All Smart Contracts Are Ambiguous

James Grimmelmann[*]

> Those who lack intimacy with the machine cannot be expected a priori to have insight into its limitations. … Even in the most formal and most mechanical of domains, trust in the machine cannot entirely replace trust in the human collectivity.[1]

---

[*] Professor of Law, Cornell Tech and Cornell Law School. I presented earlier versions of this essay at the Algorithms, Big Data, and Contracting symposium at the University of Pennsylvania Law School, at the Princeton Center for Information and Technology Policy, and to a group of students at Cornell Tech. My thanks to the participants, and to Aislinn Black, Andrew Appel, Matthew D'Amore, Karen Levy, and Stephen Sachs. This is a draft and may be substantially revised before publication.

[1] DONALD MACKENZIE, MECHANIZING PROOF: COMPUTING, RISK, AND TRUST 334 (2001).

INTRODUCTION

"Smart contracts" are neither smart nor contracts,[2] but the name has stuck. Instead, they are mechanisms that enforce agreements using software rather than with law.[3] The contracting parties write a computer program that embodies their agreement. The program updates as they perform their obligations, and automatically delivers the appropriate resources to them as they become entitled to payment. Smart contracts range from simple escrow schemes to immensely complicated joint ventures.

One argument in favor of smart contracts emphasizes the clarity and certainty of code. Legal contracts are written in natural language, which is full of ambiguity, and must be interpreted subjectively by fallible humans. Smart contracts are written in programming languages, which are unambiguous and executed objectively by infallible computers. The result is that anyone reading a smart contract can predict what it will do in response to any conceivable set of events. Legal contracts are ambiguous; smart contracts are not.

So goes the argument. But it is wrong. Smart contracts do not eliminate ambiguity — they hide it. The meaning of a legal contract is a social fact. So too is the meaning of a smart contract. It does not depend directly on what people think it means when they read it, as a legal contract's meaning does. Instead, it depends indirectly on what people think about the computer systems on which it runs. Smart contracts may in fact be more predictable and consistent than legal contracts. Or they may not. But the argument that smart contracts are not ambiguous because they cannot be is false. Worse than that, it is dangerous, because it distracts attention from the hard work required to make smart contracts work in the real world.

Part I of this essay reviews how smart contracts on blockchains work. Part II discusses ambiguity in natural and programming languages. Part III gives examples of ambiguous smart contracts. A brief conclusion then draws out some implications for blockchain governance.

---

[2] Ed Felten, Smart Contracts: Neither Smart nor Contracts? (Feb. 20, 2017), https://freedom-to-tinker.com/2017/02/20/smart-contracts-neither-smart-not-contracts/.. I will refer to "smart contracts" and "legal contracts" in this essay

[3] *Id*. For more on the terminology and a discussion of how smart contracts relate to legal contracts, see *infra* Part I.

I. SMART CONTRACTS

The defining feature of smart contracts is automation.[4] They are executed by hardware and software — physical and digital systems embedded in the world — rather than by human instructions. Thus, they provide a way for parties to enjoy the benefits of binding contracts without relying on a legal system: private law without a public authority.

The relationship between smart contracts and legal contracts is complicated.[5] It is helpful to make two additional distinctions. One is between *relations of obligation*, like the legal obligation to pay $5 on Tuesday, and the *instruments* which evidence and establish those relations, like an IOU saying "I will gladly pay you Tuesday for a hamburger today."[6] The other is between *natural* and *formal* languages. Natural languages are used by people to communicate with each other. They can evolve entirely without conscious direction, like English and Mandarin, or they can be created, like Klingon and Esperanto. Formal languages include programming languages, which consist of commands to a computer, as well as various mathematical and logical formalisms.[7]

The paradigm of a legal contract is a relation of legal obligation based on a natural-language instrument. Because legal contracts can be oral or illusory, there can be legal obligations without a corresponding instrument, and vice-versa. In additional, legal contracts can incorporate terms in formal languages. For example, the price term in a contract could be expressed using an algebraic equation or based on the output of a program. The parties' obligations would then be determined in part by the result of a computation.

Obligations can also be *technical* rather than *legal*. A technical obligation is one that is enforced immediately by a system that prevents the prohibited

---

[4] Nick Szabo, *Smart Contracts: Building Blocks for Digital Markets*, EXTROPY, no. 16, 1996 [hereinafter Szabo, *Building Blocks*]; Nick Szabo, *Formalizing and Securing Relationships on Public Networks*, 2 FIRST MONDAY (1997) [hereinafter Szabo, *Formalizing and Securing*]..

[5] *See generally* J.G. Allen, *Wrapped and Stacked: 'Smart Contracts' and the Interaction of Natural and Formal Language*, 14 EUR. REV. CONTRACT L. 307 (2018); Lauren Henry Scholz, *Algorithmic Contracts*, 20 STAN. TECH. L. REV. 128 (2017); Harry Surden, *Computable Contracts*, 46 U.C. DAVIS L. REV. 629 (2012)..

[6] Allen, *supra* note 6.

