Formal Programming Language Semantics note 2

Goals of formal semantics

Let's now look at some of the reasons why one might want to study the formal semantics of programming languages. Most of these points will be illustrated by specific examples later in the course.

Standardizing language definitions. Most obviously, a formal semantics can offer a *complete, rigorous definition of a language*. This can serve both as a reference for programmers wishing to understand subtle points of the language, and as a touchstone for implementers of the language (e.g. compiler writers), by providing a standard against which the *correctness* of an implementation may be judged. For a language with a large user community, providing a robust definition is clearly a worthwhile investment of effort, and helps to pre-empt the possibility that different implementers might interpret the definition of the language differently, with the consequent possibility that a user's program might run OK on one implementation but not on another.

The main advantages of formal versus informal definitions here are that a formal definition is mathematically precise and unambiguous (in practice, informal definitions *are* sometimes interpreted differently by different implementers), and that a formal definition is typically much more concise than a corresponding English prose description. A possible disadvantage is that the ideas of formal semantics are as yet not very widely known and understood — and many programmers are frightened of anything that looks like maths!

Proving properties about programs. A formal semantics can also provide a foundation for *mathematical proofs about programs.* Indeed, even to be able to formulate claims about programs as precise mathematical statements, one has to have a precise mathematical definition of the language in question.

The mathematical statements one might wish to prove are of two kinds: statements about the language as a whole, and statements about particular programs. As an example of the first kind, various good properties of Standard ML can be proved mathematically from the definition: for instance, the fact that type errors and null pointers can never arise at run time. Examples of the second kind include statements asserting the *correctness* of some program — that it does what it is supposed to do. Normally, "correct" will mean that the program conforms to some *formal specification* (as in Extended ML, for instance). Formal semantics provides not only a framework for making precise statements of correctness, but also many of the logical and conceptual tools needed for proving them.

The dream of being able to prove mathematically that programs are correct — the ideal of *program verification* — has been talked about for several decades now, and has been the subject of a lot of research (particularly in Edinburgh). It is still not a practical possibility for sizeable programs, but one can hope that it might become so in the future. Because of the size and complexity of real-world programs (and languages), some kind of machine assistance is essential; this means our semantics needs to be "formal" in the strong sense.

In parts of this course we will be looking briefly at how semantics is supposed to help with the problem of program verification.

Assisting in language design. The ideas of semantics can provide *guidance for language designers*. Designing a good programming language, in which the various features interact cleanly and in a principled way and unnecessary complications are avoided, is a difficult problem, and the attempt to provide a formal semantics can highlight unnecessary complications and suggest simpler, cleaner definitions.

Some people think that the program verification problem will become significantly easier if programming languages are designed to have good mathematical properties (ML is a step in this direction). The idea is that a programming language designed at random will be a very complex and artificial formal object, hence probably hard to reason about — but with the help of denotational semantics in particular, one can design languages that may be just as complex but are in some sense equivalent to a simple, natural mathematical structure; this probably means they will be relatively pleasant to reason about. (What exactly I mean by this will hopefully become clearer by the end of the course!)

Animation and language prototyping. If a semantics is formal in the strong sense, it can be *manipulated by a computer* in various ways. For example, there exist tools which take as input an operational semantics for a language and "animate" it so that the user can run programs in his language and see how they behave. This allows a language designer to experiment with various semantic choices and observe what differences they make. (I hope to make some such tool available for you to play with during the course.) Likewise, there are tools which take an operational semantics and automatically generate a compiler for the corresponding language.

Applications to compiler techniques. The theoretical ideas behind semantics can actually be applied to *make code run faster*! For example, the ideas of static semantics can be used to design *type systems* which a compiler may use to determine that a certain code optimization is legal. As another example, a technique known as *normalization by evaluation* obtains the result of a program simply by computing its value according to a denotational semantics for the language; this is often much faster than computing the result in the conventional way.

