ANASTASIA LABS

Proof of Achievement - Milestone 2

Augmenting Lucid's Utility Library Functions

Project Number 1100024
Project Manager Jonathan Rodriguez

Contents

ANASTASIA LABS

Project Name: Lucid Evolution: Redefining Off-Chain Transactions in Cardano
URL: Catalyst Proposal

Introduction

Our short-term goal with Lucid Evolution isn't to reinvent the wheel but to make it better. We're focusing on handling side effects, improving error control, offering unsafe, safe, and lazy APIs, and providing safe deserialization schemas. We have implemented an extensive utility function variety and we aim to make it easier for maintainers.

Function Design / Gap Identification

After evaluating the legacy lucid library and our initial implementations, we started working on identifying areas that needed enhancements.In this effort, a big portion of the library has been rewritten or created from scratch.

Lucid Evolution is like the legacy Lucid library but with improved APIs, better error handling, more structure, and the latest version of CML. Additionally, we're planning to introduce an abstraction layer on top, allowing users to select the serialization library that best suits their needs.

We restructured and refactored the library and have made changes to make We have a modified coinSelection algorithm, a new TxBuilder with its own function packages.

These can be grouped under

- Attach.ts
- Collect.ts
- CompleteTxBuilder.ts
- In work-Governance.ts
- Interval.ts
- Metadata.ts
- Mint.ts
- Pay.ts
- Pool.ts
- Read.ts
- Signer.ts
- Stake.ts
- TxUtils.ts

For example

in order to highlight differences between the evolution library and the legacy lucid library, we can display an example of how the two libraries handle the same transaction submission scenario:

```typescript
export type TxSigned = {
  submit: () => Promise<string>;
  submitProgram: () => Effect.Effect<string, TxSubmitError, never>;
  submitSafe: () => Promise<Either<string, TxSubmitError>>;
  toCBOR: () => string;
  toHash: () => string;
};
export const makeSubmit = (
  wallet: Wallet,
  txSigned: CML.Transaction,
): TxSigned => {
  const submit = Effect.tryPromise({
    try: () => wallet.submitTx(txSigned.to_cbor_hex()),
    catch: (error) => submitError("SubmitError", String(error)),
  });
  return {
    submit: () => makeReturn(submit).unsafeRun(),
    submitProgram: () => makeReturn(submit).program(),
    submitSafe: () => makeReturn(submit).safeRun(),
    toCBOR: () => txSigned.to_cbor_hex(),
    toHash: () => CML.hash_transaction(txSigned.body()).to_hex(),
  };
};
```

Figure 1: Lucid Evolution - TxSigned type

```typescript
export class TxSigned {
  txSigned: C.Transaction;
  private lucid: Lucid;
  constructor(lucid: Lucid, tx: C.Transaction) {
    this.lucid = lucid;
    this.txSigned = tx;
  }

  async submit(): Promise<TxHash> {
    return await (this.lucid.wallet || this.lucid.provider).submitTx(
      toHex(this.txSigned.to_bytes()),
    );
  }

  /** Returns the transaction in Hex encoded Cbor. */
  toString(): Transaction {
    return toHex(this.txSigned.to_bytes());
  }

  /** Return the transaction hash. */
  toHash(): TxHash {
    return C.hash_transaction(this.txSigned.body()).to_hex();
  }
}
```

Figure 2: Lucid - TxSigned class

Differences

We have adopted an implemented approach method closer to functional programming paradigms. We use `Effect` to handle promises and improve errors and use the latest `CML`.

As it can be seen in one of our latest release patch notes (0.2.47), we are working on enhancing and upgrading the variability of tools and services available for developers using lucid evolution.

By integrating and updating our compatibility, we are expanding the libraries reach to support different Cardano environments. We have addressed previous issues with TypeScript configuration, improving type declarations and enchancing the provider variability by integrating.

Transaction management saw significant improvements with sophisticated UTXO management, allowing efficient chaining of transactions within a single block. Memory management was optimized minimizing memory leaks and enhancing overall system stability.

These enhancements, reflect our commitment to addressing gaps and improving the library.

We want to create a library that allows, just like our design patterns repository, simplification of complex design patterns and giving developers an efficient tool.

