FAQ

Basics

Q. *I don't want to read an architecture document; how does Fae work?*
A. Fae accepts transactions, and optionally executes them. Those transactions will call some number (possibly zero) of existing contracts, then maybe do some additional work on the results, including creating more contracts. When those contracts deal with valuables, those valuables will be composed of portable contracts, called escrows, that manage the life cycle of that value.

Q. *How can it possibly work to* optionally *execute transactions?*
A. This is called "lazy evaluation", meaning that computations are deferred until their outcome is specifically required. Until that time comes, further computations can be added that use those outcomes (but are also deferred), and so on. The result is that, when the ultimate outcome of a chain of computations is required, the entire chain is executed in the correct order. Until then, it exists only in potential.

Q. *I didn't quite get how transactions work. What do they do?*
A. A transaction begins by making some contract calls; the arguments to these calls are provided in the transaction message that is exchanged over the network. Each contract call can itself receive the results of some additional calls to provide it with valuable "materials" as payment or authentication. The return values of the top-level calls are collected and passed to a function, the transaction body, that can be any kind of code that creates contracts or escrows, or uses escrows. It can't use contracts, though.

Q. *Why can't I make contract calls inside the body of a transaction?*
A. You can't make them inside a contract or escrow, either, by the way. The only place contracts can be called is before the transaction body begins. This ensures that just the knowledge of the transaction messages, without reading or executing any of the code, is enough to create the chain of lazy evaluations. If nested contract calls were possible, laziness would be pointless because just figuring out what to execute would require executing everything.

Q. *How do I write a transaction?*
A. The short answer is that transactions, and consequently the contracts and escrows they create, are written in Haskell. The long answer is to read the tutorials.

Q. *Could you explain escrows in more detail?*
A. An escrow is a construct whose life cycle is managed by Fae; it can be called like a contract, which causes it to be updated or possibly deleted. You can think of deletion as "spending" the escrow fully; it's the most basic mechanism to manipulate value in Fae. It's not likely that a casual Fae user will have to directly interact with escrows; they will just exchange tokens derived from them.

Q. *I don't know Haskell, and I'm also afraid of it from what I've heard. Am I stuck?*
A. Not necessarily; the tutorials try to be very thorough in introducing not only Fae but also Haskell. For simple transactions you barely need to write any; actually, most of the code will probably be found in Fae libraries, and unless you're writing one of those, only the barest features of Haskell are necessary. In fact, in an imaginary Fae ecosystem, standard transactions would be generated in response to human-readable requests for them. Until that exists, you do have to write a little Haskell, which I don't think will turn out to be frightening.

Q. *What do you mean by "Fae libraries"?*
A. Transactions may come with Haskell modules attached, which are made available via the standard Haskell mechanism to other transactions and their modules. In fact, some of the features of Fae rely on the ability to modularize code and hide portions of it.

Q. *How do I try out Fae?*
A. The project information sheet describes how to get the source code and the Docker images, and how to use them. You may need to write Haskell, but you definitely do not need to know how to build a Haskell project.

Q. *Why would I want to use Fae instead of Ethereum?*
A. Ethereum's scalability is limited because it forces all the network nodes to execute every transaction in order to validate the results. In Fae nodes validate only the transactions that they need in order to run their own transactions. This means that the cost of executing a smart contract in Fae amounts only to computational resources. Unlike in Ethereum, with Fae there is no need to pay miners on the network to validate your transaction. In fact there are no "miners". The closest thing to miners in Fae is the subset of the network participants who want to use the results of your transaction. These participants will then necessarily need to verify the result of your transaction so that they can view it or use its results for creating their own transactions. In summary the validators of transactions in Fae are the set of nodes which intend to use the result in someway unlike in Ethereum where the whole network all transactions and receive a fee in the form of "gas" for doing so.

Economics

Q. *Fae doesn't seem to have a currency. How does that work?*
A. Fae doesn't need a currency because there's nothing to pay for. No one is required to execute anything outside the dependencies of their direct intentions, so Fae doesn't need to charge for "gas" to perform executions.

