Summary

TZIP-015 proposes a standard for a transferlist interface: a lightweight permission schema suitable for asset allocation and transfer.

This schema is versatile and can either be used as part of a token contract (i.e. in a *monolith* configuration) or as a separate contract that's queried on demand.

This document provides an overview and rationale for the interface, which is implemented in Lorentz, SmartPy, and partially in LIGO.

Abstract

Token contracts often need to control which users can perform transfers, especially when representing permissioned assets such as digital securities. This often takes form on-chain as user "*transferlists*", i.e. lists of which users may be the sender/receiver of a transfer.

Common features include the following:

- Many users share the same permissions
- Some privileged "issuer" account distributes funds to new users
- The lists are updated using granular changes

Some optimizations implemented:

- Many users are assigned a single transferlist
- Outbound and inbound transferlists can be combined: we use just outbound transferlists
    - E.g. to allow transfers to `X` from all transferlists, allow outbound transfers to `X` on all transferlists.
- Outbound transferlists are updated by providing a "patch" of `transferlistId`'s to add and remove.
    - This allows "big" (using `big_map`'s) and "small" (using `map`'s and `set`'s) implementations to use the same interface.
- There is an issuer, who may transfer to any user that can accept transfers: this simplifies the state and is extensible:
    - The issuer can be disabled by setting the `userId` to an account that otherwise can't perform transfers, e.g. an empty contract.
    - The issuer can be split into multiple accounts by using a multisig or a "permissioned proxy", i.e. a contract acts as the issuer and forwards authenticated contract calls.
- Calling `FAILWITH` on error allows a contract to check transferlisting with a single call to `TRANSFER_TOKENS`
- Key operations to update state are commutative (up to primary keys), viz. `updateUser` and `updateTransferlist`, which allows easier batching of updates.

Specification

`userId`

`userId` is a comparable type representing a user.

The `userId` will normally be `address`, but other types can be useful:

- `nat` can be used when multiple `addresses` may be associated with a single user
- `key` (public key) can be used when users provide explicit `signatures`
- `bytes` can to avoid some overhead in parsing/preprocessing other types

Abstract storage specification

This is how the contract must behave, but no particular Michelson types or layouts are specified.

The transferlists are split up into two "tables":

- `users`:
  - `key: userId`
    - Each user has exactly one `transferlistId`
  - `val: transferlistId`
    - `transferlistId` is a `nat`
- `transferlists`:
  - `key: transferlistId`
  - `val: (unrestricted, allowedTransferlists)`
    - `unrestricted` is a `bool`. if it's `False`, the transferlist is restricted and its users will fail `assertReceivers`, `assertTransfers`. A restricted transferlist will generally behave as if its `allowedTransferlists` is empty.
    - `allowedTransferlists` is a set of `transferlistId`
      - Note: `allowedTransferlists` is not necessarily a Michelson `set` in the contract's storage, e.g. it could be implemented using a `big_map`.

In short, user `X` may transfer to user `Y` if `Y`'s `transferlistId` is in `X`'s transferlist's set of `allowedTransferlists` and both transferlists are unrestricted.

The storage also contains one variable:

- `issuer`: `userId`

The issuer is treated as a user who:

- Can't be explicitly added to `users`
- Is always `unrestricted`
- Whose `allowedTransferlists` is the set of ALL `transferlistId`'s

Ways to use the interface

The interface has three types of entrypoints: assertion, management, and informative.

It may be used in two primary ways:

- As a compile-time wrapper
  - In this case, the entire transferlist contract is inlined into another contract, e.g. a token contract.
  - The assertion entrypoints are used like library functions in the other contract.
  - The management and informative entrypoints are REQUIRED.
- As an on-chain wrapper

- In this case, the transferlist contract is deployed separately from any contract using it, e.g. a token contract.
- The assertion entrypoints are called from the other contract, without requiring callbacks since they call `FAILWITH` when they fail.
- The assertion, management and informative entrypoints are REQUIRED.

Entrypoints

Assertion

These entrypoints MUST be exposed and callable by arbitrary `address`es, except when the transferlist is used as a compile-time wrapper.

When one of these entrypoints fails, it must use the `FAILWITH` Michelson instruction.

`assertReceivers`

Succeed if and only if, for each `userId` in the list:

- The given `userId` is in `users`, and thus has a `transferlistId`
- The associated `transferlistId` refers to an existing, `unrestricted` transferlist

```
(list %assertReceivers userId)
```

`assertTransfers`

Succeed if and only if, for each `from` and their associated `to`'s in the list, either:

- Both
  - `assertReceivers` would succeed for both the `from` and `to` `userId`'s
  - `to`'s `transferlistId` is in the `set` of `from`'s transferlist's set of allowed outbound transferlists
- Or both:
  - `assertReceivers` would succeed for the `to` `userId`
  - The `from` `userId` is the `issuer`'s `userId`

This is equivalent to the following pseudocode:

```
def assertTransfers(input_list):
  for from, tos in input_list:
    for to in tos:
      if from == issuer:
        assertReceivers [to]
      else:
        assertReceivers [from, to]
        users.get(to) in transferlists.get(users.get(from)).allowedTransferlists
```

```
(list %assertTransfers (pair (userId %from)
                             (list %tos userId)))
```

See FA2's `transfer` for an example of a similarly batched entrypoint.