Use Case Scenario

Here's how the Lucid Evolution enabled input indexing could look like, making Staking Validator Design Pattern usage a breeze

```
withdraw (
  rewardAddress: RewardAddress,
  amount: Lovelace,
  redeemer?: string | RedeemerBuilder,
) => TxBuilder;

// The type which needs to be provided in case you want your redeemer to
// have input indices but would like lucid to populate them for you
// after doing the coin selection
export type RedeemerBuilder = {
  makeRedeemer: (inputIndices: bigint[]) => Redeemer;
  inputs: UTxO[];
};

const rdmrBuilder: RedeemerBuilder = {
  makeRedeemer: (inputIndices: bigint[]) => {
    return Data.to({
    nodeIdxs: inputIndices,
    nodeOutIdxs: outputIndices, // you would have this already
  })},
  inputs: selectedUTxOs // any inputs that you wish to be indexed, the inputs
}

const tx = lucid_evolution.
  .newTx()
  .collectFrom(selectUTxOs, redeemer)
  .withdraw(rewardAddress, 0n, rdmrBuilder)
  .attach.SpendingValidator(spend)
  .attach.WithdrawalValidator(stake)
  .completeProgram();
```

Outline Report for Utility Functions per Package

In this following section you will find the utility packages we have under the lucid-evolution github page with the following general format:

1. Title
2. Description
3. Key Functions
4. Code Snapshot

By clicking on the “GitHub Link” hyperlink you can view the dedicated repository section for the utility function package

Description

The `bip39.ts` module implements functions related to BIP39, which defines a way to generate the mnemonic phrase (a series of easy-to-remember words) from a random seed

- This is a partial reimplementation of BIP39 in Deno
- We only use the default Wordlist (english)

Utility Package	Directory
bip39.ts	GitHub Link

Key Functions

`mnemonicToEntropy`

Converts a mnemonic phrase back into its original entropy representation

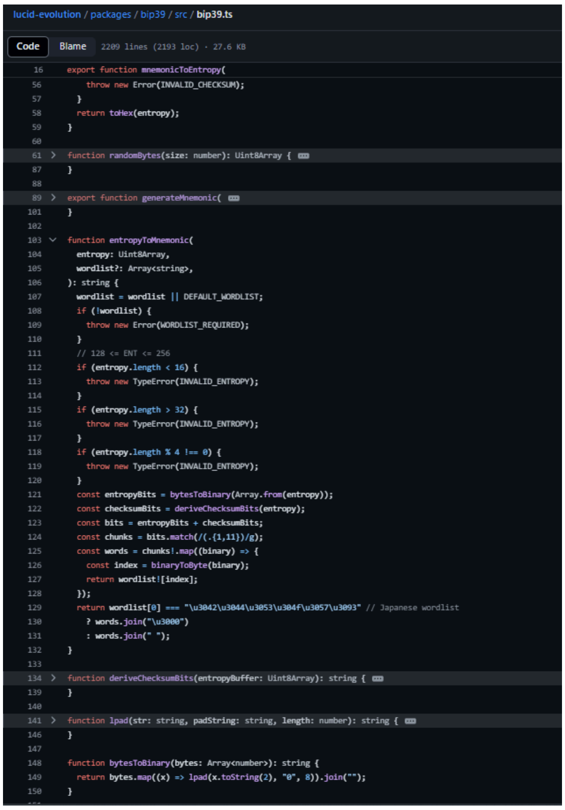
`generateMnemonic`

Generates a new mnemonic phrase from random entropy. It can be used to create new wallets or regenerate existing ones from a known entropy source

`entropyToMnemonic`

Converts entropy into a mnemonic phrase using a specific wordlist for wallet recovery or setup

