Specification

Fae is currently implemented only by myself, in Haskell; it is uncertain at this time whether it *can* be implemented in another language or even another Haskell compiler, as I use the specific capabilities of GHC Haskell extensively. This document attempts to capture the abstract capabilities that characterize it. In it, following the conventions of computer standards, **bold** imperatives are mandatory for a compliant implementation.

The nature of this specification is *inclusive* rather than *exclusive*: by and large, it demands a certain set of capabilities of any implementation, but does not demand (with a few exceptions) that the implementation be limited to these capabilities. It is therefore possible to design a system that is “a Fae” like the various *nix operating systems are each “a Unix” (if so certified, anyway). This stands in contrast with other smart contract systems that rigidly specify and strictly control every aspect of their operation. I believe that adherence to the capabilities demanded by this standard is sufficient to ensure a system supporting trustless interactions among its users.

Contracts, escrows, and transactions

These three interdependent concepts together determine the functionality of Fae. The ensuing discussion will include circular cross-references both to each of these contracts from each other, and to specific concepts within each one from each other. To make sense of this, the reader should search for the *emphasized* form of an unfamiliar term, which may occur earlier or later than any given use of that term, and which serves to define it.

CONTRACT CODE

Abstract: Contract code is the language-plus-API in which contracts, transactions, and escrows are written. The capabilities of the API are the only way that contract code may influence or be influenced by the internals of Fae.

Each of contracts, escrows, and transactions is created by providing *contract code*, which **must** satisfy the following:

- Contract code **must** be valid source code in a general-purpose hosted programming language (e.g. Haskell, Java, Go, Python), subsequently referred to as the *source language*.
- The source language **must** be strongly typed. Its type system **must** be capable of at least the following constructions:
  - The formation of “product types”, also known as “structs”: a named list of fields of various types.
  - It **must** be possible to make distinctly-typed structs with the same field types.
  - Empty structs (that is, typed symbolic constants) **must** be possible.
  - Data hiding: it **must** be possible to prevent type names, constructors, or associated functions from being available except to certain whitelisted code.
- The source language must have a subset of its builtin and library-enabled constructs that is *safe* in that:
  - Safe code **may not** directly access the ambient computing environment, hardware and software included.
  - Safe code **may not** directly construct data or function types by manipulating the memory available to the process in which it runs. Instances of a type in the source language **must** be created through source language mechanisms.
  - Safe code **may not** break type safety through unstructured “casts” that reinterpret the primitive representation of data. However, it **may** use as an intermediary for such a (source-language-mediated) conversion a source-language defined type that corresponds to a primitive data representation.
  - Definitions made within safe code **may not** shadow definitions available through the Fae implementation.
  - The use of safe code **must** be enforceable by the source language rather than by discipline on the part of programmers writing it.

- Contract code **must** be safe code.
- In addition to the native capabilities of the source language, contract code **must** have available to it *mechanisms* implementing the following capabilities:
    - Contract creation
    - Escrow creation
    - Escrow calling
    - Contract or escrow suspension, except in a transaction body
    - Contract or escrow finalization, except in a transaction body
    - Signer public-key access.
- During execution of contract code, the attribute of *context* **must** be defined, such that:
    - The context includes the executable form of the contract code.
    - The context includes the escrows available to it, and these escrows **must not** be included in any other context, nor exchanged except as mediated by the escrow transfer mechanisms; the escrow storage **must not** be accessible to contract code except through the escrow creation and calling mechanisms.
    - The context **must** include an ordered list of created contracts, which **must not** be accessible to contract code except through the contract creation mechanism.
    - The context **may** include *local variables* representing some sort of source-language-defined data or function instance storage.
    - The data or function instances that are available to contract code **must** be limited to the local variables in its context. Contract code **may not** affect local variables in another context.
- All contract code executed during the execution of a single transaction **must** have read-only access to the names and corresponding public keys of all signers of the transaction.
- Contract code **must** be capable of throwing exceptions.

CONTRACTS

Abstract: Contracts in Fae are abstract functions maintaining, as required above, an execution context. They function as “coroutines” in that they have the capability to suspend their execution, returning an intermediate value and a continuation function that resumes from after the point of suspension. Contracts may be called as functions only within a transaction's input list, but this list may contain any number of contracts and may also be followed by transaction-specific contract code.

The Fae implementation **must** maintain *contract storage* with the following properties:

- A contract in storage is retrieved by a transaction by specifying a *(long) contract ID* consisting of one of:
    - The transaction in whose body the contract was created, and its order among contracts created by that transaction body's (sole) execution;
    - The transaction and short contract ID of its input contract creating the stored contract, and its order among contracts created by that input contract call.
    - A contract ID of one of the above forms together with a nonnegative integer. This matches the identified contract **if and only if** its nonce at the time the transaction is executed is equal to this integer.
- The *short contract ID* is a digest (of fixed length but unspecified nature) of the long contract ID. Contracts **may not** be retrieved by short contract ID, but only referenced as inputs to a transaction in the second form of the long contract ID.
- The implementation **must** not provide a mechanism for retrieving contracts from storage outside the transaction input list.

