Stateful Smart Contracts On Cardano

Part 1. Minting Unique Tokens With A Shared `PolicyId`

Lorenzo Fanton

887857@stud.unive.it

Problem

Our goals:

1. We want to distribute a smart contract's state over mutiple UTxOs, *and*...
2. ... We need to ensure that it cannot be compromised by an attacker.

Otherwise, it would be pointless to use whatever abstractions we might come up with.

Problem

An informal statement of the problem might be the following.

> “ A smart contract's state, distributed over a set of UTxOs, should be modified only by transactions validated by the smart contract itself. ”

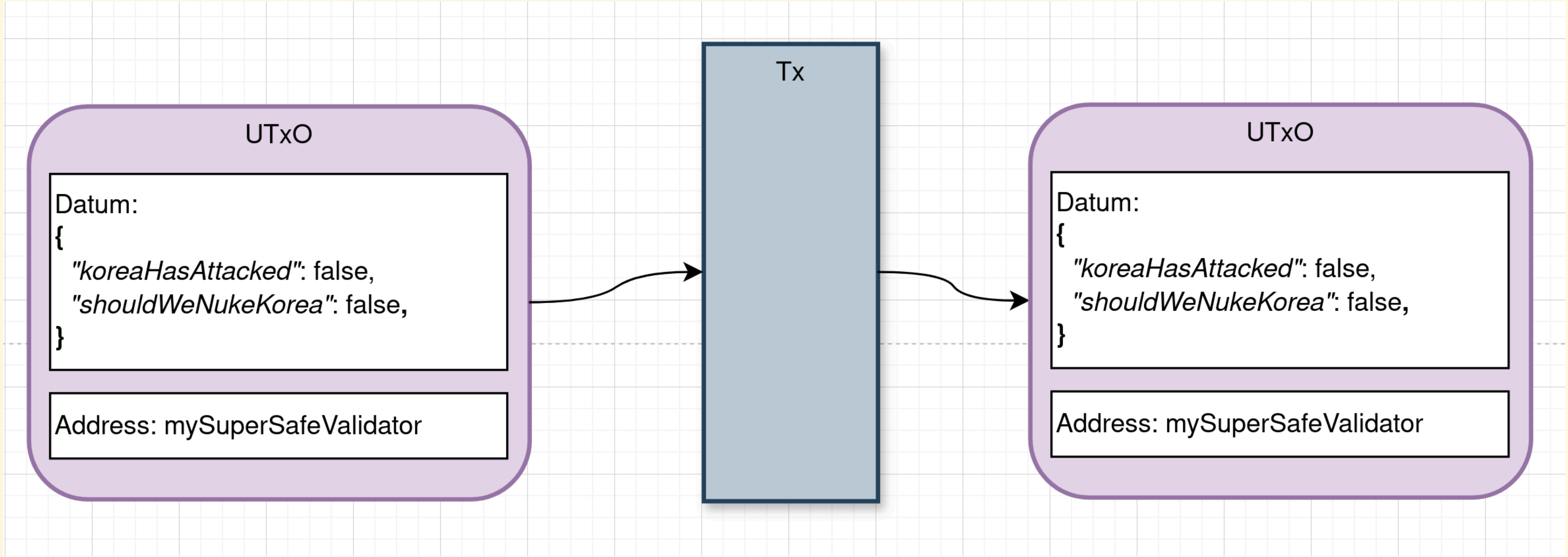
Problem

There are good news and bad news.

- **Good news**: Smart contracts (on-chain) can enforce the recipient(s) of the UTxOs being created, but...
- **Bad news**: ... They have **zero knowledge** about the sender of the UTxOs being spent.

Why is that a problem?