Code Snapshot


lucid-evolution / packages / bip39 / src / bip39.ts

Code Blame 2209 lines (2193 loc) · 27.6 KB

```
export function mnemonicToEntropy(
    throw new Error(INVALID_CHECKSUM);
  }
  return toHex(entropy);
}

function randomBytes(size: number): Uint8Array {
}

export function generateMnemonic(
}

function entropyToMnemonic(
  entropy: Uint8Array,
  wordlist?: Array<string>,
): string {
  wordlist = wordlist || DEFAULT_WORDLIST;
  if (!wordlist) {
    throw new Error(WORDLIST_REQUIRED);
  }
  // 128 <= ENT <= 256
  if (entropy.length < 16) {
    throw new TypeError(INVALID_ENTROPY);
  }
  if (entropy.length > 32) {
    throw new TypeError(INVALID_ENTROPY);
  }
  if (entropy.length % 4 !== 0) {
    throw new TypeError(INVALID_ENTROPY);
  }
  const entropyBits = bytesToBinary(Array.from(entropy));
  const checksumBits = deriveChecksumBits(entropy);
  const bits = entropyBits + checksumBits;
  const chunks = bits.match(/(.{1,11})/g);
  const words = chunks!.map((binary) => {
    const index = binaryToByte(binary);
    return wordlist![index];
  });
  return wordlist[0] === "\u3042\u3044\u3053\u304f\u3057\u3093" // Japanese wordlist
    ? words.join("\u3000")
    : words.join(" ");
}

function deriveChecksumBits(entropyBuffer: Uint8Array): string {
}

function lpad(str: string, padString: string, length: number): string {
}

function bytesToBinary(bytes: Array<number>): string {
  return bytes.map((x) => lpad(x.toString(2), "0", 8)).join("");
}
```

Figure 3: Snapshot-01-BIP39

Description

The `address.ts` module is used to handle address-related operations within the Lucid Evolution library. Its functions allow the manipulation and conversion of various address types

Utility Package	Directory
address.ts	[GitHub Link]

Key Functions

`addressFromHexOrBech32`

Converts an address from either hexadecimal or Bech32 format to a CML Address object

`credentialToRewardAddress`

Converts a stake credential into a reward address

`validatorToRewardAddress`

Converts a validator (either a certificate or withdrawal validator) into a reward address using the script hash derived from the validator

`getAddressDetails`

Extracts and returns detailed information about various address types (Base, Enterprise, Pointer, Reward, Byron), including payment and stake credentials

Code Snapshot


lucid-evolution / packages / utils / src / address.ts

Code Blame 239 lines (229 loc) · 6.97 KB

```
export function credentialToRewardAddress( ...
}

export function validatorToRewardAddress(
  network: Network,
  validator: CertificateValidator | WithdrawalValidator,
): RewardAddress {
  const validatorHash = validatorToScriptHash(validator);
  return CML.RewardAddress.new(
    networkToId(network),
    CML.Credential.new_script(CML.ScriptHash.from_hex(validatorHash)),
  )
    .to_address()
    .to_bech32(undefined);
}

/** Address can be in Bech32 or Hex. */
export function getAddressDetails(address: string): AddressDetails {
  // Base Address
  try {
    // const parsedAddress = CML.BaseAddress.from_address(
    //   addressFromHexOrBech32(address),
    // )!;
    const parsedAddress = CML.BaseAddress.from_address(
      CML.Address.from_bech32(address),
    )!;
    const paymentCredential: Credential =
      parsedAddress.payment().kind() === 0
        ? {
            type: "Key",
            // hash: toHex(
            //   parsedAddress.payment_cred().to_keyhash()!.to_bytes(),
            // ),
            hash: parsedAddress.payment().as_pub_key()!.to_hex(),
          }
        : {
            type: "Script",
            // hash: toHex(
            //   parsedAddress.payment_cred().to_scripthash()!.to_bytes(),
            // ),
            hash: parsedAddress.payment().as_script()!.to_hex(),
          };
    const stakeCredential: Credential =
      parsedAddress.stake().kind() === 0
        ? // parsedAddress.stake_cred().kind() === 0
          {
            type: "Key",
            hash: parsedAddress.stake().as_pub_key()!.to_hex(),
            // hash: toHex(parsedAddress.stake_cred().to_keyhash()!.to_bytes()),
          }
        : {
            type: "Script",
            // hash: toHex(parsedAddress.stake_cred().to_scripthash()!.to_bytes()),
```

Figure 4: Snapshot-02-Address

Description

The cbor.ts module within Lucid Evolution deals with functionalities related to CBOR (Concise Binary Object Representation), specifically focusing on encoding and decoding operations that adhere to the CBOR standard as defined in RFC 7049

Utility Package	Directory
cbor.ts	GitHub Link

Key Functions

`applyDoubleCborEncoding`

Implements a double encoding for CBOR bytestrings, which decodes an encoded string twice to ensure correct formatting

