Functional principles of contract design

With Ethereum, the blockchain concept at once entered a higher plane and a crisis mode. As a computationally complete smart contract system, Ethereum made it possible for many adversarial parties to engage in trustworthy interactions via a decentralized network. As an imperative programming language, the Ethereum VM is fundamentally designed so as almost optimally to frustrate efforts to exploit concurrency in contract execution. Its ultra-low-level nature as a virtual assembly language provides no guarantees, making the theoretically impossible problem of formal verification a hotly-sought necessity. The way forward must abandon these design decisions in favor of others with greater room for growth. I argue that the principles of functional programming are naturally suited to smart contract systems.

Contracts

We should be concerned primarily with clearly identifying what a parallelizable smart contract system *is*, and only secondarily with the machine representation of contracts. We begin with what a contract is.

LEGAL CONTRACTS

Contracts in the real world are of course dealt with extensively by lawyers and legal scholars, who have formulated a general characterization of them:

> *A **contract** is a voluntary arrangement between two or more parties that is enforceable by law as a binding legal agreement. Contract is a branch of the law of obligations in jurisdictions of the civil law tradition. Contract law concerns the rights and duties that arise from agreements. – Ryan, Fergus (2006). Round Hall nutshells Contract Law. Thomson Round Hall. p. 1. (taken from Wikipedia, "Contract")*

A contract therefore somehow connects *rights* and *duties* of the parties; the fulfillment of the contract consists of establishing these abstract rights and duties concretely. As for how this comes about:

> *One of the most important questions asked in contract theory is why contracts are enforced. –Wikipedia, "Contract"*

Fortunately, in a smart contract system this question can be answered easily: a smart contract enforces itself. That is, since the contract must be enforced, to really accept it is to accept the inexorability of fulfilling it, which is of course only possible if acceptance occurs only once the accepting party actually does their part. We take this, in fact, to be the very definition of a smart contract. This means that the actual fulfillment of the contract is part of the "offer and acceptance":

> *[An offer is] an expression of willingness to contract on certain terms, made with the intention that it shall become binding as soon as it is accepted by the person to whom it is addressed. – Treitel, GH. The Law of Contract (10th ed.). p. 8. (taken from Wikipedia, "Offer and acceptance")*

That is, a smart contract consists of an offer to connect rights and duties of various parties, which is accepted by the parties *as* the performance of these rights and duties, at which point the contract is simultaneously initiated and executed. The question of the validity of a contract is connected with (though not entirely defined by) the concept of *consideration*:

> ***Consideration** is the concept of value offered and accepted by people or organisations entering into contracts. Anything of value promised by one party to the other when making a contract can be treated as "consideration": for example, if A signs a contract to buy a car from B for $5,000, A's consideration is the $5,000, and B's consideration is the car. – Wikipedia, "Consideration"*

> *In common law it is a prerequisite that both parties offer consideration before a contract can be thought of as binding....If there is no element of consideration found, there is thus no contract formed. –Wikipedia, "Consideration"*

> *Generally, courts do not inquire whether the deal between two parties was monetarily fair–merely that each party passed some legal obligation or duty to the other party. –Wikipedia, "Consideration"*

Therefore, in at least one major legal system in the world (and in particular, that of my own country), each party in the contract must offer something of value. That a thing *has* value is determined, in this sense, as its being subject to legal obligation: a thing without limits is not valuable.

In short, a contract is:

- A commitment for some parties to provide rights to others, once the latter have performed duties for them;
- Enforceable by law once the offer by the former parties is accepted by the latter;
- Valid only if the rights and duties are valuable, in that they are constrained by law.

This is the concept that we will describe in the language of functions.

SMART CONTRACTS

As indicated above, a *smart contract* is an agreement whose context compels it to be honored once its duties are offered. It is, therefore, a function from duties to rights. Furthermore, these duties and rights may not be simple explicit values, but must be capable of *bearing value* that, in the same context as the contract, is subject to restriction. This situation resembles the concept of an *escrow:*

> *[An escrow is] a contractual arrangement in which a third party receives and disburses money or documents for the primary transacting parties, with the disbursement dependent on conditions agreed to by the transacting parties – Wikipedia, "Escrow"*

The parties in question here are: first, the one offering valuables; second, the one receiving them; and third, the context that holds and controls the actual value and its passing from one to the other. We will say, therefore, that a smart contract is a function from escrow-backed duties to escrow-backed rights, where "escrow" here means a hidden quantity connected with the representative tokens actually offered, without which those tokens are without value.