Q. *What's the point of a blockchain that doesn't manage a currency?*
A. Blockchains manage *scarcity*, which is the foundation of currency. Fae does manage scarcity in the form of "escrows", and currencies can be implemented using this mechanism. They can also implement more complex valuables, because they have all the capabilities of contracts.

Q. *But without a currency, how can you incentivize block creators?*
A. Fae does contain a concession to this need: reward tokens. They are scarce (one, or at least a limited number, per block) and therefore valuable, but explicitly not a currency. People can write functions or contracts to redeem reward tokens for something else that's valuable in their local economy (say, currency, but also privileges).

Q. *Okay, but that just passes off the problem to those people. Why would anyone write such a redemption function?*
A. It's like the economic incentive that Bitcoin uses to justify a 51% majority miner not manipulating the system: because they want it to keep working to maintain their wealth. The people writing these redemption functions are going to be merchants, entrepreneurs, the generally wealthy, even potentially social groups and governments, all of whom rely on Fae continuing to function. They will redeem reward tokens to encourage earning them through block creation, and therefore getting their own transactions into Fae.

Q. *I don't understand the point of escrows. Why can't valuables just be actual values in the code?*
A. Fae has a UTXO model of contracts, like Bitcoin, so consider how Bitcoin might work if transactions were allowed to consist of code explicitly manipulating the input UTXO values and producing outputs. Sample pseudocode:

```
x <- utxo1
y <- utxo2
let xy = x + y
let [z,w] = split xy [2,1] // Makes a 2:1 split
deposit "alice" z
deposit "bob" w
```

If *x* and *y* were both plain numbers, then we could just as easily write `xy = 2 * (x + y)` to create something out of nothing. Therefore they need to be some kind of managed quantity passed by reference, with the management

enforcing the arithmetic interface and the deletion of values after use. Generalizing “arithmetic” to “any kind of function” and “deletion” to “any kind of state modification”, you have the exact concept of escrows that Fae uses.

Q. *It seems unnecessary to have escrows if they're the same as contracts. Why not just represent value as a particular kind of contract?*
A. The reason escrows can't be the same as contracts is that they need to be subject to the above management when used *inside* contract (or transaction) code; that is the reason the management exists. And Fae, being a pure UTXO machine, does not allow contracts to be called in other contracts. Escrows therefore play the role of “portable contracts”, with the same internal functionality as contracts but subtly and importantly different semantics.

Q. *Doesn't the restriction of contract calls to before the transaction runs make it impossible to run complex interactions, like in Ethereum?*
A. Interactions like that will definitely have to be designed differently in Fae. Although it's impossible to say exactly what would constitute a proof that the two systems have equivalent power in this way, there are a few general principles in Fae that account for nested calls in Ethereum:

- *Fae software libraries do not have to be contracts.* Ethereum requires everything to be a contract; there is no concept of attached static code. Fae does have this concept, so there is no need to design contracts that function as abstract classes; instead, you design an abstract class and use it inside your contracts. The fact that everything is Haskell helps in creating really general, reusable functions.
- *Fae contracts should provide privileges in exchange for credentials.* A Fae contract is not code that *does* something, it's code that *guards* something. It may take a few rounds of calls to get it into the state where it releases the privilege it guards, but since the only interface to contracts is via their arguments and return values, they should be designed so that the return value provides further credentials (for chained calls) or simply the desired valuable.
- *Transactions with multiple signers can have complex contract calls.* Fae lets many people sign a transaction, each signature potentially providing access to different contracts. If many parties need to coordinate their actions, they can all write a transaction together. The burden is placed on humans to decide whether a transaction is suitable.
- *Contracts can rely on optional materials*. A contract can, internally, choose to look up a value by name, which value is separately provided in the transaction message as a “material”: a nested contract call that nevertheless is declared up-front and is run before the main call. The lookup then functions as, basically, a nested contract call, which it is the responsibility of the transaction author to fill in: the code itself expects only the result. If the transaction author knows that a particular material will not be needed, they can also choose not to make the call to obtain it. This “floats” the logic of nested contract calls up to the transaction message while still allowing their actual results to be intermingled with contract code.
- *If all else fails, escrows can function as callbacks.* Escrows are functionally like contracts, so if a contract call really requires many successive branching decisions, it can do the first one, then return an escrow that can be used to do the next one, possibly multiple times. Each escrow can be managed by different parties (the original contract would, of course, obtain these escrows originally by permission of those parties) and therefore give access to different privileges. This both gives more freedom to the transaction author (they can be aware of the results of the intermediate decisions) and restricts the contract author (the escrow calls can't change the state of the parent contract).

