Why Don't Programming Language Designers Use Formal Methods?[1]

David A. Watt

Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ,
Scotland.
E-mail: daw@dcs.gla.ac.uk.

Abstract. It is surprising and disappointing that programming language designers make such little use of formal methods. Most languages are specified informally. Even if a formal specification is undertaken, it tends to be too late to influence the language design. And yet informal specifications are notoriously difficult to make consistent and complete, and are not precise enough to ensure that all implementations are conformant.

This paper advocates placing formal specification at the centre of programming language development. The designer's ideas should be immediately embodied in a formal specification of the language's syntax and semantics, giving very early feedback on design irregularities and inconsistencies. The formal specification should then be used to generate a prototype implementation, allowing programmers to gain early experience with the new language and thus provide further feedback to the designer. Finally compilers should be developed, along with reference manuals and textbooks.

Such a central role for formal specification places heavy demands on the chosen formalism: too heavy demands for such as natural semantics and denotational semantics. The paper shows that action semantics, a relatively new contender, has desirable pragmatic qualities that make it an excellent tool for the language designer.

1 Introduction

There are all too many programming languages around today: probably between 500 and 1,000. Only a few of these, however, could by any reasonable measure be classified as major languages: probably about twenty. ADA, ALGOL, BASIC, C, C++, COBOL, FORTRAN, HASKELL, JAVA, LISP, ML, MODULA, PASCAL, PROLOG, SCHEME, and SMALLTALK would be on most people's lists.

With a few obvious exceptions, the major languages were designed by distinguished computing scientists. Presumably these scientists were all acquainted with well-established methods for formally specifying the syntax and semantics of programming languages. It is striking, therefore, how limited a role has been played by formal methods in the development of these languages. Formal specifications of syntax are commonplace, using BNF or one of its variants. But formal specifications of semantics are uncommon, tend to appear years after the corresponding informal specifications, are usually incomplete and/or inaccurate, and are rarely treated as definitive. Language designers still rely on informal specifications, presumably knowing very well that they strongly tend to be incomplete, inconsistent, and ambiguous.

In this paper I explore the reasons for this state of affairs, and suggest that it is time for language designers to reconsider their attitude to formal methods. Section 2 examines the life cycles of some major languages. Section 3 advocates a life-cycle model based on immediate formal specification of the newly designed language. Section 4 shows that this model places strong demands on the chosen semantic formalism, and argues that action

[1] Presented at SEMISH'96, University of Pernambuco, Recife, Brazil, August 1996.

semantics satisfies these demands better than more well-known formalisms. Section 5 concludes.

2 Programming Language Life Cycles

The notion of a software life cycle is well known, and is explained in numerous software engineering textbooks (e.g., Sommerville 1996). Requirements capture is followed by system design and specification, detailed design and implementation, installation and maintenance. Verification, validation, and prototyping during the intermediate stages stimulate early feedback to earlier stages, when problems can be fixed relatively quickly and cheaply. Later, maintenance also may uncover problems and stimulate feedback to earlier stages; but by now the software has been distributed and the problems are much more expensive to fix.

Programming languages also have life cycles. Indeed, many of the stages are broadly similar: requirements, design, specification, and implementation. Programming language life cycles, however, have not been studied anything like as intensively as software life cycles.

An interesting example is the life cycle of PASCAL, which is summarised in Figure 1. PASCAL was designed by Niklaus Wirth (1971). Its design goals were conservative: to be simple and consistent, consolidating a number of concepts already tested in other languages; to be suitable for teaching; and to be easy to implement efficiently on current machines. The language and its first compiler (targeting the CDC 6600) were designed together. This approach stimulated early feedback on implementation problems. However, the design was unduly influenced by the peculiarities of the target machine; ease of implementation tended to override the goal of consistency.

Later, Hoare and Wirth (1973) collaborated on an axiomatic specification of a large subset of PASCAL. This exercise exposed some of the design flaws, and led to a revised design. This, of course, required modifications to all the existing compilers. Even after this design iteration, the language still contained many irregularities, such as violations of the type completeness principle, and insecurities, such as variant records.

The informal specification of PASCAL (Jensen and Wirth 1974) was not actually published until several years after the initial design. Even so it proved to be thoroughly unsatisfactory: it permitted different compilers to adopt subtly different interpretations of the language's scope rules, and diametrically opposite interpretations of its type equivalence rules!

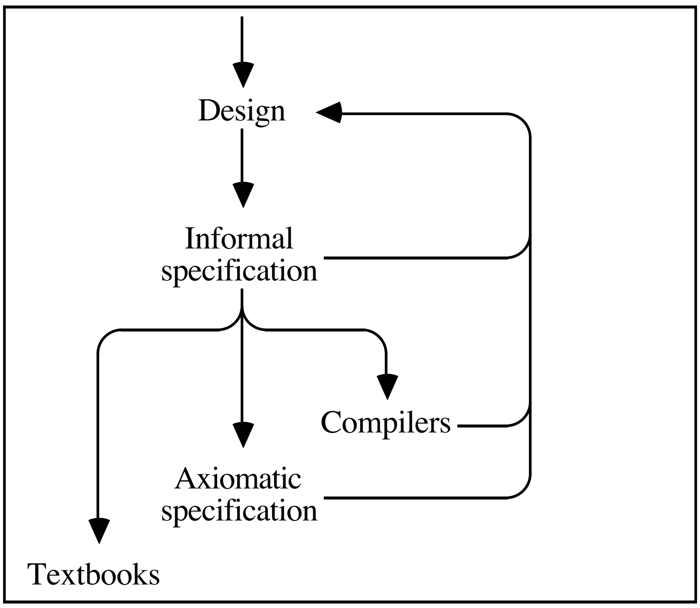

Figure 1 PASCAL's life cycle

Another instructive example is the life cycle of ADA-83, which is summarised in Figure 2. This language was designed to an ambitious set of requirements prepared by the US Department of Defense. Following a competition, the design contract was awarded to a team led by Jean Ichbiah. The team based their design partly on well-tried concepts borrowed from PASCAL and other languages, taking care to learn from these languages' mistakes. However, the team also incorporated more advanced concepts, such as tasks and rendezvous, exceptions, and generics, that were not well understood at the time.

