schools. The motivation behind this project is the perception, shared by many teachers and students, that AI is just a helpful skill with not enough deep theoretical grounding to merit the treatment as a discipline at the university level. Sloman's goal is to meet this challenge head on.

*

As always, I want to close by thanking my Dean, as well as my Department Chair, at the University of Illinois at Springfield for making it possible for me to devote more attention to this *Newsletter* than I would have been able to do otherwise. Let me end with a special note. Now when the APA site is relatively in order[2] I would want to guide the Readers towards this *Newsletter*'s history. The older issues, I think especially those edited by Jon Dorbolo,[3] will remain an excellent source of information about the history of philosophy and computing and are still very much worth browsing through.

Endnotes

1. I want to thank T. Beavers for his impromptu invitation during my visit there.
2. http://www.apaonline.org/publications/newsletters/computers.aspx
3. Jon assembled an excellent team of editors: W. Uzgalis, R. Causey, L. Hinman (as the Internet Resources Editor) and R. Barnette (as the Teaching in Cyberspace Editor). See http://www.apaonline.org/publications/newsletters/v97n1_Computers_01.aspx

From the Chair

Michael Byron
Kent State University

The Pacific Division held its 2009 meeting in April in Vancouver, BC. Last fall, the Committee voted to award the Barwise Prize to Terry Bynum of Southern Connecticut State. Unfortunately, the session had to be canceled due to travel issues. We expect to reschedule this award at the 2010 Central Division Meeting to be held next February in Chicago.

For 2009-10 the Committee welcomes two new members. David L. Anderson is in the philosophy department at Illinois State and directs the Mind Project there (http://bit.ly/DGqAH). Susan V.H. Castro (http://bit.ly/Stmia) completed her Ph.D. recently at UCLA. We are always glad to have new members.

The next divisional meeting will be the Eastern Division Meeting, to be held in December in New York City. At that meeting we will be presenting the Barwise Prize to Luciano Floridi. Beyond that, the Committee looks forward to another productive year in 2009-10.

New and Noteworthy: A Central APA Invitation

We want to invite you to two sessions at the 2010 APA Central Division meeting in Chicago.

1. North American Computing and Philosophy Conference, (NA-CAP) Thursday, February 18, 7:30-10:30 p.m.:

Machine Consciousness

Chair:

Marvin Croy

Papers:

Ricardo Sanz (Technical University of Madrid): "The Need for a Mind in Control Systems Engeneering"

Piotr Boltuc (University of Illinois at Springfield): "Non-Reductive Machine Consciousness?"

Matthias Scheutz (Indiana University–Bloomington): "Architectural Steps Towards Self-Aware Robots"

Commentators:

Thomas Polger (University of Cincinnati)

John Barker (University of Illinois at Springfield)

2. Special Session Organized by the APA Committee on Philosophy and Computers, Saturday, February 20 (morning)

Machines, Intentionality, Ethics and Cognition

Chair:

Peter Boltuc (University of Illinois at Springfield)

Participants:

David L. Anderson (Illinois State University–Bloomington): "Why Intentional Machines Must be Moral Agents (or at least Moral Patients)"

Keith Miller W. (University of Illinois at Springfield): "Truth in Advertising, or Disrespecting Robot Autonomy"

Svetoslav Braynov (University of Illinois at Springfield): "Can you Trust a Robot?"

Thomas Polger (University of Cincinnati): "Distributed Computation and Extended Cognition"

Closing comments:

Ricardo Sanz (Technical University of Madrid)

Articles

Featured Article

The Meaning of Programming Languages

Raymond Turner
University of Essex

Abstract

A folklore view has it that programming languages get their semantic interpretations layer by layer, one language getting its interpretation in the next, until the bedrock of physical reality (physical machines) provides the final and actual mechanism of semantic interpretation. We argue, based upon the normative requirements of any semantic account, that this is a false picture. We further argue that, in any adequate semantic theory of a programming language, the denotations of its constructs must be taken to be mathematical objects.

