Truth, Honesty, Geeq, and the Importance of Declarative Programming

The Problem

What is Bitcoin? Is it Satoshi's 2008 paper? Is it the program (called a client) that Bitcoin mining nodes run? The truth is, no one really knows, and at best, it is a moving target.[1]

Satoshi's paper describes a consensus mechanism, but does not specify how it should be implemented. It is more of a vision or an intention to be instantiated in code by developers and then run by the validation network.

Bitcoin is not "the code" run by nodes either for at least two reasons. First, there are more than ten Bitcoin clients currently in use written in different programming languages, for different operating systems, and optimized for different chip architectures. Second, all of these clients are patched and updated continuously. None of these versions defines what Bitcoin is. If they disagree or process transactions differently, none of them can claim that they are definitive and correct in any abstract or absolute sense.

Each of these client versions is really an interpretation of Satoshi's intention, or more accurately, what the majority of nodes agree is the best evolution of his intention. In computer science, programing code is understood to be an "imperative" instantiation of a way of doing something. That is, the code is a way of doing something, but does not in itself define what should be done. Client code does not, and is not intended to, define what Bitcoin is. More to the point, Bitcoin code is not law. How could it be? It changes almost daily.

Even if we agreed that one version of Bitcoin Core (the most popular client) defines exactly what Bitcoin is, we would still be building on sand. This is because Bitcoin depends on external code libraries to do things like encryption, signature checking, hashing, and communications. Not all of these are written and maintained by Bitcoin developers It is not clear that the existence of a privileged, centralized group with such responsibility and authority would be consistent with Bitcoin's philosophical foundation in any event. Instead, these libraries are maintained, patched, and upgraded by independent developers of all kinds.

This creates two obvious problems for projects like Bitcoin that aspire to be trustless and immutable. First, as these libraries change over time, they do different things. Even if some version of Bitcoin's code-base were to be frozen, its dependencies would not be. Second, these dependencies create a significant attack vector. One of these repositories could accidentally or intentionally deploy an update with a bug that could damage client functions or make them vulnerable to adversaries. In

1 Bitcoin is only chosen as an example because of its importance and profoundly innovative contribution. Essentially all coding projects are imperative in nature. The central thesis of this note is that imperative coding has uniquely troubling implications in blockchain space.

effect, the Bitcoin community must trust in the integrity and the competence of Bitcoin Core developers, dependency maintainers, and miners who choose which client to run.

What to do

"Code is Law" is a wonderful goal, but it is simply unattainable using conventional approaches. Technical papers are abstractions that express intentions but are neither specifications nor definitions of truth or function. Imperative code does what it does, but there is no standard that allows us to know if what it does is correct. We can't trust what we can't verify, and we can't verify what has never been defined. Clearly, what we need is a definition of what a protocol is supposed to do that can be checked against what a client actually does. Computer scientists refer to this as "declarative" programing.

Blockchain is data and data are bytes. To say a blockchain is correct is to make a declarative statement that its bytes satisfy certain logical relationships to one another. A transaction is formally be defined to be a byte string of a specific length, in a specific format, with specific subsets of these bytes being interpreted as account addresses, signatures, header data, and so on. For a transaction to be "correct" or valid, it must stand in certain well-defined relations to other data in the chain.

For example, the byte array representing the sending address must correspond to a byte array in the current ledger state representing the same account address; byte arrays representing transaction amounts and account balances also must stand in a certain, specific relation; a block must include only valid transactions; and the new ledger state must follow specified accounting rules that take the initial account balances and the transaction amounts as arguments.

This may sound similar to "unit testing", but in fact, a declarative specification builds a fundamentally different foundation. Unit testing requires developing "code criteria, or results that are *known to be good*, (and building them) into the test to verify the unit's correctness." In other words, unit testing involves choosing inputs from some domain of possibilities and then deciding that the outputs are what the developer thinks they should be.

A declarative statement of a program or protocol is a code-independent definition of what a state of a data structure must satisfy in order to be correct. How a state came into existence and how it is updated is irrelevant. Unit testing is opaque to users who can't know what criterion the developer chose for "good" results, nor how carefully or completely the tests were conducted. With a declarative statement, on the other hand, users are able independently verify that a blockchain, ledger, or any other data set, is correct without having to trust anyone but themselves.

Nodes, of course, must use client software to validate and maintain the chain. Without a fixed and complete definition of correctness, however, users can never know if the imperative instantiation of code used by a node really does what the protocol is supposed to, even when the code is "open source". Declarative code is a necessary foundation for any project that ultimately seeks not to rely on trust.

Summary and an Example

1. Declarative code is a definition of truth and correctness. There is no such objective standard when one relies solely on technical papers and imperative code implementations.

2. The declarative code never changes. Logic does not have bugs, as such, does not depend on operating systems, chip architecture, or external code libraries. It does not need to be patched or upgraded.

3. Nodes in a validation network run imperative implementations of the declarative specification. These will have bugs, need patches and upgrades, refer to external libraries, and so on. Nodes may run different clients or versions of clients as they choose. (Indeed, users can never really know what code nodes choose to run.) The key difference is that if any node, for any reason, presents an output that differs from the requirement of the declarative specification, then it is provably incorrect.

While this explanation may sound complicated, at root, it is very simple. As an example, addition is a well-defined operation. We all know that $2+2=4$ and nothing else is correct. In practice, numbers could be added together using a pencil and paper, on an abacus or a calculator, in a spreadsheet, by a computer, or even by counting fingers. In fact, we neither know nor care how our bank adds numbers together. If the bank claims anything besides $2+2=4$, however, we know, and can prove, that the bank did something wrong.

On the other hand, if we agreed that whatever number a calculator gave us was correct, we would have to trust that the calculator was bug free, working correctly, and had not been altered by someone who wanted to fool us. This is the basic difference between relying on declarative and imperative code.

Geeq and Proof of Honesty

A fundamental problem with Proof of Work, Proof of Stake, and other consensus protocols, is that there is no definition of truth. Even if one magically existed, a sufficient majority of the miners/stakeholders/nodes could simply agree to "validate" something that is clearly false or seems to violate the common understanding of correctness. Users can neither prove that such nodes are wrong, nor do would they have any meaningful recourse if they did.

Geeq uses a new consensus protocol called Proof of Honesty. In Geeq, there is no ability for a set of nodes to decide to accept (and impose on others) an alternative reality. Our goal is to empower users to protect themselves and so not need to trust in the honestly or correctness of nodes running black boxes of unknown software. Geeq's protocol is build on foundation of declarative code. If a node behaves dishonestly or even makes a mistake, it is detectable and provable by users.

It would not even matter if a majority of nodes tried to claim something false was true. Users will know, and can prove, that they are being lied to, and will refuse to accept as valid any incoming transactions on such a ledger. After all, who would accept stolen tokens when they know other users

will also be able to see that the ledger is invalid and so will rationally refuse to accept these tokens in the future? False ledgers are therefore ignored, and this means that there is no profit to nodes in creating them.

Geeq starts with declarative protocol specification which enables its Proof of Honesty consensus mechanism. In turn, this decentralizes its security guarantee to users at the edge where it belongs instead of in the center and reliant on the honesty of the majority of the nodes in the validation network.