A Theory for Lightweight Cryptocurrency Ledgers

Bill White*

February 8, 2015

Abstract

We present representations for cryptographic ledgers which do not require storing excessive information about transaction histories. Such representations may allow for the creation of cryptocurrencies which do not suffer from so-called "block chain bloat." A corresponding theory has been formalized in the proof assistant Coq. We give a high level description and a simplified example. We then give some of the main definitions and theorems which can be found in the formal development.

1 Introduction

Cryptocurrencies such as Bitcoin [9] operate via distributed construction and verification of a ledger in the form of a block chain. Each block contains a sequence of transactions, each of which contains transaction outputs which can later be spent. In order to ensure that a transaction output is available to be spent it is necessary to verify two facts:

- *Existence*: The transaction with the output is in fact in the block chain ledger.
- *Nonexistence*: There is no transaction already spending the relevant output.

In Bitcoin blocks are separated into two parts. The first part is the *block header*. The remainder of the block consists of the new transactions included in the block. We can call this second part the *block delta* since it describes the updates to the cryptographic ledger. The header contains the required proof of work, a pointer to the previous block header and a Merkle root [8] of a tree of the transactions in the block delta.

The block headers alone can be used to verify both the security of the block chain (since the proof of work can be verified using only the headers) and to

*Bitmessage: BM-2cWvQoqzQPJS8CvuRi7DZ2iCud4m9FB1HA
Email: billwhite@bitmessage.ch BTC: 12pbhpqEg7cjaCLLcvdhJhBWGUQWkRK3zS

ensure the existence of transactions in the ledger. It was part of the Nakamoto's original Bitcoin vision [9] that many nodes in the Bitcoin network would be lightweight *SPV* nodes. In order to verify the existence of a transaction someone can give the transaction along with enough hash values to compute the Merkle root of a tree containing the transaction. One can then simply check if this Merkle root is in fact in one of the block headers. That is, verifying existence of a transaction in the ledger does not require the entire block chain.

On the other hand, verifying nonexistence of a transaction spending the relevant output does require the full block chain. The only way to be certain the transaction output has not been spent is to examine each transaction in the block chain and note that it does not spend the output. In practice Bitcoin nodes keep a pool of unspent transaction outputs which are removed as transactions spending the outputs are included in the block chain.

The need for full nodes to store the entire block chain leads to a phenomenon sometimes called "block chain bloat." The Bitcoin block chain currently requires more than 30GB of storage. The problem is made even worse since, in addition to transactions intended to transmit bitcoins, there are transactions intended to record and timestamp information.

In this paper we present representations of cryptographic ledgers in which both the existence and nonexistence properties described above can be verified with very little storage. The high level idea is to consider each transaction output as an *asset* and consider the ledger as a function from addresses to lists of assets held at the address. Such a *ledger function* describes the current state of the ledger. The list of assets held at an address changes as assets are received or spent. In other words, valid transactions give ways to transform one ledger function into another ledger function.

A ledger function of this kind can be represented as a Merkle tree. Assuming there are 2^{160} addresses this would be a binary tree of depth 160. While such a tree is obviously impractical to fully represent directly, we take advantage of the fact that the ledgers are very sparse. The vast majority of addresses will never hold assets. If a node in the corresponding Merkle tree has no children holding assets, then it can be represented by a special hash value indicating the tree is empty. We call the hash value at the root of such a Merkle tree the *ledger root*.

The block chains described in this paper differ from traditional block chains. Our blocks will still be comprised of block headers and block deltas. Each of our block headers explicitly gives the Merkle root of the new ledger (i.e., the entire current state of the ledger after the block has been processed) instead of the Merkle root of the transactions in the block delta. In addition the block header includes a tree approximating the previous ledger. This would be especially useful in a proof of stake [6, 5, 2, 3, 1] currency since it implies the stake of the forger could be verified using only the header, something which is not possible with current proof of stake implementations.

The block deltas will contain the new transactions and will contain enough information to verify that the previous ledger supports the transactions. That is, the block deltas indicate how to extend the tree given in the block header to be a better approximation of the previous ledger. Each node can use the

previous ledger root and the information in the new block to independently validate the block and calculate the new ledger root.

We describe two tree representations. The first is simply a binary hash tree with empty trees represented as a special hash value. In order to reduce the amount of information needed to communicate approximations of ledgers we then describe a compact representation of such trees. Experimental results would be required to find out how much of an improvement the compact representation would be.

Each node could independently decide how much of the ledger to explicitly keep. Informally one could say that a node is "watching" certain addresses if it explicitly represents enough of the ledger to know the assets held by these addresses. In order to spend assets held by an address some node needs to produce the part of the tree corresponding to this address. Miners (or forgers) should have the incentive to keep up with the full tree in order to more easily include transactions (with transaction fees) in the blocks they mine (or forge).

