Announcing tNFTs, Validity Language & Trampoline UTXO Framework

Tempest Labs

March 2022

1 Introduction

This document specifies the semantics of a new NFT design for Nervos Network called "tNFT" which is short for "Trampoline NFT". The motivation for this is to make available a drastically simple NFT standard on CKB. While both the mNFT and NRC721 designs provide plenty of capabilities, there is still a need in Nervos Network's growing NFT ecosystem for a radically simple and easy to understand design.

The first section of this document provides a semi-formal specification of the tNFT design.

The second section provides an example of tNFT implemented in Validity, a new executable specification language for UTXO based smart contracts. Validity is under active development by the Tempest Labs team (website coming Soon™).

We have also included additional material at the end of the post as an appendix, which describes a simplified formal model of the CKB transaction environment that is used in Validity's internal semantics.

2 tNFT Specification

Each rule below describes a possible constraint on any transaction which includes a Cell that implements the tNFT standard. After defining the rules, a final invariant is defined in terms of these rules, which provides a single formula that fully specifies the tNFT standard. This is similar in ergonomics to TLA+.

2.1 Rule 1

Genesis Id cannot be changed once created.

$$\forall x \exists y (x \in Inputs \land x \in Self \rightarrow GenesisId(x) = GenesisId(y))$$

2.2 Rule 2

Genesis Id is set upon creation and is derived from the hash of the first input.

$$\neg\exists x(x \in Inputs \wedge x \in Self) \wedge$$
$$\exists y(y \in Outputs \wedge y \in Self \wedge GenesisId(y) = Hash(Inputs_0))$$

2.3 Rule 3

Data field has enough bytes to store both genesis Id and content hash (even if content hash is simply zeroed out / byte array of zeroes).

$$\forall x(x \in Self \rightarrow |Data(x)| \geq 64)$$

2.4 Rule 4

This invariant specifies the necessary and sufficient conditions under which a CKB cell is considered a tNFT cell. In other words, any cell for which this formula is true is a tNFT. The disjunction of Rule1 and Rule2 captures the fact that both need not be true at the same time (and therefore, Rule1 and Rule2 both describe a different type of operation). Since Rule3 must always be true, it is added as a final conjunct.

$$(Rule1 \vee Rule2) \wedge Rule3$$

3 Validity + Trampoline: Two Sides of the Same Coin

It would be nice if we could port the simple & precise mathematical description from the above section to a friendly programming language. Such a language would provide a host of benefits for CKB & UTXO dapp developers, such as:

1. Enable declarative-style smart contract development & make "DeFi" legos more amenable to formal verification
2. Provide the foundation of an Ethereum-like ABI for the radically different UTXO programming model
3. Enable partial automated transaction generation for easier integration third party services such as exchanges, aggregators, explorers, and bridges, etc., to interoperate with any Validity-based blockchain application.

These are just some of the goals of the language we are developing, called "Validity" (due in part to the fact that it is a logic based language which cares about "valid" formula, and in part because it is phonetically so similar to "Solidity").

3.1 How Trampoline Interacts with Validity

The Trampoline framework provides multiple capabilities for interacting with Validity contracts off-chain:

1. Trampoline's Generator SDK can automatically read Validity code to generate satisfiable transactions which use one or more Validity smart contracts.

2. Trampoline's Indexer SDK can be used to build indexers that watch the chain for transactions that match an operation defined in a Validity contract, as well as deserialize the transaction's raw data into a friendlier format based on the data types defined by a Validity contract.

Interestingly, point 2 enables similar (but not the same) functionality as Ethereum's Events API, which is crucial for usability in wallets, explorers, exchanges, and dapps that interoperate with each other.