[7] *Id*.

conduct *ex ante* rather than punishing it *ex post*.[8] All but the simplest technical obligations must be based on an instrument, and that instrument must be written in a programming language — this is just another way of saying that computers do only what they are programmed to do. The paradigm of a smart contract is thus a technical obligation based on a formal-language instrument. This is where the conflation of obligation and instrument in smart contracts comes from — and also where it breaks down. Because a legal obligation can be embodied in part in a formal-language instrument, a legal obligation may therefore "wrap" a technical obligation.[9] On the other hand, parties who enter into a technical obligation may at the same time may or may not enter into legal obligations effectively wrapping it — or they may even enter into legal obligations without knowing it or intending to. Much of the literature about whether "smart contracts" are "contracts" deals with this last question, but focusing too much on it obscures the other similarities and differences in the analogy.[10]

A. Smart Contracts

The turn to automation is motivated by three well-known difficulties with natural language and human institutions. The first is ambiguity — the fear that because legal contracts are written in natural language, they will be interpreted differently by different parties and judges.[11] The second is corruption — the fear that human judges who interpret and enforce legal contracts can be threatened or bribed.[12] A third is enforcement — the fear that parties might be able to ignore a legal judgment by fleeing the jurisdiction, delay, physical force, hiding assets, or never having assets in the first place.[13] These

---

[8] James Grimmelmann, *Note: Regulation by Software*, 114 YALE L.J. 1719, 1729–30 (2005).

[9] Allen, *supra* note 6.

[10] In this essay, I focus on the parallel with contracts, rather than with other kinds of legal instruments, such as wills, statutes, and terms of service, which raise distinct interpretive issues. I even avoid dealing with many interesting legal interpretive issues raised by smart contracts, such as whether they should be regarded as contracts of adhesion.

[11] Max Raskin, *The Law and Legality of Smart Contracts*, 1 GEO. TECH. L. REV. 305, 324–25 (2017).. *See also* Surden, *supra* note 6, at 688–89; AARON WRIGHT & PRIMAVERA DE FILIPPI, BLOCKCHAIN AND THE LAW (2018)..

[12] Raskin, *supra* note 12, at 319.

[13] *See* Szabo, *Formalizing and Securing*, *supra* note 5; Szabo, *Building Blocks*, *supra* note 5..

are opportunities for smart contracts to improve on legal contracts; they are also challenges that smart contracts must confront. In this essay, I will focus on ambiguity, although, as we will see, the three are closely related.

Smart contracts are designed to respond to all three of these concerns by expressing contractual terms in a programming language rather than in a natural language.[14] Consider a standard toy example of a smart contract: a vending machine.[15] Expressing the contract for the sale of a pack of Skittles in a programming language resolves all sources of ambiguity, because programming languages are unambiguous. The machine's code to dispense an item from row C4 when the buyer has inserted $1.50 is completely specified. Committing the contract to the software resolves the fear of corruption, because computers are incorruptible. Threats and offers of bribes literally mean nothing to the vending machine. And the smart contract resolves the concern about enforcement because it takes direct control of the relevant resources. No money, no Skittles.

The vending machine is obviously limited. Scaling it up to a true smart contract platform requires identifying and overcoming its major shortcomings:

- First, the vending machine is special-purpose: it is good only for spot candy sales. A better smart-contract platform would be general-purpose, capable of being used by many parties for many kinds of contracts.
- Second, the vending machine's code is unobservable by the user. Unambiguous code can still be malicious. Every time you put a coin in one, you are trusting that its code really does instruct it to dispense Skittles when you push C4. A better smart-contract platform would make contract code visible to affected parties.

[14] In theory, a smart contract could be implemented in hardware rather than in software. But any hardware sophisticated enough to implement a nontrivial smart contract would need to be specified in some way, and that specification is effectively equivalent to a computer program for all practical purposes. It is simply a program that is compiled into special-purpose hardware, rather than into object code for execution on general-purpose hardware.

[15] Szabo, *Formalizing and Securing*, *supra* note 5.

- Third, while the machine is by definition incorruptible, its programmer and its operator are not. You won't get anywhere pointing a gun at a vending machine, but you might if you point a gun at the technician with a key to the coinbox when he comes to restock the Skittles. A better smart-contract platform would be decentralized. The power to supervise and control the execution of the smart-contract code would be dispersed over a large population, so that no individual or small group's corruption threatens the contract.
- Fourth, the machine is physically vulnerable. If you punch a hole in the window, you can grab all the Skittles you want. A better smart-contract platform would have direct control over resources whenever possible. That is, whenever it could it would use virtual resources rather than physical ones.

All of these design goals point in the same direction: put the smart contract on a *blockchain*.

B. Blockchains

At heart, a blockchain is a ledger of transactions. It organizes digital records of transactions into discrete chunks (*blocks*), and then maintains a chronological list of those blocks (the *chain*). A chain of blocks: a blockchain. Although the basic computer-science ideas are older,[16] "Satoshi Nakamoto's" Bitcoin proposal put them together in a clever way, greater than the sum of its parts.[17]

The first important design choice is that the transactions in a blockchain are *cryptographically secure*. New transactions are allowed only if they are digitally signed by the relevant party (usually the one who pays for them or

[16] Arvind Narayanan & Jeremy Clark, *Bitcoin's Academic Pedigree*, 60 COMMUNICATIONS OF THE ACM 36 (2017).. *See generally* FINN BRUNTON, DIGITAL CASH: THE UNKNOWN HISTORY OF THE ANARCHISTS, UTOPIANS, AND TECHNOLOGISTS WHO BUILT CRYPTOCURRENCY (2019).*s*

[17] Satoshi Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*. *See generally* ARVIND NARAYANAN ET AL., BITCOIN AND CRYPTOCURRENCY TECHNOLOGIES: A COMPREHENSIVE INTRODUCTION (2016)..

transfers assets) using a private key that only they (should) know.[18] New transactions are also required to be consistent with the history of transactions on the blockchain: you can't transfer Bitcoin unless you received it in a previous transaction. Together, these consistency constraints mean that only parties who have digital assets are able to use them in transactions.