1. The Limitations of Grammar

Programming languages form part of the bedrock on which computer science is built. Their design and implementation is a core aspect of the subject, and they are the host for the day to day activity of program and software construction. Consequently, a proper conceptual analysis of the nature of these languages will form a significant part of the philosophy

of computer science. While a great many of the conceptual questions that surround them focus on their design, and their use in problem solving and software construction, here we shall concentrate on some of the issues that center upon their definition, and, in particular, on their semantic import.

In practice, programming languages get their semantic interpretation using a mixture of methods. Often, a top level natural language account guides the construction of a compiler that translates the language into the language of the implementing machine. This may or may not be direct, i.e., the interpretation may pass through several layers and several different languages, and associated compilers. It may even be cushioned by the presence of an intermediate abstract machine, but ultimately this process ends in the machine instructions of a physical machine. But how do these techniques fix the semantic content of the language? Part of our objective is to examine this question. But first we must set the scene.

Generally, a programming language, as a language, is given via a formal grammar of some sort. This spells out the legal strings of symbols of the language. For example, the following provides the syntax for a simple imperative language in a standard recursive notation, where P stands for programs, E for arithmetic expressions, and B for Boolean expressions.

$P ::= x := E \mid \textbf{skip} \mid P; P \mid \textbf{if } B \textbf{ then } P \textbf{ else } P \mid \textbf{while } B \textbf{ do } P$

$E ::= x \mid 0 \mid 1 \mid E+E \mid E*E$

$B ::= x \mid \mathit{true} \mid \mathit{false} \mid E < E \mid \neg B \mid B \vee B$

The expressions (E) are constructed from 0 and 1 by addition and multiplication. The Boolean expressions (B) are constructed from *true*, *false*, the ordering relation ($<$) on numbers and negation and conjunction. The actual programs of the language (P) are built from a simple assignment statement ($x := E$) via sequencing (P; Q), conditional programs (**if** B **then** P **else** Q) and while loops (**while** B **do** P).[1] For example, according to this grammar, the following is a grammatically legal program.

$x := 0;$ (1)
$y := 1;$
while $x < n$ **do** $(x := x+1; y := x*y)$

But by itself, the grammar does not tell us what this program does or what it is supposed to do. If you have grasped its semantic import, it is because you already have some understanding of the intended computational impact of its constructs. The point is, to construct or understand this program, or any program for that matter, one needs to know more than the syntax of its host language; one must possess some *semantic* information about the language.

2. The Normative Nature of Semantics

Any such semantics must provide an account of the intended meaning of the constructs. This must be sufficient to guide a compiler writer in implementing the language and facilitate arbitration when disputes arise over the implementation of a construct: it must enable a distinction to be drawn between the correct and incorrect implementation of a construct. It must also support a distinction between correct and incorrect programs—not just syntactically, but in the sense of meeting their intended specifications. For instance, a semantic account must determine that program (1) with input n, computes the factorial function. Likewise, it must determine that

$x := 0; y := 1;$
if $n = 0$ **then** $y := x + 1;$
while $x > n$ **do** $(x := x–1; y := (x*y)+y)$ (2) does not.

Any semantic account must act as a normative guide to the language: it must enable a distinction to be drawn between a correct and incorrect use of a language construct. This seems to apply even to natural language.[2]

> *Suppose the expression "green" means green. It follows immediately that the expression "green" applies correctly only to these things (the green ones) and not to those (the non-greens). The fact that the expression means something implies, that there is a whole set of normative truths about my behavior with that expression: namely, that my use of it is correct in application to certain objects and not in application to others. ...The normativity of meaning turns out to be, in other words, simply a new name for the familiar fact that, regardless of whether one thinks of meaning in truth-theoretic or assertion-theoretic terms, meaningful expressions possess conditions of correct use. (On the one construal, correctness consists in true use, on the other, in warranted use.) Kripke's insight was to realize that this observation may be converted into a condition of adequacy on theories of the determination of meaning: any proposed candidate for the property in virtue of which an expression has meaning, must be such as to ground the "normativity" of meaning—it ought to be possible to read off from any alleged meaning constituting property of a word, what is the correct use of that word.* [2]

This normativity constraint seems not to be a controversial one for semantics in general. However, in the case of programming languages, it has some clear, yet significant implications for any proposed semantic account.