Example: Good State



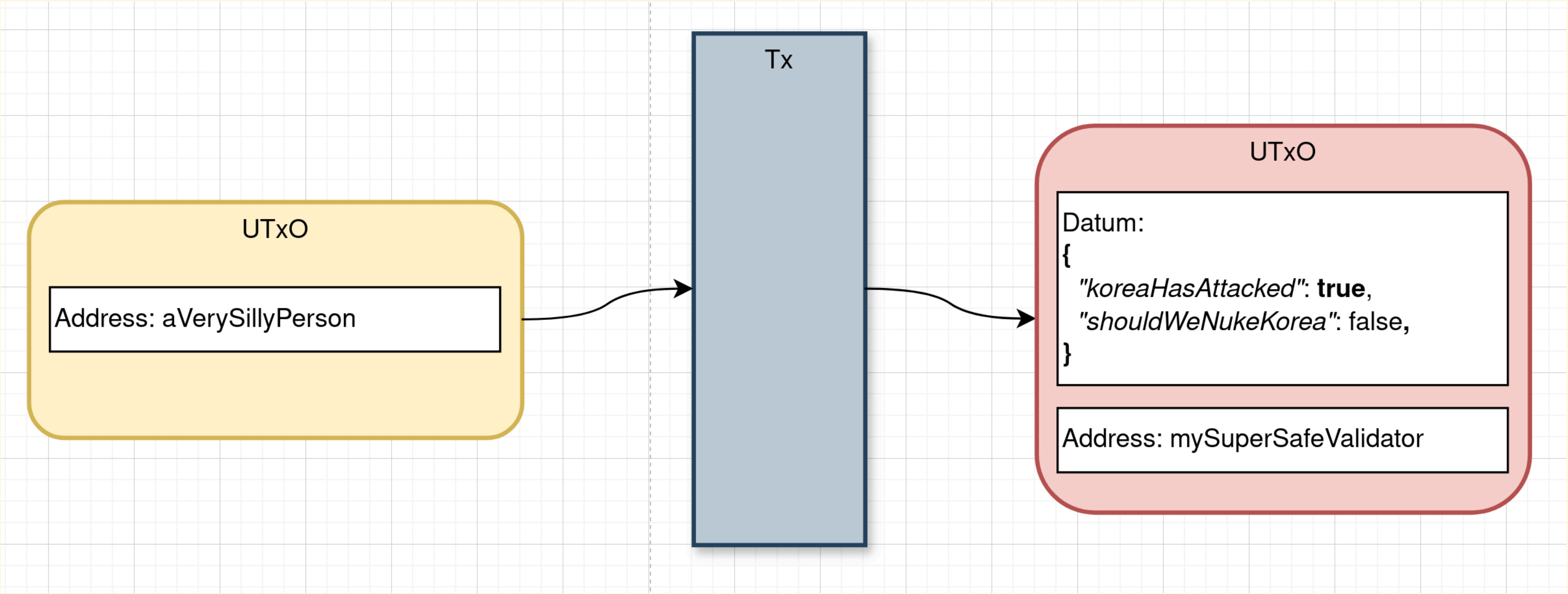
Datum:
{
"koreaHasAttacked": false,
"shouldWeNukeKorea": false**,**
}