In order to communicate design proposals among themselves, and to solicit comments from a still wider group of expert reviewers, the team developed an informal specification from the very beginning. Unusually, they also prepared a design rationale (Ichbiah 1979). The informal specification went through numerous drafts until it was finalised (Ichbiah 1983). An enormous amount of effort and care went into its preparation.

The first ADA-83 compilers appeared only after the language design was frozen. Thus implementation problems were not exposed until too late to influence the design. This was a risky strategy, and it turned out that some of the more advanced concepts were difficult to implement efficiently or had unexpected interactions.

Although ADA-83's semantic specification was informal, its syntax was as usual specified formally, in EBNF. Surprisingly, however, no-one thought to test the syntax using readily available tools. A parser generator would have quickly discovered certain syntactic problems – for example, the grammar was not LALR(1) – which could easily have been fixed.

There have been two projects to develop semantic specifications of ADA-83. The first was a denotational specification (Donzeau-Gouge *et al.* 1980), but this proved to be hugely unwieldy and was never completed. Although this project proceeded concurrently with the main project, it does not seem to have influenced the language design. The second project (DDC 1986) developed a complete specification expressed in VDM, enhanced with special features to specify concurrency. Although complete, this project came too late to influence the design of ADA-83. Significantly, however, this project did influence the development of a successful ADA-83 compiler.

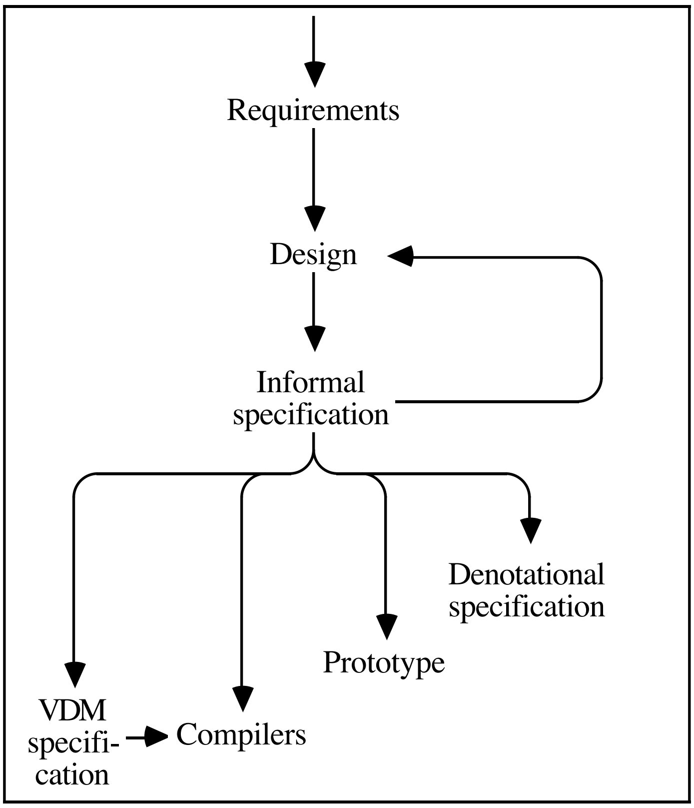

Figure 2 ADA's life cycle

These two case studies are not untypical. It is disappointing that the language design methods of the 1970s and 1980s are still being used in the 1990s. At the time of writing, even the definitive informal specification of JAVA has not yet been published, although several compilers and textbooks exist, and JAVA programs are being executed across the Internet. There is no sign of a formal specification. In these circumstances, how can anyone be confident that a JAVA program will behave as expected when executed remotely?

The functional programming community is somewhat more receptive to formal methods. Standard ML (SML) seems to have been designed with a formal semantic specification at least in mind, if not on paper. Curiously, its (concrete) *syntax* was not formally specified, and in some respects is messy! The definitive semantic specification existed as a working document long before being published (Milner *et al.* 1990). A development of this specification was used throughout the design of the successor language ML-96, apparently with success. A counter-example is HASKELL, which was designed by a large team relying exclusively on an informal specification for communication and feedback (Hudak 1992).

All our experience suggests that language design is very hard. If the designer is to do a good job, the design process must be iterative, and informed by timely high-quality feedback. Such feedback is possible only if the language design is somehow communicated to other people, whether by a formal or informal specification or by an implementation.

Feedback from a draft informal specification is of course important, but it cannot be sufficient on its own. Neither the designer nor the reviewers can actually *test* an informally specified language design. To discover problems they must rely on their own experience of similar languages (or similar language constructs), but this is no help when the new language incorporates new and untried constructs.