ESCROWS

The above legal characterization of an escrow suggests that an escrow-backed value is, itself, a contract: namely, a contract to provide some value in exchange for the duty of fulfilling its conditions. The two differ in their intended usage, though:

- A contract is offered by the seller, and its value as a fulfilled contract exists only for the buyer who accepts;
- An escrow is held by a passive party, and the seller forfeits the right to the contained value when the escrow is created. It may therefore *denote* that contained value, be exchanged between buyers, and otherwise function as a derived currency.

We will therefore separate the concepts by identifying escrows as those contracts that may be possessed by the (potential) buyer.

TRANSACTIONS

Legally, a transaction is an example of a contract; for instance:

> *A **financial transaction** is an agreement, or communication, carried out between a buyer and a seller to exchange an asset for payment. –Wikipedia, "Financial transaction"*

A transaction, though, is that which is "carried out", as opposed to simply promised, as in a contract. In executing a transaction, the buyer must demonstrate control of the payment as well as the seller demonstrating their offer to conduct such a transaction. This may entail fulfilling several contracts to obtain escrow-backed values as payment, as well as *contracting with the buyer* to engage in a transaction. This latter may seem redundant (a contract to create a contract?) but it reflects the reality that a seller may set different conditions for different buyers while still holding out a single offer to transact in general. A transaction is therefore a contract containing the fulfillment of other contracts.

We have therefore defined, in the end, just one concept, that of a smart contract, with several specializations that are important for giving it meaning legally and economically.

A functional smart contract system

We now concern ourselves with the construction of what we called, above, the “context” for a contract, namely, the system that enforces it.

LAZINESS

We desire that our smart contract system support the following property: an individual who seeks to learn the outcome of a transaction need only process the fulfillment of all contracts that are transitively fulfilled by that transaction. This means that contracts must remain *unevaluated* until their results are examined, a concept called *laziness*. Rather than dwell on how this may be achieved by a programming language, we simply assume that it *has* been achieved and can be used. This assumption is validated by the existence of, among others, the Haskell language.

STATEFULNESS

Contracts must be capable of maintaining an internal state marking the progress of the contract towards fulfillment. The format of the state is dependent on the contract, and so is the manner of progressing between states, except for the fact that this progression is sequential (rather than branching). The simplest formalism that describes these properties is to extend the notion of a contract: it must be a *coroutine* from escrow-backed duties to escrow-backed rights. A coroutine is simply a function that can be suspended, returning not only a value but also a continuation coroutine of the same type that may be resumed later. This allows the contract author to use any internal format for state, simply embedding the result state of each partial contract evaluation in the continuation to be used later. Progression of states is naturally sequential, as it is just a succession of function evaluations.

TRANSPARENCY

The interaction of laziness and statefulness presents a difficulty, namely that a transaction depending on a contract also depends implicitly on every previous transaction that also depends on the same contract. This is because each transaction causes a state update in that contract, and these updates must be sequential. Therefore *every* transaction must be examined to determine the dependency graph of any one transaction. If this examination necessarily entails *executing* the transactions, then laziness is lost; therefore, it cannot: the direct contract dependencies of any transaction must be identifiable without executing it, effectively being declared up front together with their input parameters. To rephrase: a transaction consists of two parts: first, a list of contract calls; and second, a contract accepting as its inputs the return values of those calls. The latter contract, now distinguished from a transaction by virtue of not possessing contract call declarations, may not call any other contracts.

However, it is acceptable for the body of the transaction to *create* new contracts, because the transaction must be executed anyway in order to determine the contents of the new contract before it may be called, so a dependency relation exists between them. So long as a contract's parent transaction can be determined from its identifier, that parent can be added to the dependency graph. By the same token, it is acceptable for *any* contract to create new contracts, which can always be traced back to their parents.