The ideas and results will only be described briefly and sometimes informally in this paper. Nevertheless the corresponding "ledger theory" has been fully developed and formalized in the proof assistant Coq [7, 4, 10].[1] It is time-consuming to write formal developments of this size. In general it is arguably the case that the benefits of such a formal development are outwieghed by the costs. The specific case of designing a cryptocurrency seems to be an exception. A poorly designed cryptocurrency can potentially result in a large loss of wealth. This is an argument for doing the hard work of formalizing the design and proving certain interesting properties hold in advance.

The informal descriptions will be accompanied by footnotes pointing to corresponding definitions and theorems in the Coq formalization. There are many more definitions and results in the Coq formalization which we do not describe here for the sake of brevity. The author expects to write future papers giving more detailed mathematical developments corresponding to the material in the Coq formalization.

In Section 2 we start with a simplified example in order to motivate the definitions and theorems stated later. In Section 3 we give a few preliminary concepts some of which are taken as primitive. We define the notions of assets and transactions in Section 4. The representation of a ledger as a function from addresses to lists of assets is studied in Section 5. We give a way to approximate asset lists in Section 6 and extend this to Merkle tree approximations of ledgers in Section 7. In Section 8 we give a modified version of Merkle trees, compact trees, intended to give a representation which is more space efficient. In Section 9 we discuss block headers, blocks, block chains and block header chains. We finally conclude in Section 10.

[1] The Coq formalization can be found at `github.com/billlwhite/ledgertheory`.

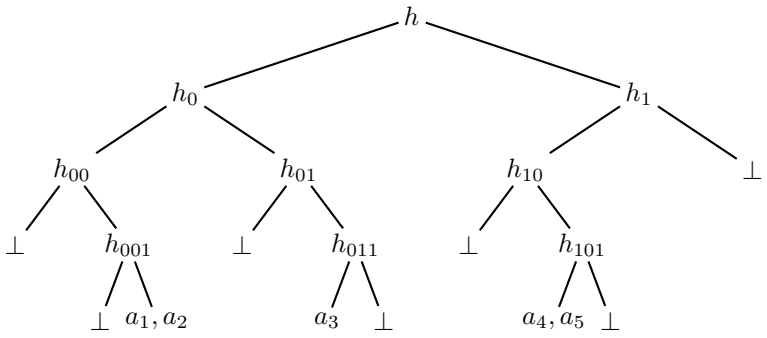

Figure 1: A Merkle Tree Ledger

2 A Simplified Example

In order to demonstrate the ideas without requiring too many details, let us consider a simplified example of the ledger representation we have in mind. Digesting this section should give a high level understanding of the ideas. Optimally this section will whet the appetite of the reader and provide encouragement to study the definitions and statements of results in the remaining sections.

For the moment, suppose we are working with a cryptocurrency for which addresses are only 4 bits long. That is, there are only 16 addresses. While this would obviously be very insecure, it suffices for purposes of illustration.

We can think of the current state of the ledger as a function f mapping each of the 16 addresses to the assets held by the address. Suppose each "asset" represents some units of currency that the address is allowed to spend. An alternative representation of the ledger is as a binary tree of depth 4. The leaf in the position indicated by the address would list the assets held by that address.

In addition to representing the ledger as a tree we make two extra decisions. First if a subtree contains no subleaves with some assets, then it is considered empty and can be represented simply as $\perp$. In practice this would mean we can represent the tree with fewer than 16 leaves and 15 internal nodes. Secondly we can add hash values to the internal nodes obtained by hashing the two children (or the list of assets at the leaves). That is we make the tree into a Merkle tree. In order to make these two decisions coherent we adopt the convention that the hash of the empty list of assets is $\perp$ and the hash of a pair of $\perp$ children is also $\perp$.

Suppose in the current state of the ledger only three addresses hold at least one asset. In particular, suppose address 0011 holds two assets a_1 and a_2, suppose address 0110 holds one asset a_3 and suppose address 1010 holds two assets a_4 and a_5. The Merkle tree representation of this ledger is given in Figure 1.

