JOURNAL OF
CROSS-DISCIPLINARY RESEARCH
IN COMPUTATIONAL LAW [CRCL]

The Structure and Legal Interpretation of Computer Programs

James Grimmelmann *

Abstract

This is an essay about the relationship between legal interpretation and software interpretation, and in particular about what we gain by thinking about computers and programmers as interpreters in the same way that lawyers and judges are interpreters. I wish to propose that there is something to be gained by treating software as another type of law-like text, one that has its own interpretive rules, and that can be analysed using the conceptual tools we typically apply to legal interpretation. In particular, we can usefully distinguish three types of meaning that a program can have. The first is *naive functional meaning*: the effects that a program has when executed on a specific computer on a specific occasion. The second is *literal functional meaning*: the effects that a program would have if executed on a correctly functioning computer. The third is *ordinary functional meaning*: the effects that a program would have if executed correctly and was free of bugs. The punchline is that literal and ordinary functional meaning are inescapably social. The notions of what makes a computer 'correctly functioning' and what makes a program 'bug free' depend on the conventions of a particular technical community. We cannot reduce the meaning and effects of software to purely technical questions, because although meaning in programming languages is conventional in a different way than meaning in natural languages, it is conventional all the same.

Keywords: interpretation, software, programming languages, natural language, semantics

Replier: Marieke Huisman, Professor of Software Reliability, University of Twente • m.huisman@utwente.nl

Journal of Cross-disciplinary Research in Computational Law

DOI: pending

www.journalcrcl.org

* Tessler Family Professor of Digital and Information Law, Cornell Tech and Cornell Law School. I presented earlier versions of this essay at the COHUBICOL Philosophers' Seminar on The Legal Effect of Code-Driven 'Law'? on 11-12 November 2021, to the Boston University Cyber Alliance on 6 March 2019, and to the Cornell Tech/Law Colloquium on 24 October 2017. My thanks to the organizers and participants, to the anonymous peer reviewers of this essay, and to Aislinn Black, Nate Foster, Jean Galbraith, Sarah Lawsky and Lawrence Solum. james.grimmelmann@cornell.edu.

Introduction

This is an essay about the relationship between legal interpretation and software interpretation, and in particular about what we gain by thinking about computers and programmers as interpreters in the same way that lawyers and judges are interpreters. I wish to propose that there is something to be gained by treating software as another type of law-like text, one that has its own interpretive rules and that can be analysed using the conceptual tools we typically apply to legal interpretation.

The point of departure is that legal texts have effects in the world that are based on their meaning. Statutes shape people's obligations, contracts give parties a right to each other's performance, deeds transfer property, and so on. In each case, lawyers, judges and laypeople must interpret these texts, giving them effect by determining their meaning.[1]

In a not completely dissimilar way, when a user causes a computer to execute a program, it has effects in the world. This too requires a kind of interpretation. The computer treats the text of the program as a series of instructions for action. Computer scientists call this process 'interpretation,' and the term is apt.[2] Whether or not a computer is the sort of entity that can ascribe meaning to a text, programmers and users certainly are.[3]

To motivate the comparison, consider *United States v. Morris*, in which Robert Tappan Morris was convicted under the Computer Fraud and Abuse Act (CFAA) for installing a software worm on thousands of computers.[4] The Second Circuit held that he acted 'without authorization' based on *how* he used existing programs on those computers to install his worm:

> He did not send or read mail nor discover information about other users; instead he found holes in both programs that permitted him a special and unauthorized access route into other computers.[5]

This line of reasoning presumes that *the programs themselves* had legal effects grounded in their functionality. Specific ways of using them were authorised; other ways were not. Consciously or not, judges are already interpreting software. Their practices require explanation and justification. This essay is a down payment on both.

In particular, I make three claims. First, programs do not have a single meaning that is appropriate under all circumstances. Just as there are different ways to interpret laws and literature, there are different ways to interpret programs, and sometimes they yield different meanings.[6] Second, although these different meanings need not be identical to the functional effects of executing a program, they are all at least derivative of those effects. The execution of programs on computers (both actual and hypothetical) is at the heart of their meaning. Third, these meanings are inescapably social. To the extent that we care about the meaning of

1 See William Baude and Stephen E. Sachs 'The Law of Interpretation' (2017) 130 Harvard Law Review 1079.

2 See e.g. Harold Abelson and Gerald Jay Sussman with Julie Sussman, *The Structure and Interpretation of Computer Programs* (2nd edn MIT Press 1996).

3 See Lawrence B. Solum, 'Artificial Meaning' (2014) 89 Washington Law Review 69.

4 *United States v. Morris* 928 F. 2d 504 (2d Cir. 1991).

5 Ibid. at 510.

6 See generally Kent Greenawalt, *Legal Interpretation: Perspectives from Other Disciplines and Private Texts* (Oxford University Press 2010).

a program, that meaning cannot be reduced to a purely technical question.

Naive functional meaning

Legal texts are addressed to people: citizens, counterparties, guests and especially judges. They provide instructions that people are expected to understand and implement. So we care about their meaning to people, and our interpretive tools are meant to ascertain these texts' meanings for appropriate audiences of people.

The analysis of program interpretation is different because it inherently involves a computer. Even when a person is reading a program, to read it *as a program* is to treat it as instructions to a computer. The interpretive tools required must take account of this fact.

This section takes a first cut at a theory of functional interpretation grounded in the idea that programs are defined by the fact that they can be executed on a computer. The simplest possible such theory is that a program's meaning consists of the effects it has when executed. I will call this theory *naive functional meaning*, and while it ultimately falls short as a theory of program meaning, it is the foundation on which better theories can be built.

Software is functional text

Software is functional text addressed to a computer. A program consists of a sequence of instructions. If the program is provided to a computer in the right way and the computer placed in an appropriate state, the program will execute: that is, the computer will interpret each instruction in turn, giving the instruction effect by changing its own state. This can cause it to display information in a form humans can understand or to take other actions humans can observe. But first and foremost, the program simply causes a computer to alter its own state. That is *what a computer program is*; that is what I mean when I say that software is primarily functional. Any other effects or meanings software has are derivative of this functional core.[7]

Programs, however, are not simply functional artifacts. Outboard motors, guillotines, Bunsen burners and other machines also do things in the world. Software is distinctive in that its functional effects are themselves derivative of its meaning. Programs are *texts*. True, they are written in programming languages rather than natural languages and they are addressed to computers rather than people. But they consist of sequences of symbols in a language, their use is to convey information, and anything they do they do because they convey information to a recipient (a computer) that acts on it. This is the starting point for analysis of programs as texts; all other meanings we may wish to ascribe to them as texts are derivative of the fact that they are functional.

To be sure, a computer does not 'understand' or 'interpret' a program in the same way that a person does. (Indeed, the point of this section is to contrast how computers interpret programs with how people interpret natural-language texts.) But the essentially textual character of software is undeniable. A program is not a tangible artifact like an automobile; it is made of intangible information. Whether or not computers understand a program by ascribing a meaning to it, programmers certainly do — both when they write code in the first place and when they read each others' code.