LAMBDA CALCULUS

As it stands, the smart contract system is unable to perform arbitrary transactions: forward instance, it is impossible to provide the output of one contract to another as input, precluding any kind of payment! However, the problem itself contains its own solution, because payment is an exchange of escrow-backed values, and escrows are the mechanism by which we may achieve economics. As described above, an escrow is simply a contract that may be possessed; the only entities in the system are contracts, and therefore each contract has as an implicit state a set of escrows that it owns. Ownership entails that only one contract at a time may ever cause a state update in the escrow, so the laziness/statefulness conflict disappears and it is possible for any contract to evaluate any of its escrows as a function, anywhere within itself. Recognizing this, we see that an economic transaction in the sense above, of *contracting with the buyer to transact*, should itself be an escrow-backed value that, as a function, conducts the transaction by accepting payment and depositing it with the seller. Here is an example in pseudocode:

```
transaction:
  call contracts:
    p <- call "pile of coins" with <some cryptographic credential>
    call "sell trinket" with () // Will sell to anyone, no credential required
  body:
    trinket <- call "cashier" with p // See "sell trinket"
    deposit trinket for <some cryptographic credential>

// Where
contract "pile of coins":
  expects: a cryptographic credential
  body:
    if (given cred differs from desired cred)
    then throw <error>
    else return <some coins>

contract "sell trinket":
  expects: // Nothing
  body:
    return escrow "cashier":
      expects: some coins
      body:
        if (given coins equal desired price)
        then
          deposit <given coins> for <some cryptographic credential>
          return trinket
        else throw <error>
```

In brief, the seller delegates the responsibility for collecting payment to an escrow that anyone can request. If the prospective buyer calls that escrow with sufficient funds, they get the contents and can keep them, in which case the escrow “deposits” the payment into a new contract signed over to the seller. This deposit contract would mirror the “pile of coins” contract from which the buyer took their own funds for the payment.

This “cashier” pattern suffers from two related exploits in which the prospective buyer vandalizes the seller:

1. By requesting to transact, thus collecting the “cashier” escrow, but without paying. If the buyer chooses to end the transaction then and there, the cashier is simply lost along with the trinket it contains.
2. As above, but depositing the “cashier” escrow into a contract keyed to the buyer, perhaps even innocently putting it aside “to pay later”. The trinket is not strictly lost, but the buyer is still denied it.

The former exploit is a clear violation of the nature of a transaction and we will show in the next section how it can be prevented by expanding the exception mechanism. The latter exploit is potentially valid, since one *should* be able to transfer escrow-backed values, but is a violation of the intentions of the seller in offering the “cashier”. To remedy it, we introduce a variant on escrows, *transactional escrows*, that may be returned only by their originator but not transferred subsequently. This forces a prospective buyer who might fall under exploit 2 to instead fall under exploit 1, hence the modifier “transactional”: the escrow contract must be fulfilled immediately.

This example demonstrates how escrows play the role of lambdas among smart contracts, but it also illustrates the necessity of breaking the pure functional paradigm with exceptions.

EXCEPTIONS

At any point during a contract's execution, the terms of the contract may be violated: a payment may be insufficient, or be offered in an unsupported currency, or, as described just above, not offered at all. If this happens, the contract execution must be terminated and its partial fulfillment reversed, so as to return assets already withdrawn to their owners. There should be at least the following two kinds of exceptions:

- User-created exceptions, like “throw <error>” in the pseudocode.
- System exceptions (arithmetic errors, errors of type representations, etc.)

Though both may be thrown anywhere, they should not be caught within a contract, lest a malicious user call a contract that would throw an exception but instead catch that exception, preventing rollback within that contract and, once again, vandalizing any resources that were partially spent. Any exception should render the entire current transaction void.

Understanding this, we see that exploit 1 from the previous section can be fixed by requiring the system to throw an exception whenever a transaction completes with any open escrows, representing resources that would be lost otherwise.

The Fae API

After the above lengthy discussion, we should summarize the resulting system from the perspective of a contract author. I like to call this API “Fae”, which might mean “functional alternative to Ethereum”, but also has the mythological connotation of a magical realm, deceptively similar to our own, but with its own rules, and which, once entered, is difficult if not impossible to leave.