Next suppose we wish to transform the ledger using a transaction. Suppose

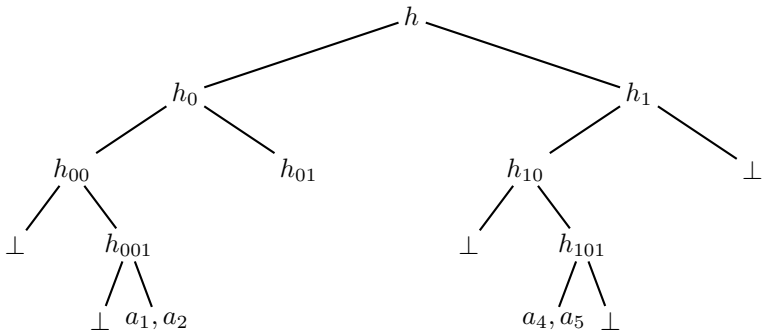

Figure 2: Merkle Tree Supporting a Transaction

the transaction τ spends assets a_1 and a_5 and creates two new assets a_6 (sent to address 1010) and a_7 (sent to address 1101). We need to validate that the two assets a_1 and a_5 are available to be spent. A complete representation of the ledger is not required for this. The Merkle tree in Figure 2 has enough information to check that the assets are available. In particular the information in the full ledger beneath node 01 is not needed to validate τ.

One can imagine a node on such a cryptocurrency network which only keeps the current ledger root h (i.e., the hash root of the current ledger). For this node to validate τ, it will need to be given a tree like the one in Figure 2. The node will know the tree is correct because it has the same ledger root h. It can then independently validate τ.

The process of transforming the ledger using τ is simple: remove a_1 and a_5 from their corresponding leaves and push the new assets a_6 and a_7 onto the leaves for the recipient addresses. In the case of a_7 this means making the leaf at the address 1101 explicit, as it was part of an empty subtree before. If the tree in Figure 2 is transformed using τ, then one obtains the tree in Figure 3. The ledger root of the new tree has, of course, changed. The hashes in the parts of the ledger root which have not been manipulated by τ (e.g., h_{01}) have not changed.

Now suppose a node on the network is run by the owner of the address 0110 who holds a_3. This node may choose to only "watch" this address. Before processing the transaction τ, this node would have a tree like the one shown in Figure 4. This is not enough information to validate τ. As before the node would need to be given a tree like the one shown in Figure 2 to validate τ. Then the node could combine the two trees to obtain the tree in Figure 1 (which in this case has the full information about the ledger, but this is only because the example is small) and transform the tree using τ to obtain the tree in Figure 5. The node could then prune back down to only keep the part of local interest as shown in Figure 6. The reason the node might want to keep this local approximation of the full ledger is because it would be needed to support

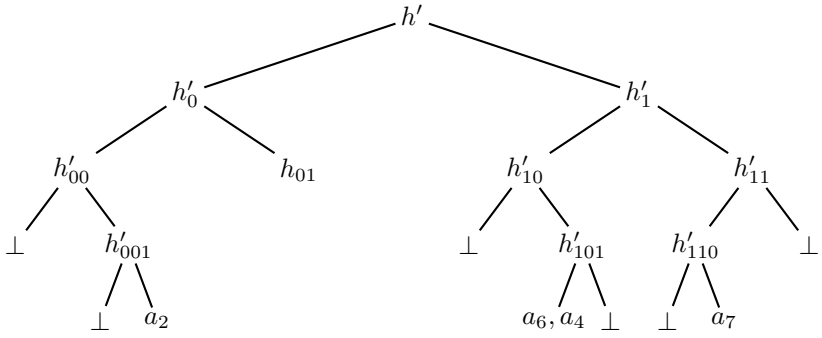

Figure 3: Merkle Tree after a Transaction

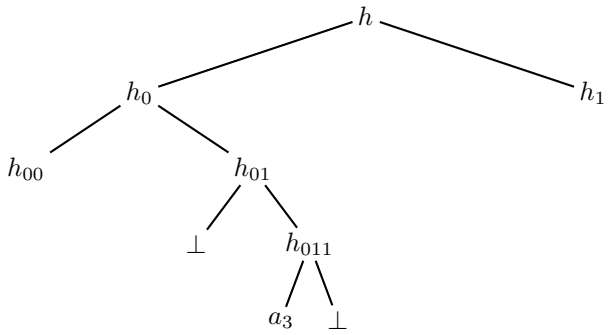

Figure 4: Local Merkle Tree Ledger

a possible later transaction spending a_3.

We have oversimplified in more ways than one in this section. In addition to passing from 160-bit addresses to 4-bit addresses we may have left the reader with the impression that signatures with the private key for α is what authorizes the spending of assets held at α. In fact we will not make this assumption. The asset could be held at an address while the condition that needs to be met in order to spend the asset could be quite different. For example an asset could be held at an address α while a signature for a different address β is required to spend the asset. This might especially be useful for proof of stake currencies so that owner of an asset could allow a forger to hold the asset and use it for forging blocks.[2]

[2]This would address one of the complaints about proof of stake cryptocurrencies: that the private keys for the address with the stake must be online in order to forge.

