History truncation in Fae (Plasma-like scaling)

There are two kinds of scalability problems in blockchains:

- Scalability in breadth, or parallelizability of independent contract and transaction executions.
- Scalability in depth, the ability to operate with truncated contract histories.

Fae is designed to provide the first kind of scalability to the maximum extent, by using lazy evaluation. However, laziness (which is actually synonymous with scalability in breadth) is not sufficient to provide the second kind of scalability. In fact, it is premised on the requirement that any execution be preceded by the execution of its entire history (and nothing else); truncating histories is not possible. However, scalability in depth *is* possible in Fae by extending the versioning mechanism slightly.

Versioning

Versioning in Fae refers to the assignment of a unique identifier to each contract return value, which can be used in subsequent contract calls to ensure the correctness of these return values when they are subject to change if the contract is not in the expected state. These identifiers, currently, are not content-based, but simply record the sequence of transaction, contract, and escrow calls that have occurred in its history, as well as the structure of the value type. I have recently extended the contract and escrow creation mechanism to allow representing them (sometimes) as concrete data types rather than function types, allowing their representation to be hashed into the version (in addition to the history).

The prototypical transaction in which versions are useful is a payment, and is structured as follows:

1. The seller has, in the past, set up a contract that will only function if the seller has signed the transaction.
2. In the present, the buyer writes a transaction message that alleges to retrieve payment from their own account contract and deliver it to the seller's contract; this delivery is denoted by passing the version ID of the payment to the seller's contract as an argument. This message must expect the signatures of both the buyer and the seller; at this stage, the buyer alone has signed it.
3. The seller receives the transaction message off-chain and runs it in a simulated environment to verify the payment's validity and computational efficiency. If satisfied, they add their signature and the transaction can be run on-chain.
4. When the transaction is run, the call-by-version of the seller's contract is correct only if the actual value returned by the buyer's account, *as determined by its execution history*, has the version ID in the transaction message.

Nearly the same sequence of actions can also implement scalability in depth.

Scalability in depth

Suppose that the seller (the one being paid) does not wish to execute the payment's full history, while still remaining certain that the payment they think they are getting is correct – the prototypical problem of scalability in depth. The seller (in fact, anyone executing the transaction, when it eventually goes on-chain) already has the alleged version ID of the payment; if this is a suitably secure cryptographic hash of the payment's value, then it is virtually impossible for an alternative value to be found with the same ID. Therefore, the buyer can offer to transmit the value, alone as a "packet", of the payment to the seller, who can then install it in the simulated environment and proceed with the transaction's execution as though the relevant history were present. In doing so, they can also verify that the version ID agrees with the version calculated from the provided "packet" value. For the same reason as in step 4, above, the seller is assured that on-chain, the value delivered to their contract will match the one they verified in simulation.

Once on chain, anyone who wants the results of the transaction but not the history of the payment can request the "packet" of its value from anyone who has executed the history and verify it against the version ID. If it matches, then by installing that packet in place of the actual executed contract's return value, the recipient will get the same future history as a more conscientious party that obtains that return value by full computation. The version ID plays the role of a checkpoint in the contract's history, from which future executions can depart after being provided just with the snapshot state at the exact time of the checkpoint; the snapshot is cryptographically linked to the version ID and cannot be forged.

The result of this protocol is that, so long as enough Fae clients execute the history of any given value, another client may avoid executing that history by requesting the "packet" from some honest one of them. This provides scalability in depth to the full extent desired by the relevant parties.

Trust

Notarizing the alleged result of a truncated history with a cryptographic hash has an obvious potential exploit: the buyer can provide both a false version ID as well as a fictitious packet that matches it. There are several potential resolutions to this:

- The seller could obtain the version ID (and possibly also the packet) from a disinterested third party, such as a commercial full Fae node, which has no reason to favor the buyer in particular with a false packet. Although this apparently defeats the purpose of the seller's node even existing, this is only true when the seller *is* selling; in transactions where it is the buyer, or for transactions without such validity concerns, they can operate without assistance.
- The seller could get the ID and the packet from the buyer but also require signatures from one or more disinterested third parties. Assuming that any honest full nodes exist, this increases the chance of having one decline the transaction, protecting the seller. It also allows the transaction to be submitted "conditionally" without further action from the seller, and it links the correctness of the version ID to the profits of the verifying parties if, say, they are offered a commission on the proceeds, providing an economic incentive to be honest.
- The buyer could supply the packet along with some form of proof of its correctness. This is impossible as a general strategy (at least, if the aim is to save the seller some work compared to simply running the relevant history; not every algorithm can be compressed) but may apply to specific situations.

The commission scheme seems quite general as an *incentive* to certify correct transactions, but does not provide a *disincentive* to certify incorrect ones – although doing so would not yield any profit to the dishonest verifiers, they may do so simply out of malice. To some extent this is on the seller, who chooses the verifiers, but in addition to somewhat defeating the ideal of trustless transactions, to put the responsibility on them would also simply be potentially ineffective, if they are not diligent or knowledgeable. Another of Fae's features may provide the answer: fallback functions.

Fae transactions are allowed to provide, essentially, `catch` clauses for transactions. These "fallback functions" are alternative transaction bodies that are run, in order, after an exception is thrown in the primary transaction. The possibility of handling transaction error states allows the seller to impose a *proof of stake* requirement on the verifiers:

- The transaction call list would include withdrawals of some kind of value for each verifier, in addition to the calls necessary for the primary transaction's functionality. Since successful contract calls are never rolled back, this stake is provided up front and independently of the result of any other activity in the transaction.
- The primary transaction body would first attempt to access the verifiers' stakes. If any are invalid, this will throw an exception; if not, it proceeds to access the buyer's payment, which likewise will throw if invalid. If no exception is thrown, then the transaction's preconditions are correct and the stakes can be re-deposited in the verifiers' names (this should be done before anything else, in case the transaction body throws an exception later) and the commissions can be distributed to them. The normal transaction operations can then proceed.

