A Tutorial on Action Semantics

– DRAFT –

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

October 1992

N.B. This draft is *incomplete* and *unpolished*.

[1]Internet: pdmosses@daimi.aau.dk

Contents

Abstract

Formal semantics is a topic of major importance in the study of programming languages. Its applications include documenting language design, establishing standards for implementations, reasoning about programs, and generating compilers.

This tutorial is about *action semantics*, a recently-developed framework for formal semantics. The primary aim of action semantics is to allow *useful* semantic descriptions of *realistic* programming languages.

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!

Action semantics is *compositional*, like denotational semantics. The main difference between action semantics and denotational semantics concerns the universe of semantic entities: action semantics uses entities called *actions*, rather than the higher-order functions used with denotational semantics. Actions are inherently more operational than functions: when *performed*, actions process information *gradually*.

Primitive actions, and the various ways of combining actions, correspond to fundamental concepts of information processing. Action semantics provides a particular notation for expressing actions. The symbols of action notation are suggestive words, rather than cryptic signs, which makes it possible to get a broad impression of an action semantic description from a superficial reading, even without previous experience of action semantics. The action *combinators*, a notable feature of action notation, obey desirable algebraic laws that can be used for reasoning about semantic equivalence.

This tutorial provides a quick (half-day) introduction to action semantics. It should be accessible to senior undergraduates, 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 [Mos92].

Chapter 1 sketches the background for action semantics, motivating the formal description of programming languages by considering the requirements of various potential users. Chapter 2 recalls the notions of abstract syntax and compositional semantics, and introduces the novel concept of *actions* as used in action semantics. Chapter 3 takes a walk through an illustrative description, explaining all the notation that it uses. Chapter 4 assesses the pragmatic qualities of action semantic descriptions, and compares the framework to VDM and RAISE. The Appendices provide a full algebraic specification of the special notation used, together with an informal summary of the standard notation.

Chapter 1

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 a particularly important feature.

1.1 Motivation

Programming languages are artificial languages. Programs written in them are used to control the execution of computers. There are many programming languages in existence. Some are simple, intended for special purposes; others are complex and general-purpose, for use in a wide variety of applications.

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 descriptions of programming languages—and hence motivate the study of appropriate frameworks for such descriptions?

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.

Once a language has been designed, it usually gets *implemented*, although in practice, design and implementation are often interleaved and iterated. A comprehensive description of the language is needed to convey the intentions of the language designers to the implementors—unless the designer and the implementor are the same person, of course. 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.

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.

Finally, theoreticians can obtain new *insight* into the general nature of programming languages by 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.

1.2 Requirements

Which properties might language designers require of language descriptions? Well, design is an iterative 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, their completed language description should provide an appropriate basis for conveying the design to the implementors, and for setting standards.

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.

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 directly related to machine operation.

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 ...

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 the above requirements can be fully satisfied simultaneously. Certainly none of the previously available frameworks appears to be suitable for use in all the above applications. 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.

The topic of this tutorial, action semantics, avoids the need for complementary descriptions by making a *compromise* between the above requirements. 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 and interpreters, so implementors should be content. 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 [Mos90], where the theory[1] is strong, but severe pragmatic difficulties hinder its application to realistic programming languages.

Some of these claims for the virtues of action semantic descriptions can be supported by looking at an example. Let us postpone consideration of the extent to which action semantics meets the stated requirements until Chapter 4, after we have seen action semantics *in action*!

1.3 Features

One especially significant feature of language descriptions is whether or not they are *formal*. 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.

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.

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 analogous performance on other programs.

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 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.

[1]at least for dealing with deterministic, sequential programs

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 formality does *not* ensure success.

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* interpreter, 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.

Chapter 2

Concepts

This chapter explains the concepts underlying action semantic descriptions of programming languages. It considers the factorization of language descriptions into syntax and semantics, and discuss the pragmatic issue of setting standards for programming languages. Readers who are already familiar with other semantic description frameworks, e.g., denotational semantics, may skip to Section 2.2.2.

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.

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.

Most frameworks for language description unfortunately do not provide a clear separation between syntactic and semantic entities, nor do they exploit compositional semantic functions. A notable exception is *denotational semantics*, from which action semantics was developed.

The distinction between the syntax and the semantics of a language is dependent on the division into structure and behaviour. At one extreme, structure could be trivial—arbitrary strings over an alphabet of symbols—and then the usual notion of program legality would have to be considered as a component of behaviour. At the other extreme, behaviour could be incorporated into a dynamic notion of structure. For comprehensive language descriptions, it is best to find a compromise such that separate descriptions of syntax and semantics provide a useful factorization of the entire description into parts with a simple interface.

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.

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 easier 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 emphasize abstract syntax, giving only a cursory explanation of concrete syntax and its relation to abstract syntax.

2.1.1 Concrete Syntax

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.

The parse tree produced by phrase-structure analysis of a program represents the recognized 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 recognized structure of the program.

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.

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`. Factors are more restricted than terms, in that any additions that occur in factors have to be enclosed in grouping parentheses, whereas the parentheses are optional when additions occur in terms. The term `a+b` is not classified as a factor, so the only possible parsing for `a+b*c` is the one where `b*c` is grouped together. 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.

The symbols used for labeling the nodes in an abstract syntax tree may be the same as the lexical symbols of the corresponding concrete syntax. This is not essential, though, as the symbols are needed only 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. Similarly, when every statement is terminated by a semicolon in concrete syntax, the semicolons may be omitted in the corresponding abstract syntax, as they have no distinguishing effect. On the other hand, the lexical symbols from the concrete syntax do have considerable suggestive and mnemonic value. By retaining them as labels in abstract syntax trees—in the same order that they occur in the corresponding parse trees—much of the relation 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 complicates the definition of semantic functions on abstract syntax. In general, however, ignoring semantically irrelevant details of parse tree structure tends to simplify the definition of semantic functions.

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.

2.1.3 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 popular BNF formalism. Aspects which fall outside context-free syntax are called context-*sensitive* and include 'declaration of identifiers before use' and 'well-typedness of expressions'. Characteristic for them is that they involve a kind of matching between distant parts of programs that is inherently more complex than mere 'parenthesis-matching'.