Feedback from an implementation is extremely valuable, indeed essential at some stage of the language's life cycle. An untried construct, however good it looks on paper, might turn out to be awkward and confusing for programmers to use in practice. Or it might turn out to be grossly inefficient to implement – perhaps even, simply by its presence, impairing the efficiency of programs that do not use it. Likewise, an untried combination of constructs might have unexpected interactions. However, the designer should take care not to rely exclusively on implementation feedback. A construct that substantially reduces the cost of programming, for example by making programs easier to write or more robust, might justify its adoption even if somewhat inefficient – especially if it does not penalise programs that do not use it. The point here is that the language designer, faced with a difficult trade-off, needs more than one kind of feedback.

A good example is PASCAL's restriction of function results to be primitive values or pointers. It is true that composite (e.g., array or record) function results are awkward to implement efficiently on some target machines. However, this problem surely does not justify such a blatant violation of the type completeness principle. This restriction is absent not only in ADA but also in many PASCAL dialects.

Feedback from a formal specification is also extremely valuable. The very act of writing a formal specification forces the specifier explicitly to address issues, such as completeness and consistency, that are all too easily fudged in an informal specification. Irregular and messy constructs tend to be exposed mercilessly by a formal specification.

Consider, for example, the mechanism by which a function result is determined in various languages. In PASCAL this is done by assignment to the function identifier, e.g., "power := x * power(x, n-1)". Any attempt to formalise this design shows how ugly it is: outside the function body, the function identifier denotes the function itself; inside it simultaneously denotes both the function and a special result variable! In ADA the function result is determined by a return-statement, e.g., "**return** x * power(x, n–1);". This is cleaner, but the semantics are complicated by the return-statement's other role as a sequencer. In ML, the function body is simply an expression, and the function returns the value yielded by this expression. Formal specification discriminates sharply among these possible designs: the PASCAL one is clumsy to formalise, the ADA one is intermediate, and the ML one is trivial. Implementation does not discriminate sharply: all three designs are easy to implement on a real or abstract target machine. The language designer might be tempted to regard the last point as conclusive; who cares whether a particular design is hard to formalise? But this is too facile. After all, the construct has to be explained to programmers, and it turns out that the PASCAL design is hardest, and the ML design easiest, to explain *precisely*.

Timeliness is just as important as the quality of the feedback. Feedback from an implementation must inevitably be slow, especially when the language contains untried constructs for which implementation techniques have not already been developed. The first PASCAL compiler took about six months to develop. The first ADA compiler was completed four years after the first complete draft of the language design.

3 A Central Role for Formal Methods

The designer has to specify the new language, either formally or informally, in order to communicate his/her ideas to other people and thus to get feedback. I believe that there are major benefits to be gained from writing a formal specification at the earliest possible stage.

A formal specification should take much less time to write than a compiler, all else being equal. The specifier is not concerned with extraneous details like the architecture and instruction set of a target machine, or the algorithms and data structures required for translation to the target machine code.

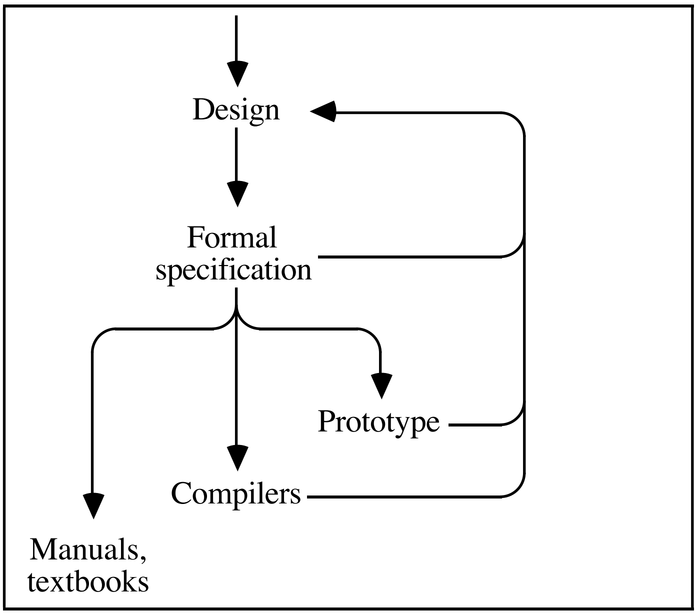

Figure 3 Proposed life cycle centred on formal specification

I propose the life-cycle model shown in Figure 3. The language designer *immediately* commits his/her ideas to a formal specification. This stimulates rapid high-quality feedback, by exposing irregularities, inconsistency, and incompleteness in the design. The designer can then improve the design, modify the specification accordingly, and iterate again. The feedback can come both from the designer/specifier him/herself and from colleagues willing to review the formal specification.

Once the design has stabilised somewhat, the formal specification can be used to generate a prototype implementation. This assumes that the formalism is equipped with suitable prototyping tools. The prototype allows programmers to gain early experience with the new language. By definition it will not be a fully-engineered implementation; it will probably be inefficient, unrobust, limited in capacity, and lacking in diagnostic facilities. Nevertheless it should provide the designer with good feedback on the pragmatics of the new language. This may suggest a further iteration with improvements to the design, modifications to the formal specification, and regeneration of the prototype.

Once the design has re-stabilised, it is time for constructing a first compiler. This will be the longest stage in the life cycle (which is the reason for leaving it to last). Nevertheless it is essential because the experience gained by the implementor will provide feedback on implementation problems. A construct might prove to be intolerably inefficient, in which case it will certainly have to be changed. Or it might prove to have tolerable but significant cost, either at compile-time or at run-time. At this point the designer has to balance the implementation cost against the merits of the proposed construct. This kind of trade-off is never easy, but at least the designer has all the information required to make a rational judgement. If the construct has to be changed, then there must be a further iteration of design, specification, and implementation.