- If any exception arises in the primary transaction, the fallback would run. This indicates that the verifiers are all dishonest, so the fallback function would simply deposit all their stakes in the seller's name. Even if all the verifiers provide invalid stakes, only a predictable, limited amount of execution would have occurred at this point, so there is no serious impact on the seller; this is no different, really, than someone writing a completely spurious transaction, as it can be ignored without negative consequences.

This disincentive limits the possibility of failure to the situation where every verifier is not only dishonest, but actually profits from acting on it. The stakes should be well-chosen (by the seller) to exceed any possible such profit, which should convert all the dishonest verifiers into honest ones, unless none at all can be found who can afford to put up such a stake. If that happens, it suggests that all the wealth in Fae is in the hands of the buyer, which is a failure mode of the entire economy.

Owing to the impossibility of general algorithm-compression, some limitation must always be present in any system purporting to allow some parties to avoid doing work while still benefiting from the results. The system presented here chooses that limitation to be the unavoidable presence of risk, albeit risk that can be decreased arbitrarily much. The alternative of limiting the system to a carefully selected subset of sequences of transactions that can be compressed is also an option, though it is unclear what that subset would have to be and whether it would still be useful.

Fae mechanism

The above protocol depends on the ability to collect an entire Fae value as a “packet” of binary data to be transmitted. This is problematic: although the packman library exists (and is amazing: it can export *any* Haskell type as binary data and then recreate it from that data), it is incomplete and in particular, cannot serialize types coming from interpreted code. Since all Fae code is run by an internal interpreter, this rules out that approach. Instead, I propose to follow the model of reward tokens.

Rewards in Fae are a special-purpose escrow type that can only be created by Fae itself, and are provided under special circumstances such as the creation of a new block (which gets to write a reward transaction). The contents of the escrow are actually trivial: it accepts a private argument type and returns a private value type, which are both handled by a public function that calls the escrow to validate it. The escrow is then destroyed; its purpose is to prove that the owner *had* a reward, so that the validation function can be used as part of a token-redemption function to award them some other quantity that has actual value. In effect, the reward token is converted into a more complex kind of escrow.

The resulting escrow, being complex, is impossible to serialize using normal methods (it certainly contains a function type, which can't be serialized, except via packman, which as noted does not work here). The token, however, is pure descriptive data and certainly can be serialized, even without packman. In fact, in combination with the redemption function, the reward token contains complete information on constructing the complex value.

The way to create depth-scaling “packets” is therefore to associate a descriptive token type with the type of value that is to be serialized. These tokens can be sent as binary data, then represented to the recipient as a special-purpose escrow like the reward token. At the same time, the author of the complex value type (and descriptive token type) will provide a redemption function that creates the value from the token. The source-code-level data of the various types and functions will of course need to be loaded from the appropriate module, which will likely be associated with a transaction whose execution is to be ignored by this mechanism, but loading the module does not require running the transaction or even saving its message.

ALTERNATIVE PACKET DESCRIPTION

A more organized way of describing the reward-token approach is to introduce a typeclass:

```
class (Serialize a) => Rebuild argType valType a where
  rebuild :: a -> Contract argType valType
```

The type parameter `a` plays the role of the token; it can be defined anew for each contract that is supposed to allow depth scaling. The `rebuild` function transforms a token into the latter portion of a contract starting immediately after a `release` call returns, accepting the return value (of type `argType`) and performing the function of the contract from that point until its final termination with `spend` (if that exists).

The token type and its `Rebuild` instance would need to be defined in a public module that can be loaded to bring it, and all the rebuilt contract function definitions, into scope (which can be done automatically by the system). A `Rebuild` constraint can be imposed on all contract definitions and a default instance (that throws an error) can be defined, to be overridden by authors who wish to enable this functionality.

Conclusion

Scaling in depth in Fae can be achieved by improving slightly two existing mechanisms:

- Versioning of contract return values, and
- Reward tokens

as well as utilizing one more in a particular way:

- Fallback functions.

The protocol is simply to allow the versioned return value of an actual contract run to be replaced with the same value obtained ahistorically. The value itself must be an escrow in the style of a reward token, and its version must be the hash of its contents, serialized somehow. The integrity of the ahistorical token is protected by the cryptographic security of the hash, which renders intentional collisions infeasible, as well as (potentially) some number of probably impartial verifiers who can be motivated to increase that probability.

This protocol makes depth-scaling completely transparent to Fae users that do not employ it. Transactions for which the full history is present will run exactly as before, according to how versioning currently works. Likewise, users that *do* employ history truncation do not use a different form of transaction, but simply arrange for the correct packets to be present during execution. No special operations are necessary to “enter” or “exit” a “sidechain” whose history is truncated. The only possible form of falsification is a conspiracy to supply a forged packet, the risk of which can be reduced as much as desired by involving any number of impartial verifiers, who need not have any official status as such but simply make themselves available for hire.

Fae's depth-scaling even allows for private subchains: the truncated history can be made actually opaque except to a whitelisted group of users. So long as the transaction that *uses* the packet value is public, no previous transaction even needs to be present for users that run it, if they can obtain the packet. So the truncated history can have an energetic participation by authorized users, who occasionally report summary values to the public portion of Fae; these values are trustworthy via versioning and may be actual financial or commercial values, or simply data that, by its participation in this scheme, is effectively notarized by the authors of the consuming transaction.