Code Snapshot


```
import { fromHex, toHex } from "@lucid-evolution/core-utils";
import { decode, encode } from "cborg";

// 1st byte (58) 0101(major type 2) , 1000 (additional info)
// 2n byte byte represents the lenght of the content
// 3rd byte represents bytestring content
// https://www.rfc-editor.org/rfc/rfc7049#section-2.1
// Apply double bytestring enconding of type `major type 2`
export const applyDoubleCborEncoding = (script: string) => {
  try {
    decode(decode(fromHex(script)));
    return script;
  } catch (error) {
    return toHex(encode(fromHex(script)));
  }
};
```

Figure 5: Snapshot-03-Cbor

Description

The `cost_model.ts` module in Lucid Evolution deals with the configuration and management of cost models related to the execution of Plutus scripts on the blockchain. These cost models are used to in determine the computational and memory costs of running smart contracts

Utility Package	Directory
cost_model.ts	[GitHub Link]()

Key Functions

`createCostModels`

Constructs a CostModels object that covers the various cost parameters for different versions of the Plutus scripts (PlutusV1, PlutusV2).

This function populates cost models from predefined settings.

1. Initializes new cost model objects for each Plutus version
2. Iteratively fills these objects with cost data parsed from input parameters
3. Handles the memory management of these operations to prevent leaks and ensure efficiency

Code Snapshot

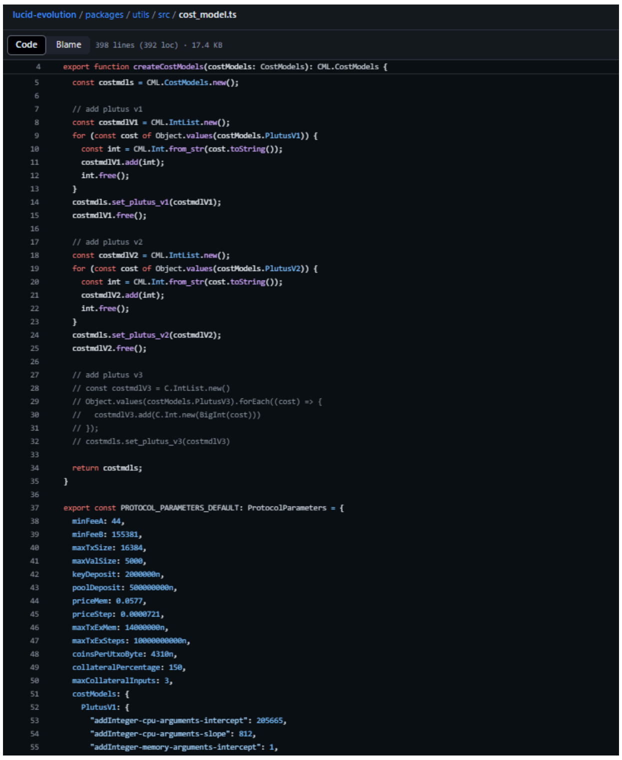


Figure 6: Snapshot-04-Costmodel

Description

The `credential.ts` module handles the creation and manipulation of credentials within the ecosystem. This module is for constructing addresses and managing their components.

Utility Package	Directory
credential.ts	GitHub Link

Key Functions

`credentialToAddress`
Converts payment and optionally stake credentials into an address

`scriptHashToCredential`
Wraps a script hash into a credential object, utilizing its use in other functions requiring a credential format

`keyHashToCredential`
Converts a key hash into a credential object, allowing for further operations that require credentials

`paymentCredentialOf`
Extracts the payment credential from an address, throwing an error if the address does not contain one

`stakeCredentialOf`
Retrieves the stake credential from a reward address

Code Snapshot

Figure 7: Snapshot-05-Credential

ANASTASIA LABS

Description

The `datum.ts` module provides functionality for handling Plutus data on the blockchain. Specifically, it includes utilities for converting Plutus data (datum) into a format that is suitable for transaction processing, like generating a hash of the datum

Utility Package	Directory
datum.ts	GitHub Link

Key Functions

`datumToHash`

Converts a datum object into its corresponding hash. This hash is used to refer to data stored off-chain.