Presumably, language designers have some semantic intentions about the computational impact of their language constructs, and one way such intentions might be articulated is via an informal natural language account of the various constructs, where these descriptions most often take the form of a reference manual for the language. And for real languages these often run to hundreds of pages, e.g., the specification of the Java Language is almost 600 pages.

What do these language specifications look like? Normally they are provided in terms of the impact of the language constructs upon an underlying machine. For our simple language we require a machine with an underlying state whose role is to store numerical values in locations, i.e., a state of the machine might look like the following.

[3 4 7 . . .]
[x y z]

where the visual display of the numerals indicates their numerical content. The full recursive language is then interpreted via its impact upon this machine. But before we embark on any further elaboration of this, we have to face a preliminary question.

Is this to be taken as a physical or abstract machine? Some authors ([4], [3], [5]), suggest that programming language constructs are ambiguous, i.e., they have two meanings, one provided by an abstract machine and one provided by physical one. For example, according to the latter, the assignment statement

$x := 10$ (3)

is given its interpretation by its impact upon a physical device, i.e., where its meaning is given as

place 10 *in location* x,

x refers to the physical location. What are the consequences of this? Presumably, that somehow the meaning is given and fixed by the physical machine. What else could it mean? Consequently,

although the intentions of the machine designer may have been used to guide the construction of the physical machine, they are no longer definitive. If when run, the instruction $x := 10$ sticks 20 in location x, then so be it; this is what it is taken to mean. The intentions of the designer are superseded by the actual impact of the instructions on the machine. But this has an important consequence, namely, there is no notion of *malfunction*. There is no alternative court of appeal. If, during the running of a machine instruction, the machine switches on and off, this is to be taken as part of the meaning of the instruction. But is this a coherent perspective? The rule following considerations ([27], [2]), and Kripke suggest not.

> *Actual machines can malfunction: through melting wires or slipping gears they may give the wrong answer. How is it determined when a malfunction occurs? By reference to the program of the machine, as intended by its designer, not simply by reference to the machine itself. Depending on the intent of the designer, any particular phenomenon may or may not count as a machine malfunction. A programmer with suitable intentions might even have intended to make use of the fact that wires melt or gears slip, so that a machine that is malfunctioning for me is behaving perfectly for him. Whether a machine ever malfunctions and, if so, when, is not a property of the machine itself as a physical object but is well defined only in terms of its program, stipulated by its designer. Given the program, once again, the physical object is superfluous for the purpose of determining what function is meant*. [11] page 34

The notion of *malfunction* must be measured against a stable account that reflects the designer's intentions. And this cannot be supplied by the physical machine. Of course, we could impose some restrictions on the device to ensure that it behaves appropriately. We might, for instance, suppose that there are physical mechanisms that enable us to perform an *Update* and a *Lookup* on the state of the machine, and these must satisfy the following requirement.

> *Suppose in the state s, the machine is updated by inserting v in location x. If in the resulting state, the value in location x is then looked up, then the value v will be returned. If the value in location y (where y is different to x) is looked up, then the value of y in the original state s is returned.*

We can put this a little more precisely. We shall use the phrase *E evaluates to v* to indicate that the expression E reduces to the value v at the end of the computation.

> $Lookup(Update(s, x, v), x)$ evaluates to v
>
> If $Lookup(s, y)$ evaluates to w then $Lookup(Update(s, x, v), y)$ evaluates to w – where x and y are distinct.

So that if $Lookup(Update(s, x, 10))$ evaluates to 20, then the physical machine has malfunctioned. But this is a definition of an abstract machine. Indeed, as Kripke observed, given the abstract machine, from the semantic perspective, the physical one is superfluous. It is the abstract machine that serves as the basis for semantics, and as a guide to the construction of the physical one.