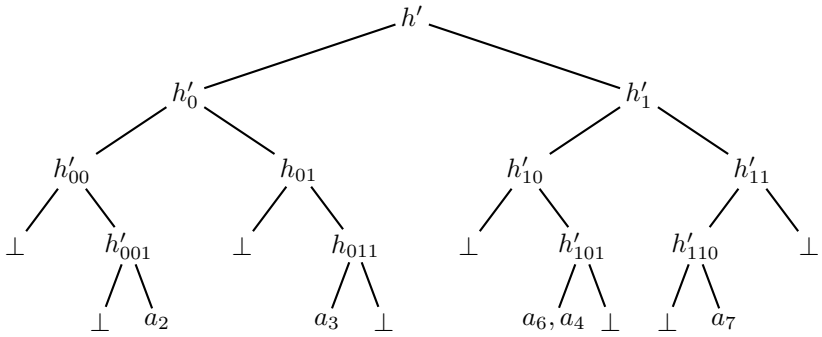

Figure 5: Full Local Merkle Tree after a Transaction

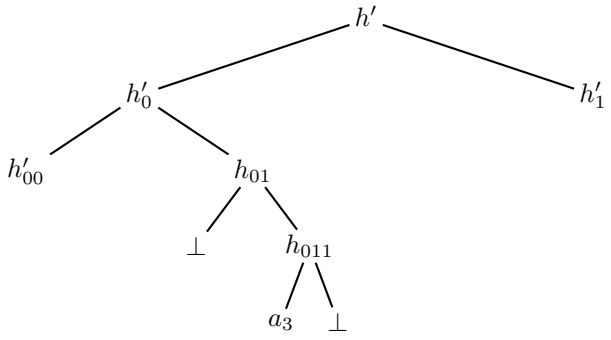

Figure 6: Local Merkle Tree after Transaction and Pruning

3 Preliminaries

We now turn to the mathematical description of the real ledger theory with 160-bit addresses. We begin with a few preliminaries. Motivations for these choices can be found in [11].

We often work with n-length bit sequences and take $\mathcal{B}_n$ to be the set (or type) of all n-length bit sequences. The set $\mathcal{A}$ of addresses is $\mathcal{B}_{160}$.[3]

We take hash values to be primitive and assume hashing functions are injective.[4] We write $\ulcorner n \urcorner$ for the the hash of integers n, write $\ulcorner \alpha \urcorner$ for the hash of addresses α and $\ulcorner h, k \urcorner$ for the hash of pairs of hash values h and k. Using these primitives it is possible to define appropriate injective hashing functions for the composite structures we define. We will not dwell on this fact, but simply use the notation $\ulcorner - \urcorner$ indicate the hashes of objects as needed.

At times we will need a special "empty" hash value which we will write as $\bot$. We will say "extended hash value" when we mean to include $\bot$ as a hash value.[5] We will use $\ulcorner - \urcorner^{\bot}$ to denote hashing operations which may return $\bot$. The intention is always that $\ulcorner x \urcorner^{\bot} = \bot$ precisely when x represents something "empty".

As an example we will use in the paper, consider $\ulcorner h, k \urcorner^{\bot}$ where h and k are extended hash values. We clearly want $\ulcorner \bot, \bot \urcorner^{\bot} = \bot$. There are three other cases to consider: $\ulcorner h, \bot \urcorner^{\bot}$, $\ulcorner \bot, k \urcorner^{\bot}$ and $\ulcorner h, k \urcorner^{\bot}$ where h and k are hash values. To ensure injectivity, we take $\ulcorner \bot, k \urcorner^{\bot} := \ulcorner \ulcorner 0 \urcorner, h \urcorner$, $\ulcorner h, \bot \urcorner^{\bot} := \ulcorner \ulcorner 1 \urcorner, k \urcorner$ and $\ulcorner h, k \urcorner^{\bot} := \ulcorner \ulcorner 2 \urcorner, \ulcorner h, k \urcorner \urcorner$.

Similarly we take the notion of a cryptographic signature to be primitive and assume we have a way of checking if a signature is valid relative to a hash value and address. That is, if σ is a signature, α is an address and h is a hash value, we assume we have a way to check if σ proves that h was signed using the private key corresponding to α.[6]

4 Assets and Transactions

Consider unspent transaction outputs in the Bitcoin block chain. In order to spend such a transaction output we need three pieces of information: a way to reference the transaction output by an identifier, a way to prove that we are authorized to spend the output and the number of currency units corresponding to the output.

We can generalize this example and consider assets to be triples (h, ω, u) where h is a unique identifier for the asset, ω indicates the requirements to spend the asset, and u is the asset value. We briefly consider what these correspond to in Bitcoin. In Bitcoin the h is a pair consisting of the transaction id (the hash of the transaction creating the asset) and an index (indicating which

[3] The Coq code for this can be found in `Addr.v`.

[4] The relevant Coq code can be found in `CryptoHash.v`.

