Reference manual

The other tutorials are better lessons in the use of Fae, but it can be difficult to reference specific features in the narrative, so they are all documented here. All the user-visible aspects of Fae are described in this manual, but none of the internals.

Monads

Fae code is written in one of the Fae monads, which are all the instances of the typeclasses

```
class MonadTX m
class MonadContract argType valType m
```

where a `MonadContract` allows the full Fae API, consisting of the functions (described in the appropriate sections below):

- *Materials:* `lookupMaterial`, `material`, `materials`
- *Signers:* `lookupSigner`, `signer`, `signers`
- *Creation:* `newContract`, `newEscrow`
- *Calling:* `useEscrow`, which includes the assignment operators `↤`, `⇤`, `⇽` and their ASCII equivalents `<-|`, `<=|`, `*<-`
- *Control flow:* `spend`, `release`

and a `MonadTX` allows all of those except the final, *control flow* category. The basic such monads are `FaeTX` and `Fae argType valType` respectively, and any other obtained by applying a monad transformer such as `StateT s` to any of the Fae monads is also a Fae monad of the same class.

EXCEPTIONS

Monads in either class are also instances of `MonadThrow`, but not `MonadCatch`, meaning that exceptions may be thrown with `throw` but may not be caught within Fae. Exceptions must therefore be used to indicate fatal errors. Due to laziness, a value that would throw an exception when evaluated may or may not actually cause the exception to be raised in code that references it, depending on whether that code actually evaluates it. It is therefore important to fully evaluate return values that might contain exceptions that should cause the present code to fail; otherwise they will cause a failure elsewhere without adequate explanation.

HELPER FUNCTIONS

A few functions are available in `Blockchain.Fae` that are not part of the Fae API but are nonetheless helpful:

```
usingState ::
  (Monad m) => s -> (a -> StateT s m b) -> (a -> m b)
usingReader ::
  (Monad m) => r -> (a -> ReaderT r m b) -> (a -> m b)
feedback ::
  (ContractVal b, MonadContract a b m) =>
  (a -> m b) -> (a -> m (WithEscrows b))
halt :: (HasEscrowIDs a, MonadContract b a m) => a -> m a
```

All of these pertain to contract functions with state. The first two make it easy to write a contract in a monad stack and supply the initial data for its mutable or immutable state; typical usage might be

```
f :: a -> m b
f = usingState True $ \x -> ...
```

where the lambda expression is of type `a -> StateT Bool m b`.

The `feedback` function helps to write contract functions that loop through a single activity repeatedly, once per call. While normally this would require careful use of recursion, with `feedback` one can specify the handling of the input argument and the return value, with the boilerplate code that closes the loop inserted behind the scenes. For instance:

```
f = feedback $ \case
  True -> return 42
  False -> halt 57
```

is a function that awaits a `Bool` and returns one of two `Integer`s via `release`, resetting to the start upon the next call until `False` is received. The `halt` function is a hack that causes this loop to terminate anyway at that point.

Transactions

THE SOURCE FILE

`TransactionSrc.hs` (for example), containing the definition of a function `body` with a signature like:

```
body :: (Show valType) => argType1 -> argType2 -> ... -> FaeTX valType
```

where any number of arguments, including zero, are allowed. The `valType` must be an instance of `Show` and is not used by any other part of Fae. This file is analogous to that containing `main` in a Haskell program and should not have a `module` header; the `Blockchain.Fae` module is imported implicitly. When the transaction is run, this function is called with the return values of the input contracts as its arguments.

THE MESSAGE FILE

`TransactionMsg`, containing at least a field `body = TransactionSrc`, identifying the source module, and possibly a block `others` containing a list of module names to be attached to the transaction:

```
others
  - OtherModule1
  - OtherModule2
  ...
```

The other modules are present at the module paths `Blockchain.Fae.Transactions.TX<this transaction ID>.<module name>`, where the module name is taken from the module headers. Any imports of these modules from each other or from the main transaction file in *this* transaction must be unqualified, and are modified upon receipt to include this transaction path (which cannot be known when the modules are written, as the contents of the source files affects the transaction ID).

Other fields may also be present, as described subsequently.