The second important design choice is that the blockchain is a *distributed* ledger. Every participant has (or could have, if they wished) a complete copy of the entire blockchain. No participant's copy is canonical; all are equally authoritative. Thus, there is no centralized recordkeeper with authority over the ledger. This is where blockchains achieve their resistance to corruption: anyone hoping to tamper with the ledger will need to suborn a significant fraction of participants, not just one.[19].

The third important design choice solves a problem introduced by the second. Distributed systems need to reach some form of consensus: if multiple parties can each have copies of the ledger, there must be some way to keep their copies in sync, or to deal with the disagreement if they are not. Bitcoin's mechanism to do so — the Bitcoin *consensus protocol* — is the most ingenious part of Nakamoto's design for Bitcoin and is in some ways the most interesting and influential thing about it.

In brief, the Bitcoin consensus protocol asks participants (called "miners") to accept any valid new block of transactions that one miner proposes — but it makes the process of generating a valid new block onerous and unpredictable. (The difficulty is regularly adjusted so that the entire community of miners can on average generate only one new block every ten minutes.) When a miner broadcasts the block to other miners, they examine it, confirm that it satisfies the consistency constraints, and then add it to the

---

[18] This is possible because with modern public-key encryption, other participants can verify that a message was properly signed by the private key holder even though they do not themselves have the private key.

[19] The redundancy also means that blockchains are practically impervious to hardware errors: any idiosyncratic faulty execution on one participant's computer will be massively outvoted by the collectivity of participants whose computers did not malfunction. Thus, in what follows, I will ignore the philosophical objection that no program can guarantee that it runs correctly on actual hardware. *See, e.g.*, James H. Fetzer, *Program Verification: The Very Idea*, 37 COMM. OF THE ACM 1048, 1059–60 (1988). When it comes to hardware faults, the objection is ontologically impeccable but practically irrelevant in this context. The more relevant objection, as I argue below, is that no program can guarantee that *people* will run it as intended.

current blockchain. Then the process begins anew to generate the next block.

Incentives are needed to make miners generate and accept blocks. The incentives to do the computational work of searching for valid blocks come from a "block reward" of new Bitcoin for each block they successfully generate, and from "transaction fees" paid by users to add their transactions to a block. Their incentives to accept valid blocks proposed by other miners come from the value of consensus itself: new blocks can only be added to what everyone else agrees is the current end of the chain. So a miner who ignores a valid block is cutting herself off from future mining rewards: any blocks she generates will not be at the end of the chain. In equilibrium, the dominant individual strategy for individual miners is to accept any valid new block and immediately start trying to trying to generate a block that follows it.

C. Smart Contracts on a Blockchain

Now let us consider how to put smart contracts on a blockchain. The basic idea is simple. There is still a ledger of transactions, maintained in the same way as the Bitcoin blockchain. The difference is that these transactions are richer: they can create and execute computer programs, not just transfer resources.

These programs run on a *virtual machine*. As the name implies, it executes instructions like an actual computer, but it is entirely simulated. The Ethereum blockchain, for example, implements the Ethereum Virtual Machine (EVM). One kind of transaction on the Ethereum blockchain simply transfers its native currency unit — called "Ether" — from one use to another. Another kind of transaction takes a program written in the EVM's native language ("EVM bytecode") and runs it on the EVM.

This last sentence is deceptively simple, so it is worth unpacking. The EVM is a simulated computer. It functions according to rules described in the Ethereum protocol[20] — that is, each participant on the blockchain independently applies those rules to each new transaction and confirms that they yield the same result. The consensus protocol ensures that each user observes

[20] *See generally* Gavin Wood, *Ethereum: A Secure Distributed Generalised Transaction Ledger*; ANDREAS M. ANTONOPOULOS & GAVIN WOOD, MASTERING ETHEREUM: BUILDING SMART CONTRACTS AND DAPPS (1 edition ed. 2018).

the same transfer and program transactions. Thus, just as the participants agree on each user's current balance of Ether because they agree on how each transfer transaction changes those balances, they agree on the EVM's current state because they agree on how each program transaction changes the EVM. The rules are significantly more complicated (though far less complicated than the circuits in a typical physical computer), but they are deterministic.

There are a few more details worth noting. First, EVM bytecode includes instructions for programs to send and receive Ether. A program can transact with users (or with other programs) by executing these instructions. Second, programs can be persistent: one user can load a program into the EVM with an initial transaction, and other users can then interact with it in subsequent transactions (if and how its code allows, that is). Together, these features enable smart contracts: I can offer you a smart contract by loading its terms into the EVM, and you can accept by sending it an appropriate transaction. Third, these program transactions are not free. Ethereum has a complicated metering scheme in which programs consume a resource called "gas" as they run: users must pay (with Ether) for enough gas for the programs they run. The design is both clever and ambitious.

II. AMBIGUITY

It might be hoped that this approach to putting smart contracts on a blockchain solves the three problems with legal contracts identified above. The smart contracts are unambiguous because they are written in programming languages. The smart contracts are incorruptible because control of the blockchain is widely distributed. And enforcement is automatic because the smart contract directly controls resources on the blockchain. I believe these hopes are overstated.

