Fae architecture

In order to make explicit what is implicit in the other documents here, this one will describe the workings of Fae, at least insofar as they visibly affect the experience.

Overview

Fae has three main executable abstractions: transactions, contracts, and escrows. Users submit transactions, while contracts and escrows are created during the course of executing the code in one of the abstractions. All three abstractions are general functions with static argument and return value types, returning values in a monad specific to Fae. They differ in their calling conventions and availability, which support different guarantees on their semantics.

Transaction messages

Transactions straddle the line between Fae-internal and Fae-external, as they are propagated across a Fae network as textual messages from which the corresponding Fae code and environment is constructed. These messages include the following data:

- The hash and file size of the transaction's main module, containing in addition to general Haskell definitions the function `body`, the Fae analogue of `main`.
- A list of *other modules*. This is simply a list of hashes of the modules as source code files, with file sizes.
- A list of *input contract calls*, each of which consists of:
  - A path contract ID, optionally annotated with a value for the contract's nonce.
  - A textual literal for the contract's argument type.
- A list of *fallback functions*, which are simply names of functions defined in the main module.
- A list of *signers*, which may *not* be empty. This has two forms, both of which have entries associating a textual name with one of:
  - **Pre-signature:** a public key.
  - **Post-signature:** the pre-signature value together with a signature by the corresponding private key. The message that is signed is the present transaction message with the pre-signature list of signers.
- An optional *salt*, which is any text whatsoever.

The signature scheme ensures that the signers' public keys can be recovered and the transaction message, including these keys, be validated. The salt value serves the purpose of ensuring that the transaction hash, which is used as the transaction's identifier, is unique; transaction messages with the same hash as a previous message are discarded. All of the cryptographic functions mentioned here may be implemented in any decent public-key system; my implementation uses Ed25519, an elliptic-curve system.

The modules that are referenced by hash and file size are expected to be distributed together with the transaction message, but possibly separately. These data allow the recipient to decide if the transaction is too large to accept, and to validate the source files if it chooses to accept them. To complete this picture, the transaction-delivery protocol should initiate with a declaration of the size of the transaction message itself, for the same purpose.

What is stored

Transactions and escrows are identified by a single hash, while contracts are identified by a “path” describing its point of creation. The structure of contract and transaction storage is:

Transaction storage

- Transaction ID: hash of the transaction message
    - Transaction return value
    - Transaction signers: public keys by name
    - **Created contracts list:** each entry
        - Ordinal of contract as created by parent contract: determines path contract ID `/transactionID/thisOrdinal`
        - Current contract function
        - Current nonce (number of times successfully called)
    - **Input contract list:** each entry
        - Path contract ID of original entry in some created contracts list
        - Map of versions to parts of the contract return value
        - **Created contracts list:** each entry
            - Ordinal of contract as created by parent contract: determines path contract ID `/transactionID/parentShortID/thisOrdinal`
            - Current contract function
            - Current nonce

The path contract ID is the two- or three-component path specifying the creating transaction, possibly the input contract that actually did the creation, and the order within the list of created contracts for this transaction or possibly contract. This allows a contract ID to precisely specify the chain of prerequisite executions to get the current contract function with that ID, and therefore eliminate the need to execute other contracts or transactions to retrieve that one.

Each contract function hides an internal state that tracks its escrows. During transaction execution, a state with the same structure is present in the transaction's memory as well. Escrow storage is:

Escrow storage

- Next escrow ID
- **Escrow map:** each entry
    - Current escrow function
    - Current escrow version

Therefore, an escrow ID only refers to a valid escrow entry in the context of the contract (or transaction) that contains that entry in its escrow map. The uniqueness of this contract-or-transaction is assured by the Fae API.