So what does this say about the so-called physical interpretation of assignment given by (3)? Clearly, it cannot function as a definitional account of the construct. So what is its relationship to the abstract normative one? Only that the normative meaning has physical implications, i.e., against the background of its normative meaning, the physical implications may be used to predict the behavior of the physical machine. In this simple case, it is the physical machine that is under scrutiny. (3) does not have two definitional readings. Indeed, why is this case not parallel to any mathematical notion and its application to the physical world? We do not feel compelled to say that the notion of a right angled triangle has two meanings: the definitional one and the one that results from its consequences when applied to the physical world. The latter does not yield a second meaning.

This is one step in our argument to the effect that any semantic theory must be an abstract one: at its base there must be an abstract machine.

3. An Informal Semantics

With this much established, we may provide some account of the semantics of our language: relative to it, we provide an evaluation mechanism for the programs of our simple language.

1. If the evaluation of E in the state s returns the value v, then the evaluation of $x := E$ in a state s, returns the state that is the same as s except that the value v replaces the current value in location x.
2. The evaluation of **skip** in a state s, returns s.
3. If the evaluation of B in s returns *true* and the evaluation of P in s returns s', then the evaluation of if B then P else Q in s, evaluates to s'. If on the other hand, the evaluation of B in s returns *false* and the evaluation of Q in s returns s', then the evaluation of **if** B **then** P **else** Q in s, returns s'.
4. If the evaluation of P in s yields the state s' and the evaluation of Q in s' returns the state s'', then the evaluation of P; Q in s, returns the state s''.
5. If the evaluation of B in s returns *true*, the evaluation of P in s returns s', and the evaluation of **while** B **do** P in s' yields s'', then the evaluation of **while** B **do** P in s, returns s''. If the evaluation of B in s returns *false*, then the evaluation of **while** B **do** P in s, returns s.

There are several assumptions built into 1-5 that need to be made explicit. First, observe that the evaluation of Boolean and numerical expressions is assumed not to change the state. This is a property of the language. Indeed, it is an essential property for the *coherence* of the semantics given by 1-5. We shall say more about such properties later. Secondly, the semantics allows for the possibility that programs may not terminate; in such cases the premises of the informal rules may not be true.

Such accounts provide our first foothold on grasping the semantic impact of the constructs of the language. But does the informality of such an account undermine its normative role? Surely normative stipulation must be exact enough to decide what is right and what is wrong. Do natural language accounts enable the articulation of a semantic theory that is simultaneously precise and transparent? Does the act of removing all possible ambiguities render the semantics unreadable and opaque? While this is not a serious issue for the present toy language, it appears to be so for commercial ones. For example, the original natural language description of Java was seriously flawed [17]. Of course, we might try to provide a more precise account by employing a programming language in which to write the semantic description. But this would just push the semantic problem onto a new language.[3]

4. A More Formal Account

There are many formal approaches ([1], [15], [23], [21], [24], [22]) but their advocates are united in the belief that natural language is not a suitable vehicle for expressing combinatorial

notions. To illustrate matters, and to serve as a vehicle for our investigation, we shall provide a slightly more formal version of our informal account. We shall write

$$< P, s > \Downarrow s'$$

to indicate that evaluating P in state s *terminates in* s' [6]. Indeed, we can almost read off the formal rules from our informal account.

1. The memory update command has the following rule of evaluation

$$\frac{< E, s > \Downarrow v}{< x := E, s > \Downarrow \mathit{Update}(s, x, v)}$$

The premise guarantees that the expression E in state s reduces to value v. The conclusion then guarantees that program $x := E$ will update the state s with value v in the location x.

2. The skip operation is given by the following transition rule.

$$\frac{}{< \textbf{skip}, s > \Downarrow s}$$

3. Sequencing is governed by the following pair of rules.

$$\frac{< P, s > \Downarrow s' \qquad < Q, s' > \Downarrow s''}{< P; Q, s > \Downarrow s''}$$

The premise of the first rule insists that the program P in state s returns the state s'; and the second that Q in state s' yields state s''. The conclusion then guarantees that the program C; D in state s will return the state s''.