A. Ambiguity in Natural Languages

Consider a famous example of the ambiguity of natural language. In *Frigaliment Importing Co. v. BNS International Sales Corp.*, the parties disagreed on the meaning of "chicken."[21] Their contract called for the delivery of

[21] Frigaliment Importing Co., Ltd., v. BNS International Sales Corp., 190 F. Supp. 116, 117 (S.D.N.Y. 1960).

100,000 pounds of "'"US Fresh Frozen Chicken, Grade A, Government Inspected, Eviscerated." The buyer thought that "chicken" meant "a young chicken, suitable for broiling and frying."[22] The seller thought it meant "any bird of that genus."[23] The court considered dictionary definitions, the text of the contract, the parties' negotiations (in a mixture of English and German), evidence of trade usage in the chicken-evisceration industry, USDA inspection standards, and prevailing market prices, only to conclude that there was evidence on both sides, so the plaintiff had failed to carry its burden of "showing that 'chicken' was used in the narrower rather than in the broader sense."[24] In short, "chicken" was ambiguous.

The parties in *Frigaliment* could have prevented their particular dispute if they had written "young chicken suitable for broiling."[25] But that would just have raised further ambiguities in other cases. What counts as "suitable for broiling?" Suitable for broiling at 500 degrees Fahrenheit? 550? For how long? Ambiguity always remains.

The problem is inherent in the nature of natural language, because natural language is inherently social. The meaning of a text is not the (single) meaning its author intended, but the (possibly different and possibly plural) meanings it has to the relevant community of readers. Even the meanings given in "objective" sources like dictionaries — putting aside all of the interpretive problems of how to read those sources — depend on how people actually use words. And since the legal effect of a contract is determined by the interpretation of its terms, the meaning of a contract irreducibly social.

B. Ambiguity in Programming Languages

To repeat, the meaning of "chicken" is a socially contingent fact. It depends on how people actually use the word in the world. Its meaning can vary and be misunderstood.

It might be argued, however, that the meaning of an expression in a programming language is a technical fact rather than a socially contingent fact. `2**3` in Python will always evaluate to `8`. Its meaning never changes,

---

[22] *Id.*

[23] *Id.*

[24] *Id.* at 121.

[25] Or "any bird of the genus *gallus gallus domesticus*" if they had settled on the seller's preferred meaning rather than the buyer's.

and if you think it means `9` you are wrong. Meanings that depend on socially contingent facts can be ambiguous, but meanings that depend on technical facts cannot.

This account is wrong. It is true that competent programmers in a given language will agree on a program's meaning (at least for simple programs). And their agreement does depend on technical facts about the language that are independent of particular programmers' idiosyncratic beliefs. But these technical facts are still social, just at a deeper level.

In a nutshell, no computer program can determine its own semantics. The program may have a fixed, objective syntax. But the act of giving meaning to that syntax — whether by talking about the program or by running it — requires something outside the program itself. Any strategy for doing so ultimately depends on social processes.

More specifically, there are three general strategies for giving a fixed meaning to programs in a given language: informal specifications, formal semantics, and reference implementations:

- A *specification* describes in natural language how programs behave. The Python reference manual says that `**` "yields its left argument raised to the power of its right argument."[26]
- A *formal semantics* for a language describes programs in that language in terms of abstract mathematical entities and operations on them. For example, here is the fragment of the EVM standard that defines the exponentiation operator:[27]

  `0x0a EXP 2 1` $\mu'_s[0] \equiv \mu_s[0]^{\mu_s[1]}$

- A *reference implementation* of a language is an interpreter for the language whose behavior is stipulated to be correct. The most common implementation of Python, called CPython, isn't officially a reference implementation, but it is widely adopted enough that it might

[26] Guido van Rossum, *The Python Language Reference* 77.

[27] Wood, *supra* note 21, at 28. For an even more rigorous version, see Everett Hildenbrandt et al., *KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine*, 2018 IEEE 31ST COMPUTER SECURITY FOUNDATIONS SYMPOSIUM (CSF) 204 (IEEE 2018). There are formal semantics for Python, *see, e.g.*, Joe Gibbs Politz et al., *Python: The Full Monty*, ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS 217 (ACM Press 2013)., but they're unofficial and less photogenic.

as well be one.[28] Any Python implementation whose behavior differs from CPython's is *prima facie* wrong. Here is what CPython does with `**`:

```
>>> 2**3
8
```

All three of these strategies have in common that they define the meaning of a program created by humans in terms of *something else also created by humans*. So the meaning of any specific program rests on a foundation of some prior agreement about how to interpret some larger class of programs. Specifications, formal semantics, and reference implementations are not authoritative as a matter of first principles; they are authoritative because people agreed that they are. Why doesn't `2**3` in Python evaluate to `9`? Not because that's what `2**3` inherently means — any more than the seven-letter sequence C-H-I-C-K-E-N inherently means any *gallus gallus domesticus*. In 1991, Guido van Rossum selected `**` as an exponentiation operator for Python and defined its behavior. He could have used `^` instead and made `**` a multiplication operator. If he had, than `2**3` would evaluate to `6`.

But, one might ask, isn't it a logical necessity that $2^3=8$? As long as the Python specification defines `x**y` as x^y, don't the laws of mathematics require that it evaluate to `8` in any correct implementation of Python? There is something to this point, which serves as the foundation of the field of *program verification*: rigorous standards of proof and truth can be applied to mathematical models of programs. Given a formal semantics of a programming language and a precise specification of a program's operating requirements, it is sometimes possible to produce a logically valid proof that the actual program correctly implements the specification.[29] But there is a crucial step missing: no formal proof is possible that the specification itself corresponds to anything in the outside world.[30] Change the language semantics

---

[28] *See Alternative Python Implementations*, https://www.python.org/download/alternatives/ (calling CPython "the 'traditional' implementation of Python).