At the very least, the formal specification should act as a guide to the implementor, the definitive source to which the implementor refers to confirm (or not) his/her intuitive understanding of the language. In the long term, perhaps, the compiler might be able to be generated automatically from the formal specification. This will depend on the success of current and future research on semantics-directed compiler generation.

When the design is frozen, it is finally time for longer-term projects such as writing of reference manuals and textbooks, re-engineering the first compiler, and developing further compilers. At last the language has matured from infancy to adolescence, if not yet adulthood!

4 The Choice of Formalism

My proposed life-cycle model clearly places formal specification right at the centre. Whether this model is at all practicable will depend critically on the pragmatic qualities of the formalism chosen for this purpose.

A formal specification should be easily *writable*. At least, it should be no harder to write than a (precise) informal specification of the same language. Ideally, the formal specification should be written by the designer personally, to make the feedback as rapid and high-quality as possible. The designer/specifier, however, should be able to concentrate on the creative aspects of language design, and this will not happen if writing the specification is a time-consuming chore.

Just as important, the formal specification should be easily *modifiable*. Language design is inherently iterative, as emphasised in the previous section. Ideally, the effort required to modify the formal specification should be commensurate with the size of the language change. It is bad news if a small language change enforces a major rewriting of the formal specification.

It is also highly desirable that (parts of) a specification should be *reusable*. In practice, every new language is based at least partly on an older language. If the older language already has a suitable formal specification, and parts of it can be reused in specifying the new language, that obviously saves the designer/specifier a great deal of work.

Last, but not least, the formal specification should be *readable*. The same specification has to be read by reviewers, by implementors, and by the authors of manuals and textbooks. Ideally, it should even be read by ordinary programmers, since it will be the definitive specification of the new language.

These pragmatic qualities are all possessed by the well-known *syntactic* formalism BNF (and its variants). Not surprisingly, therefore, these formalisms are used in almost all language specifications. Where this is not the case, for example FORTRAN, it is noteworthy that the syntax tends to be irregular and confusing.

Unfortunately, these pragmatic qualities are not possessed by the more well-known *semantic* formalisms such as natural semantics and denotational semantics. As a consequence, only a few languages have formal semantic specifications. Moreover, these specifications are generally perceived to be forbidding documents, intelligible only to a handful of experts. These formalisms are not attractive tools for the language designer.

Space does not permit me to illustrate this important point on the scale of a real programming language, so I will present a small-scale example and invite the reader to scale it up mentally. The example is inspired by the development of ML. Our objective is to design a practical functional language. The design will be incremental, starting with a very simple pure functional language, and later adding impure features such as exceptions and state.

Natural semantics

Natural semantics is one of the most well-known and popular semantic formalisms. It has been used for specifying the dynamic semantics of functional, imperative, and concurrent programming languages. It has also proved useful for specifying static semantics, including that of languages with polymorphic type inference. A major example is its use for specifying the static and dynamic semantics of SML (Milner *et al.* 1990).

The basic idea of natural semantics is to formulate the semantics in terms of *judgements*, which are typically of the form:

$$\textit{context} \vdash \textit{phrase} \Rightarrow \textit{result}$$

Here the context might be a set of bindings, a store, or both. The phrase might be an expression, a command, or a declaration. The result might be a value, a set of bindings, an updated store, or some combination of these. The semantics of the programming language is specified by a collection of *inference rules*, one or more rules for each form of phrase in the language. Each inference rule has a judgement as its consequent, and zero or more judgements (or other assertions) as antecedents.

We start our language design with an enriched call-by-value lambda-calculus, which will be version 1 of our new language. Its abstract syntax and natural semantics are shown in Figure 4. We have written the inference rules in terms of judgements of the form:

$$b \vdash E \Rightarrow v$$

meaning that expression E yields value v when evaluated in the scope of bindings b (a partial map of identifiers to values). A value is either a number n or a function $fn(b, I, E)$, where b denotes the function's static bindings, I its parameter identifier, and E its body.

Syntax:

$$E \ ::= \ I \ \mid \ \mathbf{fn}\ I\ \mathbf{.}\ E \ \mid \ E_1\ E_2 \ \mid \ E_1 \mathbin{\mathbf{+}} E_2 \ \mid \ \mathbf{let}\ I = E_1\ \mathbf{in}\ E_2$$

Semantics:

(var) $$\frac{b(I) = v}{b \vdash I \Rightarrow v}$$

(abs) $$b \vdash \mathbf{fn}\ I\ \mathbf{.}\ E \Rightarrow fn(b, I, E)$$

(app) $$\frac{b \vdash E_1 \Rightarrow fn(b', I', E') \ ; \ b \vdash E_2 \Rightarrow v \ ; \ (I' \mapsto v)b' \vdash E' \Rightarrow v'}{b \vdash E_1\ E_2 \Rightarrow v'}$$

(sum) $$\frac{b \vdash E_1 \Rightarrow n_1 \ ; \ b \vdash E_2 \Rightarrow n_2}{b \vdash E_1 \mathbin{\mathbf{+}} E_2 \Rightarrow sum(n_1, n_2)}$$

(let) $$\frac{b \vdash E_1 \Rightarrow v_1 \ ; \ (I \mapsto v_1)b \vdash E_2 \Rightarrow v_2}{b \vdash \mathbf{let}\ I = E_1\ \mathbf{in}\ E_2 \Rightarrow v_2}$$

Figure 4 Abstract syntax and natural semantics of version 1.