Address: mySuperSafeValidator

Datum:
{
"koreaHasAttacked": false,
"shouldWeNukeKorea": false**,**
}

Address: mySuperSafeValidator

Example: Attacker



UTxO

Address: aVerySillyPerson

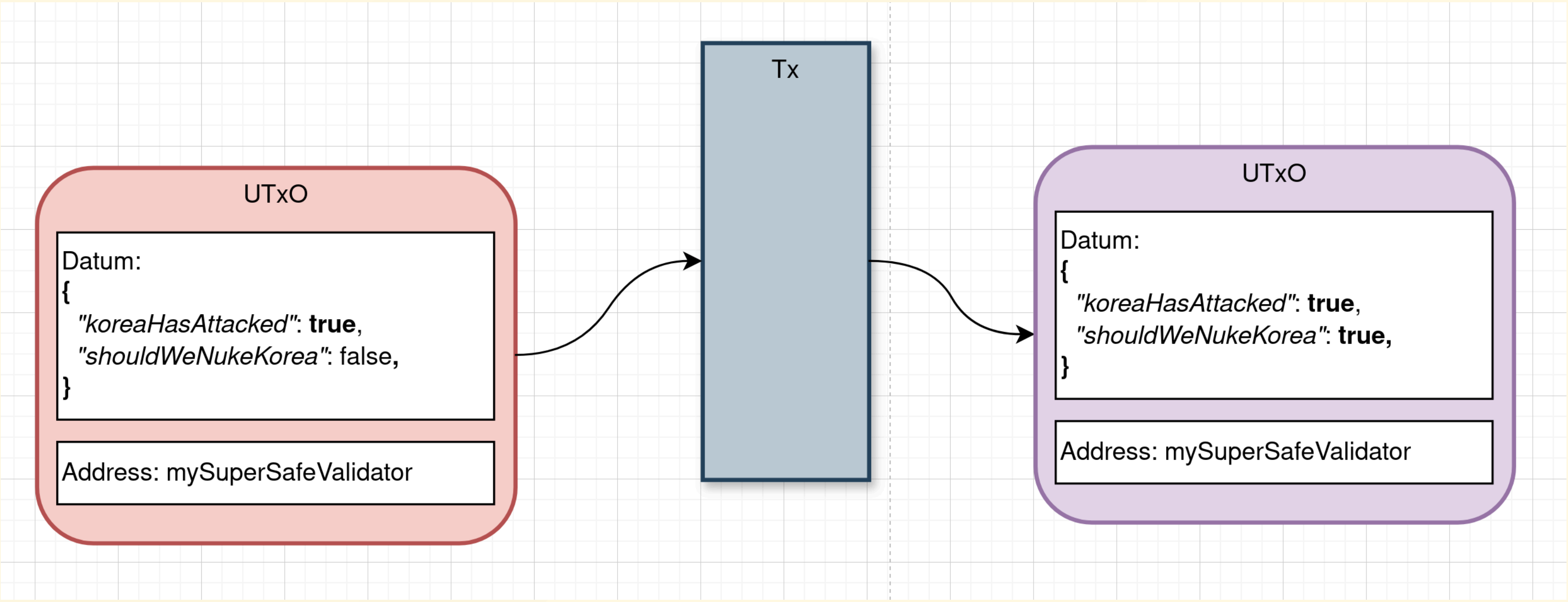
Tx

UTxO

Datum:
{
"koreaHasAttacked": **true**,
"shouldWeNukeKorea": false**,**
}

Address: mySuperSafeValidator

Example: (Very) Bad State



UTxO

Datum:
{
  "koreaHasAttacked": **true**,
  "shouldWeNukeKorea": false**,**
}

Address: mySuperSafeValidator

Tx

UTxO

Datum:
{
  "koreaHasAttacked": **true**,
  "shouldWeNukeKorea": **true,**
}

Address: mySuperSafeValidator

Problem

Compromised state could be the source of other, more serious issues.

What could be worse than WW3? Well, for example:

> *You* could be losing money!!

While total thermonuclear annihilation might be somewhat tolerable and even fun, losing money is not and we must avoid that at all costs.

Native Tokens

Native tokens, also called *assets*, are defined as:

- $\mathrm{AssetId} := \mathrm{PolicyId} \times \mathrm{AssetName}$
- $\mathrm{PolicyId} := (\mathbb{F}_{256})^{28}$ (hash of the minting validator script)
- $\mathrm{AssetName} := (\mathbb{F}_{256})^{32}$ (arbitrary string)

Where $\mathbb{F}_q := \{0, \ldots, q-1\}$.

Spending Validators

We call V the type of a Spending Validator:

$$V := (\text{Datum} \times \text{Redeemer} \times \text{ScriptContext}) \to \mathbb{B}$$

Minting Validators

We call M the type of a Minting Validator:

$$M := (\mathrm{Redeemer} \times \mathrm{ScriptContext}) \to \mathbb{B}$$

NB: Minting Validators do not take a Datum as inputs, because they do not consume UTxOs.