[29] *See generally* MACKENZIE, *supra* note 2 (describing history of controversies over program verification).

[30] Brian Cantwell Smith, *Limits of Correctness in Computers*, *in* PROGRAM VERIFICATION: FUNDAMENTAL ISSUES IN COMPUTER SCIENCE 275 (Timothy R. Colburn et al. eds., 1993).

and all you are left with is an incorrect program and an invalid "proof" of its correctness.

Here is another way of appreciating the point. Consider the Python expression `3/2`. What will happen if you evaluate it? It depends. If you run it in Python version 2.7.15, where `/` is an integer division operator, it will return `1`. But if you run it in Python version 3.7.1, where `/` is an exact division operator (and `//` is the integer division operator), it will return `1.5`. "Python" is not one thing. What we mean when we say "Python" is socially determined.[31] Under some circumstances, we mean Python 2.7.15; under others we mean Python 3.7.1. If we mean Python 2.7.15, then when we say "the value of the Python expression `3/2`" we refer to `1`, but if we mean Python 3.7.1, when we say "the value of the Python expression `3/2`" we refer to `1.5`. The value of the expression is unambiguous relative to a specific programming language, but that is like saying that the meaning of "chicken" is unambiguous relative to an interpretive convention in which it means any *gallus gallus domesticus*. All the important work is done by the claim that *this* program is written in *that* language. Such claims can only be established by reference to a community of programmers and users.

`2**3` in Python is unambiguously `8`, but that is only because Python users have already agreed on what "Python" is. If they agreed differently, "Python" would be different and so might `2**3`. Collective negotiation over the agreed meaning of "Python" is constantly taking place: in particular, it happens every time there is a new version release. Among other changes, Python 3.7 added a new function called `breakpoint`, but it also made `await` a reserved keyword.[32] Programs that call the `breakpoint` function work in Python 3.7 but not in Python 3.6; programs with a variable named `await` work in Python 3.6 but not in Python 3.7. These changes are debated at immense length on Python developer mailing lists, and each time there is a new release, everyone who is responsible for writing or running Python code decides whether or not to upgrade their version to the latest one. These choices collectively establish the meanings of Python programs — and change those meanings over time. Technical facts depend on socially determined ones.

---

[31] So for that matter is what we mean when we say "Python 2.7.15."

[32] What's New In Python 3.7 — Python 3.7.2 documentation, https://docs.python.org/3.7/whatsnew/3.7.html.

More precisely, we perceive as fixed technical facts the successful result of coming to social consensus on programming language semantics. A community of programmers and users agrees on a process to extract technical meaning from program text. Developers implement that process on different computers, with different tools, in different contexts. Most of the time, running a program on different implementations will yield the same result. When it does not, technical meaning breaks down.

III. AMBIGUITY IN SMART CONTRACTS

Back to blockchains. We might be able to ignore all of this if smart-contract blockchains never experienced breakdown.[33] But in fact there are difficulties about the meanings of blockchain programs all the time. I will present four examples, in increasingly dire order.

A. Oracles

How does a smart contract observe the world? Suppose, for example, that it needs to release funds from escrow when the seller has delivered a car. The car is a real thing in the real world, not a virtual thing defined by the blockchain VM. The smart contract cannot directly observe it.

The standard solution is to rely on an *oracle* to input real-world data in a form usable by a smart contract.[34] The simplest version of an oracle is simply a trusted user, who is asked to commit transactions verifying that a given event did or not take place, and perhaps supplying some details. This is basically a smart-contract version of an arbitrator or third-party certification. The next step up in complexity is to use a trusted data feed. Trusted software on the blockchain consults some online but off-blockchain data source — like a major financial website's stock quotations — and enters it into the blockchain.[35] The most sophisticated form of oracle is a consensus oracle: a group of users serve as oracles and the software extracts whatever value they

---

[33] *See* TERRY WINOGRAD & FERNANDO FLORES, UNDERSTANDING COMPUTERS AND COGNITION: A NEW FOUNDATION FOR DESIGN (1987) (applying Heiddeger's concept of breakdown to the skew between computer models of the world and the world itself).s

[34] *See* ANTONOPOULOS & WOOD, *supra* note 21, at 253–66.

[35] For a sophisticated authenticated data feed solution, see Fan Zhang et al., *Town Crier: An Authenticated Data Feed for Smart Contracts* 270 (2016)..

have agreed on. Even simple majority voting can make it harder to corrupt enough involved users to trick a given smart contract, and some consensus oracles use their own consensus protocols, in which the users are rewarded for their participation and for reaching agreement

Does the oracle's conclusion bear any relationship to reality? We might describe this as a problem of corruption: an oracle that says the car was delivered when it was not is mistaken or lying in the way that a bad judge will be mistaken or lying about a legal contract. Consensus oracles, following the standard blockchain mantra of decentralization, seek to limit corruption by using protocols that encourage correct agreement among the parties. But of course the oracle software has no unmediated access to the truth in the world. Instead, the best its protocols can do is encourage parties to agree — in the hopes that truth will be a more salient focal point than a lie, and that long-term incentives will lead parties to select honest oracles.