Now suppose that we wish to extend the new language with exceptions, giving us version 2. We extend the syntax with a raise-expression (which will yield an exception) and a handle-expression (which will trap one particular exception but propagate any other exception), as in Figure 5. In the semantics, we must modify the judgements for expressions to the form:

$$b \vdash E \Rightarrow y$$

where the expression's "yield" y is either a value v or an exception e. The bindings b now map identifiers to either values or exceptions. We express the semantics of the new expressions by the additional inference rules (raise) and (handle1,2,3).

We have more work to do, however. We must modify nearly all of the existing inference rules, and add further rules, to take account of the modified form of judgements. For example, consider the semantics of function application, "E_1 E_2". Not only must we slightly modify rule (app) to (app1), to allow for the possibility that the function result can now be either a value or an exception; we must also add inference rules (app2,3), to allow for the possibility that E_1 or E_2 can raise an exception, since in either case the function call will not actually take place.

Syntax:

$$E \ ::= \ \ldots \mid \textbf{raise } I \mid E_1 \textbf{ handle } I \textbf{ . } E_2$$

Semantics:

(var) *as before*

(abs) *as before*

(app1)
$$\frac{b \vdash E_1 \Rightarrow fn(b', I', E') \ ; \ b \vdash E_2 \Rightarrow v \ ; \ (I' \mapsto v)b' \vdash E' \Rightarrow y}{b \vdash E_1\ E_2 \Rightarrow y}$$

(app2,3)
$$\frac{b \vdash E_1 \Rightarrow e}{b \vdash E_1\ E_2 \Rightarrow e} \qquad \frac{b \vdash E_1 \Rightarrow fn(b', I', E') \ ; \ b \vdash E_2 \Rightarrow e}{b \vdash E_1\ E_2 \Rightarrow e}$$

(sum1)
$$\frac{b \vdash E_1 \Rightarrow n_1 \ ; \ b \vdash E_2 \Rightarrow n_2}{b \vdash E_1 \mathbin{\textbf{+}} E_2 \Rightarrow sum(n_1, n_2)}$$

(sum2,3)
$$\frac{b \vdash E_1 \Rightarrow e}{b \vdash E_1 \mathbin{\textbf{+}} E_2 \Rightarrow e} \qquad \frac{b \vdash E_1 \Rightarrow n_1 \ ; \ b \vdash E_2 \Rightarrow e}{b \vdash E_1 \mathbin{\textbf{+}} E_2 \Rightarrow e}$$

(let1,2)
$$\frac{b \vdash E_1 \Rightarrow v_1 \ ; \ (I \mapsto v_1)b \vdash E_2 \Rightarrow y}{b \vdash \textbf{let } I = E_1 \textbf{ in } E_2 \Rightarrow y} \qquad \frac{b \vdash E_1 \Rightarrow e}{b \vdash \textbf{let } I = E_1 \textbf{ in } E_2 \Rightarrow e}$$

(raise)
$$\frac{b(I) = e}{b \vdash \textbf{raise } I \Rightarrow e}$$

(handle1)
$$\frac{b \vdash E_1 \Rightarrow v}{b \vdash E_1 \textbf{ handle } I \textbf{ . } E_2 \Rightarrow v}$$

(handle2,3)
$$\frac{b \vdash E_1 \Rightarrow e \ ; \ e = b(I) \ ; \ b \vdash E_2 \Rightarrow y}{b \vdash E_1 \textbf{ handle } I \textbf{ . } E_2 \Rightarrow y} \qquad \frac{b \vdash E_1 \Rightarrow e \ ; \ e \neq b(I)}{b \vdash E_1 \textbf{ handle } I \textbf{ . } E_2 \Rightarrow e}$$

Figure 5 Abstract syntax and natural semantics of version 2.

Now suppose that we wish to extend the language with state. This introduces a new type of value, a reference to a storage cell c. We extend the syntax with a reference-allocating expression, a dereferencing expression, and an assignment expression, as shown in Figure 6. Now we must modify the form of judgements yet again:

$$b, s \vdash E \Rightarrow y, s'$$

where s and s' are the stores immediately before and after evaluating E. It turns out that this modification forces us to rewrite *all* the existing inference rules! Extracts are shown in Figure 6.

This simple example makes it very clear that modifications to a natural-semantic specification are not necessarily commensurate with the changes to the language itself. If the judgements have to be modified, most of the inference rules are going to be affected. In other words, the natural-semantic specification is unmodular.

Syntax:

$$E \ ::= \ \ldots \mid \textbf{ref}\, E \mid \textbf{!}\, E \mid E_1 \textbf{ := } E_2$$

Semantics:

(var) $$\frac{b(I) = v}{b, s \vdash I \Rightarrow v, s}$$

(sum1) $$\frac{b, s \vdash E_1 \Rightarrow n_1, s' \;;\; b, s' \vdash E_2 \Rightarrow n_2, s''}{b, s \vdash E_1 \textbf{ + } E_2 \Rightarrow sum(n_1, n_2), s''}$$

(sum2,3) $$\frac{b, s \vdash E_1 \Rightarrow e, s'}{b, s \vdash E_1 \textbf{ + } E_2 \Rightarrow e, s'} \qquad \frac{b, s \vdash E_1 \Rightarrow n_1, s' \;;\; b, s' \vdash E_2 \Rightarrow e, s''}{b, s \vdash E_1 \textbf{ + } E_2 \Rightarrow e, s''}$$