Fae provides the following set of functions:

- `newContract`, `newEscrow`: these operate the same way, except that the former deposits function code as a new contract that may be called in a transaction, and the latter deposits function code as an escrow in the present contract, returning a unique identifier as the basic “escrow-backed value” corresponding to whatever sense of value resides in the new escrow.
- `useEscrow`: this calls a given escrow ID (possibly transactional) as a function.
- `spend`: within a contract (including escrows), returns an escrow-backed value, possibly transactional, but does not provide a continuation, so that the contract is “spent” (or the escrow “closed”) and removed from storage.
- `release`: as above, but *does* provide a continuation, namely, the code following the function call. The return value of one of these functions is the argument provided to the continuation (i.e. to the contract, when it is next called).

Aside from these functions, contracts may contain any other logic written in the host language.

Host language and safety

The system as described has made no mention of the programming language it uses. This is in contrast with Ethereum (and, to the extent that it is programmable, Bitcoin), which has a formally defined virtual machine with a corresponding assembly language. In the case of Ethereum, this is necessary because only by strictly tracking individual operations is it possible to measure and thus limit the use of computational resources. The system as described has also made no mention of this issue, which we address here.

CONTRACT FAILURE MODES

As a coroutine, a contract can fail in several ways: returning an incorrect value; constructing an incorrect continuation; throwing an exception; or failing to terminate. If one of these modes is the fault of the author, this will be apparent in the contract's source code and other authors can know, by scrutinizing it, not to trust this contract (and thus, not to call it). More insidiously, the contract may fail through a malicious call.   Since the only interface between the contract and the caller is its argument, the failure modes can be due to any of the following:

- **Partiality:** the contract as a function may not handle all possible values of its input type. This is a source code error, however, and therefore detectable by others.
- **Side effects:** if the argument has side effects, the contract may fail even if its overt value is handled correctly. This is a language issue: the host language must provide for the control of side effects.
- **Vulnerability:** the argument may itself be exceptional. If it literally throws an exception, this is not problematic because it only harms the caller, whose transaction is therefore voided; the contract itself is not affected. However, laziness affords a novel opportunity for sabotage: since the argument is only evaluated when it is used, it is entirely possible for it to represent a non-terminating computation that causes the contract call itself not to terminate.

The flaw of vulnerability has serious consequences for the system as a whole, because it interrupts the sequential execution of the contract. Once a single contract call is nonterminating, the contract may *never* be called successfully afterwards because the faulty call must be resolved first. A malicious party may therefore "shut down" a contract regardless of how well-written it is.

The only recourse for this is to forbid transactions to call contracts with arguments that may lazily represent an untrusted computation. In the previous discussion, we already indicated that transactions must be constrained to call contracts only in a block before any code is run; now we make the additional restriction that the arguments to these calls must be provided as *text literals* that are parsed into the actual input types by conversion functions supplied by the contract authors.

Once this is done, a contract is exactly as good as its source code and no transaction can shut it down without the exploit being apparent in advance.

ESCROW FAILURE MODES

Escrows, being contracts, are also subject to the same failure modes. The consequences, however, are less serious: since escrows are owned by the calling contract, for the caller to pass a nonterminating argument to an escrow would do nothing but shut down the caller itself! Therefore, it is acceptable for escrows to take any programmatically constructed arguments. There is one additional failure mode that arises from the interaction between escrows and contracts:

- **Injection:** the escrow itself represents outside code injected into the contract, if it was given to the contract by another.

The avenues for this injection are limited, however. A contract may be given an escrow by its creator; if it is a transaction, then by the contracts it calls; or by another escrow that it calls within itself. It is not possible for a malicious contract to donate an escrow to another contract directly. All of these avenues are plainly visible to the contract author: in the first case, the author of the creator contract cannot, by definition, write code that is untrusted by the child contract, since they must write the code that generates that contract and are so its author as well. In the second case, the transaction has explicit control over what contracts it calls and how, so the transaction author can scrutinize them for untrustworthy code. And in the third case, the escrow that is called must have been given to the contract by its creator, if a transaction then by its called contracts, or by another escrow that it called. This reduces the third case to the others, which are already shown to be solvable by source code scrutiny.

We conclude that, despite escrows being outside code, they can always be ensured not to be *untrusted* code.