The problem of observing the world is also a problem of ambiguity. The world is complex, and contract terms map ambiguously onto the world. An oracle is a way of resolving the ambiguity in how a contract term applies to the infinite variety of factual patterns that could happen in the world. An oracle charged with determining whether the seller in *Frigalament* has performed its obligations resolves any ambiguity about the meaning of "chicken." If the oracle says the seller has performed, then what was delivered was "chicken." If the oracle says the seller has not performed as required, than whatever was delivered was not "chicken."[36]

An oracle's consensus protocol, then, is crucial to how it operates. Single-user oracles and trusted data feeds have simple trust models and consensus protocols; consensus oracles have more sophisticated ones. This leads to two points. The obvious one is that an oracle's resistance to corruption is only as good as its consensus mechanism. The subtler one is that an oracle's ability to resolve ambiguity is only as good as its consensus mechanism.

[36] Allen, *supra* note 6, at draft 24.

B. Upgrades

Blockchains also upgrade. In 2017, Bitcoin upgraded to implement "segregated witness" (also known as "SegWit").[37] Some data in transactions was moved from one portion of the block to another in a way that effectively increased the number of transactions that could fit in each block. The blockchain before SegWit and the blockchain after had different semantics.

Actually, I'm hiding the ball by saying that "Bitcoin upgraded." Blockchains don't upgrade themselves; people upgrade blockchains. Bitcoin's users collectively acted to modify Bitcoin's semantics in ways that would invalidate some transactions. A critical mass of miners announced their support for SegWit, and then on the agreed-upon date started enforcing the new rules. Everyone else went along for the ride. It was just like switching from Python 3.6 to Python 3.7, except that with a blockchain the pressure for consensus is much stronger. Today you can easily find users still happily running Python 3.6, but you will not easily find Bitcoin miners ignoring SegWit.

It's consensus all the way down. The "Bitcoin blockchain" exists only because people agree that it does and what it is. Bitcoin's consensus protocols help coordinate that agreement; indeed, they incentivize it. But the protocols themselves cannot establish their own rule of recognition. A user community can always collectively change or ignore them. This is exactly what happens in an upgrade.

C. Forks

Upgrades don't always go smoothly. SegWit was intended (by some users at least) as the first of two linked upgrades to increase Bitcoin's capacity. Following the SegWit upgrade, according to a widely reported-on compromise among various Bitcoin developers, Bitcoin was also supposed to increase its block size from 1 megabyte to 8 megabytes, octupling the number of transactions it could process per block.

This … didn't happen.[38] Instead, following the SegWit upgrade, some miners announced they were against the block size upgrade, while others

---

[37] Timothy B. Lee, *Bitcoin compromise collapses, leaving future growth in doubt*, Ars Technica (Nov. 9, 2017), https://arstechnica.com/tech-policy/2017/11/bitcoin-compromise-collapses-leaving-future-growth-in-doubt/.

[38] *Id.*

announced they were for it. Discussions and negotiations broke down, and Bitcoin forked into *two blockchains*.[39] One of these blockchains, now known as Bitcoin Cash, increased its block size to 8 megabytes (and then increased it again to 32 megabytes, having established the principle that the block size should grow as needed). The other blockchain, now known as Bitcoin, still has roughly 1 megabyte blocks.[40] The blockchains recognize the same history up until the first >1 megabyte block on Bitcoin Cash, after which they diverge.

Bitcoin and Bitcoin Cash now have different semantics. Is a block valid? The question is unanswerable in the abstract. It can only be answered with reference to a particular blockchain and its user community. A 32-megabyte block is valid according to the agreed-upon semantics of the Bitcoin Cash community, but not according to the Bitcoin community. (It should be obvious that which of them ends up with the "Bitcoin" name is a socially determined fact.)

Blockchain forks are consensus failures. Each blockchain by itself achieves local consensus, but there is no global consensus. Blockchain forks also create explicit ambiguity. The choice of blockchain exposes ambiguity not present when looking at each blockchain by itself. These two facts are inextricably linked, because it is consensus that resolves ambiguity on a blockchain.

Literally anything on a blockchain is subject to the latent ambiguity that the blockchain itself could be upgraded out from underneath it. Whether this actually happens is inescapably political. When there is a disagreement within a blockchain community about a particular upgrade, one of three things could happen. If the pro-upgrade faction backs down, the status quo prevails. If the anti-upgrade faction backs down, the upgrade happens. If neither faction backs down, the blockchain forks. (It should be obvious that which faction, if either, backs down, is an empirical and socially determined fact.)

---

[39] Benito Arrufiada, *Blockchain's Struggle to Deliver Impersonal Exchange*, 19 MINN. J.L. SCI. & TECH. 55, 73–75 (2018).

[40] I say "roughly" because SegWit complicated the formula for computing block size.

D. The DAO

The DAO — the initialism is short for "distributed autonomous organization" — was a kind of democratic online venture-capital fund.[41] A group of investors planned to join together by using a smart contract on the Ethereum blockchain to manage their affairs, rather than by forming a traditional business organization under the laws of a state. One (imperfect) analogy would be to a venture capital fund operated as a general partnership with all of the participants voting on each funding decision.[42]

In the event, it flamed out spectacularly.[43] A clever but still unidentified Ethereum user discovered a subtle bug in The DAO contract's code and was able to transfer about $60 million worth of Ether to a contract that she alone controlled.[44]

The transfers were quickly noticed, leading to a sharp debate among The DAO and Ethereum users over how to respond.[45] In the end, a large majority of Ethereum users upgraded Ethereum to recognize as valid a new special block with a transaction that unwound The DAO and returned all the funds in it to the original investors. On this blockchain, which is still known as Ethereum, The DAO and The DAO hack effectively never happened. A minority of users refused to recognize the special block because they considered it contrary to what they saw as the spirit of smart contracts, blockchains, and Ethereum.[46] On this blockchain, which is known as Ethereum Classic, The

---

[41] *See generally* Carla L. Reyes et al., *Distributed Governance*, 59 WM. & MARY L. REV. ONLINE 1 (2017)..