Authorizing NFT

Let $\mathrm{OutputReference}$ represent an on-chain reference to an UTxO.

We declare the following functions:

$$newm : (\mathrm{OutputReference}) \rightarrow M$$

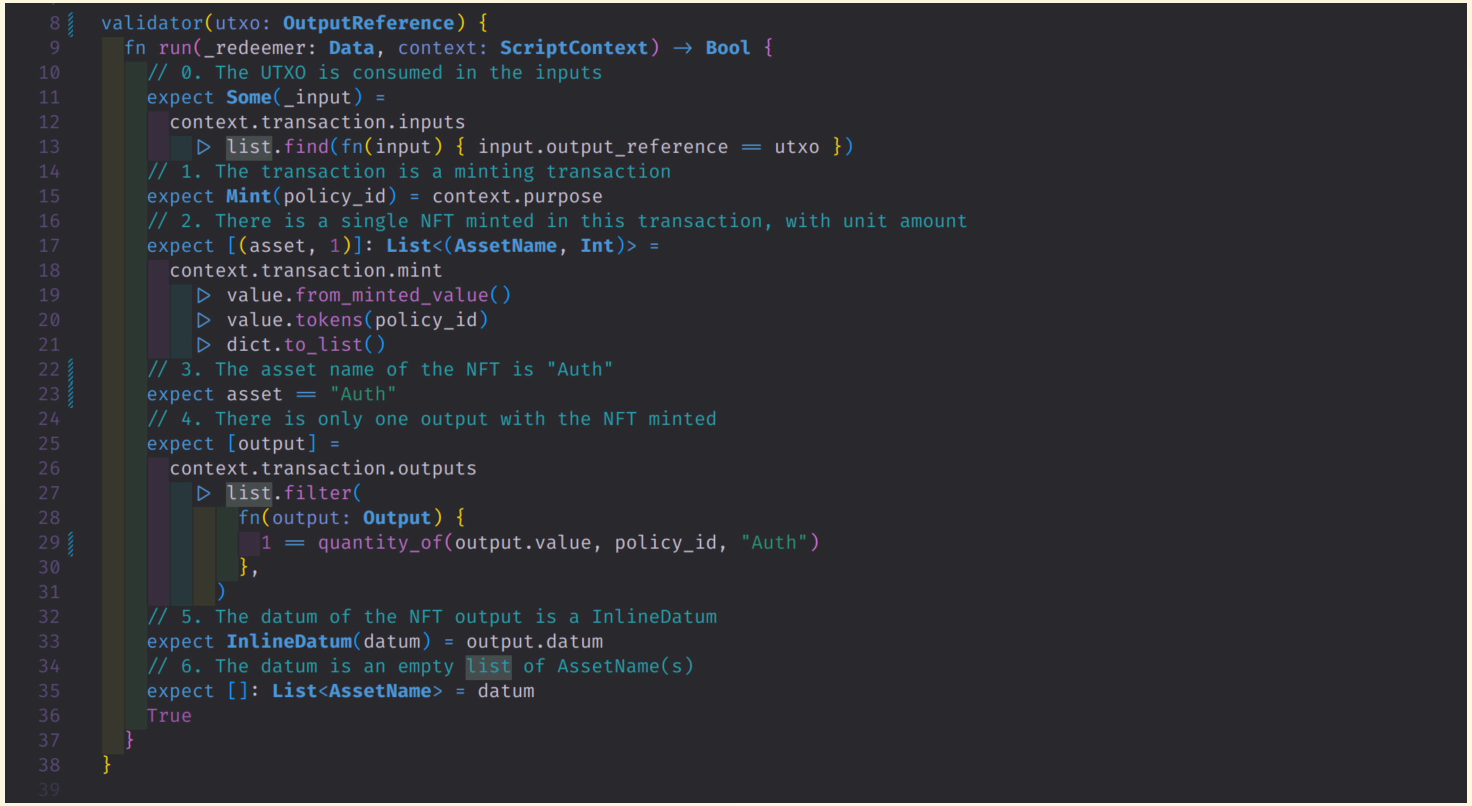
$$addv : (M \times M) \rightarrow V$$

$$addm : (M \times \mathrm{AssetName}) \rightarrow M$$

Authorizing NFT: $newm$