[5] In the Coq code, the type `hashval` is used when $\bot$ is not needed and `option hashval` is used when $\bot$ is needed.

[6] See `signat` and `check_signat` in `CryptoSignature.v`.

output created the asset). In Bitcoin ω is part of a script, the other half of which is given when the output is spent. The completed script is executed to ensure the output can be spent. Typically ω corresponds to giving an address and the completion of the script is a signature corresponding to the address. Multisignature addresses (requiring m of n signatures) are also relatively common. Finally in Bitcoin the value u is simply the number of satoshis from the transaction output.

In our design we simplify the asset identifier by taking h to be a hash value. An appropriate unique hash value for each asset is obtained by hashing the transaction hash with the index indicating the appropriate output of the transaction.

We also forego the generality of scripts and limit ω to be triples of the form $(\overline{\alpha}, m, n)$ where $\overline{\alpha}$ is a list of addresses, m is an integer indicating the number of signatures from $\overline{\alpha}$ required, and n is an integer indicating the earliest block height at which the asset can be spent. It should be clear that $\overline{\alpha}$ and m allow for multisignature assets. As for motivating the n value, it is common for cryptocurrencies to disallow the spending of freshly mined coins until they have matured. We call such a triple $(\overline{\alpha}, m, n)$ an *obligation*.[7]

Finally, we could take the asset value u to simply be an integer. However we allow for two cases to distinguish between assets intended to have value and "assets" which are intended to publish information in a cryptographically secure timestamped manner. We define a *preasset* u to either be an integer v representing v units of currency or a tagged integer $(1, d)$ where d is an integer. In the case of a tagged integer the asset has no value in terms of the currency. Instead this is meant to record the integer d in the block chain.[8]

An *asset* can now be defined as a triple (h, ω, u) where h is a hash value, ω is an obligation and u is a preasset.[9] We denote the set of all assets by $\mathfrak{A}$ and use variables such as a and b to range over assets. Likewise we denote the set of all asset lists by $\overline{\mathfrak{A}}$ and use variables such as $\overline{a}$ and $\overline{b}$ to range over asset lists.

Given an asset (h, ω, u), the value of the asset is u if u represents currency and has no value otherwise. Given a list of assets, the total value is obtained by taking the sum while skipping those with no value.

A *transaction* τ is a pair (ι, o) where ι is an *input list* and o is an *output list*.[10] An *input* is a pair (α, h) where α is an address and h is a hash value (intended as an asset identifier). An *output* is a triple (α, ω, u) where α is an address, ω is an obligation, and u is a preasset.

For the most part we only consider transactions (ι, o) satisfying two properties: The input list ι must be nonempty and contain no duplicates. We call such transactions *valid*. Note that these conditions can be checked without reference to the state of the ledger.

The condition that every transaction has at least one input is to avoid the

[7] This corresponds to the type `obligation` defined in `Assets.v`.

[8] Preassets are defined as the inductive type `preasset` with two constructors `currency` and `publication` in `Assets.v`.

[9] This corresponds to the type `asset` in `Assets.v`.

[10] See the type `Tx` defined in the file `Transactions.v`.

case that assets with the same asset identifier could be created. In Bitcoin the coinbase transactions created by miners have an artificial input. This led to at least two identical transactions being created by a miner. Since they were identical, the corresponding asset identifiers for the two outputs were the same, something that should not have happened. A fix for this was later implemented in BIP34. Such a corner case appeared in our design while doing some of the Coq proofs, leading to the explicit requirement that valid transactions have at least one input.

Note that the ledger corresponding to the initial distribution will contain assets (effectively created by initial transactions with no inputs). The conditions on nonempty inputs does not apply here.

This is not to imply that all the currency units must be initially distributed. There can still be block rewards. It only means that the initial distribution cannot be empty or no transactions would ever be possible.

A *signed transaction* is a pair $(\tau, \overline{\sigma})$ where $\overline{\sigma}$ is a list of signatures. To check the validity of the signatures we need not only the asset identifiers, but also the obligations of the corresponding assets. Looking up the assets requires a representation of the ledger. For the moment we can defer this by making the dependence explicit. Let $\mathfrak{h}$ be an integer (representing the block height) and $\overline{a}$ be a list of assets. We say the *signatures in* $((\iota, o), \overline{\sigma})$ *are valid relative to* $\mathfrak{h}$ *and* $\overline{a}$ if for each $(\alpha, k) \in \iota$ there some asset $(k, (\overline{\alpha}, m, n), u) \in \overline{a}$ such that $n \geq \mathfrak{h}$ (the asset has matured) and there are (at least) m distinct addresses from $\overline{\alpha}$ for which there is a valid signature of the (hash of the) transaction among the signatures in $\overline{\sigma}$.[11]

5 Ledger Functions

In this section we introduce our first representation of the state of a ledger using ledger functions. The representation is intended to be mathematically simple and is not intended to be implemented. When we consider practical representations later, we will relate them to the ledger function representation in order to obtain desired results.