1. Converts the datum from its CBOR hexadecimal representation to a Plutus data format
2. Uses the CML to calculate the hash of the Plutus data

Code Snapshot

lucid-evolution / packages / utils / src / **datum.ts**

solidsnakedev refactor: move CML to core file

Code Blame 6 lines (5 loc) · 232 Bytes

```
import { Datum, DatumHash } from "@lucid-evolution/core-types";
import { CML } from "./core.js";

export function datumToHash(datum: Datum): DatumHash {
  return CML.hash_plutus_data(CML.PlutusData.from_cbor_hex(datum)).to_hex();
}
```

Figure 8: Snapshot-06-Datum

Description

Utility Package	Directory
keys.ts	GitHub Link

The `keys.ts` deals with the generation and conversion of keys which are fundamental for secure transactions

Key Functions

`generatePrivateKey`
Generates a new private key using the ED25519 cryptographic algorithm. This key is used for signing transactions securely

`generateSeedPhrase`
Creates a mnemonic seed phrase based on the BIP39 standard

`toPublicKey`
Converts a given private key to its corresponding public key, allowing for the public key to be used in transaction verification without revealing the private key

An example of a key private -> public conversion would look like:

```
CML.PrivateKey.from_bech32(privateKey).to_public().to_bech32();
```

Code Snapshot

Figure 9: Snapshot-07-Keys

Description

The `native.ts` module handles operations related to Cardano's native scripts, which are used for transaction validation without the execution of Plutus smart contracts. This module provides functionality to convert custom native script objects into Cardano's native script format

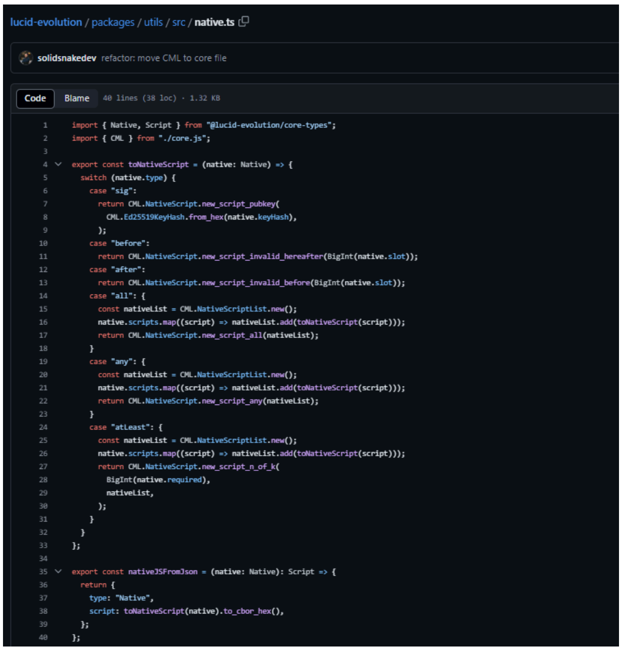
Utility Package	Directory
native.ts	GitHub Link

Key Functions

`toNativeScript`
Converts a high-level native script definition into a low-level script that the Cardano node can interpret. This function supports various types of native scripts including simple public key-based scripts, time-lock scripts, and complex multi-script conditions

`nativeJSFromJson`
Encapsulates the conversion of a Native script object into a script that is compatible with the ledger, serialized into CBOR hex format

Code Snapshot


lucid-evolution / packages / utils / src / native.ts
solidsnakedev refactor: move CML to core file
Code Blame 40 lines (38 loc) · 1.32 KB

```
import { Native, Script } from "@lucid-evolution/core-types";
import { CML } from "./core.js";

export const toNativeScript = (native: Native) => {
  switch (native.type) {
    case "sig":
      return CML.NativeScript.new_script_pubkey(
        CML.Ed25519KeyHash.from_hex(native.keyHash),
      );
    case "before":
      return CML.NativeScript.new_script_invalid_hereafter(BigInt(native.slot));
    case "after":
      return CML.NativeScript.new_script_invalid_before(BigInt(native.slot));
    case "all": {
      const nativeList = CML.NativeScriptList.new();
      native.scripts.map((script) => nativeList.add(toNativeScript(script)));
      return CML.NativeScript.new_script_all(nativeList);
    }
    case "any": {
      const nativeList = CML.NativeScriptList.new();
      native.scripts.map((script) => nativeList.add(toNativeScript(script)));
      return CML.NativeScript.new_script_any(nativeList);
    }
    case "atLeast": {
      const nativeList = CML.NativeScriptList.new();
      native.scripts.map((script) => nativeList.add(toNativeScript(script)));
      return CML.NativeScript.new_script_n_of_k(
        BigInt(native.required),
        nativeList,
      );
    }
  }
};

export const nativeJSFromJson = (native: Native): Script => {
  return {
    type: "Native",
    script: toNativeScript(native).to_cbor_hex(),
  };
};
```

