1 Specification

1.1 Syntax

$C \in \textsc{ContractNames}$ $\quad$ $T \in \textsc{TransactionNames}$

$t \in \textsc{TypeNames}$ $\quad$ $x \in \textsc{IdentifierNames}$

$n \in \mathbb{Z}$

$$\begin{array}{lll} \mathcal{Q} & ::= & \texttt{set} \mid \texttt{option} \mid \texttt{one} \mid \texttt{list} \qquad \text{(type quantities [Not sure what to call these])} \\ \tau & ::= & \texttt{bool} \mid \texttt{nat} \mid \mathcal{Q}\ \tau \mid \tau \times \tau \mid \tau \rightsquigarrow \tau \mid \{\overline{x : \tau}\} \mid t \qquad \text{(types)} \end{array}$$

Figure 1: Abstract syntax of LANGUAGE-NAME.

Note: Consider $\tau \equiv \texttt{one}\ \tau$, so $x : \texttt{nat}$ and $x : \texttt{one nat}$ are the same (and so is $x : \texttt{one one nat}$, etc.). The syntax $S : \tau \rightsquigarrow \sigma$ means S accepts τ and provides σ; that is, you can flow τ into it, and when you flow out of it, you get σ.

1.2 Statics

Some flow typing rules:

[How to handle transfer everything making it safe to ignore things?]

$$\frac{\Gamma \vdash x : \mathcal{Q}\ \tau}{\Gamma \vdash x\ \texttt{selects}\ \tau}\ \textsc{Select-Var} \qquad \frac{}{\Gamma, x : \tau \vdash x : \texttt{demote}(\tau)}\ \textsc{Lookup-NoConsume}$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau \dashv \Gamma}\ \textsc{Lookup} \qquad \frac{\Gamma \vdash x : \tau \qquad \neg(\tau\ \texttt{asset})}{\Gamma \vdash x : \tau \dashv \Gamma}\ \textsc{View}$$

$$\frac{\Gamma, x : \tau \vdash e : \sigma \dashv \Gamma}{\Gamma \vdash (\lambda x : \tau . e) : \tau \rightsquigarrow \sigma}\ \textsc{Flow-Transformer} \qquad \frac{}{\Gamma, S : \mathcal{Q}\ \tau \vdash S : \tau \rightsquigarrow \tau}\ \textsc{Flow-Storage}$$

$$\frac{\Gamma \vdash S : \tau \rightsquigarrow \sigma \qquad \Gamma \vdash f\ \texttt{selects demote}(\sigma) \qquad \Gamma \vdash D : \sigma \rightsquigarrow \pi}{\Gamma \vdash S \xrightarrow{f} D : \tau \rightsquigarrow \pi}\ \textsc{Flow}$$

[Probably should only be possible to transfer resources via. "external" calls to calls to the same contract.] [We could also guard on things like $A \to B; A \to B$ where $A : \text{one}\ \tau$, because the second one will always fail. Not sure if we care to put this in the type system, or just have a linting check. It's safe either way because of dynamic checks. Probably would be nice to check for, we need to worry about this for all $\mathcal{Q}$ except for `set`; but also only for nonfungible types. Fungible types can be flowed from infinitely many times, the later flows just don't do anything. That seems like a better candidate for a lint check.]

Demote definition **[TODO: Finish]**

$$
\begin{aligned}
\texttt{demote}(\texttt{nat}) &:= \texttt{nat} \\
\texttt{demote}(\texttt{bool}) &:= \texttt{bool} \\
\texttt{demote}(t) &:= \texttt{demote}(\tau) && \text{where } t \text{ is } \texttt{asset}\ \tau \\
\texttt{demote}(\mathcal{Q}\ \tau) &:= \mathcal{Q}\ \texttt{demote}(\tau)
\end{aligned}
$$

[Selecting from or with a list should yield a list, probably] [Asset retention theorem?] [Resource accessiblity?] [NOTE: Transformers must be consumed, because they might be holding resources if partially applied. This is just like any asset local variable.]