[42] Christoph Jentzsch, *Decentralized Autonomous Organization to Automate Governance*.

[43] Matt Levine, *Blockchain Company's Smart Contracts Were Dumb*, BLOOMBERG.COM, Jun. 17, 2016, https://www.bloomberg.com/opinion/articles/2016-06-17/blockchain-company-s-smart-contracts-were-dumb.

[44] Nathaniel Popper, *A Hacking of More Than $50 Million Dashes Hopes in the World of Virtual Currency*, N.Y. TIMES, Jun. 17, 2016, https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html.. For technical details, see Phil Daian, *Analysis of the DAO exploit*, Hacking Distributed (Jun. 18, 2016), http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[45] Joon Ian Wong & Ian Kar, *Everything you need to know about the Ethereum "hard fork,"* Quartz (Jul. 18, 2016), https://qz.com/730004/everything-you-need-to-know-about-the-ethereum-hard-fork/.

[46] The Ethereum Classic Declaration of Independence, https://ethereumclassic.github.io/assets/ETC_Declaration_of_Independence.pdf..

DAO and The DAO hack did happen. The two blockchains have different semantics. Indeed, they are incompatible. Transactions now can be entered on the Ethereum blockchain and conform to its views of which transactions have happened (including The DAO, the hack, and the rollback) or on the Ethereum Classic blockchain and conform to its views (including The DAO and the hack but not the rollback).[47]

The DAO was also (purportedly) governed by a legal contract, although its main job was to defer as much as possible to the smart contract. It stated:

> The terms of The DAO Creation are set forth in the smart contract code existing on the Ethereum blockchain at `0xbb9bc244d798123fde783fcc1c72d3bb8c189413`. Nothing in this explanation of terms or in any other document or communication may modify or add any additional obligations or guarantees beyond those set forth in The DAO's code.[48]

In hindsight, this passage is ambiguous. The phrase "the Ethereum blockchain" does not uniquely refer. Does it mean Ethereum or Ethereum Classic?[49] It uniquely referred when the contract was drafted, but no longer. It became ambiguous — just like any reference to a blockchain could, at any time.

CONCLUSION

We began with three motivations for smart contracts: ambiguity, corruption, and enforcement. It is obvious that protocol changes, forks, 51% attacks, and other consensus breakdowns are a kind of corruption threat to smart contracts. They subject smart contracts to abrogation or alteration at the whims of other blockchain users.[50] It is also obvious that the difficulty

---

[47] Aaron van Wirdum, *Ethereum Classic Community Navigates a Distinct Path to the Future*, Bitcoin Magazine, https://bitcoinmagazine.com/articles/ethereum-classic-community-navigates-a-distinct-path-to-the-future-1471620464/..

[48] The DAO - Explanation of Terms and Disclaimer< (Aug. 3, 2016), https://web.archive.org/web/20160803111447/https://daohub.org/explainer.html.

[49] It should be obvious that the social fact that one of the blockchains is commonly called "Ethereum" and the other is not is relevant but not conclusive in resolving this ambiguity.

[50] As I write this, Ethereum Classic was subjected to a $500,000 double-spending attack based on a well-executed deep by users who temporarily dominated its mining power.

of getting people to use a blockchain at all is an enforcement threat. It doesn't matter what a smart contract controlling asset-title tokens on a blockchain says if no one in the physical world pays any attention to the blockchain.

My argument in this essay is that we should also understand the problem of consensus as an ambiguity threat. Natural languages are embedded in communities of people who use and understand those languages. This introduces ambiguity and uncertainty, because people may use and understand the same words in different ways. But it also provides a backstop on how badly natural-language contracts can fail. In many cases, the meaning of a contract is clear to a large fraction of people in the relevant linguistic community. If a contract isn't worth the paper it's printed on, it is because of corruption or enforcement problems, not because of ambiguity.

Programming languages appear to reduce linguistic ambiguity. In many cases, they do. Relative to a given implementation, a computer program's meaning is far more definite than a typical natural-language term's. Indeed, the very process of reducing a term to a formal-language expression requires a degree of precision from its drafters that can itself force them to understand and express their intentions more clearly.

But because programming languages are formal, constructed systems, when the bottom drops out, it can really drop out. The relevant community can redefine the programming language in a way that radically alters the meaning of programs written in it. Smart contracts on a blockchain are particularly vulnerable to this. First, the same consensus mechanism that keeps them in a local equilibrium can lock them quickly into a new and very different equilibrium; second, there are often powerful incentives for users to push the blockchain into a different equilibrium. Blockchain-based smart-contract programming languages don't have continual linguistic drift; they have occasional earthquakes.