[7] For a parallel example of how a family of distinct concepts can be derivative of a core concept, see Peter Westen, *The Logic of Consent: The Diversity and Deceptiveness of Consent as a Defense to Criminal Conduct* (Routledge 2004).

Indeed, programmers create and analyse software in a fundamentally written-linguistic way. The process starts with individual symbols, which natural linguists would call graphemes and programming linguists would call characters. These symbols are assembled into units (morphology or lexical analysis, respectively), these units are themselves assembled into larger units (syntax), the larger expressions are given abstract meanings (semantics), and those meanings are filled in with specific contextual details (pragmatics or the runtime, respectively). The terminology is different, but the processes are structurally analogous.[8]

Meaning and effect

There is an obvious way to ascertain the effects of a program: run it and see what happens. The computer will do something. That 'something' is the program's functional effect. Since programs, by definition, are the class of texts that cause computers to do things, we can use those effects to attribute meanings to the programs that produced them.

I will call this theory of interpretation *naive functional meaning* and it is beautifully straightforward. It asserts that what a program means is what it does, and what a program does is what it means. There is no daylight between the two.

Naive functional meaning is conceptually simple because it equates one concept of interest (a program's meaning) with another (a program's effects). It is also operationally simple because it supplies a real-world procedure to answer questions about program meaning: execute the program and observe what happens. (We will see in a moment that this 'straightforward real-world procedure' is not quite so simple.)

The cardinal virtue of naive functional meaning as a theory of interpretation is that it is unusually clear. Natural language is inherently vague. No listener ever understands an utterance in perfectly the same way as the speaker meant it. Patterns of usage are always contested around the margins and always in flux. Words never fully capture the messy complexity of reality. Any natural language is at best an approximation.

Programming languages do less but within their domain they are comparatively more precise. They do not aspire to describe the world in all its detail, to express the richness of subjective experience, to make threats or issue warnings. They are good for one thing and one thing only: issuing commands to a computer. Within their circumscribed domain, programming languages avoid many of the pitfalls of natural language.

This is the sense in which computers are 'rational,' 'logical,' and 'objective.' One can argue about whether a person is 'tall,' but there is no point in arguing about what a program does, when we can run it and find out. Leibniz wanted to make language precise and computable, so that 'when there are disputes among persons, we can simply say: Let us calculate, without further ado, to see who is right.'[9] Programming languages, it appears, achieve that ambition.

The interpretation-construction distinction provides an illuminating perspective on naive functional meaning. In the theory of legal interpretation, it is conventional to distinguish *interpretation* from

[8] Compare Adrian Akmajian, Ann K. Farmer, Lee Bickmore, Richard A. Demers and Robert M. Harnish, *Linguistics: An Introduction to Language and Communication* (7th edn MIT Press 2017) (natural linguistics) with Robert Sebesta, *Concepts of Programming Languages* (11th edn Pearson 2015) (programming linguistics).

[9] Gottfried Wilhelm Leibniz, 'The Art of Discovery (1685)' in Philip P. Wiener (ed), *G. W. Leibniz: Selections* (Scribner's 1951).

construction as stages of giving legal effect to a text.[10] Interpretation is the process of determining the linguistic meaning of a text. Construction is the process of translating that linguistic meaning into its legal effects. The conventional account of the distinction is that construction is necessary because texts can be both ambiguous and vague. Interpretation can resolve ambiguities and select the most appropriate of several possible meanings. But, when a text is vague and has no determinate meaning in some respect, interpretation cannot help. The meaning 'runs out', and the process of construction fills in the gaps by consulting sources beyond the text's linguistic meaning. These can include the expectations and goals of the text drafters, historical practice, normative theories, policy consequences, administrability and many other practical considerations.

In terms of the interpretation-construction distinction, then, naive functional meaning says that programs require interpretation (by a computer) but not construction (by anyone). They are neither vague nor ambiguous. A program has exactly one (technical) meaning, which corresponds to the (functional) effects that it has when run. Thus, naive functional meaning allows the computer to do the work of interpretation (by executing the program) and then considers any further work of construction to be unnecessary (since the computer has arrived at a single determinate meaning for the program).

Naive functional meaning as a foundation

Naive functional meaning is obviously insufficient as a theory of interpretation for a simple but devastating reason: programs are buggy. Real-world programs go spectacularly wrong all the time in all kinds of ways. Under naive functional meaning, every bug has binding legal effect. If a bank's ATM software dispenses USD 1000 in cash to anyone who holds down the right combination of six buttons at once, naive functional meaning would say that the entrepreneurs who go around town draining every ATM they can find are entitled to keep every dollar.[11] The software's naive functional meaning is that anyone who holds down the right six buttons receives USD 1000; it provides no basis on which to say, 'But that *shouldn't* work!' Naive functional meaning takes Lessig's slogan *code is law* to the natural extreme: *bug is law*.[12]

But for all its flaws, naive functional meaning contains a crucial core of insight. While legal meaning cannot always be *identical* to technical meaning, it must at least be *grounded* in technical meaning. Not everything goes with software. Video poker is not video backgammon. A smart contract is not a potato-salad recipe. We can argue over whether a website allows or prohibits access to a file, but both of these arguments presuppose the possibility that *such questions about software can be answered at all*. An ordinary video-poker wager produces a clear win or loss, which is to say that sometimes, in fact most of the time, technical meaning does determine legal meaning.[13]

The problem with naive functional meaning is that it is committed to the actual effects a program has when run, *whether those effects are right or wrong*. To improve on it, we must find a way to recognise and isolate cases in which a program has somehow gone wrong, to keep the program's meaning from going

[10] Lawrence B. Solum, 'The Interpretation-Construction Distinction' (2010) 27 Constitutional Commentary 95.

[11] For an actual case with similar facts, see *Kennison v. Daire* [1986] HCA 4.

[12] Lawrence Lessig, *Code: And Other Laws of Cyberspace* (Basic Books 1999).

[13] See James Grimmelmann, 'Computer Crime Law Goes to the Casino' (Technology | Academics | Policy (TAP), 24 May 2013) <https://www.techpolicy.com/Grimmelmann_ComputerCrimeLawGoesToCasino.aspx> accessed 12 April 2023.

down with the ship. In other words, we require a theory of program correctness: meaning equals effects only when the program executes correctly according to an externally specified standard.

Programming-language specification

Fortunately, computer science has a theory of correctness via specification: indeed, it is the conceptual foundation on which the entire discipline is built. By specifying precisely what a program is *expected* to do, programmers can treat any actual execution that deviates from the specified expectation as incorrect, whose effects can be disregarded. Indeed, with a proper specification, there is a sense in which actual execution becomes unnecessary. Instead, the program's meaning can be defined in terms of the effects it would have on a properly functioning computer that performs according to the specification.