Figure 10: Snapshot-08-Native

ANASTASIA LABS

Description

The `network.ts` module is to map high-level network identifiers to their corresponding numeric identifiers

Utility Package	Directory
network.ts	GitHub Link

Key Functions

`networkToId`

Converts a network name into its corresponding numeric ID

Mapping process

```
export function networkToId(network: Network): number {
  switch (network) {
    case "Preview":
      return 0;
    case "Preprod":
      return 0;
    case "Custom":
      return 0;
    case "Mainnet":
      return 1;
    default:
      throw new Error("Network not found");
  }
}
```

This function’s purpose is to ensure that transactions are correctly associated with the appropriate network

Description

The `scripts.ts` module offers a range of functions to manage and transform scripts used in smart contracts. It handles various script types including native, Plutus V1, and Plutus V2 scripts, facilitating their usage in transactions and smart contracts

Utility Package	Directory
scripts.ts	GitHub Link

Key Functions

`validatorToAddress`

Converts a validator script into a Cardano address

`validatorToScriptHash`

Generates a script hash from a validator object. This function supports multiple script types including Native, Plutus V1, and Plutus V2

`toScriptRef / fromScriptRef`

Converts a script into a CML.Script object and vice versa, facilitating the use of scripts in a format suitable for transactions

`mintingPolicyToId`

Converts a minting policy into a policy ID using the script hash functionality

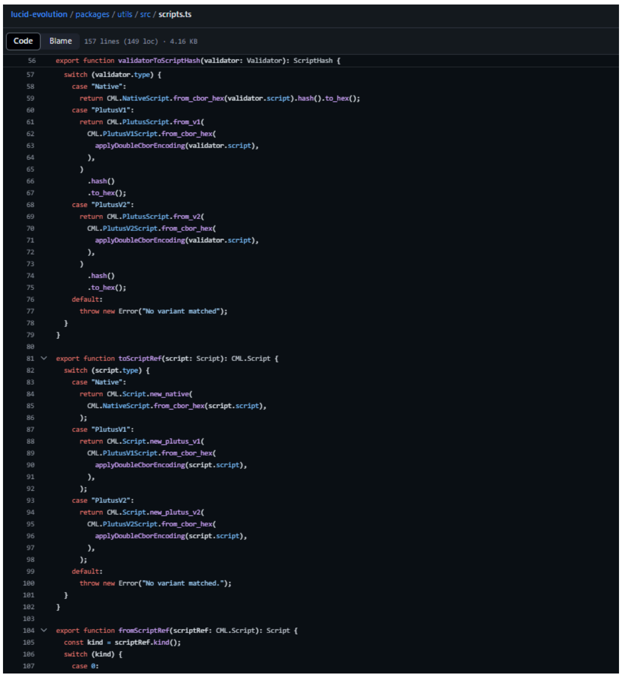
`nativeFromJson / nativeScriptFromJson`

Converts JSON representations of native scripts into script objects, so that scripts can be handled in a standardized format across the system

`applyParamsToScript`

Applies parameters to a Plutus script

Code Snapshot



lucid-evolution / packages / utils / src / scripts.ts

Code Blame 157 lines (149 loc) · 4.16 KB

```
export function validatorToScriptHash(validator: Validator): ScriptHash {
  switch (validator.type) {
    case "Native":
      return CML.NativeScript.from_cbor_hex(validator.script).hash().to_hex();
    case "PlutusV1":
      return CML.PlutusScript.from_v1(
        CML.PlutusV1Script.from_cbor_hex(
          applyDoubleCborEncoding(validator.script),
        ),
      )
        .hash()
        .to_hex();
    case "PlutusV2":
      return CML.PlutusScript.from_v2(
        CML.PlutusV2Script.from_cbor_hex(
          applyDoubleCborEncoding(validator.script),
        ),
      )
        .hash()
        .to_hex();
    default:
      throw new Error("No variant matched");
  }
}

export function toScriptRef(script: Script): CML.Script {
  switch (script.type) {
    case "Native":
      return CML.Script.new_native(
        CML.NativeScript.from_cbor_hex(script.script),
      );
    case "PlutusV1":
      return CML.Script.new_plutus_v1(
        CML.PlutusV1Script.from_cbor_hex(
          applyDoubleCborEncoding(script.script),
        ),
      );
    case "PlutusV2":
      return CML.Script.new_plutus_v2(
        CML.PlutusV2Script.from_cbor_hex(
          applyDoubleCborEncoding(script.script),
        ),
      );
    default:
      throw new Error("No variant matched.");
  }
}

export function fromScriptRef(scriptRef: CML.Script): Script {
  const kind = scriptRef.kind();
  switch (kind) {
    case 0:
```