In addition to the Fae abstractions, Fae also stores plain Haskell modules (the “other modules” in the transaction message) associated with each transaction. These modules can be imported either by that transaction or any later one, using the following convenient conventions:

- In the original transaction, a module named `ModuleName` may be imported via `import ModuleName`.
- In other transactions, the same module may be imported via `import Blockchain.Fae.Transactions.TX<transactionID>.ModuleName`; i.e. the module is placed in a namespace below the same one containing the Fae libraries, tagged by the transaction ID.
- The main module itself is also available to other transactions under the qualified import name but without the final component.

Escrow-backed value

Valuable quantities in Fae are constructed from escrows, as recognized by having an instance of `HasEscrowIDs`; the escrow IDs actually discovered by the traversal in that class denote the *backing escrows* for the value. Types without any backing escrows are not valuable because they may be copied "sneakily" as ordinary Haskell values; at the other extreme, a type with many valuable fields is a "derived value". Escrows protect value in several ways:

- An escrow is owned by exactly one contract.
- Escrow IDs are parametrized by the argument and return types of the escrow.
- Escrow calls are deterministically sequenced.
- Escrow calls are type-checked against the escrow ID.
- Backing escrows are transferred automatically via contract and escrow returns.

The combination of these effects is that a Haskell value is "valuable" in the Fae sense in exactly one contract, which if no sneaky copying was performed is the contract holding the Haskell value; if a sneaky copy is returned, then its backing escrows come with it and the original loses its value; if a valuable and its sneaky copy are both used in their original context, the first use affects the state of the backing escrows and the sneaky copy therefore does not necessarily have an equivalent value to the original, i.e. is not a genuine copy.

The inner workings of the escrow function, which is provided by the creator, determine the nature of the escrow as a valuable. The most extensive of these workings is the function's ongoing internal state, which persists between escrow calls; a blunt example of state is whether the escrow exists at all, which is the source of the name of the `spend` API function that returns a value and deletes the returning escrow (or contract).

A more subtle feature is the combination of strictly checked argument and return value types with private types that the escrow author uses to implement public interface functions but does not export to users. Without access to the private types that run the escrow, users cannot create a valid counterfeit escrow, nor "hack" a valid escrow by calling it in an insecure manner. This technique is easily implemented by defining the private types and the interface in modules attached to some transaction, and providing restricted export lists that include the interface functions but not the types.

Contract-like and transaction-like code

There are two classes of monad in which Fae makes its API available, called `MonadContract argType valType` and `MonadTX`, where `argType` and `valType` parametrize the monad according to the contract's argument and return value types. The base instances of these classes are monads called, respectively, `Fae` (which is also in `MonadTX`) and `FaeTX`, and any monad transformer stack on top of one of these is also in the same class. The base instances are state monads tracking the following data:

- Escrow storage (as above)
- List of created contracts
- **Read-only:** current transaction data
    - Transaction signers
    - Transaction ID
- Current continuation function (void for `FaeTX`)

These monad classes differ in how much of the Fae API (described in the haddocks) they provide; contracts and escrows operate in a `MonadContract`. with the appropriate parameters, while transactions operate in a `MonadTX`.

In `MonadContract argType valType`,, the `spend` and `release` functions are available, which create return values of a wrapper type `WithEscrows valType`; a value so wrapped is accompanied by its backing escrows and can be constructed in no other way, so that a contract or escrow execution must end with one of these function calls. The

difference is that `release` updates the current continuation function, while `spend` causes the contract-or-escrow to be deleted from storage.

A `MonadTX` does not necessarily include these functions; in particular, `FaeTX` does not, and is therefore not allowed to return a `WithEscrows`-wrapped value; transactions terminate with a simple `return` statement or monadic value, as usual for Haskell. It provides the API functions `newContract`, `newEscrow`, and `useEscrow` that function as described, as well as various functions to obtain the transaction signers. Note that there is no `useContract`, which is one of the calling conventions that differentiate contracts from escrows.

