WHAT IS A MODEL?
A CONSUMER'S PERSPECTIVE ON SEMANTIC THEORY
Jon Shultis
Department of Computer Science
University of Colorado
Boulder, Co 80309, USA

Introduction

By a "consumer" of semantic theory, I mean anyone who, like myself, is not primarily concerned
with proving new results or otherwise contributing directly to the theory of semantics, but who finds
(or could find) the concepts and methods of semantics useful for doing other work. Currently, consum-
ners include programming language designers and, to some extent, language implementors. The
results and methods of semantics could serve a much wider audience.

Insofar as everyday programming is a process of (language) specification and implementation,
why should semantics seem so irrelevant to programmers? Whenever I treat semantic theory in my
graduate programming languages class, the students listen patiently for about a week, and then
begin asking: "What is this stuff good for?" I am getting better at finding answers to pacify my stu-
dents, but the question nags at me more all the time. Take money, for example. How many cases can
we point to where a result from semantic theory has saved someone a penny? Not one.

A thing is useful if it solves a problem that someone wants solved. So far, semantics has solved
problems that are of interest mainly to semanticists: How can this kind of language be modelled?
When are two programs equivalent? What do all models of this language have in common? When
does a language have a relatively complete theory? And so forth. The answers that have been found
for these and many other questions have unquestionably improved our understanding of the basic
issues and methods of semantic theory.

Should I then tell my students that these concepts and techniques are good for doing more
semantic theory? In other words, "this stuff is good for solving problems that most of you don't care
about". If that's the best I can do, I shouldn't be teaching semantics to computer scientists.

Semantic theory can be far more relevant and useful to the practically-oriented computer scien-
tist than it is at present. To see how, we need to take a careful look at the kinds of problems faced by
consumers of semantics, and understand why the theory sometimes helps solve those problems, and
sometimes doesn't.

1. Metalanguage Semantics and Constructivity

The controversy and confusion over how to interpret the definition of Algol 60 posed a problem
that computer scientists wanted solved: some means had to be found of defining the meaning of a
programming language in a precise and unambiguous way.

Strachey's solution [21] was to associate a mathematical denotation with each phrase of a
language. In order to do this, he faced a subsidiary problem: how to write down the denotations, and
how to write down the mapping from language structures to their denotations. He settled on
Church's $\lambda$-notation for writing down the denotations, and syntax-directed translation for writing
down the mapping. A third problem, noticed by Scott, is that the metalanguage used to specify the
semantics also needs a semantic definition; reflexive domains provided the solution for $\lambda$-calculus.
The method of denotational semantics is now routinely used to define small languages and study their properties. Several examples of such applications are contained in these proceedings.

The similarity between the specification of a denotational semantics and a model has led to the identification of the two. Recall that, in mathematical logic, a domain together with a homomorphism from the syntax to that domain is called an interpretation. Given a notion of truth in the domain, an interpretation is a model of a logic (theory) if all of the axioms are true and all of the inference rules preserve truth. In the case of programming languages, we can think of a denotational semantics as giving a model of the language's Floyd-Hoare theory. In the case of λ-calculus models, we might use realizability as our notion of truth, in which case all that really needs to be checked is that the conversion rules are valid.

The identification of denotational semantics with models is unfortunate, however, because it misses a point of paramount importance to consumers. The success of denotational semantics is largely due to the fact that the λ-calculus semantics we use is constructive, in the sense that writing down the semantic equations for a language gives an effective way of computing the meaning of any phrase.

To illustrate the importance of this point, consider the denotational semantics for Communicating Sequential Processes given by Brookes et al. in [5]. The metalanguage used there is the language of classical first-order logic and set theory. Ignoring the philosophical issues, one thing is clear: whatever semantics we give to this metalanguage, it will not be effective. There is nothing "wrong" with this, unless we want to use the semantics as a basis for an implementation, or use some theorems about the semantics to manipulate actual programs. Because these are precisely the things that consumers want semantics for, however, a non-effective semantics is less useful to them than it might be.

Among other things, the non-effectiveness of the semantics causes trouble when operations for combining processes are defined. For example, Brookes et al.'s Theorem 2 asserts that the intersection of an arbitrary family of processes is a process. In order to understand why this is so from a computational standpoint, one needs a proof that yields a construction of the intersection, showing why the result is again a process. Unfortunately, the proof offered uses the method of contradiction, which is not effective. The burden of devising a constructive proof is therefore laid squarely on the shoulders of anyone who might wish to use the result as part of a computer program. Since there is no guarantee a priori that such a proof can be found, the "result" might just as well have been stated as a conjecture.

By contrast, consider the theorem that for every nondeterministic finite automaton there is an equivalent deterministic one. The usual proof of this gives a direct construction of the deterministic automaton from the nondeterministic one, and shows that the construction leaves invariant the language accepted. The theorem is useful to consumers of automata theory because its proof supplies an algorithm.

By analogy, denotations (for programming semantics, at least!) should be effective objects, effectively given, and effectively reasoned about. In short, a denotational semantics should provide an effective model, not just any model. For a discussion of effectiveness, see [24]. Adherence to this principle would go a long way toward making semantic theory more useful. But is it enough?

2. What is a Model?

Consider the category N≤ having the natural numbers as objects, the arrows being given by the usual ordering. It is easy to model addition using N≤ as the domain. Let the numeral n denote the corresponding arrow \( n:0 \leq \bar{n}, \) where \( \bar{n} \) is the (semantic) natural number corresponding to the (syn-