Hardware and software as abstractions

Everyone who has ever worked with computers knows that computers do not always work. One way in which computers can fail is that, just like any other physical device, they malfunction. Just as screwdrivers sometimes slip, gears sometimes break and engines sometimes lock up, computers sometimes go wrong even if perfectly programmed. A stray cosmic ray can cause a bit in memory to flip from 1 to 0 or 0 to 1. A power surge can cause a computer to crash. Semiconductor chips have manufacturing defects, rats chew through wires and hard drives wear out. These are simply facts of life to be managed. Computer and software engineering are disciplines dedicated to systematically overcoming the fallibility of physical hardware.

The central idea of computer science is a response to this problem: modelling computers as mathematical abstractions. An idealised 'stored-program computer' consists *only* of a memory unit which contains numerical data and a processor that can carry out very simple instructions, like 'add numbers x and y.'[14] The processor fetches an instruction from the memory unit along with any data the instruction needs, then carries out the instruction. New values can be written back to the memory and an instruction can designate what location in the memory holds the next instruction to be executed.

The stored-program computer model is artificial. But it occupies an appealing middle ground. On the one hand, it is a faithful model of actual real-world computers: although they are vastly more complicated, their basic operations are faithfully represented at an abstract level as stored-program computers. On the other hand, stored-program computers can themselves be elegantly modelled by simple mathematical abstractions like finite-state machines, Turing machines and the lambda calculus. Thus, they provide a bridge between actual computers in all their messy complexity and clean, well-behaved mathematical models.

The stored-program computer model has a number of important features. First, it abstracts away from irrelevant physical details. It does not matter whether the voltage level in a transistor is 1.49 volts or 1.51 volts: both will be regarded as simply representing a zero. This is the key move that simplifies the immense complexity of reality to the point where it is theoretically

[14] See generally David Money Harris and Sarah L. Harris, *Digital Design and Computer Architecture: ARM Edition* (Morgan Kaufmann Publishers 2015).

tractable. Second, it abstracts away from specific hardware. Two Intel chips may have subtle manufacturing differences but they ought to be functionally identical as far as the model is concerned. Indeed, so should an Intel chip and an AMD chip implementing the same instruction set. The actual hardware is irrelevant; only its behaviour on an abstract level is relevant. In theory, you could build a stored-program computer out of water wheels or Tinkertoys. Third, it abstracts away from faulty hardware. The memory chip hit by a cosmic ray deviates from the correct behaviour of a memory chip. Chip architects and engineers will work to produce chips that do not suffer such faults and software engineers will work to write programs that can detect and fix them but all of this work is so that programmers working with these chips can *ignore* the possibility of cosmic-ray bit flips (most of the time). The mathematical idealisation of the chip is free from these physical risks entirely and programmers spend most of their time working with the idealisation.

Another feature of the stored-program computer model is a little subtler. It is a *general-purpose* computer. By putting different instructions in the computer's memory at the outset, it is possible to make the computer carry out different functions. Unlike a special-purpose device which is designed to compute a single function, like 'find the square root of a number', a general-purpose computer can compute any function for which it is given suitable instructions. Or, in more familiar terminology, it can be 'programmed'.

This, then, is the source of the distinction between hardware and software. A general-purpose device, the hardware, is a physical system with physical properties. It can be programmed by loading it with specific instructions, the software. The actual line between hardware and software is vague and flexible; today it is easy to find examples in which each can and often does take on jobs usually associated with the other. But the distinction itself is central to modern computer science. It allows programmers to work with software as text: sequences of symbols, abstracted away from the specific machines on which they will run and from the specific media on which they are stored. It also allows us as theorists to work with software as text and to focus on what that text communicates.

Programming language semantics

Now that we have isolated programs as texts in our description of software, we can confront the question of how a program comes to have a specific and specified meaning. It is all well and good to say that a programmer can write the Python program

```
2 ** 3
```

but what does this program *do*? In one sense the answer is trivial. Any competent Python programmer is able to say that this program yields the value `8`. But in a deeper sense, this answer just raises further questions. Why is it that all Python programmers agree? What had to take place for this remarkable uniformity to be possible?

A descriptivist might say that the pattern of regularity in the programmers' answers is itself the only relevant fact; there is nothing further to be posited or explained. But while this works as a theory of natural language — language consists of usage norms — it fails as a theory of programming languages. I could sit a total novice down at a keyboard, have them type `2 ** 3` in a Python interpreter and ask them to report back on what happens. This person who has never programmed before, who does not 'speak' Python and who has no knowledge whatsoever of what the symbol `**` means or does, will nonetheless still report back

that the interpreter prints '8'. So there is apparently a fact about what `2 ** 3` means in Python that is independent of the usage habits of Python programmers. The meaning of a Python program inheres at least in part in Python, not just in programmers' minds.

A prescriptivist might then argue that Python defines what Python programs mean and that the programmers' usages and expectations are irrelevant. But where did 'Python' come from? In 1917, long before computers and long before Python, `2 ** 3` had no meaning as a Python program. Or what if I modify my Python interpreter (it is an open-source program, after all) so that `**` is a multiplication operator rather than an exponentiation operator, in which case my interpreter will print `6` instead of `8`? So something outside of the interpreter itself must determine the meaning of this program. (Once again, 'whatever the program does' is an incomplete answer.)

The answer to both questions is that programming language semantics is both a social and technical process: people agree about the meanings of programs not directly, by saying 'expression *E* in programming language *L* means *M*' but indirectly, by codifying their agreements in technical processes that assign meanings to classes of expressions in a programming language.[15] The community of Python programmers agree on *what Python is*, and the meanings of specific Python expressions (such as `2 ** 3`) follow from that agreement.[16]

There are four broad classes of techniques programmers use to express their agreement on programming-language semantics: informal natural-language descriptions, formal mathematical semantics, reference implementations and test cases. It is worth looking at them each in more detail. I will refer to them collectively as *specification.*

Natural-language descriptions

A description can be a few lines in a README file telling a user what a program does in very general terms. But others are more ambitious: they strive to describe a language in sufficient detail such that one could create an implementation by reading the description and making the implementation conform to it. Java, for example, has an 848-page language reference,[17] supplemented by a 624-page virtual machine reference.[18] These detailed descriptions are primarily written in natural language, generously sprinkled with mathematical notation and technical terms of art. Here is an example from the 2,279-page Python Library Reference:

> `Math.atan2(y, x)`
> Return `atan(y / x)`, in radians. The result is between `-pi` and `pi`. The vector in the plane from the origin to point `(x, y)` makes this angle with the positive X axis. The point of `atan2()` is that the signs of both inputs are known to it, so it can compute the correct quadrant for the angle. For example, `atan(1)` and

[15] See generally Donald A. Mackenzie, *Mechanizing Proof: Computing, Risk, and Trust* (MIT Press 2001).