A *ledger function* is a function $f : \mathcal{A} \to \overline{\mathfrak{A}}$ from addresses to lists of assets.[12] We write $a \in f(\alpha)$ to mean a is an asset which occurs in the asset list $f(\alpha)$.

We say a hash value h is *spendable according to* f if there exist α, ω and u such that $(h, \omega, u) \in f(\alpha)$. In other words, f explicitly contains an asset with the identifier h.[13]

We can also define the notion of a hash value h being *spent according to* f. This is defined inductively by considering two cases.[14]

[11] See `tx_signatures_valid` in `Transactions.v`.

[12] See `statefun` in `LedgerStates.v`. Note that we are using the fact that functions are first-class values in Coq.

[13] See `sf_unsp_txout` in `LedgerStates.v`.

[14] See the inductively defined relation `sf_spent` in `LedgerStates.v`.

- If there is a transaction (ι, o), an integer i and an address α such that $(\alpha, h) \in \iota$ and $\ulcorner\ulcorner \iota, o \urcorner, \ulcorner i \urcorner\urcorner$ is spendable according to f, then h is spent according to f.

- If there is a transaction (ι, o), an integer i and an address α such that $(\alpha, h) \in \iota$ and $\ulcorner\ulcorner \iota, o \urcorner, \ulcorner i \urcorner\urcorner$ is spent according to f, then h is also spent according to f.

Note that it is infeasable to "check" if a hash value has been spent according to f, since would would need to recover a sequence of transactions leading from the transaction being spent to some currently spendable asset. Fortunately we will not need to check this. It is defined to state certain conditions on ledger functions, to which we turn next.

A ledger function f is *valid* if it satisfies the following properties:[15]

1. Each asset list $f(\alpha)$ has no duplicates.

2. If $a \in f(\alpha)$, $a' \in f(\alpha')$ and a and a' have the same asset identifier, then $\alpha = \alpha'$ and $a = a'$. That is, asset identifiers are unique.

3. For each asset $(h, \omega, u) \in f(\alpha)$ there is a transaction (ι, o)[16] and an integer i where the i^{th} member of o is (α, ω, u) and h is $\ulcorner\ulcorner \iota, o \urcorner, \ulcorner i \urcorner\urcorner$.

4. If h is a spent hash value according to f, then it is not spendable according to f.

Sometimes we will only be interested in second property, so we introduce a term for this. We say a ledger function f is *semivalid* if $\alpha = \alpha'$ and $a = a'$ whenever $a \in f(\alpha)$, $a' \in f(\alpha')$ and a and a' have the same asset identifier. Clearly every valid f is semivalid.

In order for a transaction to be valid relative to a ledger function f, we need to ensure that the sum of the asset values in the input (plus fees) is equal to the sum of the asset values in the output (plus rewards). The output asset values can be computed indepedently of the ledger since the preassets are explicit in the output list. Let $\overleftarrow{\Sigma}(o)$ be the sum of the values of the preassets in an output list o.[17]

The input asset values must be looked up in the ledger. Suppose ι is an input list for a transaction. Attempts to define this as a total function could be problematic. One would need to handle cases for which a ledger has no asset with the asset identifier. We can avoid this problem by defining a relation $\vec{\Sigma}_f(\iota, t)$ meaning *the inputs ι have total value t according to f*. The relation can be defined inductively as follows:[18]

- $\vec{\Sigma}_f(\cdot, 0)$ – no inputs have zero value.

[15] See `sf_valid_` and `sf_valid` in `LedgerStates.v`.

[16] Note that we do not require the transaction to be valid, as otherwise it would be impossible to bootstrap.

[17] See `asset_value_out` in `Assets.v`.

[18] See the inductively defined relation `statefun_asset_value_in` in `LedgerStates.v`.

- If $\vec{\Sigma}_f(\iota, t)$, $a \in f(\alpha)$, the asset a has identifier h and value u, then $\vec{\Sigma}_f((\alpha, h) :: \iota, u + t)$.
- If $\vec{\Sigma}_f(\iota, t)$, $a \in f(\alpha)$, the asset a has identifier h and no value then $\vec{\Sigma}_f((\alpha, h) :: \iota, t)$.

Note that if there is an input (α, h) in ι such that no asset with identifier h is in $f(\alpha)$, then there will be no t such that $\vec{\Sigma}_f(\iota, t)$.