(ref1) $$\frac{b, s \vdash E \Rightarrow v, s' \;;\; (c, s'') = allocate(s') \;;\; s''' = update(s'', c, v)}{b, s \vdash \textbf{ref}\, E \Rightarrow c, s'''}$$

(ref2) $$\frac{b, s \vdash E \Rightarrow e, s'}{b, s \vdash \textbf{ref}\, E \Rightarrow e, s'}$$

(deref1,2) $$\frac{b, s \vdash E \Rightarrow c, s' \;;\; v = fetch(s', c)}{b, s \vdash \textbf{!}\, E \Rightarrow v, s} \qquad \frac{b, s \vdash E \Rightarrow e, s'}{b, s \vdash \textbf{!}\, E \Rightarrow e, s'}$$

(assign1) $$\frac{b, s \vdash E_1 \Rightarrow c, s' \;;\; b, s' \vdash E_2 \Rightarrow v, s'' \;;\; s''' = update(s'', c, v)}{b, s \vdash E_1 \textbf{ := } E_2 \Rightarrow v, s'''}$$

(assign2,3) $$\frac{b, s \vdash E_1 \Rightarrow e, s'}{b, s \vdash E_1 \textbf{ := } E_2 \Rightarrow e, s'} \qquad \frac{b, s \vdash E_1 \Rightarrow c, s' \;;\; b, s' \vdash E_2 \Rightarrow e, s''}{b, s \vdash E_1 \textbf{ := } E_2 \Rightarrow e, s''}$$

Figure 6 Abstract syntax and natural semantics of version 3 (extracts only).

In the natural-semantic specification of SML (Milner *et al.* 1990), the authors side-step the problems we have discussed by adopting a couple of conventions. Their *exception convention* allows them to omit rules, like (app2,3) and (sum2,3) in the example, where evaluation is terminated by a subexpression raising an exception. Their *state convention* allows them to omit the stores s, s', … in judgements and inference rules, except in those rules where the store explicitly plays a part, like (ref1), (deref1), and (assign1) in the example. Both these conventions capture the essence of left-to-right evaluation. They undoubtedly reduce the clutter of inference rules, and without them the natural-semantic specification of SML would hardly have been practicable. However, these conventions are not themselves formal, so the SML specification is only semi-formal! Furthermore, these conventions are unlikely to be supported by any natural-semantic tools.

Action semantics

Action semantics (Mosses 1992, Mosses and Watt 1987, Watt 1991) was developed with pragmatic qualities as a major objective. Experience has shown that action semantics is suitable for specifying the semantics of imperative, object-oriented, functional, and concurrent programming languages, e.g., a subset of ADA (Mosses 1992), PASCAL (Mosses & Watt 1993), and SML (Watt 1988). Moreover, these specifications are unusually writable, modifiable, reusable, and readable. These qualities are consequences of the design of action notation. Familiar programming language concepts like binding, storing, sequencing, iteration, etc., are directly expressed in action notation. (In other semantic notations they have to be encoded.) Moreover, the action notation is English-like, and an action-semantic specification reads rather like an operational specification expressed in English.

An *action*, when performed, will either complete, escape, fail, or diverge. If it completes or escapes, it may give out transient data, for immediate use by another action. If it completes, it may produce bindings, whose scope will be another action. It may update stored data. The action may use incoming transients, bindings, and/or stored data.

A *primitive action* has a single effect. "give *d*" gives out the data yielded by the term *d*. "bind *I* to *d*" produces a binding of identifier *I* to the data yielded by *d*. "store *d* in *c*" updates the storage cell yielded by *c* to contain the data yielded by *d*. "check *d*" completes if *d* yields true, otherwise it fails.

An *action combinator* mediates control flow and data flow between its subactions. "A_1 or A_2" performs either A_1 or A_2, but if that subaction fails then it performs the other one instead. "A_1 and A_2" performs A_1 and A_2 in no particular order, and if both complete it merges their outgoing transients and bindings. Other combinators perform their subactions in sequence. "A_1 and then A_2" is otherwise similar to "A_1 and A_2". "A_1 then A_2" makes transients flow from A_1 into A_2. "A_1 hence A_2" makes bindings flow from A_1 into A_2, which thus becomes the scope of these bindings.

Syntax:

```
Expression = Identifier | ⟦"fn" Identifier "." Expression⟧ |
             ⟦Expression Expression⟧ | ⟦Expression "+" Expression⟧ |
             ⟦"let" Identifier "=" Expression "in" Expression⟧ .
```

Semantics:

```
        evaluate _  ::  Expression →
                             action [giving a value] [using current bindings] .

(var)   evaluate I:Identifier  =
              give the value bound to I .

(abs)   evaluate ⟦"fn" I:Identifier "." E:Expression⟧  =
              give closure of abstraction of
                 ( furthermore  bind I to the given value
                   hence  evaluate E  ) .

(app)   evaluate ⟦E1:Expression E2:Expression⟧  =
              ( evaluate E1  and  evaluate E2  )  then
              enact application of the given abstraction#1 to the given value#2 .

(sum)   evaluate ⟦E1:Expression "+" E2:Expression⟧  =
              ( evaluate E1  and  evaluate E2  )  then
              give sum (the given number#1, the given number#2) .

(let)   evaluate ⟦"let" I:Identifier "=" E1:Expression "in" E2:Expression⟧  =
              furthermore  (  evaluate E1  then  bind I to the given value  )
              hence  evaluate E2 .
```

Figure 7 Abstract syntax and action semantics of version 1.