Operational, denotational and axiomatic semantics

In principle, any mathematical description of a programming language that correctly predicted the behaviour of programs would count as a "semantics". But whatever technique we use, we want to strive for *clarity* and *simplicity* in our description, avoiding unnecessary detail as far as possible. (It is obvious in view of the applications listed above why this is so desirable.) For this reason, an approach such as defining a language via the behaviour of some particular compiler is to be regarded as a bad (nay, disgusting) way to do semantics.

In practice, there are three particular styles of semantics that are traditionally considered. (We are really talking about *dynamic semantics* here.) Let's consider these in turn:

Operational semantics: Here one specifies the way in which programs run or execute by means of symbolic rules. Or at least, we specify *one* possible way in which they might execute which yields the intended behaviour — it need not be the way they actually run in a particular implementation. Normally we specify a kind of "idealized implementation", going for simplicity and clarity rather than efficiency. Nevertheless, operational definitions generally have the feel of a description of a process involving symbol manipulation that "happens in time".

The are two main kinds of operational semantics:

- The *abstract machine* approach. Here we specify a way of implementing the language on some lower-level (idealized) computing machine (e.g. the SCM or SECD machine), or else of translating (or "compiling") it down to a lower-level language for which we already have an operational semantics. Approaches of this kind have the advantage that they are close to actual implementations, but their big disadvantage is that they tend to involve a lot of arbitrary choices and irrelevant detail.
- *Structural operational semantics,* introduced by Plotkin in the early '80s. Here our symbolic rules work directly with the syntax of the language in question, and so much of the syntactic structure of programs is "preserved" — it is rather like giving an idealized interpreter for the language. This is a bit more abstract and generally cleaner than the abstract machine approach. In this course we will concentrate on this latter kind of operational semantics.

Denotational semantics: Here one looks for a mathematical structure $\mathcal{M}$ which constitutes an answer to the question "What is it that program phrases actually *mean*?" In very simple cases, for example, we could take the meaning of a program to be simply a function from input-values to output-values, in which case we might take $\mathcal{M}$ to be just the set of all functions of the appropriate type.[1]

[1] Normally, $\mathcal{M}$ has to be something more complex than this. Much work in denotational semantics uses various kinds of *complete partial order (CPO)* or *domain* — see later in the course.

We then define a mapping which associates to each program phrase P an element $[\![P]\!]$ of $\mathcal{M}$, called the meaning or *denotation* of P. The intended behaviour of a program (e.g. the result it returns) can typically be read off easily from its denotation. Almost always, we try to give a definition of $[\![-]\!]$ which is *compositional*, in the sense that the denotation of a program is completely determined by the denotation of its constituent subprograms; this means that $[\![P]\!]$ contains all the information necessary to determine how P will behave in any program context.

In contrast to the "dynamic" feel of operational semantics, denotational semantics has a "static" feel: we are describing program behaviour in terms of mathematical structures fixed in eternity. This means that ordinary kinds of mathematical reasoning can be applied to denotations: we don't have to worry about things "happening in time". When it works well, denotational semantics offers the deepest mathematical understanding of any of the three approaches. Much of the interest of the subject lies in the problem of trying to find a good mathematical structure $\mathcal{M}$ for modelling a given programming language. (In general, for a given language many choices of $\mathcal{M}$ are possible, but — as we shall see — some are definitely "better" than others.)

Axiomatic semantics: Here one specifies the behaviour of a program indirectly, by giving a system which says what *properties* programs have. Typically, we give a *program logic* which allows us to generate formal proofs of such properties; we then declare that the properties provable in the system have to be true. Among these properties will be certain "basic" statements about the behaviour of particular programs, so giving a program logic is enough to give a semantics.

These three styles are not in competition, but are mutually complementary and serve different purposes. Operational semantics is the most immediately useful as a standard for implementers. Axiomatic semantics is the most suitable for program verification. Denotational semantics provides a logical and conceptual link between operational and axiomatic semantics, and is also the best in terms of helping us to design both programming languages and program logics in a clean way. The best situation is to have all three kinds of semantics for a given language, together with proofs that the they "agree" (that is, define the same program behaviour). [Exercise: in the remaining inch or so of space, copy down the diagram from the blackboard (this will save me typesetting it!).]

John Longley