2 Introduction

LANGUAGE-NAME is a DSL for implementing programs which manage resources, targeted at writing smart contracts.

2.1 Contributions

We make the following main contributions:

- **Safety guarantees**: similar to **[or maybe just exactly]** linear types**[or maybe uniqueness types? need to read more about this]**, preventing accidental resource loss or duplication. Additionally, provides some amount of reentrancy safety.
  - We can evaluate these by formalizing the language and proving them; the formalization is something that would be nice to do anyway.
- **Simplicity**: The language is quite simple—it makes writing typical smart contract programs easier and shorter, because many common pitfalls in Solidity are automatically handled by the language, such as overflow/underflow, checking of balances, short address attacks, etc.
  - We can evaluate these by comparing LOC, cyclomatic complexity, etc. Not sure what the right metric would be. **[Or how cyclomatic complexity would work exactly in this language.]**
  - We can also evaluate via a user study, but that will take a long(er) time.
- **[Optimizations?]** Some of the Solidity contracts are actually inefficient because:
  1. They use lots of modifiers which repeat checks (see reference implementation of ERC-721).
  2. They tend to use arrays to represent sets. Maybe this is more efficient for very small sets, but checking containment is going to be much faster with a `mapping (X => bool)` eventually.
  - We can evaluate this by profiling or a simple opcode count (which is not only a proxy for performance, but also means that deploying the contract will be cheaper).

3 Language Intro

The basic state-changing construct in the language is a *flow*. A flow describes a transfer of a *resource* from one *storage* to another. A *transaction* is a sequence of flows. **[sort of...except for when we use for loops. It's still a set, but it can change based on the inputs/current state.]**

A flow can have one of four *modes*:

1. send
2. merge
3. hold
4. consume

Each flow also has a *source*, a *destination*, and a *selector*. The source and destination are two storages which hold a resource, and the selector describes which part of the resource in the source should be transferred to the destination. A flow may optionally have a *name*.

Note that all flows fail if they can't be performed. For example, a flow of fungible resources fails if there is enough of the resource, and a flow of a nonfungible resource fails if the selected value doesn't exist in the source location.

NOTE: If we wanted to be "super pure", we can implement preconditions with just flows by doing something like:

```
consume true from { contractCreator = msg.sender }
```

This works because `{ contractCreator = msg.sender } : set bool` (specifically, a singleton), so if `contractCreator = msg.sender` doesn't evaluate to true, then we will fail to consume true from it. I don't think actually doing this is a good idea; at least, not in the surface language. Maybe it would simplify the compiler and/or formalization), but it's interesting/entertaining.

Actions In addition to flows, transactions may contain *handlers*, which are a pair of a *trigger* and an *action*. Triggers specify when an action should be executed. An action can be:

1. An event
2. An external call
3. An error handler

[Can you do actions other than just providing an error message in on fail?] [Sending ether will also trigger an external call, which should be considered as being part of the `on success` block, probably]

For example, below, we create a flow with a name `F`.

```
F: voterSource --- newVoterAddress --> authorizedVoters
```

We can then create handlers for this flow, such as the following:

```
on fail F with Err:
    revert("This address is already authorized! Do not re-authorize it.")
on success F:
    emit AuthorizedVoter(newVoterAddress)
    call newVoterAddress.receiveAuthorization()
        returning resultCode such that resultCode = "SUCCESS"
```

It's possible to have triggers which only occur when some subset of actions occurs. Below, we create two flows. The trigger `on fail {F1,F2}` triggers when any one of the actions `F1` or `F2` fails. The trigger `on success {F1,F2}` triggers when **both** the actions `F1` and `F2` succeed. **[Is this confusing, or is this how you would expect it to work?]**

```
F1: A --- x --> B
F2: C --- y --> D
on fail {F1,F2}:
    // Stuff
on success {F1,F2}:
    // Stuff
```