The description of context-sensitive syntax can, in principle, be accomplished by use of so-called *attribute grammars*. Unfortunately, these are not always as perspicuous as context-free grammars. Moreover, context-sensitive abstract syntax makes a more complicated interface between concrete syntax and semantics than context-free abstract syntax does. Of course, this is to be expected, because the former generally captures more information than the latter. 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.

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 Semantics

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

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.

2.2.1 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.?

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 semantics, provided that program behaviour does not take account of efficiency. Any compositional semantics for phrases that has this property is called *fully abstract*.

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. But when they have different semantics, they may or may not be interchangeable: a compositional semantics is not necessarily fully abstract.

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.

Unfortunately, full abstractness is often difficult to obtain, at least when semantics is defined explicitly. In fact it has been shown *impossible* to give fully abstract *denotational* semantics for some rather simple programming languages [Plo77, Sto88]. In any case, a less-than-fully abstract semantics can be much simpler to specify, and the semantic equivalence that it provides between phrases may be adequate for most purposes. For instance, a semantics that is not fully abstract sets exactly the same standard for implementations as one that is fully abstract, provided the semantics of entire programs is the same, since requirements on implementations do not directly involve the semantics of phrases such as statements and expressions. So let us not demand full abstractness at all.

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 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.

Actions

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 abandoning the performance of an action (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.

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.

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.

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.

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

Data

The information processed by actions consists of items of *data*, organized 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 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

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.

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 yields themselves when evaluated.

2.3 Pragmatics

Let us conclude this chapter by considering the use of comprehensive language descriptions for setting *standards* for implementations.

The syntax of a programming language defines the set of legal programs, and the semantics of each program gives a representation of its implementation-independent behaviour. A standard for a programming language relates its syntax and semantics to properties of physical implementations, defining the class of *conforming* implementations. Thus it is concerned with the *pragmatics* of the language.

With syntax a standard may, for instance, require implementations to reject (with an informative error message) illegal programs, and perhaps allow them also to reject legal programs whose processing exceeds the available resources. It may allow national or typographical variations in the characters used to write programs.

It is important to realize in connection with semantics that the actual behaviour of a particular program may be allowed to vary between implementations—even between different runs on the same implementation! This variation may be represented in the semantics by loosely-specified entities, i.e., parameters, or by a nondeterministic relation between input and output. A standard may require certain parameters to be *implementation-defined*; the remaining ones are left *undefined*, and an implementation is free to choose. In ADA, for instance, the value of MAX_INT is implementation-defined, whereas the order of expression evaluation is generally left undefined.

Finally, note that although it may be feasible for a standard to *define* a class of implementations on the basis of syntactic and semantic descriptions, one may still not be able to *verify* that a particular implementation belongs to that class. In practice, it is feasible to verify only those implementations that have been developed systematically from language descriptions. A *validation suite* for a language is a particular set of programs that an implementation must process 'correctly' so as to be regarded as valid. The use of validation suites to test conformance to standards is a rather weak approximation to verification.

Chapter 3

An Illustrative Example

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. Summaries of the standard action notation and data notation used in the description are given in the Appendices.

The language used here to illustrate action semantic descriptions is a medium-scale, 'ideal' programming language. Syntactically, it is a sublanguage of ADA. However, the specified action semantics for the constructs does not always correspond exactly to the semantics described in the ADA Reference Manual. For instance, parameter passing modes are left implementation-dependent in ADA, but not here.

The language is a cut-down version of that described in [Mos92, Appendix A]. The main constructs omitted here, for simplicity, are: 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 illustrative 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 .

An action 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 may themselves be divided into *submodules*, just as we normally divide technical reports and textbooks into sections and subsections.

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.

We write titles of modules using initially capitalized words in **This Bold Font**. The specification above has three modules, with the obvious titles. We could give a title to the whole specification, if we wished; here let us simply inherit the title of the present chapter. The abstract syntax module and the semantic entities module are self-contained, and could be used independently. On the other hand, the semantic functions module *needs* the notation introduced by both the other modules, so its use implies their use too. Our notation for indicating modular structure is intended to be unobtrusive, and most of the time we shall disregard the modularization and focus on what is specified in the bodies of the modules.

We allow modules to be *mutually* dependent, and the order in which we present modules is immaterial. In abstract syntax, for instance, the specification of procedure declarations involves statements, and that of block statements involves declarations, so the corresponding modules have to be mutually dependent; similarly for the corresponding parts of the semantic equations. Most previous frameworks for modules insist on a strict hierarchy, thus forbidding mutual dependence.

A related point is that we are free to present modules in whatever order is most convenient for the reader. In semantic descriptions, it is preferable to present the specification of abstract syntax first, followed by the semantic functions, leaving the semantic entities to the end. This is assuming that the notation for semantic entities is well chosen, and its intended interpretation strongly suggested by the symbols used. When semantic entities are presented before the semantic functions, it can be difficult to appreciate them, independently of their usage. However, there is no need for us to be dogmatic about such matters, because the order in which modules are written has no effect at all on what they specify.

Finally, we use various devices to indicate that modules are submodules of other modules. For small specifications, such as that of the overall modular structure above, indentation (formally equivalent to putting grouping parentheses around the indented part) is adequate for showing nesting structure. For larger specifications we use numbered titles $m_1.\cdots.m_n$ M, as in ordinary technical documents.

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.

The grammar-like specification given below consists mainly of a set of (numbered) equations. Ignoring the double brackets $[\![\ldots]\!]$, equations 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, differing a little from the variant used in the ADA reference manual. Terminal symbols are written as quoted strings of characters, such as "(" and "or". The use of ordinary lexical symbols as terminal symbols in the grammar specifying abstract syntax makes it rather easy to imagine a corresponding concrete syntax (up to disambiguation of grouping, at least).

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. (Such conventions have no formal significance, and may be varied as desired.) The alternatives for each nonterminal symbol are started by =, separated by **|** , and terminated by a period.

grammar:
closed.

There is a precise formal interpretation of a grammar as an *algebraic specification of sorts of trees*; interested readers may consult [Mos92]. Here, it is enough to know that occurrences of $[\![\ldots]\!]$ indicate the construction of nodes of trees. (In denotational semantics such brackets merely separate abstract syntax from semantic notation, and cannot be nested.) We make a distinction between a character, such as '0', and the string consisting of just that character, "0", following most programming languages. (Actually, strings are simply nodes whose branches are all single characters.) We write **grammar:** to ensure this interpretation of the subsequent equations. We also write **closed** in a grammar module when all the productions are being given in full.