TYPE SAFETY

Source control scrutiny being the solution to all the exploits discussed above, we must address the question of how this scrutiny is to be performed. This entails at least a degree of *formal verification*, a problem that is, in general, unsolvable, but which can at least be made more tractable by appropriately structured code. We have seen two kinds of scrutiny, each influenced as follows by a strong type system:

-

Explicit scrutiny: the manual examination of source code by a human. In the tradition of the Penrose theory of mind, this is supposed to be the panacea for all uncomputable ills (we consider this kind of scrutiny, jokingly, to constitute a *Penrose oracle* for programs). However, the experience of weakly typed languages such as Javascript (with its total type fungibility) or, even, C (with its implicit conversions and limited types), suggests that correctly analyzing the behavior of an expression requires classifying its components into pieces whose values can be fully described. In other words, it requires inserting a strong type system into the mental model of the language. A language with an explicit type system can serve to communicate the intentions of the author in this matter to the reader.

- **Implicit scrutiny:** the automatic examination of source code by a machine. We reject runtime scrutiny of the code because, either, the code structure is lost at runtime and scrutiny can only take the form of automated testing, or the structure is not lost and the scrutiny is the same as at compile-time. Compile-time scrutiny can provide, for instance, a guarantee that the literal arguments of a transaction's contract calls will be parsed into the correct input types, because those input types are known strongly and the correct parser chosen automatically. It can also allow the author to guarantee that they are handling the expected kind of escrow-backed value by specifically naming the type of that value; this is strengthened by the possibility of the value's author utilizing *private types* that cannot be constructed by anyone else, proving that a correctly typed escrow value is actually their own.

A strong type system is not a full solution to formal verification, because it cannot access the particular values that are used (for instance, it cannot catch an off-by-one error). A *stronger* type system, such as dependent typing (meaning types that can reflect runtime values) could do this, though still, provably, not enough to help completely. We again point to Haskell as a language whose type system is strong enough to generate (somewhat exaggerated) claims among its adherents that "if it compiles, it works".

HIGH-LEVEL CODE

Explicit scrutiny can be aided by two things in addition to merely strong typing:

- **Encapsulation:** as in object-oriented programming, methods or other "functional values" that represent specializations of more generic operations can make it easier to express the specific intentions of the author. Rather than a generic loop with a boilerplate body whose function can only be discerned by pattern recognition, use a fold or a map. Even merely specializing a generic function to constrained argument and value types can make it specific enough to rule out entire classes of unintentional errors.
- **Sandboxing:** or, in other words, a virtual machine. Ethereum recognizes the necessity of this but does not go far enough in equipping the machine. Ideally, the host language should remove any mechanism by which code may access the machine directly, while at the same time making it possible to operate in a simulated machine with other capabilities. This includes the control of side-effects, allowing the user of a particular value to know the limits of that value's behavior entirely from its declared type. A contract can declare its own internal mutable state that is entirely managed by the coroutine mechanism, so that the system itself can know that it is correctly handling any kind of state the author desires.

Together with our advocacy of strong typing, these are of course merely arguments for the use of functional languages. Nonetheless, in a system that requires a high level of control over the behavior of code for the safety of its mutually untrustworthy users, a language that affords arbitrary levels of abstraction is to be preferred over one that prioritizes arbitrary levels of machine control. This is why we promote Haskell over the C language and its descendants; the jury remains out on high-level imperative languages like Python.

COMPUTATIONAL RESOURCES

Returning to the particular concerns of a smart contract system, we consider at last a major component of Ethereum: computational resource constraints. Familiarly, Ethereum assigns a "gas" value to each of its assembly operations as well as some system-level operations like contract creation and transaction execution, and each transaction must declare in advance, as well as pay for, its gas usage. This is bluntly effective at limiting the amount of computation that a single author, however malicious, can force the network to perform, though it is difficult (and as usual, impossible in theory) to know exactly how much gas is necessary.

The principle of laziness changes the calculus entirely. Since transactions are only executed when their results are required, *no one* is forced to execute *anything*. In fact, a client of such a smart contract system might continuously absorb new transactions without ever executing any of them, because simply adding a new transaction does not qualify as using any other. Use consists of a user of the specific machine operating the client executing a program that, in effect, *links* to the smart contract storage, extracts transaction results, and evaluates them, say to be printed to the screen or delivered via the internet.