In fact, all handlers are internally translated into this form, so `on fail F` becomes `on fail {F}`, and `on fail` becomes `on fail {F1,F2 ,..., Fn}`, where `F1`, `F2`, ..., `Fn` are all the flows preceeding the handler. **[I think this is the right way to do it.]**

We can also allow the following syntax to be used, with minimal additional implementation troubles:

```
handle {
    A --- x --> B
    C --- y --> D
    E --- z --> F
} with {
    on fail :
        // Stuff
    on success:
        // Stuff
}
```

This can be mechanically rewritten to:

```
F1: A --- x --> B
F2: C --- y --> D
F3: E --- z --> F
on fail {F1,F2,F3}:
    // Stuff
on success {F1,F2,F3}:
    // Stuff
```

[random note] The following pattern, while popular in Python **[and probably some other languages]**, is not possible with this system.

```
try
    A --- x --> B
    C --- y --> D
catch
    E --- z --> F
```

So that the second flow only happens if the first fails, you should instead just check whatever condition you're actually interested in. I don't think this will greatly impact usability, and in fact is probably easier to read because the condition has to be specified, and you won't get unexpected failures causing the flow. For example, say you want to do the third flow only when there isn't enough money in `A` and `C` for first two flows to occur. But the above implementation would also cause the money to be taken from C in the case that there's, for example, and overflow when you transfer to B, or if A has enough money, but not C, which may is desirable.

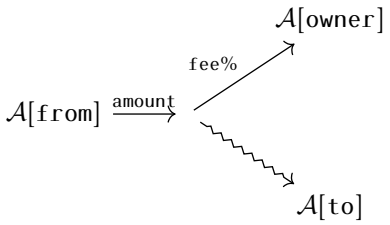
Transformers Partially applied transformers: suppose you want to flow two resources into pairs (e.g., a zip). If you use a partially applied transformer, this might cause issues for several reasons:

1. If I have a $f : \tau \rightsquigarrow \sigma \rightsquigarrow \pi$, and I flow into f, then $\sigma \rightsquigarrow \pi$ flows out. However—in what order?
2. We need a "list selector" that lets us control what order things flow in.

4 Examples

Transfer with fees This is fairly common: for example, the contract with the most transactions does this, as do many gambling/auction contracts.

`transfer(from, to, amount)`:



4.1 The DAO attack

We can prevent the DAO attack (the below is from `https://consensys.github.io/smart-contract-best-practices/known_attacks/`):

```
function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    // At this point, the caller's code is executed, and can call withdrawBalance again
    require(msg.sender.call.value(amountToWithdraw)(""));
    userBalances[msg.sender] = 0;
}
```

In LANGUAGE-NAME, we would write this as:

```
transaction withdrawBalance():
    userBalances[msg.sender] -~-> msg.sender.balance
```

Not only is this simpler, but the compiler can automatically place the actual call that does the transfer last, meaning that the mistake could simply never be made.

4.2 approveAndCall

Many token contracts include the concept of `approveAndCall`, typically similarly to the following (taken from: `0x174bfa6600bf90c885c7c01c7031389ed1461ab9`, one of the most popular contracts on the blockchain by transaction count):

```
function approveAndCall(address _spender, uint256 _value, bytes memory _extraData) public returns (bool success) {
    tokenRecipient spender = tokenRecipient(_spender);
    if (approve(_spender, _value)) {
        spender.receiveApproval(msg.sender, _value, address(this), _extraData);
        return true;
    }
}
```

In LANGUAGE-NAME, we can do the same thing by associating a call with a flow.

```
transaction approveAndCall(_spender : address, _value : uint256, _extraData : bytes):
    consume everything from allowance[msg.sender, _spender]
    approvalSource --[ _value ]-> allowance[msg.sender, _spender],
    on success: call  receiveApproval(msg.sender, _value, address(this), _extraData)
```