SIGNERS

The message file may contain a block of keys to sign the transaction:

```
keys
  role1 = name
  role2 = <public key>
  ...
```

where `role1` is assigned the public key corresponding to the private key with the (textual) `name`, as present locally, and `role2` is "declared" to require the signature of the given public key, but does not actually perform the signing and does not require the key to be present locally (this is very specific to the `postTX` utility; other key management policies are possible in a different implementation and only the eventual public keys are important to Fae.
Additionally, `postTX` signs the transaction with the key and role `self` if this block is absent. Finally, in the case of `role2`, the transaction will be rejected unless sent with the `--fake` flag). The roles are exposed in any Fae monad via the APIs

```
lookupSigner :: (MonadTX m) => String -> m (Maybe PublicKey)
signer :: (MonadTX m) => String -> m PublicKey
signers :: (MonadTX m) => m (Map String PublicKey)
```

The `PublicKey` type is opaque (essentially a Fae primitive) and can only be constructed via one of these functions. `lookupSigner` is a total function to dereference a role name; `signer` is a partial function that throws an exception where `lookupSigner` would return `Nothing`; `signers` just fetches the entire role-key mapping.

REWARDS

A special `Reward` type is provided and may be used as the first argument to `body`. If the transaction receives a reward (in `faeServer`, from the `reward` parameter) then an additional value of this type is passed as the first argument when `body` is called. This type has a single API:

```
claimReward :: (MonadTX m) => Reward -> m ()
```

that destroys this value and, in the process, validates it. A `Reward` is escrow-backed and as such can only be obtained via the reward mechanism.

FALLBACK FUNCTIONS

A transaction may exit with an uncaught exception (catching any exception is impossible in Fae), which may leave any valuables returned from the input or materials calls in a state where they have already been obtained and the calls cannot be reverted, but have not yet been transferred appropriately to their ultimate recipients. To alleviate this problem, a transaction may include any number of *fallback functions*, via a block of names in the message file:

```
fallback
  - name
  ...
```

where each name is a function accepting the same argument types as `body` and that is visible in the same module. When an exception occurs in `body`, the fallback functions are called in order with the same arguments until one succeeds; their return type must be `()` and is ignored except for being checked for exceptions. These functions can then re-deposit various values that would otherwise be lost.

Escrows

ESCROW IDS

An escrow is a “portable contract” whose state is tracked by Fae and that can be accessed by its *escrow ID*, of type `EscrowID name`, where `name` is the contract name of the escrow. Valid escrow IDs can only be obtained as the return value of a call to `newEscrow`, or as part of the return value of a contract or escrow call. The escrow itself is implicitly present in the “context” of a single contract, escrow, or transaction code and an escrow ID is meaningless in any other context. Escrows are transferred between contexts whenever their escrow IDs are part of the return value as passed to `spend` or `release`.

The presence of escrow IDs in a return value is determined by a class

```
class HasEscrowIDs a
```

of which every type with an instance of `Generic`, as well as any of the primitive types or basic Fae types, is an instance. Types such as `Map` that are not `Generic` may have an instance applied to them by wrapping them in the parametrized type

```
newtype Container a = Container { getContainer :: a }
```

which simply traverses the values inside. Manual instances of `HasEscrowIDs` cannot be defined in order to prevent dishonest reporting of escrow IDs.

The escrows found in a given value are said to be *backing escrows*.

CONTRACT FUNCTIONS

A Fae contract or escrow is a suspendable function: the `spend` and `release` APIs cause the call in the current transaction to terminate, to resume at the same point in a subsequent call.

```
spend ::
  (HasEscrowIDs valType, MonadContract argType valType m) =>
  valType -> m (WithEscrows valType)
release ::
  (HasEscrowIDs valType, MonadContract argType valType m) =>
  valType -> m argType
```

`spend` causes the contract to be deleted, so that it cannot be called again, and returns its argument as the contract return value. The return type `WithEscrows valType` is another opaque type and can only be constructed by this function, cannot be deconstructed, and can only be used to end a contract function. It indicates that the return value's backing escrows have been extracted to be transferred alongside it.

`release` allows resumption of the contract body, returning its argument from the current call and evaluating to the *next* contract call's argument. The resumed contract begins execution with this value.

Contracts and escrows are functions with the signature

```
type Contract argType valType = argType -> Fae argType valType (WithEscrows valType)
```

meaning that the type variables have roles corresponding to their names and the function must end with a `spend` unless it is nonterminating (meaning that it may be repeatedly called indefinitely, not that any single call is nonterminating); an expression like `forever (release 0)` can have any type, even though there is no `spend`.