[16] See generally Stanley Fish, 'Interpreting the "Variorum"' (1976) 2 Critical Inquiry 465, 483 (defining an 'interpretive community' as 'those who share interpretive strategies not for reading (in the conventional sense) but for writing texts, for constituting their properties and assigning their intentions').

[17] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith and Gavin Bierman, *The Java® Language Specification* (Java SE 17th edn 2021).

[18] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley and Daniel Smith, *The Java® Virtual Machine Specification* (Java SE 17th edn 2021).

`atan2(1, 1)` are both `pi/4`, but `atan2(-1, -1 )` is `-3*pi/4`.[19]

To understand the expected meaning of `atan2(1)` in Python, a programmer would need to know enough mathematics to know what the 'arc tangent' is. To write their own implementation of Python that includes this function, they would also need to know enough about numerical methods to compute it accurately.

Formal mathematical semantics

A second way to achieve consensus on the meaning of programs in a given programming language is to give a *formal semantics* for the language, which identifies programs with abstract mathematical objects and states rigorous theorems about those mathematical objects.[20] For example, the following rule defines the semantics of an **add** operator in a hypothetical programming language by adding (+) the values (v_1 and v_2) of the expressions (e_1 and e_2) that **add** operates on.

$$\frac{e_1 \to v_1 \quad e_2 \to v_2}{\textbf{add}\ e_1 e_2 \to v_1 + v_2}$$

Formal semantics are not typically intended for everyday use by programmers. Instead, they are useful in establishing desirable properties of programs and implementations. One use is in establishing that a language as a whole has some nice feature, e.g. that programs written in it will never crash because they tried to access a forbidden part of memory. These proofs depend on the formal semantics of the languages to which they apply. Another use is to automate the process of looking for possible bugs in a code fragment — or proving that it can contain no bugs of a given type — a process that depends on having a good formal semantics for the language in question. And a third is to support the process of writing good implementations of the language: a well-done formal semantics can achieve high standards of unambiguity in describing correct program behaviour.

Reference implementations

A third way for programmers to express their agreement about a programming language is for them to agree on a specific instantiation of that programming language, a *reference implementation*. The reference implementation is a version of the interpreter that executes programs in the language (or a compiler that translates them into executable forms, or a translator that transforms them from one language to another with its own agreed-upon semantics). The community of programmers agrees to treat the reference implementation as authoritative as to the meaning of programs in the language: whatever the reference implementation does is considered the correct behaviour. Other implementations are possible, and for many languages are quite common, but when another implementation differs in its behaviour from the reference implementation, the other implementation is considered to be the buggy one. This is an appeal to 'whatever the program does', but with a crucial difference: it is an appeal to a socially agreed program, and therefore it excludes my idiosyncratic modified version of Python.

This is a common strategy, so common in fact that programmers do not always realise they are relying on it. Python does not have an official reference implementation but it does have one so widely used that it is *de facto* the reference implementation, called CPython. It is open-source software, written in a mixture of Python and C, and if you are curious, you can browse its source code at

[19] Guido van Rossum and the Python Development Team, *The Python Library Reference* (Release 3.10.1, 16 December 2021).

[20] See generally e.g. Glynn Winskel, *The Formal Semantics of Programming Languages: An Introduction* (MIT Press 1993).

<https://github.com/python/cpython>. If you install 'Python' on your computer, you are likely to get a version of CPython and many other Python implementations are modified versions of CPython. When I teach a programming course in Python, we spend time in the first week making sure that every student has access to the same version of CPython. The result is that I can tell students in class, 'open up Python and type in (...)' and be confident that everyone in the class will see the same result. This is, on a smaller scale, the kind of consensus that agreement on a reference implementation provides.

Test cases

A fourth way of establishing consensus on program meaning is to describe the behaviour of a language using *test cases*. A test case for a program is an input together with the expected results of running the program on that input. Because test cases are examples, they technically do not say anything about what programs should do on other inputs. It is very rare to specify a language's semantics entirely via test cases. Instead, test cases are used in conjunction with other approaches. For one thing, test cases can help make informal descriptions and formal semantics more intelligible to human readers by providing concrete examples of abstract rules. (The description of `atan2` above has three test cases embedded in the English text.) For another, test cases can be used as double checks for languages specified via these other methods: *if* the language is implemented correctly, *then* these test cases will be correctly computed. An implementor who sees a test case fail knows they must have made a mistake somewhere.[21] Test cases can even be used to verify that a reference implementation conforms to *itself*. A 'regression test' is a test that a programmer runs after making some changes to their program: if the results of the test change, it is an immediate warning that something in the program's behaviour has changed. On the assumption that the program was working correctly before the change, the programmer immediately knows that it is now working incorrectly.

Literal functional meaning

However achieved, specification of a programming language provides a new way to ascribe meaning to a program written in that language. Consider two cases: in one, a bit flips in a computer's memory as the result of an addition specified by a program, while in the other, a bit stored in the same physical memory location flips as the result of being hit by a cosmic ray. Naive functional meaning had no way to distinguish these two cases, but now that we have a specification for the relevant programming language — in this case, the low-level binary language of machine code that the physical computer implements — there is an obvious difference between the two.[22] Only the first result is consistent with the specification, that is, with the ideal stored-program computer that the physical computer implements and approximates.

The same reasoning applies to other cases where the actual computer and the idealised computer differ: when a physical computer has a manufacturing defect, or when there is a bug in the software toolchain that turns a programmer's source code into executable machine code, or when the programmer executes their code on an idiosyncratic non-standard system.

[21] See Kent Beck, *Test-Driven Development by Example* (Addison-Wesley Professional 2002).

[22] See generally Randall Bryant and David O'Hallaron, *Computer Systems: A Programmer's Perspective* (3rd edn Pearson 2015) (describing the low-level abstractions of a computer).

In each case, reality departs from the model, but because the model defines the standard of correctness, it is reality that is wrong.

Specification, then, provides a new interpretive strategy. The *literal functional meaning* of a program is the effect that a program would have when executed on a computer that correctly implements the specification of the programming language in which the program is written. Literal functional meaning retains much of naive functional meaning's virtue in minimising ambiguity: the meaning of a program is simply what the specification says.

While naive functional meaning is tied to specific executions of a program on specific systems on specific occasions, literal functional meaning abstracts away from particular executions, computers and occasions. In doing so, it avoids the pitfalls of following a specific broken computer into the abyss; it is not subject to problems of hardware malfunctions and idiosyncratic variations in computer systems. Because of this abstraction, literal functional meaning is a genuine theory of the meaning of programs as *texts*, rather than of computers as physical machines.

Literal functional meaning is in a sense an extreme version of natural-language literalism.[23] It is both simple and rigorous. Although not quite as simple as naive functional meaning, it is simple because of how comparatively little it requires from judges and other legal interpreters. Find an implementation of the relevant programming language that the relevant technical community recognises as correctly implementing the language specification. Then run that implementation and see what it does. And it is rigorous because it leaves so little wiggle room. Even fairly pure variants of literalism have always included escape hatches like the doctrine of scrivener's error: sufficiently obvious mistakes in a text will be corrected on the (almost always fictional) theory that the mistake is the product of a later copyist's mistranscription.[24] Literal functional meaning, for better or worse, has no such outs.