A ledger function f *supports the transaction* (ι, o) *with fee* q *and reward* r if there is some t such that $\vec{\Sigma}_f(\iota, t)$ and $t + r = \overleftarrow{\Sigma}(o) + q$.[19]

It is easy to informally describe how a transaction $\tau = (\iota, o)$ can transform a ledger function f. For each input $(\alpha, h) \in \iota$, the corresponding asset is removed from $f(\alpha)$. Likewise for each output (α, ω, u) the asset $(\ulcorner\ulcorner\tau\urcorner, i\urcorner, \omega, u)$ (where i is the index of the output) is inserted into the list $f(\alpha)$. We denote the result of transforming f by τ by $[f]^\tau$.[20]

The following result is provable.[21]

Theorem 5.1. *If f is a valid ledger function, τ is a valid transaction and f supports τ with fee q and reward r, then $[f]^\tau$ is a valid ledger function.*

In addition to knowing the transformed ledger function is valid, we would like to know the total value of the assets in the ledger have been manipulated in accordance with the fees/rewards in the transaction. To obtain the total value of the assets in the ledger function f is theoretically easy: we append all the lists $f(\alpha)$ as α varies over all addresses in $\mathcal{A}$ and then sum the value of all the assets in this list.[22]

The following result is provable.[23]

Theorem 5.2. *Let f be a valid ledger function, τ be a valid transaction. Suppose f supports τ with fee q and reward r. Let t be the total value of the assets in f and t' be the total value of the assets in $[f]^\tau$. We have the following equation:*

$$t + r = t' + q$$

In other words, the value before plus the rewards is the same as the value afterwards plus the fees.

6 Approximations of Asset Lists

Let f be a ledger function. Consider a transaction τ with one input (α, h) and one output (β, ω, u) (where α and β are different). In order to check if the transaction is supported, we need to check if an asset a in the asset list $f(\alpha)$

[19] See `statefun_supports_tx` in `LedgerStates.v`.
[20] See `tx_statefun_trans` in `LedgerStates.v`.
[21] See `sf_tx_valid_thm` in `LedgerStates.v`.
[22] See `statefun_totalassets` and `total_units` in `LedgerStates.v`.
[23] See `sf_tx_valid_thm` in `LedgerStates.v`.

with identifier h. Assuming there is such an asset, we will need to remove it from $f(\alpha)$ and add an asset (k, ω, u) to $f(\beta)$.

Clearly we do not need full information about the asset lists $f(\alpha)$ and $f(\beta)$ in order to perform these tasks. Suppose $f(\alpha)$ is a list with four assets a_0, a_1, a_2 and a_3 where a_1 is the asset with identifier h. In order to verify a_1 is in this list it is enough to have an approximation of the list which contains a_0, a_1 and a hash value summarizing the rest of the list. When we transform the ledger function $[f]^\tau(\alpha)$ will be the asset list with three assets a_0, a_2 and a_3. If we only have the approximation, we will not be able to compute the new asset list, but we will be able to compute a new approximation consisting of a_0 along with the same hash value summarizing the rest of the list as before.

Likewise when we want to add the asset (k, ω, u) to $f(\beta)$ we do not need details of the asset list $f(\beta)$ beyond a hash value summarizing the list. The updated approximation would consist of the new asset (k, ω, u) along with the hash value summarizing what becomes the rest of the list.

We call such approximations $\mathcal{H}$-lists. $\mathcal{H}$-lists ($\mathcal{H}$) are defined by the following grammar:

$$\mathcal{H} ::= h| \cdot |(a, \mathcal{H})$$

where h ranges over hash values (not including $\bot$), a ranges over assets and $\cdot$ represents the empty $\mathcal{H}$-list.[24]

As usual we write $\ulcorner a \urcorner$ for the hash of an asset a. It is easy to use hashing of assets to be able to (injectively) hash asset lists in such a way that the empty list is sent to the empty hash value $\bot$. We write $\ulcorner \overline{a} \urcorner^\bot$ for such a hash of an asset list $\overline{a}$, where $\ulcorner - \urcorner^\bot$ applied the empty list gives $\bot$. It is defined as follows:[25]

- $\ulcorner \cdot \urcorner^\bot = \bot$.
- $\ulcorner a :: \overline{b} \urcorner^\bot = \ulcorner \ulcorner 3 \urcorner, \ulcorner a \urcorner \urcorner$ if $\ulcorner \overline{b} \urcorner^\bot = \bot$.
- $\ulcorner a :: \overline{b} \urcorner^\bot = \ulcorner \ulcorner 4 \urcorner, \ulcorner \ulcorner a \urcorner, k \urcorner \urcorner$ if $\ulcorner \overline{b} \urcorner^\bot = k$ where k is not $\bot$.