CONTRACT NAMES

Escrows (and contracts) are abstract functions but are declared via concrete data types, their *contract names*, which are instances of the class

```
class
  (HasEscrowIDs a, Typeable (ArgType a), HasEscrowIDs (ValType a)) =>
  ContractName a where

  type ArgType a
  type ValType a
  theContract :: a -> Contract (ArgType a) (ValType a)
```

ESCROW CREATION

Contract names are used in the escrow creation function:

```
newEscrow :: (ContractName name, MonadTX m) => name -> m (EscrowID name)
```

The `name` must normally be given a `ContractName` instance in advance, but as a convenience, a `Contract argType valType` may be used directly with the obvious instance, so that the contract function (normally `theContract name`) can be given inline as a lambda expression. This disables the import/export feature, however, and so is considered a form of unfinished prototyping.

USING ESCROWS

Escrows, but not contracts, can be called inside the Fae monads using the API

```
useEscrow ::
  (ContractName name, HasEscrowIDs (ArgType name), MonadTX m) =>
  [Assignment] -> EscrowID name -> ArgType name -> m (ValType name)
```

The `Assignment` list renames the signer roles and materials names. An assignment is of one of the following forms:

```
newSignerRole <-| oldSignerRole      or  newSignerRole ↤ oldSignerRole     (↤ = U+21a
newMaterialName <=| oldMaterialName  or  newMaterialName ⇤ oldMaterialName (⇤ = U+290
newMaterialName *<- materialValue    or  newMaterialName ⇜ materialValue   (⇜ = U+291
```

(the Unicode variants are cosmetic). The first two ones extract a value at an existing name and assign it a new name; the third one creates a new name with a new value; this may be necessary to operate escrows that expect certain values to be present as materials and not arguments. Any names that are not explicitly assigned in this list are not present in the escrow call, in order to prevent escrows from having backdoors that are activated by materials inherited from the current contract code's own caller (and thus are not under the control of the current code's author).

The `EscrowID` must refer to an escrow in the current context; once these two parameters are passed, `useEscrow` is an ordinary call to the escrow function.

Contracts

CONTRACT CREATION

Contracts, similarly to escrows, are created by a call to the function

```
newContract ::
  (ContractName name, Read (ArgType name), Exportable (ValType name), MonadTX m) =>
  name -> m ()
```

which may occur in the `body` of a transaction or in the contract function of one of the input calls. The `Exportable` class will be described below. The argument is an instance of `Read` in order to parse the argument literal in the input call. As for escrows, a lambda expression may serve as the contract name at the cost of disabling the export feature.

INPUT CONTRACT CALLS

In the transaction message, contract calls are declared in a block of the form:

```
inputs
  contractID : argument
  ...
```

The inputs are implicitly numbered starting at 0 according to their order in this list. Multiple calls to the same contract ID are permissible and are resolved in numerical order, which can affect the contract's internal state on subsequent calls. Contracts may only be called in the transaction message, unlike escrows, which may never be called in the transaction message.

A contract ID is a "path" of the form `<transaction ID>/<transaction part>/<output #>/<contract version>`, identifying the exact point of origin of this contract (the location of the `newContract` call that created it), where:

- The `<transaction id>` is the unique identifier of the transaction where the contract was created;
- The `<transaction part>` is either `Body` or `Input n`, where the former indicates that the `newContract` was called in the transaction's `body`, and the latter indicates that it was called during the `n`th input contract call.
- The `<output #>` is the index in the 0-based sequence of `newContract` calls in the `<transaction part>` where the particular call that created this contract occurs.
- The `<contract version>` is either `Current` or `Version <hex>`, where the former indicates the absence of a constraint on the contract version, and the latter indicates that the version must match the `<hex>`.

Contract calls themselves introduce a block that may contain signer name assignments:

```
contractID : argument
  newRole1 = oldRole1
  newRole2 = oldRole2
  ...
```

with the same interpretation as the `newSignerRole ↤ oldSignerRole` assignment syntax.

CONTRACT VERSIONS

Contracts are created with versions that are updated on each call by hashing with the transaction ID, and so reflect the entire call history of the contract. This version may be referenced in the `Version <hex>` alternative for the last component of the contract ID; if the given version differs from the internal one then the call fails. This is useful to ensure that the contract is in the same state as it was when the transaction was tested.

MATERIALS

Contract calls must have literal arguments but they can receive Fae-internal values from the return values of other calls via *materials*, analogous to signers' public keys. Materials are provided alongside signer assignments with a similar syntax:

```
contractID1 : argument1
   materialName = contractID2 : argument2
     ...
   ...
```

Note the indented `...` underneath the material: it functions syntactically as a contract call and can have its own indented list of assignments and materials.