4. The conditional is governed by the following two rules that cover the *true* and *false* cases for the evaluation of the Boolean expression.

$$\frac{< B, s > \Downarrow \mathit{true} \qquad < P, s > \Downarrow s'}{< \textbf{If } B \textbf{ do } P \textbf{ else } Q, s > \Downarrow s'}$$

$$\frac{< B, s > \Downarrow \mathit{false} \qquad < Q, s > \Downarrow s''}{< \textbf{If } B \textbf{ do } P \textbf{ else } Q, s > \Downarrow s''}$$

5. Finally, we provide the rules of the while command. The first rule deals with the case where the Boolean is true and the second where it evaluates to false. Note that the premise in the first assumes that the while loop terminates in the state derived from evaluating P.

$$\frac{< B, s > \Downarrow \mathit{true} \qquad < P, s > \Downarrow s' \qquad < \textbf{while } B \textbf{ do } P, s' > \Downarrow s''}{< \textbf{while } B \textbf{ do } P, s > \Downarrow s''}$$

$$\frac{< B, s > \Downarrow \mathit{false}}{< \textbf{while } B \textbf{ do } P, s > \Downarrow s}$$

The semantics is structural, because the meaning of a program is defined by the meaning of its components. This form of (big step) *Structural Operational Semantics* (SOS) was introduced by Gordon Plotkin [16]. It provides a clear guide to the implementor without imposing how exactly the latter is to proceed on a given concrete machine.

Indeed, the semantics can be looked at as an axiomatic theory of operations: the language enables the construction of complex operations from simple ones, and the rules provide their content. In particular, they tell us what the termination conditions for the constructs are. Of course, in order to constitute a useful theory, it requires some development. In particular, we may introduce a notion of equivalence.

$$P \bullet Q \times \text{œ} s_1 \text{A} \text{œ} s_2 \text{A} < P, s_1 > \Downarrow s_2 \leftrightarrow < Q, s_1 > \Downarrow s_2$$

i.e., two programs are *behaviorally equivalent* if they behave identically, i.e., started in the same state, they terminate in the same state. This provides us with a notion of *partial equality*. Given this we may show that the following rules may be established

(*i*) **while** B **do** $P \bullet$ **if** B **then** (P; **while** B **do** P) **else skip**

(*ii*) **if** *true* **then** P **else** $Q \bullet P$

(*iii*) **if** *false* **then** P **else** $Q \bullet Q$

And though not a deep and exciting theory, this is beginning to look much like any other mathematical theory. Indeed, we seemed to have arrived at the conclusion that any purported semantic theory, i.e., one that meets our normativity requirements, must be mathematical in nature. It would seem to follow that programming languages are definitionally, mathematical objects, i.e., their very definition as semantic objects makes them so. We shall refer to this as our *central claim*. Unfortunately, there are some possible objections. We shall consider these in the next two sections.

5. Informal Mathematics

One concerns the move from informal to formal semantics. The practicing programmer (or even compiler writer) might well claim that formal accounts are unnecessary. Indeed, our move from the informal account to the formal one was not justified by the present language, but by the vagaries that might creep in with real, large, and complex languages. However, so it may be argued, while one may have to take great care in formulating matters, and mistakes may be made, even for the most syntactically complex languages, there is nothing in principle that blocks the development of an informal normative account.

Does this mean that our central claim fails? No. We may accept this criticism but argue that even informal accounts are mathematical. More exactly, we might be persuaded that the move to a formal semantics is unnecessary, but still claim that the informal semantics is mathematical. To see why such account must be taken to be mathematical, consider again the problem of its *coherence*. To establish that latter, we need to make explicit some of the rules for the evaluation of expressions.

1. To evaluate a variable, in a state s, lookup its value. And return the same state.
2. If in state s, E evaluates to (v, s') and E' in state s' evaluates to (v', s'') then $E + E'$ evaluates to $(v + v', s'')$ and $E * E'$ evaluates to $(v * v', s'')$.