- A contract is stored along with a *nonce*, a nonnegative integral value that **must** equal the number of times the contract has been updated. As a corollary, contracts are created with nonce zero.
- When called as a transaction input, a stored contract is *updated* if it reaches suspension or finalization without raising an exception. This update consists of:
    - If execution terminated in contract suspension, the contract context in storage is replaced by the continuation following the suspension directive.
    - If execution terminated in contract finalization, the contract is deleted from storage. Further references to it by contract ID **must** be invalid, but short contract ID references to it from within a long contract ID given as a transaction input must continue to function normally.
    - The contract nonce is incremented.
- When a contract is updated, or when a transaction completes without raising an exception, the created contracts list is removed from its context and each entry placed in storage.
- A contract may not be referenced, executed, or retrieved from storage within contract code. The only contract code that **may** affect contract storage is the contract creation mechanism.
- The retrieval or execution of any single contract by a transaction from storage **must not** force the retrieval or execution of any other contract or transaction, with the following exceptions:
    - The contract or transaction execution that created the retrieved contract **may** be performed.
    - Any appearance of the retrieved contract in the input list a transaction prior to the current one **may** be executed.
    - If one input contract refers, via versions, to the return value of another (in the same transaction), then that other input may also be executed.
- Contract calls are *immutable*: regardless of future updates to a contract, any contract call must always function using the contract as it existed in storage when the transaction in which it is called was entered.

Contracts are placed in storage via the *contract creation mechanism*:

- The contract code is specified. This code must define an abstract function with specific argument and return types.
- A set of escrows available within the context performing contract creation, called the *endowment*, is specified.
- A context is created for the new contract code, containing only the escrows given. This context **may**, optionally, also contain copies of certain local variables in the creating context, according to the features of the source language.
- The new contract-as-context is placed into the *created contracts list* present in the creating context.

The argument type of the contract **must** have a method parsing it from a string literal. Arguments to input contracts are passed to it via literals and converted to the correct type.

A contract execution that raises an exception is terminated and its context discarded; the stored contract remains unchanged.

ESCROWS

Abstract: Escrows are contracts-within-contracts that implement the concept of “persistent value” within Fae. Like a contract, an escrow is an abstract suspendable function, or coroutine, but the requirements and effects of calling it are different. This difference allows escrows to represent scarce quantities.

Each contract **must** maintain its context across successive calls, including the escrows available to the contract; a transaction body, during its sole execution, also maintains such a context. These context-specific escrow storages have the following properties:

- Escrows within the context are identified by an *escrow ID*, which is unique among all escrow IDs in any context and which never changes throughout the lifetime of the escrow.

- Any escrow present in a context may be *called* in that context via its escrow ID, by supplying an argument of the correct type; the call evaluates to a value of the correct return type.
- The storage entry for an escrow includes its current version, and upon update, this version is also updated in a manner that preserves its distinction from every other version, including the versions of compound values.
- Like contracts, escrows are updated in their storage when a call returns without raising an exception. This update process is the same as that for contracts, except that the nonce increment is replaced by the version update, and the created contracts list is not committed to contract storage but appended to the list in the calling context as it exists at the location of the call.
- Escrows may be transferred between contexts only under the following conditions:
    - An escrow or contract call terminates without exception, returning a compound value. All the backing escrows of the compound value are transferred to the calling context.
    - During contract creation, when the endowment is passed to the new contract.
    - During escrow creation, when the endowment is passed to the new escrow.

The *escrow creation mechanism* is identical to the contract creation mechanism with only the difference that the created escrow-as-context is immediately placed in escrow storage in the creating context.

Escrows give rise to the concept of a *compound value*, which is a struct type containing either escrow IDs or other compound values. The set of all escrow IDs that are ultimately referenced by descent through the fields of the struct is known as the set of *backing escrows* for the value. It **must** be possible to obtain the set of backing escrows for any compound value; this capability **need not** be available within contract code but only to Fae internally.

The type system of the source language **must** be sufficiently strong to construct *opaque values*: escrows of a type signature for which both the argument and return types are private, available only to designated functions of the author of these types. These escrows, as well as less restrictive variants, serve as scarce quantities that can only be created through a specific interface, potentially only under specific circumstances.

TRANSACTIONS

Abstract: A transaction is a one-time contract that is capable of calling and using the return values of stored contracts. In addition to these contract calls, a transaction may subsequently execute contract code to perform further, custom actions. In the event of an exception terminating the transaction, execution may continue through fallback transactions.

Transactions consist of two parts: the *input list* and the *transaction body;* an optional third part is the list of *fallback transactions*, which are alternative transaction bodies (without input lists). The transaction body is contract code that specifies an abstract function (possibly empty) of a particular signature; however, the contract suspension and finalization mechanisms **must** not be available in the transaction body or fallback transactions. The transaction's signature and the input list have the following properties:

- The argument type of the transaction signature must possess a constructor from the ordered list of return types of the input contracts. The return values of the inputs are passed to the transaction body via this constructor.
- The elements of the input list are pairs consisting of: a long contract ID, and a string literal that is parsed by the argument type of the referenced contract.
- When the long contract ID is the form with a nonce, the return value of the contract call **must** be augmented by an implicit mapping of *versions* to values determined from the return value. Versions **must not** be produced for calls via other contract IDs. Versions **must not** be duplicated between distinct values.
- A value's version **must** faithfully reflect all data contained in the value (e.g. all its fields, as well as their data, and so on), the structure of the value type, and the versions of all backing escrow IDs.