Examples

Consider the following setup where `userId` is `string`:

Users:

- `"alice": 0`
- `"bob": 0`
- `"charlie": 1`
- `"dan": 2`

Transferlists:

- `0: (unrestricted: True, allowedTransferlists: {0, 2})`
- `1: (unrestricted: True, allowedTransferlists: {1})`
- `2: (unrestricted: False, allowedTransferlists: {1, 2})`

Then suppose the following call to `assertTransfers` were made:

```
assertTransfers
  { Pair "alice" { "bob", "dan" }
  , Pair "bob" { "alice" }
  , Pair "charlie" { "charlie", "dan" }
  }
```

- `alice -> bob`: `alice` and `bob` are on the same transferlist (`0`), which contains itself in its `allowedTransferlists` and is `unrestricted`, so this succeeds
- `alice -> dan`: `alice` is on a transferlist (`0`) that contains `dan`'s `transferlistId` (`2`) in its `allowedTransferlists` and is `unrestricted`, but it fails because `dan`'s transferlist is restricted
- `bob -> alice`: This succeeds by the same logic as `alice -> bob`: they're on the same `unrestricted` transferlist that contains its own `transferlistId` in its `allowedTransferlists`
- `charlie -> charlie`: This succeeds since `charlie`'s transferlist is unrestricted and contains its own `transferlistId` in its `allowedTransferlists`
- `charlie -> dan`: This fails because `dan`'s transferlist (`2`) is restricted

Thus the above call to `assertTransfers` will fail.

Management

These entrypoints MUST be exposed, but need not be callable by arbitrary `userId`'s.

For example, they all may be callable by a single administrator address (not necessarily the issuer).

`setIssuer`

Set the issuer's `userId`

```
(userId %setIssuer)
```

`updateUser`

Add, update, or remove a user:

- To add or update a user, provide `Some transferlistId`
- To remove a user, provide `None` for `transferlistId`
- This must fail with `FAILWITH` if the issuer's `userId` is provided.
- This must NOT fail if the `transferlistId` is NOT in `transferlists`. In other words, it must be possible to create the transferlist *after* calling `updateUser`.

```
(pair %updateUser (userId %user)
                  (option (nat %transferlistId)))
```

`updateTransferlist`

Add, update, or remove a transferlist:

- To add or update a transferlist, provide `Some`:
  - `disallowTransferlists` is a `list` of `transferlistId`'s to remove from the `allowedTransferlists`
  - `allowTransferlists` is a `set` of `transferlistId` to add to the `allowedTransferlists`
  - NOTE: `disallowTransferlists` *must* run before `allowTransferlists`. In other words, if a `transferlistId` is in both `disallowTransferlists` and `allowTransferlists`, it will be idempotently added to `allowedTransferlists`.
- To remove a transferlist, provide `None` for the `option`
- This must NOT fail if any `transferlistId` provided in `allowedTransferlists` is NOT in `transferlists`. In other words, it must be possible to create the transferlists in `allowedTransferlists` *after* calling `updateTransferlist`.

```
(pair %updateTransferlist (nat %transferlistId)
                          (option (pair (bool %unrestricted)
                                        (pair (list %disallowTransferlists nat)
                                              (set %allowTransferlists nat)))))
```

Informative

These entrypoints MUST be exposed and callable by arbitrary `userId`'s.

`getIssuer`

Get the issuer's `userId`

```
(pair %getIssuer unit
                 (contract %callback userId))
```

`getUser`

Get `Some user`'s `transferlistId` if and only if the provided `userId` exists in `users`, or `None` otherwise

```
(pair %getUser (userId %user)
               (contract %callback (option %transferlistId nat)))
```

`assertTransferlist`

Succeed if and only if:

- `Some` is provided and:
  - The given `transferlistId` exists in `transferlists`
  - The given `unrestricted bool` matches its `unrestricted` state
  - The given `allowedTransferlists` is a subset of its `allowedTransferlists`
- `None` is provided and the `transferlistId` does *not* exist in `transferlists`

```
(pair %assertTransferlist (nat %transferlistId)
                          (option (pair (bool %unrestricted)
                                        (set %allowedTransferlists nat))))
```

Test Cases

Test cases for each entrypoint:

- Implemented using Lorentz, may be found here

NOTE: These test cases reflect the version of this TZIP *before* `1720bd90b55e6ba8d7667c643cefa7929282fdb8`.

- Implemented using Taquito, may be found here

Implementations

- An implementation of compile-time wrapping and separate-contract transferlists in Lorentz may be found here

NOTE: These implementations reflect the version of this TZIP *before* `1720bd90b55e6ba8d7667c643cefa7929282fdb8`.

- A partial implementation of a compile-time wrapper in LIGO may be found here
- An implementation of the separate-contract transferlist in SmartPy may be found here

Copyright