To show that this coheres with 1-5 for the evaluation of programs, we need to show that expression evaluation does not change the state. For this, we argue by induction on the structure of expressions. The case of variables is clear. Here there is no change to the state, just a simple lookup. Moreover, if two expressions do not change the state, then neither does their sum or product. The same may be argued for Boolean expressions. This is consistent with the evaluation of Boolean expressions in the evaluation of conditional programs. Consequently, the whole semantics fits together.

At some level such checking is not really optional; it is an integral part of the activity of specifying the language. But the important point is that such arguments are mathematical; just because they are expressed in English does not mean that they are not so. Indeed, they involve a form of structural induction, and it would be a strange notion of mathematics that ruled them out. Furthermore, most of actual mathematics is conducted at this level of informality. Few areas of mathematics are practiced with the rigor of formal logic. Indeed, even our informal semantic account provides, albeit informal, an axiomatic theory of operations. Moreover, many other theories have started out as informal axiomatic theories, and are only later made more precise when the appropriate formal language

is invented. Geometry is an obvious example. It was and still is mathematical, and its entities are mathematical ones.

So the informality of the semantics does not obviously undermine the claim that normative accounts of programming languages will render them mathematical objects.

6. Are Operational Accounts Mathematical?

However, some would deny this. They would do so, not by drawing a distinction between our formal and informal accounts, but by insisting that neither is mathematical. In ([21], [14]) such a criticism is aimed at Landin's operational approach [12].

> *We can apparently get quite a long way expounding the properties of a language with purely syntactic rules and transformations.....But we must remember that when working like this all we are doing is manipulating symbols—we have no idea at all of what we are talking about. To solve any real problem, we must give some semantic interpretation.* [21]

> *The motivation for the Scott-Strachey approach to semantics stems from a recognition of the fact that programs and the objects they manipulate are in some sense concrete realisations or implementations of abstract mathematical objects.* [14]

Apparently, operational accounts do not provide an interpretation into the abstract mathematical world of numbers and sets. Likewise, our rules of evaluation do not yield a mathematical account. In the end, we are still left with uninterpreted systems of rules. And a formal system, with no intended interpretation, is still an uninterpreted formal system, i.e., the rules themselves need to be interpreted. Moreover, any such interpretation relies on a further interpretation that relies on more rules, etc. Consequently, so the argument goes, we still do not have an account where this regress is blocked by reference to abstract mathematical objects.[4] If this is so, while we have provided a system of rules that constitute a normative account of the language, this system does not constitute a mathematical theory.

Apparently, for a semantic account to be mathematical, it must be based upon *mathematical objects*. For example, one such would have the semantics associate, with each program, a mathematical function from states to states, i.e., the semantic function C would have the form

$$C : Program \Rightarrow (State \Rightarrow State)$$

where *State* is a set and $State \Rightarrow State$ represents some class of set theoretic functions ([26], [1],[10]). Underlying this demand is the view that set theory is taken to be more than just a formal theory. In particular, the language of set theory is taken to refer beyond syntax to the abstract world of sets, an objective world, independent of language and our knowledge of it. Hence, it is taken to block any such regress of languages. Its language refers to an abstract pre-existing world of sets [13]. In particular, the axioms of set theory are taken to be justified by the picture of the cumulative hierarchy.

In general, genuine mathematical systems are taken to point beyond the syntax of the formal system to an abstract mathematical world. This is a realist view of mathematical structures ([13], [20]). In contrast, our fledgling theory of operations, which is a theory of operations that is determined by the programs of our simple language, is taken not to be a theory that refers to an abstract world. Indeed, if it were, it would seem to follow that every programming language gives rise to such a theory. And does this not lead to the implausible view that programming languages are not designed but discovered? Well, not quite.