3.1.1 Expressions

(1) Identifier = $[\![$ letter (letter **|** digit)* $]\!]$.
(2) Literal = $[\![$ digit$^+$ $]\!]$ **|** $[\![$ digit$^+$ '.' digit$^+$ $]\!]$.

The standard nonterminals digit and letter are always implicitly available in our grammars, for convenience when specifying the lexical syntax of identifiers and numerals. The terminal symbols that they generate are single characters, rather than strings of characters.

The equations above 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* $(R_1$ **|** $\ldots$ **|** $R_n)$, an *optional* part $R^?$, an *optional repeatable* part R^*, or an *obligatory repeatable* part R^+. We do not use the rather inelegant notation for optional and repetitive parts provided by so-called EXTENDED BNF (EBNF), despite its familiarity from reference manuals, because we have a better use for the brackets it uses for optional parts $[R]$, and its $\{R\}$ is hardly suggestive of ordered repetition! Moreover, EBNF requires $R\ \{R\}$ to express that R is an *obligatory* repeatable part, whereas our R^+ avoids writing R twice.

```
(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, and could be just as well specified in the module for statements.

(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" ⟩$^?$ ";" ⟧ .

One could save some effort by regarding an if-then statement as a formal abbreviation for an if-then-else statement with a null else part; similarly, a loop statement could abbreviate a while-statement with an always-true expression part.

(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]] .

In legal ADA programs, the top-level declarations are compilation units, which are essentially just packages, subprograms, and tasks. Here we do not bother to exclude other sorts of declaration, such as constant and variable declarations. Let us assume that the identifier of a program indicates a main procedure, without parameters, to be called when the program is run.

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

3.2 Semantic Functions

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.

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.

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.

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; generally the author tries to maximize their suggestiveness, at the expense of conciseness, but this is not obligatory. Each symbol may consist of several words, e.g., the value of _ , and the place-holder _ indicates where the argument goes.

It is usual to specify the *functionality* of each semantic function. For instance,

evaluate _ :: Expression → 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.

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.

The right hand sides of the semantic equations involve the standard notation for actions and data provided by action semantics. It must be emphasized 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. The following hints about the general form of action notation may be helpful.

Action notation consists mainly of action *primitives* and *combinators*. Each primitive is concerned with one particular kind of information processing, and makes no contribution to the other kinds. Each combinator, on the other hand, expresses a particular mixture of control flow and how the various kinds of information flow. Action notation was designed with sufficient primitives and combinators for expressing most common patterns of information processing straightforwardly, i.e., not *simulating* one kind of information processing by another.

Action notation also incorporates a basic notation for *data*, including truth-values, rational numbers, lists, and finite maps.

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 [Goo87], and of Cobol.)

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 emphasize by vertical rules, as illustrated in the semantic equations for statements given earlier.

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 $\lambda\epsilon_1.\lambda\rho.\lambda\kappa.A_1\varepsilon_1\rho(\lambda\varepsilon_2.A_2\epsilon_2\rho\kappa)$. 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 abbreviations 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.

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.

The intended interpretation of the standard notation for actions is specified operationally, once and for all, in [Mos92, Appendix C]. All that one has to do before using action notation is to specify the information that is to be processed by actions, which may vary significantly according to the programming language being described. This may involve *extending* data notation with further sorts of data, and *specializing* standard sorts, using sort equations. Furthermore, it may be convenient to introduce formal *abbreviations* for commonly-occurring, conceptually-significant patterns of notation. Extensions, specializations, and abbreviations are all specified *algebraically*, as illustrated in Section A.1.

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 $\rightarrow$ 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 canonicalize 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.

3.2.1.2 Literals

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

The sort number is specified in Section A.1 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! No abstractness is lost, though, because leading zeros in digit sequences *are* significant in the fractional parts of real numbers.

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.

3.2.1.3 Evaluating Expressions

• evaluate _ :: Expression $\rightarrow$ 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. But note that failure is *always* an implicit possibility.

(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 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 [[ E₁:Expression O:Binary-Operator E₂:Expression ]] =
        | evaluate E₁ and evaluate E₂
        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 [[ E₁:Expression "or" "else" E₂:Expression ]] =
        evaluate E₁ then
        | | check the given truth-value then give true
        | or
        | | check not the given truth-value then evaluate E₂ .
(7) evaluate [[ E₁:Expression "and" "then" E₂:Expression ]] =
        evaluate E₁ then
        | | check the given truth-value then evaluate E₂
        | 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.1. 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,value)] .

(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) the binary-operation-result of " & " =
  the concatenation of (the given array#1, the given array#2) .

(4) the binary-operation-result of "*" =
  the product of (the given number#1, the given number#2) .

(5) the binary-operation-result of "/" =
  the quotient of (the given number#1, the given number#2) .

(6) the binary-operation-result of "mod" =
  the modulo of (the given number#1, the given number#2) .

(7) the binary-operation-result of "rem" =
  the remainder of (the given number#1, the given number#2) .

(8) the binary-operation-result of "=" =
  the given value#1 is the given value#2 .

(9) the binary-operation-result of "/=" =
  not (the given value#1 is the given value#2) .

(10) the binary-operation-result of "<" =
  the given number#1 is less than the given number#2 .

(11) the binary-operation-result of "<=" =
  not (the given number#1 is greater than the given number#2) .

(12) the binary-operation-result of ">" =
  the given number#1 is greater than the given number#2 .

(13) the binary-operation-result of ">=" =
  not (the given number#1 is less than the given number#2) .

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

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

(16) 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⁺ | storing]
        [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.1.

```
(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 below. The yielder 'it' formally abbreviates the given datum; here, we could also write the given mode.

```
(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.

```
(3) moderate ⟨ E₁:Expression "," E₂:Expressions ⟩ =
        moderate E₁ and
        | give the rest of the given mode⁺ then moderate E₂ .
```

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⁺ → action
        [completing | escaping with an escape-reason | diverging | storing | communicating]
        [using current bindings | current storage | current buffer] .

(1) execute ⟨ S₁:Statement S₂:Statement⁺ ⟩ = execute S₁ and then execute S₂ .
```

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.1.

```
(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 them 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" S1:Statement+ "else" S2:Statement+ "end" "if" ";" ⟧ =
         evaluate E then
         | | check the given truth-value and then execute S1
         | or
         | | check not the given truth-value and then execute S2 .
(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 .
```

```
(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 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.1.