Consensus and correctness

Literal functional meaning's freedom from the physical comes at a steep price. Naive functional meaning is tied to an actual execution: a program's effects and hence its meaning can be determined by examining a specific physical computer, whose state can be objectively observed (insofar as anything can be). But literal functional meaning depends on an abstraction and there is no guarantee that any actual computer correctly implements the language specification that defines the meaning of a program. The program must be interpreted in light of the specification, which requires knowing what specification to interpret it against.

Programs are not self-defining. We can give them a meaning only with respect to a specific programming language. Consider 'polyglot' programs, which are valid programs in multiple different languages. The program below, for example, is valid in six different programming languages (Perl, C, the Unix shell, Brainfuck, Whitespace, and Befunge).[25] Its semantics are well-defined only once one chooses one of these six languages in which to interpret it as a program.

[23] See François Recanati, *Literal Meaning* (Cambridge University Press 2004).

[24] See John David Ohlendorf, 'Textualism and the Problem of Scrivener's Error' (2011) 64(1) Maine Law Review 119.

[25] See Thomas Schoch, 'computer/programs/useless/misc/polyglot' <https://retas.de/thomas/computer/programs/useless/misc/polyglot/index.html> accessed 12 April 2023.

```
# define x u /*      v
# ::::::::::::::::::>>>>>>>$$$a"muroftih"#[>:#,_@]
eval 'echo "hitforum";exit';sub echo { print "@_\n"}
__END__>++++++++++>++++++++++[>++++++++++>++++++++++
+<<-]>------.+.>++++++.<---.+++++++++.>--.+++
.<--.<<. */
main() { printf ("hitforum\n"); }
```

This is an example of a deeper problem. It is not enough for literal functional meaning to say that a program is written in a given language. To interpret a program, one must also know the contents of that language's specification. This is a conventional fact, not a natural one: it depends on the practices of the people who use the language.[26] All four methods of programming-language specification described above — reference implementations, informal descriptions, formal semantics and test cases — are fundamentally social processes. `2**3` in Python returns `8` not because it has to in any metaphysical sense, but because a community of Pythonistas agreed that it should. They agreed that 'Python' is defined by what the Python Language Reference says and what CPython does. What CPython does on a given input is almost entirely insulated from social processes, but the underlying agreement that what CPython does is constitutive of Python is not.

The source of meaning for computer programs, then, is consensus within a technical community: the language maintainers who write implementations of a programming language, the programmers who write programs in that language and the users who run those programs using the implementations. Through a combination of implementations, descriptions, semantics and test cases the members of that community agree in broad strokes about a process for extracting functional effects from the text of programs. Community members instantiate that process on different computers, using different implementations, and so on. Most of the time, running the same program on these different instantiations will produce (what community members agree is) the same result. This is how the meaning of programs is fixed.

More precisely, to say that some combination of reference implementation, description, formal semantics, and test cases is *the* specification S for a programming language L is to say that the community of programmers and users of L have agreed that the correct behaviour of programs in L is defined by S. Even more precisely, they have agreed that S provides a general, effective, and authoritative procedure for determining the effects of any program P that is written in L. If applying S to P yields effects E, then P means E in L, full stop, end of story. You can assert that P has some other effects E' by explaining that P is actually a program in some other language L'. You can try to persuade the relevant community to adopt some other specification S' for L. But you cannot accurately assert that P, regarded as a program in L, yields E'. It does not. The community has agreed otherwise and the community's consensus does not support your idiosyncratic meaning.

Breakdown

The dependence of literal functional meaning on conventional facts has real consequences for interpretation. One is that it opens up a gap through which

[26] See Andrei Marmor, *Social Conventions: From Language to Law* (Princeton University Press 2009).

ambiguity can re-enter.[27] Reference implementations can contain bugs; specifications can contain ambiguous or vague phrases; even formal semantics can contain mistakes (just as mathematical proofs can). Trying to resolve these ambiguities can introduce others. If you provide a specification for a new language, and then try to iron out any glitches in the specification by also providing a reference implementation, the question will naturally arise: which controls when the two of them differ? The same problem arises if one gives a formal semantics and a natural-language description of the formal semantics, and so on. Under many circumstances, these ambiguities will remain latent. For all practical purposes, the meaning of `2**3` in Python is completely settled, despite the almost certain existence of ambiguities elsewhere in the Python specification. But they cannot be eliminated entirely. The crooked timber of humanity is visible even in the code we write.

Another way in which literal functional meaning can break down is that programming-language specifications are often incomplete. Web pages can look different in Firefox and Chrome because these two browsers implement the Cascading Style Sheets language standard with slight differences.[28] Exhibiting a CSS program (technically a 'style sheet') is not sufficient to resolve the ambiguity about how it will appear on the user's screen without also specifying what browser the user is running and other details of the execution environment. The CSS 'language' is actually a large family of closely related languages implemented by different browsers in different versions. A complete specification of a programming language often must include context-specific details.

Literal functional meaning can also change over time, simply because programming-language communities collectively decide to change the specification of a language. If everyone agreed tomorrow that 'Python' should be defined differently, *then it would be*. For example, Python versions 3.6 and above allow the use of underscores in numbers as a kind of visual separator, for example '`1_000_000`' for one million, which is easier to read than '`1000000`'.[29] Previous versions disallowed underscores in numbers, so a program containing '`1_000_000`' will produce an error if run in Python 3.5.2 but will work in Python 3.6.1. These changes, almost by definition, create ambiguities about what 'Python' does on particular inputs; one must specify a version number to resolve the ambiguity.

If one is confronted with a 'Python' program and asked what it means, the only sensible way to answer the question is to look to the specific setting in which the program is to be used and to the usage patterns of the community running similar software, to determine what version they are using, and how they understand it to be defined.[30] Literal functional meaning commits us to asking empirical questions about social facts.

Ordinary functional meaning

Literal functional meaning is not a complete theory of software interpretation either. The process of filling its

[27] See generally Terry Winograd and Fernando Flores, *Understanding Computers and Cognition: A New Foundation for Design* (Ablex 1986).

[28] See 'Can I Use' <https://caniuse.com> accessed 12 April 2023 (documenting browsers' different support for Web standards).

[29] See Georg Brandl and Serhiy Storchaka, 'Python Enhancement Protocol 535 — Underscores in Numeric Literals' (Feb. 10, 2016).

[30] See James Grimmelmann, 'All Smart Contracts Are Ambiguous' (2019) 2 Journal of Law & Innovation 1.

most obvious gap bears a striking resemblance to the process by which we developed it in the first place.