In a legal system, the way to change the consequences of contracts is to *change the law*. The natural-language terms in legal contracts still mean what they used to, but their legal effects are different. But on a blockchain, the way to change the consequences of contracts is to *change the semantics*. The

---

Dan Goodin, *Almost $500,000 in Ethereum Classic coin stolen by forking its blockchain*, Ars Technica (Jan. 8, 2019), https://arstechnica.com/information-technology/2019/01/almost-500000-in-ethereum-coin-stolen-by-forking-its-blockchain/.

programming-language terms in smart contracts mean something different than they used to, and they have different technical effects, and these two differences are the same thing. Interpretation and construction collapse.[51]

This is neither the first nor the last word on ambiguity in smart contracts. I have argued the narrow point that perfect unambiguity is impossible even in theory, because the technical layer ultimately rests on a social one.[52] There is a complementary and broader critique of smart contracts — spelled out best in papers by Karen Levy,[53] Jeremy Sklaroff,[54] and Kevin Werbach and Nicholas Cornell[55] —that even where they do provide unambiguous incorruptible automatic enforcement, this may not be what contracting parties want or need. Writing code is hard, and debugging it is even harder: one advantage of vague and ambiguous natural language is that it is cheaper and faster to negotiate and write down. And sometimes flexibility is good. As Levy explains of legal contracts,

> As such, it can be both operationally and socially beneficial to leave some terms underspecified; vagueness preserves operational flexibility for parties to deal with newly arising circumstances after an agreement is made, and sets the stage for social stability in an ongoing relationship.[56]

---

[51] Lawrence B. Solum, *The Interpretation-Construction Distinction*, 27 CONST. COMMENT. 95 (2010). Note that in a *legal* contract incorporating a formal-language term there is still room for construction. As I have argued, these terms are not ambiguous relative to a given formal language; they are ambiguous when there are multiple plausible formal languages in which they could be interpreted and the court (or other legal actor) must select among them. The court might also decide that a term's meaning is clear but nonetheless disregard it for any of the reasons it might disregard a natural-language term. *See, e.g.*, Levine, *supra* note 44.

[52] This is hardly unique to smart contracts or to blockchains. It is a general characteristic of social software. *See* James Grimmelmann, *Anarchy, Status Updates, and Utopia*, 35 PACE L. REV. 135 (2015).

[53] Karen E.C. Levy, *Book-Smart, Not Street-Smart: Blockchain-Based Smart Contracts and The Social Workings of Law*, 3 ENGAGING SCIENCE, TECHNOLOGY, AND SOCIETY 1 (2017)..

[54] Jeremy M. Sklaroff, *Smart Contracts and the Cost of Inflexibility*, 166 UNIVERSITY OF PENNSYLVANIA LAW REVIEW 263 (2017)..

[55] Kevin Werbach & Nicolas Cornell, *Contracts Ex Machina*, 67 DUKE L.J. 70 (2017)..

[56] Levy, *supra* note 54, at 8.. An interesting line of research involves trying to write more deliberately flexible smart contracts. *See, e.g.*, Bill Marino & Ari Juels, Setting Standards for Altering and Undoing Smart Contracts, presented at RuleML 2016.

And this is to say nothing of the use of smart contracts for socially harmful purposes,[57] the environmental costs of blockchain mining, or the recent blockchain investment bubble.[58]

All is not lost for the smart-contract project. Smart contracts cannot be perfectly unambiguous. But they do not need to be perfect to be useful. Socially determined facts are empirically contingent: they are always open to contestation and change. But legal contracts also depend on socially determined facts, and this has not stopped them from having an extremely successful multi-thousand-year run. Much of the time, legal contracts work well enough, despite the ambiguities of natural language. If smart contracts can do as well or better in even a single domain, they will have a worthwhile role to play.

For better and for worse, blockchains make consensus explicit. The mechanism that holds a blockchain together is the process for agreeing on the next block. Whatever that process yields — in all of its technical and social complexity — is the next block. Every smart contract is therefore only as resilient as its underlying blockchain. Contract law depends on social institutions, particularly those that establish and limit the governments which enforce contracts. Smart contracts depend on social institutions too, particularly those that establish and limit blockchain communities. A blockchain whose governance fails will collapse, fork, or be vulnerable to hijacking. All of these threaten the smart contracts that run on it. There is no escape from politics.[59]

What, then, does good blockchain citizenship demand? Smart contracts that are unambiguous enough to be usable require correct and stable blockchains. These properties are only possible with a good blockchain community, one that has the wisdom and will to preserve correctness by making

---

[57] Ari Juels et al., *The Ring of Gyges: Investigating the Future of Criminal Smart Contracts*, PROC. ACM CONF. COMMPUTER AND COMMUNICATIONS SECURITY 283 (ACM Press 2016).

[58] *See, e.g.*, Shaanan Cohney et al., *Coin-Operated Capitalism*, COLUM. L. REV. (forthcoming) (identifying lack of investor protections in numerous smart contracts).

[59] Curtis Yarvin, *The DAO as a Lesson in Decentralized Governance*, Urbit.org (Jun. 24, 2016), https://urbit.org/blog/dao/; Steve Randy Waldman, *A Parliament Without a Parliamentarian*, Interfluidity (Jun. 19, 2016), https://www.interfluidity.com/v2/6581.html; Grimmelmann, *supra* note 53.

good and necessary changes, and to preserve stability by not making bad and unnecessary changes.

Very little of this is to be found in the consensus protocols: those are just one part of a rich institutional support system. What makes a blockchain work are the human institutions behind it: the deep developer community, the mailing lists where they work through changes and controversies, the users with a long-term commitment to the blockchain's health, the platforms that make the blockchain usable by non-experts, and so on. Blockchains are made out of people.