```
(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.1.

```
(15) execute ⟦ "accept" $I$:Identifier "end" ";" ⟧ =
         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 initialization, 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 ⟨ D1:Declaration D2:Declaration+ ⟩ = elaborate D1 before elaborate D2 .
```

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. Mutually-recursive declarations are not considered here.

```
(2) elaborate ⟨ ⟩ = 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!

Somewhat contrary to the explanation of ADA constants in the Reference Manual, we let constants be bound directly to values, rather than to special 'variables' that cannot be assigned new values. Thus our constants resemble *named numbers* in ADA.

```
(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.1. 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. [Mos92, Appendix A] shows how this can be done, using *indirect* bindings (directly!).

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.1. 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.1.

```
(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 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.1.

```
(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. It is a bit complicated to describe the action semantics of distributed tasks that have access to shared variables—the task that declares a variable has to act as a server for assignments and inspections—so we let our illustrative language deviate from ADA in this respect.

3.2.3.2 Relinquishing Variable Declarations

- relinquish _ :: Declaration$^+$ $\rightarrow$ 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 $\langle$ "(" Formals ")" $\rangle^?$ "return" Identifier "is" Block ";" $]\!]$ |
$[\![$ "procedure" Identifier $\langle$ "(" Formals ")" $\rangle^?$ "is" Block ";" $]\!]$ |
$[\![$ "task" "body" Identifier "is" Block ";" $]\!]$ $\Rightarrow$
relinquish D = complete .

3.2.3.3 Synchronizing Task Declarations

- synchronize _ :: Declaration$^+$ $\rightarrow$ 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 $\langle$ D_1:Declaration D_2:Declaration$^+$ $\rangle$ =
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 $\langle$ ":=" Expression $\rangle^?$ ";" $]\!]$ |
$[\![$ "task" Identifier "is" Entry$^+$ "end" ";" $]\!]$ |
$[\![$ "function" Identifier $\langle$ "(" Formals ")" $\rangle^?$ "return" Identifier "is" Block ";" $]\!]$ |
$[\![$ "procedure" Identifier $\langle$ "(" Formals ")" $\rangle^?$ "is" Block ";" $]\!]$ $\Rightarrow$
synchronize D = complete .

3.2.3.4 Modes

- the mode of _ :: Mode$^?$ $\rightarrow$ mode .

(1) the mode of $\langle$ $\rangle$ = the constant-mode .

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

(3) the mode of $\langle$ "in" "out" $\rangle$ = the copy-mode .

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

3.2.3.5 Modes of Formal Parameters

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

(1) the modes of $[\![$ I_1:Identifier ":" M:Mode$^?$ I_2:Identifier $]\!]$ = the mode of M .

(2) the modes of $\langle$ F_1:Formal ";" F_2:Formals $\rangle$ = (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 the end of Section A.1 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 Section A.1, which also refers to the standard action notation and data notation used in action semantics, summarized informally in Appendix B and Appendix C respectively.

Chapter 4

Conclusion

Here, the pragmatic qualities of action semantic descriptions will be assessed, and compared with those of other frameworks such as VDM and RAISE.

Appendix A

An Illustrative Example, ctd.

The modular structure of this Appendix is as follows:

Semantic Entities

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.**

A.1 Semantic Entities

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

includes: [Mos92]/Action Notation.

A.1.1 Sorts

introduces: entity .

- entity = value | variable | type | procedure | task (*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 | □ .

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, and idempotent.[1] Although sort equations look a bit like the so-called domain equations used in denotational semantics, their formal interpretation is quite different. The use of □ above formally expresses an inclusion, as it leaves open what other sorts might be included in datum and sendable.

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

A.1.2 Values

introduces: value .

includes: **[Mos92]/Data Notation/Instant/Distinction** (value *for* s , _ is _).

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

A.1.3 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≤value) assigned to (Y:yielder [of variable]) =
the (v & storable) stored in the cell yielded by Y .

(4) allocate (v≤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.1.4 Types

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

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

A.1.5 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 _ .

- 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 .

(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) ;
 (3) product (integer-number of i_1, integer-number of i_2) = integer-number of product (i_1, i_2) ;
 (4) quotient (integer-number of i_1, integer-number of i_2) =
  integer-number of integer-quotient (i_1, i_2) ;
 (5) modulo (integer-number of i_1, integer-number of i_2) =
  integer-number of integer-modulo (i_1, i_2) ;
 (6) remainder (integer-number of i_1, integer-number of i_2) =
  integer-number of integer-remainder (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) difference (real-number of r_1, real-number of r_2) : real-number of difference (r_1, r_2) ;
 (3) product (real-number of r_1, real-number of r_2) : real-number of product (r_1, r_2) ;
 (4) quotient (real-number of r_1, real-number of r_2) : real-number of quotient (r_1, r_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.1.6 Subprograms

A.1.6.1 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.1.6.2 Arguments

introduces: argument , copy-back _ .

- argument = value | variable .
- copy-back _ :: 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.1.6.3 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] $\Rightarrow$
procedure of abstraction of a : procedure .

(2) p = procedure of a $\Rightarrow$ the procedure-abstraction of p:procedure = a .

(3) p = procedure of a $\Rightarrow$
parameterless p:procedure : procedure ; parameterized p:procedure : procedure .

(4) p = parameterless procedure of a $\Rightarrow$ the procedure-abstraction of p:procedure = a .

(5) p = parameterized procedure of a $\Rightarrow$ the procedure-abstraction of p:procedure = a .