Just as naive functional meaning lacks a satisfying treatment of hardware malfunctions — cases in which the *computer* fails to function as expected — literal functional meaning lacks a satisfying treatment of bugs — cases in which the *program* fails to function as expected. In a malfunction, the computer diverges from its specification; in a bug, the program diverges from its intended functionality. Just as naive functional meaning treats the computer's actual behaviour as authoritative even when we know better about what the computer should have done, literal functional meaning treats the program's behaviour according to the language specification as authoritative, even when we know better about what the program should have done. Just as we developed literal functional meaning by looking closely at the source of our knowledge about what the computer should have done, we can develop a theory of *ordinary functional meaning* by looking closely at the source of our knowledge about what the program should have done. Even the price we pay is similar: ordinary functional meaning will have to take on board additional conventional facts about the practices of programmers and users.

Bugs

Suppose that I am writing a program to draw an octagon. But when I run my program, an eight-pointed star appears on the screen. My code has a bug. So I look closely at the code and I find that I have gotten the math wrong: at each corner, the line should turn by 45 degrees, not 135 degrees. So I delete `135` and replace it with `45`. Now, when I run my program again, an octagon appears. I have fixed the bug.

The literal functional meaning of my original program was to draw an eight-pointed star. The literal functional meaning of my revised program was to draw an octagon. Two programs, two meanings. From the point of literal functional meaning, these two are equally valid. But from my point of view as a programmer, the two are not equally valid: one is buggy and one is correct.

The concept of a bug presumes a distinction between the actual and intended behaviour of a program. Program *P* actually does *E* but the programmer intended *E'* and they can achieve it by changing the program from *P* to *P'*. There are many kinds of bugs. A programmer could type the wrong expression: ** instead of *. They could misunderstand how the language they are using works. They could misunderstand how the algorithm they chose works. They could misunderstand the problem they are trying to solve, fail to anticipate a possible user input, make an incorrect assumption about the world, misunderstand a library or API they relied on, miscommunicate with a colleague, forget what they were doing at a previous time and do something inconsistent with it, run the program on hardware that violates their expectations for how it works or regret doing something they fully intended at the time, to name just a few.

This list bears more than a passing resemblance to the list of ways to misspeak in a natural language. The distinction between actual and intended meaning, then, carries over from natural to programming languages. Just as a speaker might produce an utterance that their human audience understands differently than they intended, a programmer might produce a program that computers interpret differently than they intended. The program has a determinate meaning in the programming language they are using; it is just not the meaning they were trying to express. As programmers can attest, this divergence between what a

program does and what you want it to do is common. Indeed, by some estimates programmers spend as much time debugging — i.e. trying to close this gap — than writing code in the first place.[31]

Debugging is characteristic of programming but not of natural language. It is worth asking why. In conversations, and even to a significant extent in writing, people detect and correct misstatements by noticing incongruities and through discussion. Computers, however, are ill-positioned to do either. Instead, error detection and correction for programs is a matter of second-best approaches. They can be coded to accept a wider range of inputs and make different assumptions about what the user might have meant: a spell-checker with autocorrect is a simple example. But they still have to be programmed to do so; they lack (for now at least) the broad adaptability humans have in understanding the nuances of what a speaker might have intended.[32]

Ordinary functional meaning

Now it should be clear how literal functional meaning goes wrong in a normative sense when it deals with buggy code. It is oblivious to the knowledge that a relevant community of programming-language users brings to the task *about how the program is intended to function*. It is precisely because technical communities are capable of recognising and fixing bugs that we can appeal to those abilities in constructing another theory of program meaning, one that makes appropriate corrections for buggy code.

The ordinary linguistic meaning of an utterance is the meaning that a reasonable audience would give it.[33] The audience, as competent speakers of the relevant language, make allowances for slips of the tongue, grammatical mistakes, confusions about word meanings, and more. They attempt to reconstruct, as best they can, the meaning that the speaker intended to convey by means of the utterance.

The audience for a program consists of the relevant technical community of programmers and users. And that technical community is familiar with the distinction between actual and intended program meaning; it expects (as a predictive matter) that programs contain bugs. Sometimes, when looking at a program, readers can tell not just what it actually does, but what its programmer intended for it to do. Not all bugs are of this sort, but many are, and when programmers encounter one, they will reliably agree on what the actual programmer probably wanted.

Thus, the ordinary functional meaning of a program is what reasonable people in the position of its programmer and knowing what they know would expect the program to do, *if it were free of bugs*. Ordinary functional meaning is more prone to ambiguity than literal functional meaning — witness the old joke, 'That's not a bug, that's a feature!' — but it captures the collective

[31] E.g. Chris Grams, 'How Much Time Do Developers Spend Actually Writing Code?' (The New Stack, 2019) <https://thenewstack.io/how-much-time-do-developers-spend-actually-writing-code/> accessed 12 April 2023 (32% 'Writing new code or improving existing code' versus 31% 'Code maintenance' and 'testing').

[32] Cf. Karen E.C. Levy, 'Book-Smart, not Street-Smart: Blockchain-Based Smart Contracts and the Social Workings of Law' (2017) 3 Engaging Science, Technology & Society 1 (discussing why contracting parties may not want the literal exactitude of computer interpretation of software).

[33] See generally Brian G. Slocum, *Ordinary Legal Meaning: A Theory of the Most Fundamental Principle of Legal Interpretation* (University of Chicago Press 2015).

expectations and the collective wisdom of a technical community.

Choosing an interpretive strategy

To abuse the terminology of interpretation only slightly, naive functional meaning adopts the viewpoint of a specific actual computer, however unreasonable that computer's interpretation may be. Literal functional meaning adopts the viewpoint of a reasonable computer, where what is 'reasonable' is judged with reference to a community of programming-language users. Ordinary functional meaning adopts the viewpoint of a reasonable programmer, where what is 'reasonable' is judged with reference to a community of programmers and users.

The choice between these interpretive strategies is not one that can be made in the abstract; it depends on one's reasons for asking. Programming as a profession *depends on adopting all three as needed*, sometimes even simultaneously. Indeed, the very practice of debugging is unintelligible without both literal and ordinary theories of functional meaning. To debug a program is to change its behaviour by changing its text. Without literal functional meaning, debugging would be unnecessary — the programmer already knows what the program is intended to do. Without ordinary functional meaning, debugging would be pointless — there is no reason to prefer one program text to another. Only if a program can have both kinds of meanings at once *and the two can diverge* does debugging make sense. Even naive functional meaning has a role to play. A programmer whose code is not working needs to be alert to the possibility that the problem is with their computer (naive and literal functional meaning diverge) or with their code (literal and ordinary functional meaning diverge).

The same is true for legal interpreters of software. The choice between interpretive strategies is inextricably bound up with the normative goals of interpretation. Consider Morris and his CFAA conviction for misusing programs like sendmail.[34] The *literal* functional meaning of sendmail allowed him to install his worm program on computers, but its *ordinary* functional meaning did not. The court's holding that he 'did not use [sendmail] in any way related to [its] intended function' is a choice for ordinary functional meaning — i.e. what a reasonable user would have known sendmail was 'intended' to do, not what it actually did.[35] This choice was based on the court's interpretation of 'without authorisation' in CFAA, and that interpretation in turn implements Congress's policy decision to deter the exploitation of bugs in programs like sendmail.