An action-semantic specification is formulated in terms of *semantic functions*, each of which maps phrases (of a particular syntactic class) to actions. Each semantic function is defined by one or more *semantic equations*.

Recall the language design example. Our objective is to design the same functional language as before. The abstract syntax and action semantics of version 1 are shown in Figure 7. This specifies that the semantic function "evaluate _" maps each expression to an action. This action will give out a value, and may use the incoming bindings (but will not itself produce bindings). A value will be a number or an abstraction. An *abstraction* is an action encapsulated as a data item, and is here used to model a programming language function.

Next we extend the language with exceptions, giving us version 2. As shown in Figure 8, expressions are now mapped to actions that might escape (giving an exception) rather than complete (giving a value). This possibility is evident in semantic equation (raise). The primitive action "escape with *d*" escapes, giving out the value yielded by *d*. Semantic equation (handle) shows the use of the combinator "A_1 trap A_2": if A_1 completes, the combined action completes, but if A_1 escapes, A_2 is performed.

Syntax:

```
        Expression  =  ... | ⟦"raise" Identifier⟧ |
                             ⟦Expression "handle" Identifier "." Expression⟧ .
```

Semantics:

```
        evaluate _   ::  Expression →
                                 action [giving a value | escaping with an exception]
                                        [using current bindings] .

(var)       as before

(abs)       as before

(app)       evaluate ⟦E1:Expression E2:Expression⟧ =
                  ( evaluate E1  and then  evaluate E2 )  then
                  enact application of the given abstraction#1 to the given value#2 .

(sum)       evaluate ⟦E1:Expression "+" E2:Expression⟧ =
                  ( evaluate E1  and then  evaluate E2 )  then
                  give sum (the given number#1, the given number#2) .

(let)       as before

(raise)     evaluate ⟦"raise" I:Identifier⟧ =
                  escape with the exception bound to I .

(handle)    evaluate ⟦E1:Expression "handle" I:Identifier "." E2:Expression⟧ =
                  evaluate E1
                  trap
                  ( ( check (the given exception is the exception bound to I )  then
                        evaluate E2 )
                    or
                    ( check not (the given exception is the exception bound to I )  and then
                        escape with the given exception ) ) .
```

Figure 8 Abstract syntax and action semantics of version 2.

What about the existing semantic equations? We could possibly leave them all untouched. This is because every combinator behaves reasonably if either of its subaction escapes. Consider semantic equation (let), for example: if the subaction "evaluate E_2" escapes, then the whole action will escape – which is exactly what we want. The only problem is "evaluate E_1 and evaluate E_2" in equations (app) and (sum). In version 1 it did not matter whether E_1 or E_2 was evaluated first. In version 2 it does matter, because either E_1 or E_2 or both could raise an exception. In order to avoid non-determinism, however, let us insist that E_1 be evaluated before E_2. This is simply achieved by replacing the combinator "and" by "and then", as shown in Figure 8.

Our final language change is to add state. As shown in Figure 9, expressions are now mapped to actions that might use and/or update stored data, in addition to their existing possible behaviours. The new possibilities are evident in semantic equation (ref) where a cell is allocated and initialised, in (deref) where the content of a cell is fetched, and in (assign) where the content of a cell is updated.

What about the existing semantic equations? The good news is that none of them need be modified in any way! This is because every combinator behaves reasonably if either or both of its subactions update stored data: if A_1 and A_2 are performed sequentially, then A_2 will see any updates made by A_1.

Syntax:

```
Expression = … | ⟦"ref" Expression⟧ | ⟦"!" Expression⟧ |
                 ⟦Expression ":=" Expression⟧ .
```

Semantics:

```
evaluate _  ::  Expression →
                    action [giving a value | escaping with an exception | storing]
                           [using current bindings | current storage ] .
```

(var) *as before*

(…) …

```
(ref)     evaluate ⟦"ref" E:Expression⟧ =
              ( allocate a cell and then evaluate E ) then
              ( store the given value#2 in the given cell#1 and give the given cell#1 ) .

(deref)   evaluate ⟦"!" E:Expression⟧ =
              evaluate E then give the value stored in the given cell .

(assign)  evaluate ⟦E1:Expression ":=" E2:Expression⟧ =
              ( evaluate E1 and then evaluate E2 ) then
              ( store the given value#2 in the given cell#1 and give the given value#2 ) .
```

Figure 9 Abstract syntax and action semantics of version 3.

Comparison

This example will, I hope, make it clear that action-semantic specifications are significantly more readable and modifiable than natural-semantic specifications. As an example of language design and specification, it is of course unrealistically small. However, the problems it has exposed are truly characteristic of iterative language design and specification. Certain major language changes, such as adding side effects, exceptions, or concurrency, tend to force global modifications to the formal specification, not just in parts of the specification that are directly concerned with the language changes.

Only if the formal specification has a truly modular structure can modifications to the specification be localised and commensurate with the scale of the language changes.

If the designer/specifier chooses natural semantics, major language changes force modifications to the forms of the judgements. These in turn force modifications to most or all of the inference rules. It might be argued that the designer/specifier should anticipate the final forms of the judgements, but generally this is too much to expect. The design of a real programming language is a huge and complex task; even if the designer starts off with a clear vision of the new language, this might have to be changed substantially if feedback shows up major problems. In any case, the designer/specifier might wish to reuse (parts of) the formal specification of an older language, and the writer of that older specification could not possibly have anticipated the shape of the new language!

