- signal = begin-signal | done-signal | terminated-signal $(individual)$.
- initial task-action : action .
- entry of _ :: token $\rightarrow$ entry $(total)$.
- _ is entered in _ :: entry, buffer $\rightarrow$ truth-value $(total)$.

(1) t = task of a $\Rightarrow$ task-abstraction t:task = a .

(2) initial task-action =
| send a message [to the contracting-agent] [containing the performing-agent] and then
| receive a message [from the contracting-agent] [containing a task]
then
| send a message [to the contracting-agent] [containing the begin-signal] and then
| enact the task-abstraction of the task yielded by the contents of the given message .

(3) entry of k_1:token is entry of k_2:token = k_1 is k_2 .

(4) e:entry is entered in empty-list = false .

(5) e:entry is entered in list of m:message [containing an entry] = e is the contents of m .

(6) e:entry is entered in list of m:message [containing a signal | agent | task] = false .

(7) e:entry is entered in concatenation (b_1:buffer, b_2:buffer) =
either (e is entered in b_1, e is entered in b_2) .

A.8 Escapes

introduces: escape-reason , exit , procedure-return .

- escape-reason = exit | procedure-return $(individual)$.

A.9 Required Bindings

introduces: required-bindings .

- required-bindings : map [token to value | type] .

(1) required-bindings = disjoint-union of (map of "TRUE" to true,
map of "FALSE" to false,
map of "BOOLEAN" to boolean-type,
map of "MININT" to integer-number of min-integer,
map of "MAXINT" to integer-number of max-integer,
map of "INTEGER" to integer-type,
map of "REAL" to real-type) .

Bibliography

[1] E. Astesiano. Inductive and operational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Report, pages 51–136. Springer-Verlag, 1991.

[2] E. Astesiano and G. Reggio. SMoLCS driven concurrent calculi. In *TAP-SOFT'87, Proc. Int. Joint Conf. on Theory and Practice of Software Development, Pisa, Vol. 1*, volume 249 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

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

[4] D. Bjørner and O. N. Oest, editors. *Towards a Formal Description of Ada*, volume 98 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[5] D. F. Brown, H. Moura, and D. A. Watt. Actress: an action semantics directed compiler generator. In *CC'92, Proc. 4th Int. Conf. on Compiler Construction, Paderborn*, volume 641 of *Lecture Notes in Computer Science*, pages 95–109. Springer-Verlag, 1992.

[6] D. Goodman. *The Complete HyperCard Handbook*. Bantam, 1987.

[7] Y. Gurevich. Evolving algebras, a tutorial introduction. *Bull. EATCS*, (43):264–284, Feb. 1991.

[8] B. S. Hansen and J. Bundgaard. The role of the ANDF formal specification. Technical Report 202104/RPT/5, issue 2, DDC International A/S, Lundtoftevej 1C, DK–2800 Lyngby, Denmark, 1992.

[9] B. S. Hansen and J. U. Toft. The formal specification of ANDF, an application of action semantics. In P. D. Mosses, editor, *Proc. First Intl. Workshop on Action Semantics (Edinburgh, April 1994)*, number NS-94-1 in BRICS Notes Series, pages 34–42. BRICS, Dept. of Computer Science, Univ. of Aarhus, Denmark, 1994.

[10] G. Kahn. Natural semantics. In *STACS'87, Proc. Symp. on Theoretical Aspects of Computer Science*, number 247 in Lecture Notes in Computer Science. Springer-Verlag, 1987.

[11] P. Klint. A meta-environment for generating programming environments. In *Algebraic Methods II: Theory, Tools, and Applications*, volume 490 of *Lecture Notes in Computer Science*, pages 105–124. Springer-Verlag, 1991.

[12] E. Moggi. Computational lambda-calculus and monads. In *LICS'89, Proc. 4th Ann. Symp. on Logic in Computer Science*, pages 14–23. IEEE, 1989.

[13] P. D. Mosses. Denotational semantics. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume B, chapter 11. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[14] P. D. Mosses. A practical introduction to denotational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Report, pages 1–49. Springer-Verlag, 1991.