But in other contexts, computer users need to be able to rely on the exact behaviour of a program, without having look behind its literal text to guess what its creator intended. If you misspell `Disallow` as `Dsallow` in the `robots.txt` file that tells search engines which directories of your website they should not index, search engines are free to index those directories.[36] Even though anyone who examines your `robots.txt` file would be able to tell that `Dsallow` is a typo, it would be *normatively* unreasonable to expect every search engine operator to program their indexing software to recognise all of the possible misspellings of `Disallow`. This is a choice for literal functional meaning over ordinary functional meaning.

[34] Morris (n 4) at 510.

[35] See James Grimmelmann, 'Consenting to Computer Use' (2016) 84 George Washington Law Review 1500.

[36] See 'The Web Robots Pages' <https://www.robotstxt.org> accessed 12 April 2023.

Conclusion

Thinking about software interpretation as though it were legal interpretation may seem artificial, but it has many applications.

- First, as discussed above, sometimes it *is* legal interpretation: lawyers and judges are placed in positions where they must determine the legal effects of software and thinking of software as a meaning-bearing text provides a coherent and principled way of doing so. Other contexts where similar issues arise include copyright,[37] the First Amendment[38] and smart contracts.[39]
- Second, it sheds light on computer science. Legal interpretation is a pragmatic enterprise, in a way that is a good fit for the pragmatism of programming. Using legal theory's conceptual toolkit to talk about software engineering helps us understand the linguistic, social and normative aspects of technical processes, from standard setting to software testing.
- Third, it sheds light on law. Literal functional meaning is and is not like literal natural-linguistic meaning; ordinary functional meaning is and is not like ordinary legal meaning. Software interpretation gives legal interpretation a mirror with which to consider itself from a new and different angle.

And this is just the beginning. We have not yet considered the meanings that programs convey to their users directly: a webpage with the text 'ACCESS PROHIBITED' means something different than a page that refuses to load with an HTTP 403 error. Nor have we considered the meanings that programmers can hide in the source-code comments of their programs that have no functional consequences whatsoever. All of these, and more, are necessary for a full legal understanding of software interpretation. The work awaits.

References

Kennison v Daire [1986] HCA 4.

United States v Morris 928 F2d 504 (2d Cir 1991).

Abelson H, Sussman GJ, and Sussman J, *The Structure and Interpretation of Computer Programs* (2nd edn MIT Press 1996).

Akmajian A and others, *Linguistics: An Introduction to Language and Communication* (7th edn MIT Press 2017).

Allen JG, 'Wrapped and Stacked: 'Smart Contracts' and the Interaction of Natural and Formal Language' (2018) 14(4) European Review of Contract Law 307.

Baude W and Sachs SE, 'The Law of Interpretation' (2017) 130(4) Harvard Law Review.

Beck K, *Test-Driven Development: By Example* (Addison-Wesley Professional 2002).

Brandl G and Storchaka S, 'Python Enhancement Protocol 535 – Underscores in Numeric Literals' (10 February 2016), https://peps.python.org/pep-0515/.

Bryant R and O'Hallaron D, *Computer Systems: A Programmer's Perspective* (3rd edn Pearson 2015).

'Can I Use' <https://caniuse.com>.

[37] See e.g. Pamela Samuelson, 'Functionality and Expression in Computer Programs: Refining the Tests for Software Copyright Infringement' (2016) 31 Berkeley Technology Law Journal 1215.

[38] See e.g. Lee Tien, 'Publishing Software as A Speech Act' (2000) 15 Berkeley Technology Law Journal 629.

[39] See e.g. Jason G. Allen, 'Wrapped and Stacked: "Smart Contracts" and the Interaction of Natural and Formal Language' (2018) 14 European Review of Contract Law 307.

Fish SE, 'Interpreting the "Variorum"' (1976) 2(3) Critical Inquiry 465.

Gosling J and others, 'The Java® Language Specification' (9 August 2021) <https://docs.oracle.com/javase/specs/jls/se17/ html/>.

Grams C, 'How Much Time Do Developers Spend Actually Writing Code?' (2019) <https://thenewstack.io/how-muchtime-do-developers-spend-actually-writing-code/>.

Greenawalt K, *Legal Interpretation: Perspectives from Other Disciplines and Private Texts* (Oxford University Press 2010).

Grimmelmann J, 'Computer Crime Law Goes to the Casino' (Technology | Academics | Policy (TAP), 24 May 2013) <https://www.techpolicy.com/Grimmelmann_ComputerCrimeLaw-GoesToCasino.aspx>.

— 'Consenting to Computer Use' (2016) 84(6) The George Washington Law Review 1500.

— 'All Smart Contracts Are Ambiguous' (2019) 2(1) Journal of Law & Innovation 1.

Harris S and Harris D, *Digital Design and Computer Architecture: ARM Edition* (Morgan Kaufmann Publishers 2015).

Leibniz GW, 'The Art of Discovery (1685)' in Wiener PP (ed), *G.W. Leibniz: Selections* (New York: Scribner's 1951).

Lessig L, *Code: And Other Laws of Cyberspace* (Basic Books 1999).

Levy KEC, 'Book-Smart, Not Street-Smart: Blockchain-Based Smart Contracts and the Social Workings of Law' (2017) 3 Engaging Science, Technology & Society 1.

Lindholm T and others, 'The Java® Virtual Machine Specification' (2021) <https://docs.oracle.com/javase/specs/jvms/se17/html/index.html>.

Mackenzie D, *Mechanizing Proof: Computing, Risk, and Trust* (MIT Press 2001).

Marmor A, *Social Conventions: From Language to Law* (Princeton University Press 2009).

Ohlendorf JD, 'Textualism and the Problem of Scrivener's Error' (2011) 64(1) Maine Law Review 119.

Recanati F, *Literal Meaning* (Cambridge University Press 2003).

Rossum G van and the Python Development Team, The Python Library Reference (Release 3.10.1, 16 December 2021).

Samuelson P, 'Functionality and Expression in Computer Programs: Refining the Tests for Software Copyright Infringement' (2016) 31(3) Berkeley Technology Law Journal 1215.

Schoch T, 'computer/programs/useless/misc/polyglot' <https://retas.de/thomas/computer/programs/useless/misc/polyglot/index.html>.

Sebesta R, *Concepts of Programming Languages* (11th edn Pearson 2015).

Slocum BG, *Ordinary Meaning: A Theory of the Most Fundamental Principle of Legal Interpretation* (University of Chicago Press 2015).

Solum LB, 'The Interpretation-Construction Distinction' (2010) 95 Constitutional Commentary.

— 'Artificial Meaning' (2014) 89 Washington Law Review 69.