There might be an abstract theory of operations that is given independently of these languages. A theory from which programming language designers select constructs. This seems to have been the view of Strachey [22]. Indeed, put this way, it is unclear why this is less plausible than the case of set theory. We certainly require some positive metaphysical reason to put set theory in a different class to theories of operations. Gödel provides a possible one.[5]

> *Despite their remoteness from sense experience, we do have something like a perception also of the objects of set-theory, as is seen from the fact that the axioms force themselves upon us as being true. I don't see any reason why we should have less confidence in this kind of perception, i.e., in mathematical intuition than in sense perception, which induces us to build up physical theories and to expect that future sense perceptions will agree with them, and moreover, to believe that a question not decidable now has meaning and may be decided in the future.* [9]

But, even if we accept something like it, there is still a puzzle about the difference between a theory of operations and a theory of sets; even under such an interpretation, it seems hard to see how the difference could be made out. It is certainly not clear that Gödel would have supported such a distinction. Referring to Turing's analysis of mechanical computability, he writes:

> *The greatest improvement was made possible through the precise definition of the concept of finite procedure, which plays a decisive role in these results. There are several different ways of arriving at such definition, which, however, all lead to the same concept. The most satisfactory way, in my opinion, is that of reducing the concept of finite procedure to that of a machine with a finite number of parts, as has been done by the British mathematician Turing.* [8]

Consequently, one assumes that he would have assigned the notion of *finite procedure* a similar metaphysical status to sets. In Wang's words, Gödel saw the problem of defining computability as:

> *an excellent example of a concept which did not appear sharp to us but has become so as a result of a careful reflection.* [25]

It would seem that any theory of operations that achieved definitive status would have the same metaphysical status as sets. Indeed, if one finds the concept of pre-existing notions of operations implausible, it is hard to see how one can defend the view of a pre-existing world of sets [24]. It would seem that the two theories stand or fall together: either they are both mathematical theories or both are not. But since set theory is a paradigm case of a mathematical theory, to deny it mathematical status would be to deny almost anything mathematical status.

But the story does not end here. Even though the case of mathematical status for these toy languages seems plausible, whether this can be maintained for real languages is less clear.

Acknowledgements

Rosana Turner provided detailed and very insightful comments on various drafts of the paper.

References

[1] Allison, L. 1990. *A Practical Introduction to Denotational Semantics*. Cambridge: Cambridge Computer Science Texts.

[2] Boghossian, P.A. 1989. The rule-following considerations. *Mind* 89.

[3] Colburn, T. 2004. Methodology of Computer Science. *The Blackwell Guide to the Philosophy of Computing and Information*, edited by Luciano Floridi. 318-26. Malden, MA: Blackwell.

[4] Colburn, T.R. 2000. *Philosophy and computer science. Explorations in Philosophy* Series. New York: M.E. Sharpe.

[5] Fetzer, J.H. 1988. Program verification: the very idea. *Communications of the ACM* 31(9): 1048-63.

[6] Fernandez. M. 2007. *Programming Languages and Operational Semantics: An Introduction*. Oxford.

[7] Gödel, K. 1934. Undecidable diophantine propositions. In *Collected Works III*. 164-75.

[8] Gödel, K. 1934. Some basic theorems on the foundations of mathematics and their implications. In *Collected Works III*. 304-23.

[9] Gödel, K. 1983. What is Cantor's continuum problem? Reprinted in Benacerraf and Putnam's collection *Philosophy of Mathematics*, 2nd ed., Cambridge University Press.

[10] http://en.wikibooks.org/wiki/Haskell/Denotational_semantics

[11] Kripke, S. 1982. *Wittgenstein on Rules and Private Language*. Harvard University Press.

[12] Landin, P.J. 1964. The mechanical evaluation of expressions. *The Computer Journal* 6(4): 308-20; doi:10.1093/comjnl/6.4.308.

[13] Maddy, P. 1992. *Realism in Mathematics*. Oxford: Oxford University Press.

[14] McGettrick, A.D. 1980. *The Definition of Programming Languages*. Cambridge University Press New York, NY: Cambridge University Press.

[15] Milne, R. and Strachey, C. 1977. *A Theory of Programming Language Semantics*. New York, NY: Halsted Press.

[16] Plotkin, G. 2004. A structural approach to operational semantics. *J. Log. Algebr. Program* 60-61: 17-139.

[17] Pugh, W. 2000. The Java, memory model is fatally flawed. *Concurrency: Practice and Experience* 12(6): 445-55.