The API also depends on the following type classes, none of which may be explicitly instantiated:

- `HasEscrowIDs :: * -> Constraint`: an instance of this class may be traversed for all escrow IDs it contains. It is accompanied by a type `BearsValue` that is dynamically resolved to a static instance of `HasEscrowIDs`. Automatic instances are provided for `Generic` types (e.g. `data T = ... deriving Generic`, which simply recurse into an ADT to extract the fields with escrow ID type.
- `Versionable :: * -> Constraint`: an instance of this class may be folded into a map of the versions of its component values. It is accompanied by a newtype `Versioned`. Automatic instances are provided for `Generic` types, which construct the map by structural induction that terminates at any `Versioned`-wrapped value.
- `GetInputValues :: * -> Constraint`: an instance of this class may be constructed from a list of `BearsValue`s. Automatic instances are provided for `Generic` types, which construct the value by assigning each element of the list to each field of a product type, or reading a sum type from a single element. Too-long or too-short lists yield errors.

As described, Fae users do not need to even mention these classes, but simply derive `Generic`. This also provides an automatic instance of `Control.DeepSeq.NFData` (or rather, a shadowing class with an identical interface) that is guaranteed to force each field by structural induction. Manual `NFData` instances are not allowed.

Since all of the classes above are automatic for any `Generic` type, they may be safely ignored by Fae users who are not attempting to write a polymorphic contract library; type signatures are the only place they need appear.

Contract execution

Contracts are executed as input contracts for a transaction, and only in this way. Any transaction may call any contract with any argument; it is looked up in storage by the provided ID, potentially validated against the optional nonce annotation, and called on the value obtained by parsing the literal argument. To enable this parsing, contract argument types must be instances of the `Read` type class. The automatic `deriving Read` instance allows contract arguments to be given as Haskell record syntax, but as this instance is implicitly part of the contractual behavior, users may also write their own parser. The rationale for this strict calling convention is to prevent transaction authors from writing nonterminating code as part of the contract arguments; instead, all the code is under the control of the contract author.

Contract calls are resolved as follows:

- Each call is lazy: the execution occurs not when the transaction is received but when the return value of the contract is used in the transaction body.
- The argument is scanned for versions and the corresponding values are fetched from the current version map to replace the identifiers.
- The argument with versions resolved is scanned for escrow IDs via its `HasEscrowIDs` instance and these escrows are transferred from the context of the ongoing transaction into the contract. (The only place an escrow ID may appear in a contract argument is in a versioned value.)

- The escrows accompanying the contract's return value are deposited in the context of the ongoing transaction.
- If the contract call included a nonce, the version map of the return value is exposed to the input contracts. The versions are available to future input contracts.

Transaction execution

Transaction execution builds on the input contract calls to construct an argument to the transaction body via its `GetInputValues` instance. The transaction's return value must be an instance of the `Show` type class to display the outcome to external users. This display has no internal use, so manual `Show` instances are purely for informational purposes; the automatic `deriving Show` instance is expedient but not pretty.

The course of transaction execution is:

- The input contract list is run and the return values used to construct the argument to `body` (defined in the main module) as above.
- When the transaction terminates, its return value is placed in storage under the newly-created entry for this transaction.
- If any exception is thrown in any code in the transaction body, including in escrow calls, this exception terminates the transaction body; the return value in storage is then taken to be this exception since no valid value is returned.
- In the event of an exception, the fallback functions given in the transaction message are run in order. Each of these functions accepts the same argument as the transaction body, and returns nothing; if an exception occurs during any one, it terminates and is rolled back, but the exception is ignored and the next fallback function is run.

The purpose of the fallback functions is to reclaim valuables that are returned from the input contracts in the event that the transaction raises an exception before they can be deposited as intended. Since they have no return value, their effect can only be the creation of new contracts, which are intended to contain unused valuables for later retrieval. This is effectively a `catch` mechanism, which is otherwise forbidden in contract code because it is an impure action and contract (and escrow and transaction) code is intentionally pure, but transactions are run in an impure environment that is capable of catching exceptions.