(6) p = parameterized modalized (m:modes, procedure of a) $\Rightarrow$

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

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

A.1.7 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 → task (*total*) .
- task-abstraction _ :: task → abstraction (*total*) .
- signal = begin-signal | done-signal | terminated-signal (*individual*) .
- initial task-action : action .
- entry of _ :: token → entry (*total*) .
- _ is entered in _ :: entry, buffer → 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.1.8 Escapes

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

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

A.1.9 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 min-integer,
                                             map of "MAXINT" to integer-number max-integer,
                                             map of "INTEGER" to integer-type,
                                             map of "REAL" to real-type ) .
```

Appendix B

Action Notation

The systematic informal description of (almost all of) action notation summarizes the explanations given in Chapter 3, and gives further details. It is intended for reference. The usage of the notation is illustrated in the semantic description in Chapter 3.

The symbols of action notation are explained in the order indicated below. See Section 2.2.2 for the general concept of actions.

Action Notation

- **Basic.**
- **Functional.**
- **Declarative.**
- **Imperative.**
- **Reflective.**
- **Communicative.**
- **Hybrid.**
- **Facets**
  - **Outcomes.**
  - **Incomes.**
  - **Actions.**
  - **Yielders.**

B.1 Basic Action Notation

Basic action notation is primarily concerned with specifying flow of control in performances of actions. It includes basic notation for data as well, see Appendix C.

B.1.1 Actions

- *All primitive basic actions:*
  - Give no transients, except for escape.
  - Produce no bindings.
  - Make no changes to storage.
  - Do not communicate.
- *All basic action combinators are:*
  - Functionally conducting (see the basic action A_1 and A_2), except for A_1 trap A_2, which is functionally composing (see the functional action A_1 then A_2 in Section B.2.1).

  - Declaratively conducting (see the basic action A_1 and A_2).

- complete: a primitive basic action. Represents normal termination. Unit for A_1 and A_2, as well as for A_1 and then A_2.
  - Indivisible. Always completes.
- escape: a primitive basic action. Represents abnormal termination. Unit for A_1 trap A_2.
  - Indivisible. Always escapes.
  - Gives any given transients.
- fail: a primitive basic action. Represents abortion of the current alternative. Unit for A_1 or A_2.
  - Indivisible. Always fails.
- commit: a primitive basic action. Represents commitment to the current alternative, cutting away other current alternatives.
  - Indivisible. Always commits and completes.
- unfold: a dummy action, standing for the innermost enclosing unfolding.
- unfolding A: Represents the (in general, infinite) action formed by continually substituting A for unfold. (To avoid singularities in pathological cases, substitute complete and then A, rather than just A.)
  - Performs A, but whenever the dummy action unfold is reached, A is performed in place of unfold.
- indivisibly A: a basic combination of action A. Represents that the steps of performing A are not interleaved with those of other actions performed by the same agent. Also ensures that the performance of A cannot be interrupted by an escape or failure occurring outside A. For use only when A cannot diverge.
  - *Indivisible*: A is performed as a single step.
- A_1 or A_2: a basic combination of actions A_1, A_2. Represents implementation-dependent choice; specializes to deterministic choice when one or the other of A_1, A_2 must fail.
  - Performs either A_1 or A_2. When the performed alternative fails without committing, it is ignored and the other alternative is performed instead.
  - All the transients given to the combination of A_1, A_2 are given to the performed alternative. On normal or abnormal termination, all the transients given by the performed alternative are given by the combined action.
  - All the bindings received by the combination of A_1, A_2 are received by the performed alternative. On normal termination, all the bindings produced by the performed alternative are produced by the combined action.
- A_1 and A_2: a basic combination of actions A_1, A_2. Represents implementation-dependent order of performance of indivisible subactions, specializing to independent performance when there is no interference between A_1 and A_2.
  - *Basically interleaving*: Performs both A_1, A_2, with arbitrary interleaving of their indivisible steps. An escape or a failure causes any remaining parts of the subactions to be skipped.
  - *Functionally conducting*: The transients given to the combination of A_1, A_2 is given to both A_1, A_2. On normal termination, all the transients given by A_1, A_2 is collected and given by the combined action—if both give one or more items of transients, these are tupled in the given order. On escape, only the transients given by the escape is given by the combined action.
  - *Declaratively conducting*: The bindings received by the combination of A_1, A_2 are received by both A_1, A_2. On normal termination, all the bindings produced by A_1, A_2 are collected and produced by the combined action—provided that the bindings are all for distinct tokens, otherwise the combined action fails.
- A_1 and then A_2: a basic combination of actions A_1, A_2. Represents dependency on normal termination.

  - *Basically (normal) sequencing*: Performs A_1 first. If A_1 completes, performs A_2.

- A_1 trap A_2: a basic action combination. Represents recovery from abnormal termination.
  - Performs A_1 first. If A_1 escapes, performs A_2.
  - Functionally composing (see the functional action A_1 then A_2 in Section B.2.1).
- action: the sort of all actions. See Section B.8.3 for notation for subsorts of action.

B.1.2 Yielders

- the d yielded by Y: a yielder, where d is a sort of data and Y is a yielder. When Y yields an individual, it yields that individual, provided that the individual is included in the sort, otherwise it yields nothing.
- Every *data-operation* (i.e., operation specified for arguments included in data) is extended to arguments of sort yielder. The application of a data operation to yielders yields whatever is yielded by applying the data operation to the data yielded by the arguments. For instance, sum (Y_1, Y_2) yields the numerical sum of whatever Y_1 and Y_2 yield.
- yielder: the sort of all yielders. See Section B.8.4 for notation for subsorts of yielder.

B.1.3 Data

- datum: a sort. Its individuals represent items of data. Left open, as it depends on the variety of information processed by the programs of a programming language. Includes generally-useful sorts from data notation (see Appendix C), except for tuples. Similarly for distinct-datum, the sort of datum whose individuals are distinguished by the operation _ is _ .
- data: a sort. Its individuals represent tuples, i.e., ordered collections of individuals of sort datum, which may also be processed as transient information (see Section B.2.3).
- a d: the same data as d. Only used to improve the readability of the notation. Similarly for an d, the d, of d, and some d. Thus an application of an operation *op* to arguments x can be written as *op* x, *op* of x, and the *op* of x. Note that the words 'the' and 'of' are obligatory parts of some other operation symbols. (Compare the HyperCard scripting language, HyperTalk [Goo87].)