[18] Rapaport, W.J. 1999. Implementation is semantic interpretation. *Monist* 82: 109-30.

[19] Rapaport, W.J. 2005. Implementation is semantic interpretation: Further Thoughts. *Journal of Experimental and Theoretical Artificial Intelligence* 17(4): 385-417.

[20] Shapiro, S. 2004. *Philosophy of Mathematics: Structure and Ontology*. Oxford.

[21] Stoy, J. 1977. *The Scott-Strachey Approach to Programming Language Semantics*. MIT Press.

[22] Strachey, C. 1966. Towards a formal semantics. In *Formal Language Description Languages for Computer Programming*. 198-220. North Holland.

[23] Tennent, R.D. 1991. *Semantics of Programming Languages*. London: Prentice-Hall International.

[24] Turner, R. 2007. Understanding programming languages. *Minds and Machines* 17(2): 129-33.

[25] Wang, H. 1974. *From Mathematics to Philosophy*. London: Routledge & Kegan Paul.

[26] http://en.wikipedia.org/wiki/Denotational_semantics

[27] Wittgenstein, L. 1953. *Philosophical Investigations*. Oxford: Blackwell Publishing.

Endnotes

1. We shall use parenthesis to disambiguate.
2. Presumably, in natural language, the semantics does not play a definitional role. It merely codifies what we take to be correct and incorrect use.
3. [18], [19] offer a detailed discussion of the nature of implementation as semantic interpretation.
4. Moreover, at this point in the argument, we have no other obvious way out of the regress since we have given up the bedrock of physical reality as a mechanism to fix the meaning.
5. Maddy [13] defends a version of Gödel's view.

Leibniz, Complexity, and Incompleteness[1]

Gregory Chaitin
IBM Research

Let me start with Hermann Weyl, who was a fine mathematician and mathematical physicist. He wrote books on quantum mechanics and general relativity. He also wrote two books on philosophy: *The Open World: Three Lectures on the Metaphysical Implications of Science* (1932), a small book with three lectures that Weyl gave at Yale University in New Haven, and *Philosophy of Mathematics and Natural Science*, published by Princeton University Press in 1949, an expanded version of a book he originally published in German.

In these two books Weyl emphasizes the importance for the philosophy of science of an idea that Leibniz had about complexity, a very fundamental idea. The question is what is a law of nature, what does it mean to say that nature follows laws? Here is how Weyl explains Leibniz's idea in *The Open World*, pp. 40-41: The concept of a law becomes vacuous if arbitrarily complicated laws are permitted, for then there is always a law. In other words, given any set of experimental data, there is always a complicated *ad hoc* law. That is valueless; simplicity is an intrinsic part of the concept of a law of nature.

What did Leibniz actually say about complexity? Well, I have been able to find three or perhaps four places where Leibniz says something important about complexity. Let me run through them before I return to Weyl and Popper and more modern developments.

First of all, Leibniz refers to complexity in Sections V and VI of his 1686 *Discours de métaphysique*, notes he wrote when his attempt to improve the pumps removing water from the silver mines in the Harz mountains was interrupted by a snow storm. These notes were not published until more than a century after Leibniz's death. In fact, most of Leibniz's best ideas were expressed in letters to the leading European intellectuals of his time, or were found many years after Leibniz's death in his private papers. You must remember that at that time there were not many scientific journals. Instead, European intellectuals were joined in what was referred to as the Republic of Letters. Indeed, publishing could be risky. Leibniz sent a summary of the *Discours de métaphysique* to the *philosophe* Arnauld, himself a Jansenist fugitive from Louis XIV, who was so horrified at the possible heretical implications that Leibniz never sent the *Discours* to anyone else. Also, the title of the *Discours* was supplied by the editor who found it among Leibniz's papers, not by Leibniz.

I should add that Leibniz's papers were preserved by chance, because most of them dealt with affairs of state. When Leibniz died, his patron, the Duke of Hanover, by then the King of England, ordered that they be preserved, sealed, in the Hanover royal archives, not given to Leibniz's relatives. Furthermore, Leibniz produced no definitive summary of his views. His ideas