3.2 tNFT in Validity

```
def TrampolineNFT {
    def Source GenesisSeedInput {
        is Hash::Blake2b,
        From Field(OutPoint, Inputs::Index(0))
    }

    def Source GenesisId {
        is Bytes(32),
        From Self::Data::Offset(0)
    }

    def Rule IdOnCreation {
        And(
            Not(
                Some(x in Self::Inputs)
            ),
            Some(x in Self::Outputs | GensisId(x) = GenesisSeedInput)
        )
    }

    def Rule SufficientDataSize {
        ForAll(x in Self | |Self::Data| >= 64)
    }

    def Operation CreateNFT {
        ConstrainedBy(IdOnCreation)
    }
```

```
    def Operation UseNFT {
        ConstrainedBy(GenesisIdNeverChanges)
    }

    def Invariant GenesisIdNeverChanges {
        ForAll(
            x in Self::Inputs | Some(y in Self::Outputs | GenesisId(x) = GenesisId(y))
        )
    }
}
```

The Validity specification above is a complete implementation of tNFT functionality. It can be deployed on-chain and can be read by Trampoline framework to automatically generate transactions for tNFT operations as well as to subscribe to tNFT events.

In the next article, we will dive deeper into the semantics of tNFT, principles of UTXO-based smart contract development, and how to realize the potential of generalized UTXOs with Trampoline and Validity.

If you're interested in following the development of both Trampoline Framework & the Validity language, make sure to star the repository here.

4 Appendix: Definition of Built-in Validity Terms

4.1 Inputs, Outputs, Transaction Context

Inputs refers to the inputs in a CKB transaction. Likewise, *Outputs* refers to the outputs field in a CKB transaction. For simplicity, we can consider a transaction to be a tuple of inputs, outputs, scripts, and a script lookup function.

$$Tx := (Inputs, Outputs, Scripts, F)$$

Where *Inputs* and *Outputs* are both finite N-indexed sets and *Scripts* is a set of predicate functions for which the domain is a tuple containing the Tx context and binary data (script args).

$$Script := Tx \times \{0,1\}^* \to \{true, false\}$$

F is an especially important function that maps each cell in both the inputs and outputs to a pair of Scripts to execute for that Cell (representing the Cell's lock script and type script, respectively). In this case, since Type Script is optional, we account for that by having a tautological script in the Scripts set by default (one that always returns true).

$$F := (Inputs \times \{In\}) \cup (Outputs \times \{Out\}) \times \{Lock, Type\} \to Scripts$$

A cell is modeled as a 5-tuple representing capacity (in CKBytes), cell data, lock script args, type script args, and its outpoint (an ordered pair representing its transaction hash and its index in outputs).

$$Cell := (a, b, c, d, e)$$

The $Cell$ fields are defined as:

$$\begin{aligned} a &\in N^{+} \\ b, c, d &\in \{0,1\}^{*} \\ e &:= (x, y) \\ x &\in \{0,1\}^{256} \\ y &\in \{0,1\}^{64} \end{aligned}$$

4.2 The "Self" Keyword

The set of inputs and outputs can be partitioned into two sets of equivalence classes: one for Type scripts and one for lock scripts . In either case, $Self$ refers to a particular equivalence class such that all cells in that set have the same script. This is essentially the semi-formal description of *Script Groups* as implemented in the CKB node and as described in CKB RFC 22.

More precisely, $Self$ is a function that first partitions the inverse relation of F into a set of equivalence classes partitioned by each $Script$. Since each of these partitions contains sets which are indexed by lock or type, input or output, and offset into the transaction's inputs or outputs, $Self$ provides a family of functions to perform look-ups on various cells where all the cells use the same script in one of their fields.

To properly declare $Self$ as a function, though, we can say that $Self$ is a function that maps every Script to a subset of the **superset** of the domain of F.

$$\begin{aligned} Self &:= Scripts \to G \\ &where\ G \supset dom(F) \end{aligned}$$

Specifically, for any script:

$$Self(script) = \{x \in G \supset dom(F) \mid \forall y \in x\ (F(y) = script)\}$$

Thus, $Self(script)$ produces a family of sets which are indexed by $\{Lock, Type\}$, then by $\{In, Out\}$, and then by $(0..n)$. This is why Validity provides various methods on $Self$ such as Self::Inputs, Self::Outputs etc. Note that $\{Lock, Type\}$ denotes whether the CKB VM is currently evaluating the Validity script as a Lock or Type script, and is therefore implicitly applied before applying any other methods on Self.