Similarly, if the designer/specifier chooses denotational semantics, major language changes force modifications to the functionalities of the semantic functions, and in turn to most or all of the semantic equations. In the example, the denotation of an expression would be bindings → value in version 1, bindings → (value + exception) in version 2, and bindings × store → (value + exception) × store in version 3. At each step, nearly all of the existing semantic equations would have to be modified.

In action semantics, however, the modifications to the specification are commensurate with the language changes, even major language changes. Typically, the modifications are localised to the clauses that declare the semantic functions and to the semantic equations that are directly concerned with the language change. Occasionally minor modifications to other semantic equations are required, such as a new choice of combinator; even so, the readability of the action notation makes such modifications comparatively easy to locate.

What gives action semantics its desirable but elusive pragmatic qualities? The answer lies in the logical and systematic design of action notation.

The readability of action notation is a consequence of two factors. Grammatically it is English-like: primitive actions are verbs, unary combinators are adverbs, binary combinators are conjunctions, data terms are nouns. At a deeper level, the primitive actions and combinators correspond explicitly to the fundamental concepts by which we understand programming languages: bindings, scope, storage, sequencing, selection, iteration, recursion, and so on.

The modifiability and reusability of action notation stem from the polymorphism of the combinators. Each combinator operates smoothly and consistently over all actions, no matter how simple or complex their possible behaviours. First consider actions with only transients and bindings. There are basically three ways to combine a pair of such actions: make transients flow from one to the other ("then"), make bindings flow from one to the other ("hence"), or merge both transients and bindings ("and"). Now consider normal control flow. The combinators "then" and "hence" must logically be sequential. The other combinator could logically be either interleaving or sequential; to avoid arbitrariness, it is separated into two combinators, "and" and "and then", respectively. Now consider exceptional control flow. If one subaction escapes, the combined action also escapes. Moreover, if the first subaction of a sequential combinator escapes, the second subaction is not performed at all. Similar points apply to divergence and failure as well as escape. Finally, consider what happens if the subactions update stored data. Whether the combinator is sequential or interleaving, the storage updates are ordered correspondingly. In view of all this, the language designer can start with a simple language design and specification, and then proceed to develop them incrementally, secure in the knowledge that the initial choice of combinators will almost certainly hold good even as the actions involved in the specification become more and more complex in their possible behaviours.

5 Conclusion

In this paper I have advocated an "engineering" approach to programming language development, with a central role for formal specification. This should provide timelier and better-quality feedback to language designers than has been evident in the past. It is critically important, however, that the formal specification should be a help and not a hindrance to the language designer, and this demands excellent pragmatic qualities. Action semantics demonstrates that such qualities are attainable.

Further work is required. A well-developed formalism should be equipped with an integrated tool set for composing, editing, checking, and prototyping formal specifications. Some action semantics tools exist (Mosses 1994), but integration is lacking. In this respect action semantics lags behind some other formalisms such as VDM.

Acknowledgements

Peter Mosses led the development of action semantics, and is responsible for its theoretical foundations. I would like to thank Peter Mosses, Martin Musicante, and Simon Peyton Jones for commenting on this paper at very short notice.

References

DDC (1986) The draft formal definition of ANSI/MIL-STD-1815A Ada, Report to the Commission of the European Communities, DP Project 782, Dansk Datamatik Center, Denmark.

Donzeau-Gouge, V., Kahn, G., Lang, B., and Krieg-Brückner, B. (1980) Formal description of the ADA programming language, INRIA, Rocquencourt, France.

Hoare, C.A.R., and Wirth, N. (1973) An axiomatic definition of the programming language PASCAL, *Acta Informatica 2*, 335–355.

Hudak, P. (ed.) (1992) Report on the functional programming language HASKELL, *ACM SIGPLAN Notices 27*, 5.

Ichbiah, J. (ed.) (1979) Rationale for the design of the ADA programming language, *ACM SIGPLAN Notices 14*, 6.

Ichbiah, J. (ed.) (1983) *ADA Programming Language*, ANSI/MIL-STD-1815A, Ada Joint Program Office, Department of Defense, Washington, DC.

Jensen, K., and Wirth, N. (1974) *PASCAL User Manual and Report*, Springer, Berlin.

Milner, R., Tofte, M., and Harper, R. (1990) *The Definition of Standard ML*, MIT Press, Cambridge, Massachusetts.

Mosses, P.D. (1992) *Action Semantics*, CUP, Cambridge, England.

Mosses, P.D. (ed.) (1994) Proc. 1st International Workshop on Action Semantics, Report BRICS NS-94-1, Aarhus University, Denmark.

Mosses, P.D., and Watt, D.A. (1987) The use of action semantics, in *Formal Description of Programming Concepts III* (ed. Wirsing, M.), North-Holland, Amsterdam, 135–163.

Mosses, P.D., and Watt, D.A. (1993) Pascal action semantics, version 0.6, Computer Science Department, Aarhus University.

Sommerville, I. (1996) *Software Engineering* (5th edn.), Addison–Wesley, Reading, Massachusetts.

Watt, D.A. (1988) An action semantics of Standard ML, in *Mathematical Foundations of Programming Language Semantics* (eds. Main, M., Melton, A., Mislove, M., and Schmidt, D.A.), Springer, Berlin, 572–598.

Watt, D.A. (1991) *Programming Language Syntax and Semantics*, Prentice Hall International, Hemel Hempstead, England.

Wirth, N. (1971) The programming language PASCAL, *Acta Informatica 1*, 35–63.