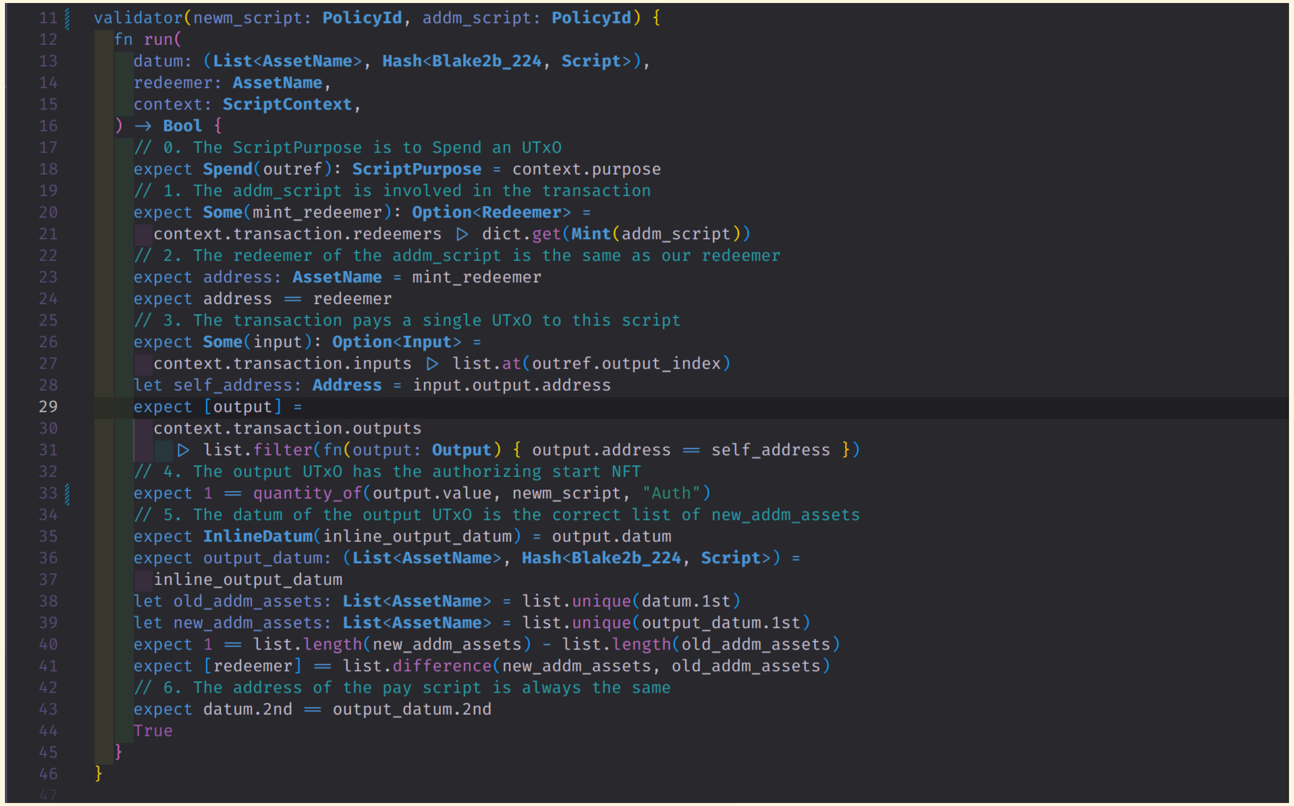
$newm$ takes an $\mathrm{OutputReference}$ `utxo` and returns a Minting Validator that produces a single token with $\mathrm{AssetName}$ `"Auth"`. The script is a one-time minting validator, because it ensures that `utxo` is consumed in the transaction (which can happen only once).