B.2 Functional Action Notation

Functional action notation is primarily concerned with specifying the processing of transient information (data).

B.2.1 Actions

- *All primitive functional actions*:
  - Do not commit.
  - Produce no bindings.
  - Make no changes to storage.
  - Do not communicate.
- *All functional action combinators are*:
  - Declaratively conducting (see the basic action A_1 and A_2 in Section B.1.1).
- give Y: a primitive functional action, where Y is a data yielder. Represents creating a piece of transient information.
  - Indivisible. Completes when Y yields data. Fails when Y yields nothing.
  - Gives the data yielded by Y.

- escape with Y: a primitive functional action, where Y is a data yielder. Represents escaping with a piece of transient information, which may be used to distinguish different reasons for escape.
  - Indivisible. Escapes when Y yields data. Fails when Y yields nothing.
  - Gives the data yielded by Y.
- regive: a primitive functional action. Represents propagation of transient information, i.e., data. Unit for A_1 then A_2.
  - Indivisible. Always completes.
  - Gives any given data.
- choose Y: a functional action, where Y is a data yielder. Represents implementation-dependent choice between a possibly infinite collection of individual items of data.
  - Indivisible. Completes when Y yields a sort including a data individual. Fails when Y yields nothing.
  - Gives any individual data of the sort yielded by Y.
- check Y: a functional action, where Y is a truth-value yielder. Represents a guard checking that a condition is true.
  - Indivisible. Completes when Y yields true. Fails when Y yields false (or nothing).
  - Gives no data.
- A_1 then A_2: a functional combination of actions A_1, A_2. Represents passing on transient information normally.
  - Basically sequencing (see the basic action A_1 and then A_2 in Section B.1.1).
  - *Functionally composing*: The transients given to the combination of A_1, A_2 are given to A_1. Only the transients given by A_1 are given to A_2 (provided that A_2 is performed). Only the transients given by A_2 are given by the combined action.

B.2.2 Yielders

- given d: a data yielder, where d is a sort of data. Yields the transient data given to its evaluation, provided that the data is of sort d.
- given $d\#p$: a datum yielder, where d is a sort of datum and p is a positive integer. Yields the p'th component of the transient data given to its evaluation, provided that the datum is of sort d.
- it: a datum yielder. Yields the single datum given to its evaluation as a transient.
- them: a data yielder. Yields all the data given to its evaluation as transients.

B.2.3 Data

- data: a sort. Its individuals represent ordered collections, i.e., tuples, of individuals of sort datum, processed as transient information.

B.3 Declarative Action Notation

Declarative action notation is primarily concerned with specifying the processing of scoped information (bindings).

B.3.1 Actions

- *All primitive declarative actions*:
  - Do not commit.
  - Give no transients.

  - Make no changes to storage.
  - Do not communicate.

- ***All declarative action combinators are***:
  - Functionally conducting (see the basic action A_1 and A_2 in Section B.1.1).
- bind T to Y: a primitive declarative action, where T is a token and Y is a yielder of bindable data. Represents creating a piece of scoped information.
  - Indivisible. Completes when Y yields data of sort bindable. Fails otherwise.
  - Produces the binding of the token T to the bindable data.
- unbind T: a primitive declarative action, where T is a token. Represents hiding a piece of scoped information, making a hole in its scope.
  - Indivisible. Completes.
  - Produces the binding of the token T to the datum unknown.
- rebind: a primitive declarative action. Represents propagation of scoped information. Unit for the declarative action A_1 hence A_2.
  - Indivisible. Always completes.
  - Produces all received bindings.
- produce Y: a primitive declarative action. Represents production of reified scoped information.
  - Indivisible. Completes when Y yields a datum of sort bindings.
  - Produces the bindings yielded by Y.
- furthermore A: a declarative combination of the action A. Represents propagating the received bindings, but letting bindings produced by A take precedence when there is a conflict.
  - Basically as A.
  - Functionally as A.
  - Declaratively as rebind moreover A.
- A_1 moreover A_2: a declarative combination of actions A_1, A_2. Like A_1 and A_2, but gives priority to bindings produced by A_2.
  - Basically interleaving (see the basic action A_1 and A_2 in Section B.1.1).
  - *Declaratively overlaying*: The bindings received by the combination of A_1, A_2 are received by both A_1, A_2. On normal termination the bindings produced by A_1, overlaid with those produced by A_2, are produced by the combined action.
- A_1 hence A_2: a declarative combination of actions A_1, A_2. Represents passing on scoped information, restricting the bindings received by A_2.
  - Basically sequencing (see the basic action A_1 and then A_2 in Section B.1.1).
  - *Declaratively composing*: The bindings received by the combination of A_1, A_2 are received by A_1. Only the bindings produced by A_1 are received by A_2 (provided that it is performed). Only the bindings produced by A_2 are produced by the combined action.
- A_1 before A_2: a declarative combination of actions A_1, A_2. Represents accumulating scoped information.
  - Basically sequencing (see the basic action A_1 and then A_2 in Section B.1.1).
  - *Declaratively accumulating*: The bindings received by the combination of A_1, A_2 are received by A_1. The bindings received by the combined action, overlaid with the bindings produced by A_1, are received by A_2 (provided that it is performed). On normal termination the bindings produced by A_1, overlaid with those produced by A_2, are produced by the combined action.

B.3.2 Yielders

- current bindings: a yielder of bindings maps. Yields the collection of bindings received by its evaluation.
- the d bound to T: a yielder of bindable data, where d is a sort of data and T is a token. Yields the data of sort d to which T is bound by the received bindings, if any.

B.3.3 Data

- bindings: a subsort of map. Its individuals represent collections of associations between tokens and bindable (or unknown) individuals.
- token: a subsort of distinct-datum. Left unspecified, as it depends on the variety of identifiers of a programming language. (Usually, it is a subsort of string.)
- bindable: a subsort of data. Left open, as it depends on the variety of scoped information processed by the programs of a programming language.

B.4 Imperative Action Notation

Imperative action notation is primarily concerned with specifying the processing of stable information (storage).

B.4.1 Actions

- *All primitive imperative actions*:
  - Give no transients.
  - Produce no bindings.
  - Do not communicate.
- *There are no special imperative action combinators.*
- store Y_1 in Y_2: a primitive imperative action, where Y_1 is a yielder of storable data and Y_2 is a yielder of cells. Represents changing an atomic piece of stable information.
  - Indivisible. Commits and completes when Y_2 yields a reserved cell and Y_1 yields storable data. Fails otherwise.
  - Stores the storable yielded by Y_1 in the cell yielded by Y_2.
- reserve Y: a primitive imperative action, where Y is a yielder of cells. Represents extending stable information with an extra, uninitialized piece.