Figure 11: Snapshot-09-Scripts

Description

The `time.ts` module in our library handles the conversion between blockchain-specific slot numbers and Unix timestamps. This functionality is important for scheduling and timing events within the blockchain, where time is often expressed in terms of slots

Utility Package	Directory
time.ts	GitHub Link

Key Functions

`unixTimeToSlot`
Converts a Unix timestamp to the corresponding slot number in the blockchain. It is to determine when specific events or transactions should occur relative to blockchain time

`slotToUnixTime`
Converts a slot number to the corresponding Unix timestamp. This allows applications to interpret blockchain time in terms of real-world time

What are slots and how do they serve a role in time?

These functions use `SLOT_CONFIG_NETWORK`, a predefined mapping specific to each network configuration that defines the relationship between slot numbers and Unix time. This ensures accurate time calculations across different network settings

```
export function slotToUnixTime(network: Network, slot: Slot): UnixTime {
  return slotToBeginUnixTime(slot, SLOT_CONFIG_NETWORK[network]);
}
```

Description

The `utxo.ts` module provides functionality for managing UTxOs. It supports creating transaction inputs and outputs, converting UTxOs to different formats, and sorting or selecting UTxOs based on specific criteria

Utility Package	Directory
utxo.ts	GitHub Link

Key Functions

`utxoToTransactionOutput` / `utxoToTransactionInput`
These functions convert UTxO data into transaction outputs and inputs, facilitating the integration of UTxOs into new transactions

`utxoToCore` / `utxosToCores`
Converts UTxOs to CML.TransactionUnspentOutput objects, standardizing UTxOs for transaction processing

`coreToUtxo` / `coresToUtxos`
Reverses the conversion process, transforming CML.TransactionUnspentOutput objects back into UTxO format

`selectUTxOs`
Selects UTxOs from a list that meet specified asset requirements, useful in transaction construction where specific asset amounts are required

`sortUTxOs`
Sorts an array of UTxOs according to a specified order, either largest first or smallest first, based on the amount of Lovelace

Code Snapshot

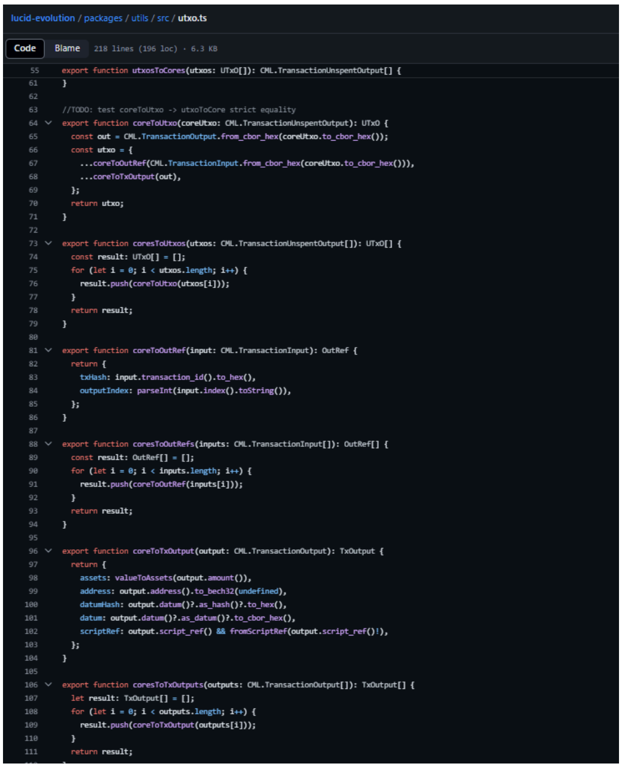


Figure 12: Snapshot-10-Utxo

Description

The `value.ts` module provides functions to manipulate and convert between the blockchain's internal value representation and a more accessible assets format. This serves the purpose for managing transaction outputs and state transitions in smart contracts