The return value of the call to `contractID2` is supplied to the call to `contractID1` with the given `materialName`. Materials may also be passed directly to the transaction body in a top-level block:

```
materials
  materialName = contractID : argument
  ...
```

Materials can be accessed in the Fae monads via the APIs

```
lookupMaterial :: (MonadTX m, Typeable a) => String -> m (Maybe a)
material :: (MonadTX m, Typeable a) => String -> m a
materials :: (MonadTX m, Typeable a) => m (Map String a)
```

which function similarly to the like-named functions for signers, except for the type variable `a`, whose purpose is to specify the type of material sought; an exception is raised in the event of a mismatch in `material`, and `Nothing` is returned by `lookupMaterial` in the same event. The `materials` function is intended to be useful in transaction bodies that logically would accept a list of arguments with indeterminate length, which would be generated when the transaction message is written and used with a single generic `body` (also avoiding the need to have a `body` with a very large number of function arguments). It never throws an exception, but may return the empty map.

Supplied server and client

The documentation above assumes the `faeServer` implementation of Fae and its matching `postTX` utility for submitting transactions (via an HTTP POST, hence the name).

FAESERVER

The general command-line operation of `faeServer` is described in its documentation, printed with the `--help` flag.

Operationally, `faeServer` is an HTTP server accepting connections by default on port 27182, and 27183 for import/export requests. It handles POST requests with the following structure:

- A `parent` parameter, indicating the transaction ID of the parent for the current transaction (any may be specified). If this field is absent it selects the one at the end of the “best chain”: the chain that became the current longest one at the earliest time.
- A `view` parameter, indicating the ID of an existing transaction in order to view its results, rather than submitting a new transaction.
- A `lazy` parameter, which is boolean (default `False`), indicating that the new transaction should be processed but the results not evaluated. The response when this parameter is set is a simple message `Transaction`

`<transaction ID>`.

- A `fake` parameter, also boolean (default `False`, except in Faeth mode, when it is always `True` no matter how specified), indicating that the transaction should be run one-shot and then dropped. In this mode, public-key signer declarations are allowed and the corresponding roles are given those keys without verification.
- A `reward` parameter, also boolean (default `False`), indicating whether the transaction is passed a `Reward` parameter. This simulates such a feature that might be present in a blockchain-based Fae.
- An attached file with parameter `body`; the filename is ignored. This is the source code defining the `body` function.
- Any number of attached files with parameter `other`; the filenames should match the module names. These are the source code for any modules intended for import by this or other transactions.
- An attached file with parameter `message`, whose filename is ignored, containing the formal transaction message structure.

Except when `lazy` is specified, the response is a JSON structure describing the results of the transaction and all its contract calls, which is displayed as described in the `postTX` help message.

Import/export requests have the following simpler structure:

- A `parent` parameter, with the meaning that the value being exported should be taken from the Fae state immediately following execution of the parent transaction, and the value being imported should be inserted into that state.
- An `import` parameter, containing a 4-tuple
- (
  <exported contract ID>,
  <Updated|Failed|Deleted status of call>,
  <list of required module names to interpret the contract name's value type>,
  <the contract name's value type>
)
- An attached file `valuePackage`, associated with `import`, containing the binary encoding of the value to be imported. The response for an `import` is the same as that for a `lazy` transaction.
- An `export` parameter, mutually exclusive with `import`, containing a pair `(<calling transaction ID>, <input number of contract call>)` indicating a particular contract call whose return value is to be exported. The response to this request is a 5-tuple containing the values for `import` and `valuePackage`.

Finally, `faeServer` may be run in “Faeth mode”, in which it receives all its transactions from an Ethereum client running in parallel, but can also receive `fake` transactions via HTTP for testing, which can set `parent`s from the history of Fae transactions carried by Ethereum.

POSTTX

This tool constructs the requests that are sent to `faeServer` as described above, and operates as described by its `--help` text.