- Fae **must** provide a one-field type `Versioned` that controls the generation of versions. The versioned values derived from a top-level struct always include the top-level value itself, as well as the versioned values derived from each of its fields, except when the top-level struct is wrapped in a `Versioned` constructor. For the purposes of versioning, the `Versioned` type itself is ignored and only the inner type is considered.
- The parser method of the `Versioned` type **must** accept versions and construct the `Versioned` from the value with that version present in the same input list, if present. An implementation **must** limit the allowable versions in a given input's argument to those appearing in the return values of previous inputs.
- The termination, successful or not, of a contract call appearing as a transaction input **must** be independent of the transaction body.
- The implementation **must not** provide any facility allowing transaction-specific code to be executed during the evaluation of the input list.
- Escrows transferred via the input contract return values **must** be placed into the context of the transaction. The implementation **must** ensure that, if a `Versioned` contract argument is a compound value, then its backing escrows are present in the transaction context at the time that
- If the transaction body raises an exception, it **must** be terminated immediately and the fallback transactions run. The fallback transactions are alternate transaction body functions whose signature has the same argument type as the main one but which return nothing. When run, each function is passed the same input value as the original transaction body. If a fallback transaction raises an exception, it **must** terminate immediately and the next one, if any, **must** be run.
- The return value of a transaction (or the exception, if one was raised) **may** be stored in an implementation-defined manner for retrieval outside Fae.

SUMMARY

The three concepts of contracts, escrows, and transactions interconnect deeply. Only transactions may call contracts and only before their own code and with a restricted argument-passing convention, but all of contracts, escrows, and transactions can create other contracts. They may also create and call escrows, which function like contracts but are “owned” by a single contract, escrow, or transaction and transferred by passing their immutable ID. The contract calling conventions ensure that a contract call only executes known code, not that which is inserted by a transaction. They also allow contract executions to be run concurrently or independently. The transaction fallback mechanism allows users to plan for failure and appropriately save any valuables that are present in the transaction context.

Blocks and transaction messages

Some aspects of the mechanism of transaction distribution are relevant to Fae, though in general it concerns itself only with the execution of transactions that are already received and appropriately prepared.

TRANSACTION MESSAGES

A transaction, as communicated from one client to another, is transmitted in a message that **must** contain at least the following elements:

- A list of input contract calls; that is, a list of pairs of a long contract ID and a string literal.
- The source code of the transaction function and, if present, the fallback transactions.
- Optionally, the contents of additional source modules (of whatever nature corresponds to the source language). This **must** be possible if separate modules are necessary to implement data hiding.
- A list of signers, consisting of pairs of a text string (the *signer name*) and a cryptographic signature of the modified transaction where each signature is replaced in this list by the corresponding public key.

NEGOTIATIONS AMONG SIGNERS

The presence of multiple signers in a transaction serves to give the “owner” of a contract the ability to protect it. This protection has two parts:

- The contract itself is guarded by checks against the transaction signers to limit access to the owner(s).
- All parties to a transaction have the option of verifying its correctness, performance, security, and so on before signing it and allowing it to be committed to Fae. This is intended to include executing the transaction in a sandbox.

This dynamic makes Fae a trustworthy *codification* engine, immutably recording a transaction that all parties agree to externally before it is entered.

BLOCKS AND BLOCK REWARDS

According to the consensus protocol used, transactions will be grouped into blocks. Fae imposes no necessary structure on blocks but **may** optionally provide a mechanism for instantiating a reward within Fae for supporting the continued operation of the protocol (e.g. for mining a block). If this mechanism is present, it **must** take the following form:

- There **must** exist types `Reward` and `Token` whose only instances are symbolic constants, also denoted `Reward` and `Token`, both of which instances must be hidden from contract code.
- A *reward escrow* is the escrow whose function accepts a `Token` and simply finalizes, returning a `Reward`.
- The *reward claim mechanism* is a function that accepts a reward escrow ID and calls the corresponding escrow, discarding the result. This mechanism **must** exist within the implementation, and other means of manipulating reward escrows **must** not exist.
- A *reward transaction* is any transaction, designated in the manner defined by the implementation (e.g. added by the block miner), expecting, in addition to its explicit input list, an implicit “zeroth input return value” that is prepended to the list, whose type is a reward escrow ID. The implementation **must** place a reward escrow into the reward transaction's context and pass its escrow ID in this value.

This mechanism can be elaborated on by users of Fae to build functions awarding more general kinds of value that are defined by the ongoing operations of Fae. These functions may accept a reward escrow ID via the reward claim mechanism and return their corresponding specialized reward in return. The incentive for doing so is the same as the incentive for participants to support the consensus protocol (e.g. by mining): it is only through the continued operation of Fae that their assets within it have any value at all.