Utility Package	Directory
value.ts	GitHub Link

Key Functions

`valueToAssets`
Converts a CML.Value object, which represents the amount of different tokens in a transaction output, into an Assets object that is easier to manipulate and display

`assetsToValue`
Converts an Assets object back into a CML.-Value object for use in transaction creation or other on-chain activities

`fromUnit` / `toUnit`
These functions handle conversion between a unit representation (combining policy ID and asset names) and its constituent parts, helping in asset identification and manipulation

`addAssets`
Aggregates multiple Assets objects into a single object, summing up quantities of the same assets

Code Snapshot


lucid-evolution / packages / utils / src / value.ts

Code Blame 101 lines (96 loc) · 3.21 KB

```
      const assetNames = policyAssets.keys();
      for (let k = 0; k < assetNames.len(); k++) {
        const policyAsset = assetNames.get(k);
        const quantity = policyAssets.get(policyAsset)!;
        //FIX: report to dcspark policyAsset.to_cbor_hex() adds the head byte twice eg. MyMintedToken -> (to Hex) -> 4d4d794d69
        const unit = policy.to_hex() + fromText(policyAsset.to_str());
        assets[unit] = quantity;
      }
    }
  }
  return assets;
}

export function assetsToValue(assets: Assets): CML.Value {
  const multiAsset = CML.MultiAsset.new();
  const lovelace = assets["lovelace"] ? assets["lovelace"] : 0n;
  const units = Object.keys(assets);
  const policies = Array.from(
    new Set(
      units
        .filter((unit) => unit !== "lovelace")
        .map((unit) => unit.slice(0, 56)),
    ),
  );
  for (const policy of policies) {
    const policyUnits = units.filter((unit) => unit.slice(0, 56) === policy);
    const assetsValue = CML.MapAssetNameToCoin.new();
    for (const unit of policyUnits) {
      assetsValue.insert(
        CML.AssetName.from_bytes(fromHex(unit.slice(56))),
        BigInt(assets[unit]),
      );
    }
    multiAsset.insert_assets(CML.ScriptHash.from_hex(policy), assetsValue);
  }
  return CML.Value.new(lovelace, multiAsset);
}

/**
 * Splits unit into policy id, asset name (entire asset name), name (asset name without label) and label if applicable.
 * name will be returned in Hex.
 */
export function fromUnit(unit: Unit): {
  policyId: PolicyId;
  assetName: string | null;
  name: string | null;
  label: number | null;
} {
  const policyId = unit.slice(0, 56);
  const assetName = unit.slice(56) || null;
  const label = fromLabel(unit.slice(56, 64));
  const name = (() => {
```

Figure 13: Snapshot-11-Value

Testing Suite

Our testing suite, integrated into the Lucid Evolution library through GitHub Actions, automatically runs on `push` to the main branch and during `pull_request` events. It includes tests in order to ensure each function performs as expected.

Automated tests are triggered to validate code functionality.

Packages

Lucid

- coinselection.test.ts
- onchain.test.ts
- read.test.ts
- tx.test.ts
- txHash.test.ts
- wallet.test.ts

Utils

- apply-param.test.ts
- cbor.test.ts
- native.test.ts
- utxo.test.ts

Provider

- koios.test.ts
- kupmios.test.ts

GIF Testrun

Test Result

This GIF, **an automated test** running in terminal,
showcases that our test packages are working as intended

Brief overview of test cases in Lucid

Coin Selection

This test case ensures the functionality of the `coinSelection`. By checking for various scenarios, each test case focuses on specific aspects of the selection algorithm, ensuring that the function works correctly under different conditions and efficiently selects the appropriate UTxOs based on the input criteria

Onchain Tests

This comprehensive test suite ensures that various functionalities related to transactions, staking, minting, burning, and parameterized contracts work as expected in the lucid-evolution library

Read Tests

In order to ensure that the library correctly integrates with a provider API to perform operations like wallet selection, UTxO retrieval

Tx Test

This scripts is designed to validate the minting and burning functions of tokens by verifying transaction creation signing and submission.

Tx Hash Test

It ensures the correctness of the transaction signing and hash computation-. It uses a predefined transaction and signs it with a selected wallet, than computes the hash in order to compare the computed hash with the signed transaction hash

Wallet test

To validate wallet management. Things like switching providers, generating seed phrases and correctly selecting a wallet