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

[16] P. D. Mosses, editor. *Proceedings of the First International Workshop on Action Semantics (Edinburgh, Scotland, April 1994)*, number NS-94-1 in BRICS Notes Series. Dept. of Computer Science, Univ. of Aarhus, 1994.

[17] P. D. Mosses and D. A. Watt. Pascal action semantics, version 0.6. Available by FTP from ftp.daimi.aau.dk in pub/action/pascal, Mar. 1993.

[18] H. Moura and D. A. Watt. Action transformations in the Actress compiler generator. In *Compiler Construction (Proc. 5th Intl. Conf., CC'94, Edinburgh, U.K., April 1994)*, volume 786 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1994.

[19] J. P. Nielsen and J. U. Toft. Formal specification of ANDF, existing subset. Technical Report 202104/RPT/19, issue 2, DDC International A/S, Lundtoftevej 1C, DK–2800 Lyngby, Denmark, 1994.

[20] P. Ørbæk. OASIS: An optimizing action-based compiler generator. In *Compiler Construction (Proc. 5th Intl. Conf., CC'94, Edinburgh, U.K., April 1994)*, volume 786 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1994.

[21] J. Palsberg. A provably correct compiler generator. In *ESOP'92, Proc. European Symposium on Programming, Rennes*, volume 582 of *Lecture Notes in Computer Science*, pages 418–434. Springer-Verlag, 1992.

[22] G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN–19, Dept. of Computer Science, Univ. of Aarhus, 1981. Now available only from University of Edinburgh.

[23] G. D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Formal Description of Programming Concepts II, Proc. IFIP TC2 Working Conference, Garmisch-Partenkirchen, 1982*. IFIP, North-Holland, 1983.

[24] The RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice-Hall, 1992.

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

[26] J. U. Toft. Feasibility of using RSL as the specification language for the ANDF formal specification. Technical Report 202104/RPT/12, issue 2, DDC International A/S, Lundtoftevej 1C, DK–2800 Lyngby, Denmark, 1993.

[27] D. A. Watt. Executable semantic descriptions. *Software: Practice and Experience*, 16:13–43, 1986.

Recent Publications in the BRICS Notes Series

NS-96-14 **Peter D. Mosses.** ***A Tutorial on Action Semantics*****. December 1996. 46 pp. Tutorial notes for FME '94 (Formal Methods Europe, Barcelona, 1994) and FME '96 (Formal Methods Europe, Oxford, 1996).**

NS-96-13 **Olivier Danvy, editor.** ***Proceedings of the Second ACM SIGPLAN Workshop on Continuations, CW '97*** **(ENS, Paris, France, 14 January, 1997), December 1996. 166 pp.**

NS-96-12 **Mandayam K. Srivas.** ***A Combined Approach to Hardware Verification: Proof-Checking, Rewriting with Decision Procedures and Model-Checking; Part II: Articles. BRICS Autumn School on Verification*****. October 1996. 56 pp.**

NS-96-11 **Mandayam K. Srivas.** ***A Combined Approach to Hardware Verification: Proof-Checking, Rewriting with Decision Procedures and Model-Checking; Part I: Slides. BRICS Autumn School on Verification*****. October 1996. 29 pp.**

NS-96-10 **Robert Pollack.** ***What we Learn from Formal Checking; Part III: Formalization is Not Just Filling In Details. BRICS Autumn School on Verification*****. October 1996. iv+42 pp.**

NS-96-9 **Robert Pollack.** ***What we Learn from Formal Checking; Part II: Using Type Theory: An Introduction. BRICS Autumn School on Verification*****. October 1996. iv+71 pp.**

NS-96-8 **Robert Pollack.** ***What we Learn from Formal Checking; Part I: How to Believe a Machine-Checked Proof. BRICS Autumn School on Verification*****. October 1996. iv+19 pp.**

NS-96-7 **Tom F. Melham.** ***Some Research Issues in Higher Order Logic Theorem Proving. BRICS Autumn School on Verification*****. October 1996. 15 pp.**

NS-96-6 **Gerard J. Holzmann.** ***On-the-Fly Model Checking Tutorial. BRICS Autumn School on Verification*****. October 1996. 31 pp.**