'The Web Robots Pages' <https://www.robotstxt.org>.

Tien L, 'Publishing Software As A Speech Act' (2000) 15(2) Berkeley Technology Law Journal 629.

Westen P, *The Logic of Consent: The Diversity and Deceptiveness of Consent as a Defense to Criminal Conduct* (Routledge 2004).

Winograd T and Flores F, *Understanding Computers and Cognition: A New Foundation for Design* (Ablex 1986).

Winskel G, *The Formal Semantics of Programming Languages: An Introduction* (MIT Press 1993).

A reply: On the need for program contracts

Marieke Huisman • University of Twente, m.huisman@utwente.nl

I enjoyed reading this paper, in particular because it discusses a field that I know very well from a completely different perspective. There are several points that I would like to respond to.

First of all, the paper introduces several ways to look at the meaning of a computer program. However, in computer science, the meaning of a program is a well-defined concept: it is what is defined as the formal semantics of the program. Formal program semantics defines the behaviour of a program in terms of a mathematical object. All other ways that can be used to describe the meaning of a program, need to be equivalent to, or can be derived from this formal semantics. If there is a discrepancy between those, then normally the formal semantics is considered to be the ground truth.

In his paper, James Grimmelmann takes a rather basic view of what a computer program is. Modern software is becoming very complex, however, which has an impact on the meaning of the program. I will describe several of these additional complexities.

The paper never mentions explicitly that most computer programs take some input, and thus the meaning of the program depends on the input that is provided. When we take this into account, we see that we have to think about questions such as:

- what are legal inputs for the program?
- what will the program do if it is called with illegal input?

Capturing this is part of describing the meaning of a program.

Moreover, modern programming languages have additional control features like exceptions and support for parallelism. Exceptions are a way to handle exceptional situations, e.g. if a program is called with illegal input, if internally a computation goes wrong, or if the program runs out of memory. When describing the meaning of a program, this means that different modes of termination have to be considered. In particular, if a program does terminate in an exceptional state, is this considered acceptable behaviour or not?

Moreover, if a program supports parallelism, there might be multiple different behaviours, all of which are acceptable. In a parallel program, multiple computations happen in parallel, and no guarantees are given about the respective speed of these individual computations. For example, suppose that we have a variable `x`, whose initial value is `0`, and which is shared by two parallel computations. Now we could have the following behaviour: one computation first reads the current value of `x`, and then adds 4 to it and stores that in `x` again, while a second computation first reads `x`, then multiplies it by 4 and then stores the result in `x`. Depending on which computation happens first, the final value of variable `x` can be either `4` or `16`. When describing the meaning of the program, we have to capture the fact that both outcomes are possible and acceptable. This example also illustrates why the meaning of a program can never be described by looking at a single execution only. Moreover, looking at multiple executions is usually insufficient for parallel

programs: on specific hardware, one computation might always be executed first, resulting in one specific behaviour, but if the program is later executed on different hardware, it might suddenly exhibit a different behaviour. Thus, the behaviour of such a parallel program should be captured independently of the specific hardware on which the program is executed.

The paper ends with a discussion of buggy programs, and states that a program with a bug might not capture the intended meaning of the program. This is an important problem, which receives a lot of attention within computer science, and in particular within the field of formal methods. The idea is that we should decouple *what* the program is computing from *how* it is computing it. What is computed is considered to be the program specification. Ideally, this is described in a formal language (although it happens frequently that the program's intent is captured only in informal documentation), while how it is computed is described as the program implementation. Formal methods are concerned with establishing a formal relationship between the two, e.g. providing a guarantee that a program correctly respects its specification. This is of course closely connected to the program semantics, because the guarantees have to be provided for all possible program behaviours.

In my own research, the intended behaviour of a program is typically described as a program contract, which consists in essence of a precondition and a postcondition for a program. The precondition specifies what the legal inputs are for the program, while the postcondition specifies what guarantees are provided by the program. If the program is correct, it means that for all legal inputs the implemented program will achieve the properties specified in the postcondition. This formalism supports a separation of concerns: any other program that uses this program does not have to understand the implementation of the program, but can rely on everything that is specified in the contract. Contracts also provide an answer to the legal question of what the meaning of a program is. The intended meaning is described as the contract, and this is what users of the program should be able to rely upon. It is the programmer's responsibility to guarantee that the program implementation respects the program's contract.

I believe that to continue the work on legal interpretation of software, these program specifications have to be taken into account as well. This will also provide an incentive for programmers to actually describe the intended meaning of their programs as a formal specification, rather than in informal documentation only. A major advantage of writing down the formal specification is that it becomes possible to use tools to check that the program respects its specification. This will help to increase the quality of programs, and to reduce the number of software bugs that occur.

Author's response

James Grimmelmann

Marieke Huisman's thoughtful reply makes a number of good points that illustrate why the analogy between software interpretation and legal interpretation is so fruitful.

First, she mentions that 'most computer programs take some input', so that the output is a function of the input. In a similar way, legal rules are applied to many different fact patterns, so that the effects of a rule depend on the facts of a particular case. In this way, both programs and legal rules define functions from contexts to consequences.

Second, she observes that programs can be written with control features such as exception handling, so that exceptional situations can still lead to well-defined outputs. Here too there is a useful parallel to law. Many contracts, statutes, and other legal texts are written with savings clauses, which explain how the remainder of the text should be applied if part of it is held to be illegal or unenforceable.

Third, Huisman notes that programs can use parallelism such that different executions yield different results where 'both outcomes are possible and acceptable'. Here the contrast with law is striking and illuminating. This kind of nondeterminism is usually considered problematic whenever it occurs in law, even though it seems to be an inevitable characteristic of any legal system operated by humans. She succinctly reminds us that nondeterminism is not a uniquely human characteristic; it is so pervasive in computing that there are extensive formalisms to describe it.

Fourth, and most profoundly, Huisman explores the use of formal methods to 'guarantee that a program correctly respects its specification'. This is a powerful idea, because, in her words, it enables us to 'decouple *what* the program is computing from *how* it is computing it.' From the perspective of a programmer using a library, only the specified interface and not its implementation matters.

Huisman suggests that these program contracts could serve as 'an answer to the legal question of what the meaning of a program is'. This is a delightful observation. The use of the same word to describe a program 'contract' and a legal 'contract' gets at a deeper truth: they both involve a promise to behave in a particular way. I think that she is right, and in many cases the technical commitments made by the creator of an interface about its parameters and return values will be the best evidence available to a court as to what that code was intended to do, and what its users expect it do.

The final fascinating issue she raises, which I can touch on only briefly here, involves the use of formal verification to improve program quality. For all that I have emphasized the similarities between programs and legal texts as the products of human communication and consensus, this difference remains. We know how to run verification tools against software, and how to reduce or even eliminate certain kinds of errors in programs. We have only the faintest idea of how such a thing might even be feasible in law — and much to learn from computer scientists about how to tackle such a task.