TRANSACTION MESSAGE STRUCTURE

Formally, a Fae transaction is the following data type:

```
data TXMessage a =
  TXMessage
  {
    salt :: a,
    mainModulePreview :: ModulePreview,
```

```
    otherModulePreviews :: Map String ModulePreview,
    materialsCalls :: InputMaterials,
    inputCalls :: [Input],
    fallbackFunctions :: [String],
    signatures :: Map String (PublicKey, Maybe Signature)
  }

data ModulePreview =
  ModulePreview
  {
    moduleDigest :: Digest,
    moduleSize :: Integer
  }

data Input =
  InputArgs
  {
    inputCID :: ContractID,
    inputArg :: String,
    inputRenames :: Renames,
    inputInputs :: InputMaterials
  } |
  InputReward

type Module = ByteString
type ModuleMap = Map String Module
```

This has four principal aspects:

- The calling data: `materialsCalls`, `inputCalls`, and `fallbackFunctions`. The `Input` should always be `InputArgs`, as the `InputReward` variant is only generated by the reward mechanism.
- The module data: `mainModulePreview` and `otherModulePreviews`. These are defined in terms of the `ModulePreview` type, which records not the contents of the module but data usefully identifying it: its unique digest and its size in bytes. This data can be used by a Fae instance to decide whether to accept a transaction (due to resource constraints) and to validate the source files it does accept. The source files themselves should be transmitted in parallel with the message.
- The signature data: `signatures`, possibly containing the signatures of each of the named roles, but definitely containing the public key that these roles are required to use. In "production", a transaction message is invalid if these signatures do not validate against the given public keys, but in testing it is permissible for signatures to be absent, as the public key is still known.
- The "salt": `salt`, whose type is polymorphic. For `faeServer` in normal mode, this is just a `String` containing a timestamp, while in Faeth mode, it contains other data that is described there. The timestamp helps ensure that transaction messages that might otherwise contain identical data (but operate differently due to the effects of evolving historical state) will have distinct transaction IDs.

TRANSACTION ID

The *normalized* version of a transaction message is obtained from it by setting all signatures to `Nothing`. The transaction ID is the digest of this normalized message, and the signatures are all derived against this digest: thus, the ID depends on the identities of the signers (as well as everything else), and not on the exact signatures, which in many digital signing schemes can involve a nondeterministic element; likewise, the signatures may be added in any order, as they do not affect the data that must be signed.

Import and export

In its normal mode of operation, Fae requires the presence of the full transaction history, so that contract creations and calls are queued up for lazy evaluation. An instance may elide part of this history by receiving just the *return value* of a contract call out-of-band with the ordinary transactions. No Fae side effects are performed by importing this value, so contract creations that occur in the actual call are not performed by an imported return value. However, the value's backing escrows are passed with it, so that it is usable in the normal way. This feature is completely invisible to Fae transactions, which are the same messages regardless of imported values, but import and export hooks (not described here) are provided for use in `faeServer` (or any other implementation of Fae) to handle these values.

Export is enabled by the class

```
class (Typeable a) => Exportable a
```

that, like `HasEscrowIDs`, is opaque and cannot be explicitly instantiated, but has default instances for `Generic`, primitive, and `Container`-wrapped types. The functionality of this class is to convert values of its instance types to and from a binary encoding, including their backing escrows; as a special case, function types are `Exportable` but any attempt to use the exported value throws an exception. This class appears as a constraint on `newContract` (but not `newEscrow`), so valid contracts may be higher-order functions that return values containing other functions, though this defeats the export mechanism.

The class is *not* a constraint on the `ContractName` class, but backing escrows are exported via their contract names, so any escrows that represent transmissible valuables must have `Exportable` contract names. Currently, there is an additional restriction that an escrow may only be exported if it is in its *initial state*, never having been called, because the contract name represents initial data that does not describe the escrow's state after being used. (Removing this restriction is a high priority.)

FAESERVER IMPORT/EXPORT OPERATION

There is only one parameter to `faeServer` that affects imports and exports, the `--import-export-port=`, which defaults to 27183. `faeServer` is always listening on this port even if the option is missing.

POSTTX IMPORT/EXPORT OPERATION

To perform complete export/import from one `faeServer` to another, run `postTX` with the two arguments `--import-host=` and `--export-host=`, which set the hostname and possibly port of the two `faeServer` instances. The main argument to `postTX` is then of the form `<transaction ID>:<input #>`, indicating which input call in which transaction is to be exported. This value is then available for this, and only this call in the target instance.

It is often necessary for certain source files to be present in the target to interpret the imported value (specifically, any module defining a type appearing in the full type of this value). These files, which occur in various previous transactions, can be transferred using a separate mode `--resend`, which causes `postTX` to find in its history the exact transaction whose ID is the main argument, and send it to the desired host. This can be combined with `--lazy` so that the target instance can get the source files it contains without running the transaction itself (which is the point of importing a return value).