  - Indivisible. Commits and completes when Y yields an unreserved cell. Fails otherwise.
  - Reserves the cell yielded by Y and stores the datum uninitialized in it.
- unreserve Y: a primitive imperative action, where Y is a cell yielder. Represents destroying stable information.
  - Indivisible. Commits and completes when Y yields a reserved cell. Fails otherwise.
  - Unreserves the cell yielded by Y.

B.4.2 Yielders

- current storage: a yielder of storage maps. Yields the state of storage.
- the d stored in Y: a yielder of storable data, where d is a sort of data and Y is a yielder of cells. Yields the data of sort d stored in the cell yielded by Y according to the current storage, provided that the cell is currently reserved.

B.4.3 Data

- storage: a subsort of map. Its individuals represents states of stable information, associating cells with storable (or uninitialized) individuals.
- cell: a subsort of distinct-datum. Its individuals represent the locations of pieces of stable information. Left loosely-specified, as the details are implementation-dependent.
- storable: a subsort of data. Left unspecified, as it depends on the variety of stable information processed by the programs of a programming language.

B.5 Reflective Action Notation

Reflective action notation is concerned with specifying the *reification* of actions and information as abstractions, and with the *reflection* of abstractions as actions.

B.5.1 Actions

- enact Y: a reflective action, where Y is a yielder of abstractions. Represents performing an action in a context different from that of its occurrence.
  - Performs the action incorporated in the abstraction yielded by evaluating Y.
  - The performance of the incorporated action is given no data. (But see the abstraction yielder application d_1 to d_2.)
  - The performance of the incorporated action receives no bindings. (But see the abstraction yielder closure d.)

B.5.2 Yielders

- application d_1 to d_2: an abstraction, when d_1 is an abstraction and d_2 is data. Incorporates the same action as d_1, except that the incorporated action is given d_2 as transients whenever the abstraction is enacted. Represents supplying transients to an abstraction (precluding the supply of further transients).

  This operation extends to yielders Y_1, Y_2 in the usual way: it is evaluated by applying the above operation to the data yielded by evaluating Y_1, Y_2.
- closure d: an abstraction yielder, where d is an abstraction. Yields an abstraction which incorporates the same action as d, except that the incorporated action receives particular bindings whenever the abstraction is enacted. The bindings are those current for the evaluation of closure d.

  This operation extends to yielders Y in the usual way: it is evaluated by applying the above operation to the datum yielded by evaluating Y.

  The yielder closure abstraction of A represents reification of an action as an abstraction with static bindings. The action enact the closure of a given abstraction represents reflection of an abstraction with dynamic bindings (unless static bindings were already supplied to it). The action enact a given abstraction represents reflection of an abstraction with no bindings (unless static bindings were already supplied to it).

B.5.3 Data

- abstraction: the sort of datum that incorporates (i.e., reifies) an action. The incorporated action is performed when the abstraction is enacted (i.e., reflected).
- abstraction of A: an abstraction, where A is an action. Incorporates A. Represents (a pointer to) the 'code' implementing A. Yielders occurring in A do *not* get evaluated when the abstraction is evaluated: they are left for evaluation during the performance of the incorporated action.

B.6 Communicative Action Notation

Communicative action notation is primarily concerned with specifying the processing of permanent information (communications).

B.6.1 Actions

- *All primitive communicative actions*:
  - Give no transients.
  - Produce no bindings.
  - Make no changes to storage.
- *There is only a unary communicative action combinator.*
- send Y: a primitive communicative action, where Y yields a sort of message. Represents initiating 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 yield individuals.
  - Indivisible. Commits and completes when the sort of message yielded by Y includes a message from the current performer (with the next serial number). Fails otherwise.
  - Emits a message of the sort yielded by Y. The serial number of the message is the successor of that of the previous message sent (or contract offered) by the performing agent. Message transmission is reliable, but each message takes an implementation-dependent 'time' to arrive (so message transmission between two particular agents is not necessarily order-preserving).
- remove Y: a primitive communicative action, where Y is a yielder of individual messages. Represents that a particular message in the buffer has been processed and is to be discarded.
  - Indivisible. Commits and completes when the message yielded by Y is in the buffer. Fails otherwise.
  - Removes the message yielded by Y from the buffer.
- offer Y: a primitive communicative action, where Y yields sort of contract. Represents initiating 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.
  - Indivisible. Commits and completes when the sort of contract yielded by Y includes an individual contract from the performer. Fails otherwise.
  - Gives no data.
  - Requests the arrangement of a contract of the sort yielded by Y. Offered contracts are distinguished by serial numbers, as with sent messages. The arrangement of a contract takes an implementation-dependent 'time', which must be finite when there is a continually uncontracted agent of the specified sort.
- patiently A: a communicative action, where A is an action. Represents *busy waiting* while A fails. Only useful when A refers to information that may change due to some other action, for instance, the messages in the buffer.
  - Performs A indivisibly, but not indivisible itself. If the performance fails it tries again. Thus it diverges when A always fails.
  - Functionally as A.
  - Declaratively as A.

B.6.2 Yielders

- current buffer: a yielder of buffers. Yields the list of messages that have appeared in the buffer, but which have not yet been removed. The messages are listed in the order of their arrival in the buffer.
- performing-agent: a yielder of agents. Yields (the identity of) the agent that is performing the enclosing action.
- contracting-agent: a yielder of agents. Yields (the identity of) the agent that offered the contract to the performer.

B.6.3 Data

- agent: a sort of datum. An agent identifies a potential process of performing an action, representing a piece of distributed processing. It is loosely-specified, as the maximum number and distribution of processes is usually implementation-dependent.

  Each agent has its own buffer and storage. It is inactive until it accepts a contract to perform an action, whereafter it remains active for ever, even after the termination of the contracted action.
- user-agent: a distinguished agent. It corresponds to the environment of a program, providing input and accepting output. The user agent is initially the only agent with a contract.
- buffer: a sort of datum. A buffer is a list of messages sent to the same agent, in the order of their arrival.
- communication: a sort of datum. Individual communications represent information that can be transmitted by agents. Communications have components indicating their sender, receiver, and contents. Moreover, each communication is distinguished by a serial number determined by the sender.