Code: *newm*



```
validator(utxo: OutputReference) {
  fn run(_redeemer: Data, context: ScriptContext) -> Bool {
    // 0. The UTXO is consumed in the inputs
    expect Some(_input) =
      context.transaction.inputs
        |> list.find(fn(input) { input.output_reference == utxo })
    // 1. The transaction is a minting transaction
    expect Mint(policy_id) = context.purpose
    // 2. There is a single NFT minted in this transaction, with unit amount
    expect [(asset, 1)]: List<(AssetName, Int)> =
      context.transaction.mint
        |> value.from_minted_value()
        |> value.tokens(policy_id)
        |> dict.to_list()
    // 3. The asset name of the NFT is "Auth"
    expect asset == "Auth"
    // 4. There is only one output with the NFT minted
    expect [output] =
      context.transaction.outputs
        |> list.filter(
             fn(output: Output) {
               1 == quantity_of(output.value, policy_id, "Auth")
             },
           )
    // 5. The datum of the NFT output is a InlineDatum
    expect InlineDatum(datum) = output.datum
    // 6. The datum is an empty list of AssetName(s)
    expect []: List<AssetName> = datum
    True
  }
}
```

Code: *addv*



```
validator(newm_script: PolicyId, addm_script: PolicyId) {
  fn run(
    datum: (List<AssetName>, Hash<Blake2b_224, Script>),
    redeemer: AssetName,
    context: ScriptContext,
  ) -> Bool {
    // 0. The ScriptPurpose is to Spend an UTxO
    expect Spend(outref): ScriptPurpose = context.purpose
    // 1. The addm_script is involved in the transaction
    expect Some(mint_redeemer): Option<Redeemer> =
      context.transaction.redeemers |> dict.get(Mint(addm_script))
    // 2. The redeemer of the addm_script is the same as our redeemer
    expect address: AssetName = mint_redeemer
    expect address == redeemer
    // 3. The transaction pays a single UTxO to this script
    expect Some(input): Option<Input> =
      context.transaction.inputs |> list.at(outref.output_index)
    let self_address: Address = input.output.address
    expect [output] =
      context.transaction.outputs
        |> list.filter(fn(output: Output) { output.address == self_address })
    // 4. The output UTxO has the authorizing start NFT
    expect 1 == quantity_of(output.value, newm_script, "Auth")
    // 5. The datum of the output UTxO is the correct list of new_addm_assets
    expect InlineDatum(inline_output_datum) = output.datum
    expect output_datum: (List<AssetName>, Hash<Blake2b_224, Script>) =
      inline_output_datum
    let old_addm_assets: List<AssetName> = list.unique(datum.1st)
    let new_addm_assets: List<AssetName> = list.unique(output_datum.1st)
    expect 1 == list.length(new_addm_assets) - list.length(old_addm_assets)
    expect [redeemer] == list.difference(new_addm_assets, old_addm_assets)
    // 6. The address of the pay script is always the same
    expect datum.2nd == output_datum.2nd
    True
  }
}
```

Code: $addm$

Examples

$$DS(\ ,\sigma = 100, \epsilon = 0.0001, \theta = 0.3, k = 2) \mapsto$$

$$DS(\ ,\sigma = 100, \epsilon = 0.001, \theta = 0.4, k = 3) \mapsto$$

$$DS(\ ,\sigma = 100, \epsilon = 0.001, \theta = 0.1, k = 3) \mapsto$$

$$DS(\ ,\sigma = 100, \epsilon = 0.001, \theta = 0.08, k = 1) \mapsto$$