Faeth

Fae has a limited ability, dubbed *Faeth*, to exist on the Ethereum blockchain and to interact with Ethereum transactions. Like import/export, this feature is invisible to Fae code, but it does have consequences for the meta-coding that gives transactions practical meaning.

FAETH BLOCKCHAIN

Fae alone is not a blockchain, but a smart contract engine, like a standalone EVM. Faeth embeds it within the Ethereum blockchain: Faeth transactions are a particular type of Ethereum transaction, and so synchronize Fae with Ethereum, taking the Fae transaction ordering from the one obtained by consensus via the Ethereum blockchain.

FAETH TRANSACTION STRUCTURE

Faeth transactions are Ethereum transactions with a Fae transaction message embedded within their `data` field. The Fae transaction has a `salt` that includes data relevant to Ethereum:

```
data Salt =
  Salt
  {
    ethArgument :: ByteString,
    ethFee :: Maybe HexInteger,
    ethRecipient :: Maybe EthAddress,
    faeSalt :: String
  }
```

The `ethArgument` is of course the sequence of bytes that would otherwise go in the Ethereum `data` field and be interpreted by the contract it calls. Because of its position as the first field in the first field of the Fae message that is actually put in `data`, it appears verbatim as the initial portion of `data`, and that contract can still process it obliviously to Fae, so long as it expects an argument that is self-delimiting (i.e. is not just a stream of bytes without a declared length). This applies to any Solidity contract, for instance.

(The length of the `ethArgument` is stored as a single machine word shifted to the end of `data`, so that the `Salt` value can be parsed but the `data` seen by the Ethereum contract is still the expected byte string, which may not be length-prefixed in the same way.)

The other two Ethereum fields represent constraints on the `value` and `to` fields of the Ethereum transaction; these constraints are optional. If present, the Fae transaction is invalid unless they match the values in the Ethereum transaction; because Faeth does not modify Ethereum's operation at all, the reverse cannot be true. Their presence establishes an economic exchange: the Ethereum transaction, to run with the required argument and making the required payment to the required contract or address; the Fae transaction, to do whatever it is to do in return. A failure due to a mismatch in these fields will still allow the Ethereum portion to run, so it is strictly detrimental to the Ethereum-based party.

The `faeSalt` field is the same one that appears as the entire `salt` in a `faeServer` transaction: a timestamp for uniqueness.

FAESERVER FAETH OPERATION

The `faeServer` daemon operates in Faeth mode if started with the `--faeth-mode` command-line argument. In this mode, it communicates with a Parity client (whose location is controlled by the `--faeth-hostname=` and `--faeth-port=` arguments), subscribing to new blocks via JSON-RPC and scanning their transactions for those containing a Faeth-formatted `data` field. This operation builds the Fae transaction sequence, but does not submit transactions; this is the job of `postTX`.

In Faeth mode, `faeServer` continues to accept transactions submitted directly by `postTX`, but considers them all to have been sent with the `--fake` flag, and so never enters them into its transaction sequence. This is purely for testing purposes, or to use the other features of `postTX` such as `--view` or import/export.

POSTTX FAETH OPERATION

The `postTX` tool has several arguments for interacting with Faeth; it expects to do so by interacting with a Parity client as well, which may or may not be the same as the one that `faeServer` is listening to. Transactions submitted via Faeth are propagated via Ethereum, so any Ethereum client may receive a new one. This allows multiple parties to interact:

- Party A creates a dummy Faeth transaction with the correct Fae transaction inside, including its `Salt`, but with Party B's signature missing from it and with incorrect `to` and `value` fields (likely ones that have no effect, such as Party A's own address and 0, respectively). The Fae transaction is therefore invalid and not run, but can be received by Party B via Ethereum's networking.
- Party B then signs the Fae transaction with their private key, corresponding to the required public key it already contains, and adds the correct Ethereum parameters. The transaction then goes through as intended in both Fae and Ethereum.

The `postTX` parameters that implement this flow are:

- `--faeth-fee=` and `--faeth-recipient=`, and `--faeth-eth-argument=`, which Party A can use to set the corresponding fields of the `Salt`;
- `--faeth-eth-value=` and `--faeth-eth-to=`, which set the corresponding fields of the Ethereum transaction and can be used in the “Party B” scenario above. Syntactically, `--faeth-eth-argument` belongs with these, but since it affects the `Salt`, it is unchangeable after the Fae transaction is first created (and signed by the author).
- `--faeth-add-signature=`, which Party B can use to add a signature to the embedded Fae transaction. This affects the Ethereum transaction ID but not the Fae one.