- message: a subsort of communication. Messages can be sent directly from one agent to another.
- sendable: a sort of data. The data that can be the contents of messages sent between agents. Left open, as it depends on the variety of permanent information processed by the programs of a programming language. (Specified to include abstraction and agent to allow proper use of subordinate Y.)
- contract: a subsort of communication. Contracts can be offered by one agent to another (sort of) agent. The contents of a contract is the abstraction to be enacted by an agent accepting the contract.
- contents d: data, where d is a communication. The data contained in d.
- sender d: a datum, where d is a communication. The agent that sends d.
- receiver d: a datum, where d is a communication. The agent that receives d.
- serial d: a datum, where d is a communication. The serial number of d, determined locally when it is emitted.
- d [containing d_1]: a subsort of communication, where d_1 is a (sort of) data, and d is a sort of communication. It includes only those communications in d whose contents is (of sort) d_1.
- d [from d_1]: a subsort of communication, where d_1 is a (sort of) agent, and d is a sort of communication. It includes only those communications in d whose sender is (of sort) d_1.
- d [to d_1]: a subsort of communication, where d_1 is a (sort of) agent, and d is a sort of communication. It includes only those communications in d whose receiver is (of sort) d_1.
- d [at d_1]: a subsort of communication, where d_1 is a (sort of) natural number, and d is a sort of communication. It includes only those communications in d whose serial number is (of sort) d_1.

B.7 Hybrid Action Notation

Hybrid action notation consists of some useful abbreviations for compound actions involving more than one kind of information, together with some hybrid combinators that mix various facets of the other combinators. There are also some hybrid data operations.

B.7.1 Actions

- allocate d: an imperative and functional action, where d is a sort of cell. Represents implementation-dependent choice and reservation of a cell of sort d.
  - Indivisible. Commits and completes when there is an unreserved cell of sort d. Fails otherwise.
  - Reserves some cell of sort d.
  - Gives the reserved cell.
- receive Y: a communicative and functional action, where Y yields a sort of message. Represents waiting for a message to arrive in the buffer. The usual form of Y is a restriction such as a message [from Y_1] [containing Y_2], where Y_1, Y_2 may yield sorts or individuals.

  - Patiently waits for a message of the sort yielded Y to arrive, then commits and completes. Otherwise diverges.
  - Chooses and gives any received message of the sort yielded by Y.
  - Removes the chosen message from the buffer.

- A_1 thence A_2: a declarative and functional hybrid combination of actions A_1, A_2. Like A_1 then A_2 for control and transients, and like A_1 hence A_2 for bindings.

B.7.2 Yielders

- *There are no hybrid yielders.*

B.7.3 Data

- owner d: an agent, where d is a cell or an indirection. The agent where it is located.
- d [on d_1]: a subsort of cell, where d_1 is a (sort of) agent, and d is a sort of cell. It includes only those cells in d whose owner is (of sort) d_1. Similarly when d is a (sort of) indirection.

B.8 Facets

The facets of actions and yielders are obtained by focusing on particular kinds of information. The notation summarized below is used for expressing ***sorts*** of actions and yielders.

B.8.1 Outcomes

- outcome: a sort of auxiliary entities, used for specifying sorts of actions. Sort union O_1 **|** O_2 combines outcomes. (The various outcomes below are disjoint, so there is no use for specifying their intersections.)
- giving d: allows normal termination that always gives transients included in the data sort d. completing abbreviates giving ( ), where ( ) is the empty tuple of data.
- escaping with d: allows abnormal termination that always gives transients included in the data sort d. escaping abbreviates escaping with data.
- binding: allows normal termination that optionally produces some bindings. The union outcome giving d **|** binding allows normal termination that always gives transients of sort d and optionally produces some bindings. The use of binding alone implies completing.
- failing: ignored. (Most compound actions that use information fail when that information is not made available to their performance.)
- committing: allows commitment, independently of termination. Included in storing, and communicating, which are analogous.
- diverging: allows nontermination. Actions without this sort of outcome are guaranteed to terminate, whereas those with this outcome might or might not terminate. Actions that contain any occurrence of unfolding or enact Y have this sort of outcome, unless they are equivalent to other actions that do not have it.

B.8.2 Incomes

- income: a sort of auxiliary entities, used for specifying sorts of actions and yielders. Sort union I_1 **|** I_2 combines incomes. (The various incomes below are disjoint, so there is no use for specifying their intersections.)
- given d: allows use of given transients of the data sort d. More specifically, the income given $d\#p$ specifies possible use of the p'th component of the given transients, this being of datum sort d.
- current bindings: allows use of received bindings.

- current storage: allows use of storage.
- current buffer: allows use of the message buffer.

B.8.3 Actions

- action: the sort of all actions. Its subsort primitive-action includes not only the actions described in this Appendix as primitive, but also compound actions that are equivalent to them, e.g., complete and then escape, which is equivalent to escape.
- A [O]: a sort of action, where A is a sort of action and O is a sort of outcome. Restricts A to those actions which, whenever performed, either fail or have an outcome whose termination properties and kind of information processing are included in O. Note that A [O_1] | A [O_2] is generally a proper subsort of A [O_1 | O_2], whereas A [O_1] & A [O_2] is the same sort as A [O_1 & O_2], which can also be written as A [O_1] [O_2].
- A [using I]: a sort of action, where I is a sort of income. Restricts A to those actions which, whenever performed, perhaps evaluate yielders that refer to the current information indicated by I. Compare Y [using I], where Y is a sort of yielder, below.

B.8.4 Yielders

- yielder: the sort of all yielders.
- Y [d]: a sort of yielder, where Y is a sort of action and d is a sort of data. Restricts Y to those yielders which, whenever evaluated, yield data included in d. Note that the union sort Y [d_1] | Y [d_2] is generally a proper subsort of Y [d_1 | d_2], whereas Y [d_1] & Y [d_2] is the same sort as Y [d_1 & d_2], which can also be written as Y [d_1] [d_2].
- Y [of d]: equivalent to Y [d], but a yielder [of an integer] reads a bit more naturally than yielder [integer].
- Y [using I]: a sort of yielder, where I is a sort of income. Restricts Y to those yielders which, whenever evaluated, refer at most to the current information indicated by I.

Appendix C

Data Notation

This Appendix will provide an informal summary of the data notation used in the example.

Bibliography

[Goo87] Danny Goodman. *The Complete HyperCard Handbook.* Bantam, 1987.

[Mos90] Peter 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.

[Mos92] Peter D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Sto88] Allen Stoughton. *Fully Abstract Models of Programming Languages.* Pitman & John Wiley, 1988.