Operation

Q. *You say that no one has to execute anything they don't want to. How do they decide if they want to?*
A. Manual scrutiny and automatic testing. Either they read the code behind the contracts they want to call, or they run them in a sandbox with appropriately limited resources to see if they fit.

Q. *So if I want to write a transaction I have to read the source code of every previous transaction?*

A. You could if you wanted to uncover anything that might be wrong in a sneaky way, but 99% of transactions are going to be calling well-known contracts (accounts containing currency, stores, and so on), so what you really want to guard against is nonterminating code. That can be detected by automatic testing and takes no more time than you would spend running your transaction if it were actually included in Fae.

Q. *Well, what if I'm the owner of one of those stores and someone tries to pay me with currency tainted by a nonterminating transaction? My contract will run autonomously and be blocked by this one call.*
A. If your contract is going to be taking money from untrusted sources, you're going to want to secure it to you: prevent it from accepting the currency unless your signature is on the transaction. Then you will have the opportunity to run some tests on the proposed transaction before submitting it.

Q. *So I'll have to look at every transaction that might affect me?*
A. In a way. That automated testing can include automated signing, too. The whole thing could be attached to the website you're running your store from: the customer makes a purchase and submits payment; the website writes a transaction representing this, and runs it in a sandbox; if all is well, the merchant's signature is attached and the transaction is sent to Fae. That's not really any different from how stores usually work.

Q. "Gas" in Ethereum ensures that no non-terminating contract can be included on chain. Does Fae have built in automated testing to prevent client's from running non-terminating contracts?

Q. *What if I'm not a merchant, but just trying to accept some payments?*
A. If you're talking about individual payments from known individuals, then you *can* afford to fire them off in your local Fae client before signing. Or you may simply trust those people. Or, if this is actually a burden and you don't trust them, but you aren't big enough to set up an automated web store, it seems like it would be a pretty useful niche for someone to run their own automated test client and sell you time on it. This is not a contract question; it's a tooling question.

Network

Q. *Why is it important that Fae doesn't execute every transaction?*
A. In a word, scalability. It means that each Fae client in a network is allowed to choose how much of the burden of running Fae they want to assume. Even if one corner of the Fae universe increases explosively, quiet people just trying to have a few interactions among a small group will be able to ignore it. The capacity of the Fae network therefore increases, potentially, as the sum of the capacities of all its participants, though in practice, of course, many of them will be executing the same transactions.

Q. *If Fae clients don't have to execute every transaction, how can the network remain consistent?*
A. There is something all clients have in common, but it's not the fully executed state of the transaction list. It is, rather, the "stubs" of transactions that have been lazily added but not executed. All clients have the same set of transaction code, and their executions are queued so that each transaction comes after its dependencies, but left in this prepared state until they are forced. The storage states of the various clients are consistent in that *if* two of them were to execute the same transaction, they would get the same result.

Q. *That's a bit technical. Is there a more intuitive explanation?*
A. Sure! Here's a metaphor: the Fae network is like the universe as described by quantum mechanics; there is one deterministic "wave function" (the lazy queue of transactions) and many "observers" (Fae clients) who "measure" the wave function (partially execute the queue).

Q. *That's the first time I've heard quantum mechanics described as intuitive. Maybe something more familiar?*
A. There's also the metaphor of five blind people examining an elephant. One grabs the trunk and says it's a snake; one grabs the tail and says it's a worm; one feels a foot and says it's a stone; one feels a leg and says it's a tree; one touches a tusk and says it's a spear. But it's the same elephant for everyone; they just examine different parts of it.

Q. *What consensus algorithm does Fae use?*
A. None, or perhaps any. Fae is a virtual machine operating on transactions it's given; the consensus algorithm determines what those transactions are, and is independent of Fae. It can be attached to any one.