When such a use occurs, the burden is entirely on the user to determine whether the resulting cascade of transaction and contract evaluations is satisfactorily efficient. They may, of course, simply kill such a program if it takes too long, without affecting the synchronicity of the network, because the actual contracts remain the same, and a more powerful computer or more patient user may successfully evaluate them, with the same results for anyone who completes the computation. A malicious contract author can be safely ignored: a particular transaction may even fail to terminate at all and not lock up the entire system, since its evaluation is decoupled from the evaluation of other, independent transactions that are wise enough to ignore it.

The conclusion is that in a lazy system, there is no need to limit computational resources. Each client can choose independently how much load they wish to assume, without compromising the integrity of the network.

PULL, NOT PUSH

The considerations above all center on the discretion of the contract author in choosing which other contracts to trust, while being free from the unilateral interference of other contracts. We denote this principle by *pull, not push*, meaning that contractual operations should center on making resources available for autonomous users to collect at their leisure, and not on accepting deposits and instructions from users who wish to cause a certain effect. This is also called “inversion of control”, another typical feature of functional languages over imperative languages. Clearly, this affects the structure of contractual interactions. For example, consider the following descriptive implementation of the “two-party swap”, or perhaps “ransom demand”, where two parties each put forward a valuable but neither can collect until both agree:

- anyone: create a "voting" contract accepting yes/no votes from parties A and B only and returns an opaque token with two values: A and B. If either party votes “no”, both of them can collect their corresponding tokens. If both vote “yes”, then they can collect the *other* party's token.
- party A: create a contract accepting the opaque token A and returning valuable A.
- party B: create a contract accepting the opaque token B and returning valuable B.
- both parties: vote, then collect their tokens.
- If there was a “no” vote, then party A can take back valuable A and party B can take back valuable B; they lose nothing. If both voted “yes”, then party A gets valuable B and party B gets valuable A; the swap is completed.

With this design, it is impossible for either party to cheat by withdrawing their valuable during the deal, or to steal the other's valuable without the other one being able to collect their own. The valuables are *not* “deposited with” the voting contract because it is not possible to call a contract with a programmatic argument, and a value that is given by a literal is not valuable. So the “push” model of each party giving their valuable to the neutral voting party, then voting and collecting their results, is impossible. Instead, they create contracts with trustworthy, coordinated credentials independently and then collect them when they obtain those credentials from the vote.

Native currency and rewards

After the discussion of computational resources, it becomes clear that there is no inherent use for a currency. Users may, of course, define their own currencies which, through economics, can acquire value, but these currencies need not be woven into the system. No part of the execution of smart contracts requires payment.

The external concern of managing a blockchain that synchronizes the contract network does, however, present a need for the system to provide some kind of valuable resource so that it may be awarded to block creators as an incentive to keep the network going. Thus, the system *will* provide a minimal such resource: an abstract, non-numerical token that we call `Reward` that is made available in unit quantity to a “reward transaction” that the block author may insert via an escrow that is opened in that transaction. This token, which is obviously scarce, has just one operation: `claim`, which accepts a `Reward` escrow ID, checks that it contains an actual value (and is not, say, a type-system trick like `undefined`), and then destroys it. After this call, one can be sure that one had a valid reward token. Anyone can use this mechanism to accept (but not to accumulate) `Reward`s in exchange for any other valuable they wish to offer in exchange; reward transactions can voluntarily benefit from any of these reward contracts to gain sums of various user-defined currencies, privileges in some user-defined community, and so on.

There is indeed an incentive for creating blocks with such a reward system, so long as desirable award contracts are offered. The incentive for individuals to write reward contracts is simply that, by the same token, it incentivizes block creators to do their work, continuing the operation of the smart contract system within which their holdings have value. This symbiotic scheme amounts to a community-organized system of block rewards, letting the market determine the value of a new block and the corresponding inflation of its assets.

Conclusion

We have described here a complete re-imagining of the design of a computationally complete smart contract system organized around the principles of functional programming. This system possesses numerous additional benefits over existing systems, allowing it to solve virtually all of the problems plaguing them while at the same time expanding their economic power. Hopefully the community of blockchain enthusiasts and investors will realize the power of this redesign and adopt this or a similar system in the future, preventing this powerful technology from being hobbled by a reliance on increasingly obsolete computing models.