We write $\mathcal{H} \blacktriangleright \overline{a}$ if $\mathcal{H}$ approximates $\overline{a}$. This can be defined inductively as follows:[26]

- $\cdot \blacktriangleright \cdot$. (The empty $\mathcal{H}$-list approximates the empty asset list.)
- If $\ulcorner \overline{a} \urcorner^\bot = h$ (where h is not $\bot$), then $h \blacktriangleright \overline{a}$.
- If $\mathcal{H} \blacktriangleright \overline{a}$, then $(a, \mathcal{H}) \blacktriangleright a :: \overline{a}$.

In essence, $\mathcal{H} \blacktriangleright \overline{a}$ if $\mathcal{H}$ explicitly lists a prefix of $\overline{a}$ ending either with the empty list or a hash of the remainder of $\overline{a}$.

Let us return to our example transaction. Suppose we have two $\mathcal{H}$-lists $(a_0, (a_1, \mathcal{H}_1))$ and $\mathcal{H}_2$ where

$$(a_0, (a_1, \mathcal{H}_1)) \blacktriangleright f(\alpha) \text{ and } \mathcal{H}_2 \blacktriangleright f(\beta).$$

[24] See `hlist` in `MTrees.v`.
[25] See `hashassetlist` in `Assets.v` and `ohashlist` in `CryptoHashes.v`.
[26] See `approx_assetlist` in `MTrees.v`.

It is easy to see that we must have

$$(a_0, \mathcal{H}_1) \blacktriangleright [f]^\tau(\alpha) \text{ and } ((k, \omega, u), \mathcal{H}_2) \blacktriangleright [f]^\tau(\beta).$$

In general we will be able to operate on $\mathcal{H}$-lists instead of asset lists with the security of knowing there are (unique) asset lists being approximated.

There is another way to characterize when $\mathcal{H} \blacktriangleright \overline{a}$ holds. For each $\mathcal{H}$-list we can define and compute its *hash root* $\mathbf{R}\mathcal{H}$ as follows:[27]

- $\mathbf{R}\cdot = \bot$
- $\mathbf{R}h = h$
- $\mathbf{R}(a, h) = \ulcorner 3, \ulcorner a \urcorner \urcorner$ if $\mathbf{R}\mathcal{H} = \bot$.
- $\mathbf{R}(a, h) = \ulcorner 4, \ulcorner\ulcorner a \urcorner, \mathbf{R}\mathcal{H} \urcorner \urcorner$ if $\mathbf{R}\mathcal{H} \neq \bot$.

It is possible to prove $\mathbf{R}\mathcal{H} = \ulcorner \overline{a} \urcorner^\bot$ if and only if $\mathcal{H} \blacktriangleright \overline{a}$.

We write $a \in \mathcal{H}$ if a occurs in the explicit prefix of $\mathcal{H}$. Suppose $\mathcal{H} \blacktriangleright \overline{a}$. It is clear that if $a \in \mathcal{H}$, then $a \in \overline{a}$.[28] The converse does not generally hold.

7 Merkle Trees

In a first attempt to obtain practical representations of ledgers we consider a form of Merkle trees. In particular we will define $\mathcal{M}$-trees of depth n and represent ledgers as $\mathcal{M}$-trees of depth 160. The idea is that an address indicates which leaf of the $\mathcal{M}$-tree holds the asset.

The collection of $\mathcal{M}$-trees of depth n are defined recursively on n. An $\mathcal{M}$-tree of depth 0 is an $\mathcal{H}$-list. An $\mathcal{M}$-tree of depth $n+1$ is either an extended hash value h (possibly $\bot$) or a pair $(\mathcal{M}_0, \mathcal{M}_1)$ of $\mathcal{M}$-trees of depth n.

As with $\mathcal{H}$-lists, we define the *hash root* $\mathbf{R}\mathcal{M}$ of an $\mathcal{M}$-tree $\mathcal{M}$. The definition is by recursion on n as follows:[29]

- If $n = 0$, then $\mathcal{M}$ is an $\mathcal{H}$-list $\mathcal{H}$ and we take $\mathbf{R}\mathcal{M} := \mathbf{R}\mathcal{H}$.
- Suppose $\mathcal{M}$ is an $\mathcal{M}$-tree of depth $n+1$. There are two cases.
  - If $\mathcal{M}$ is an extended hash value h, then $\mathbf{R}\mathcal{M} := h$.
  - If $\mathcal{M}$ is $(\mathcal{M}_0, \mathcal{M}_1)$ where $\mathcal{M}_0$ and $\mathcal{M}_1$ are $\mathcal{M}$-trees of depth n, then $\mathbf{R}\mathcal{M} := \ulcorner \mathbf{R}\mathcal{M}_0, \mathbf{R}\mathcal{M}_0 \urcorner^\bot$.

We next wish to define when an $\mathcal{M}$-tree of depth 160 approximates a ledger function f. We will generalize and define $\mathcal{M} \blacktriangleright f$ for $\mathcal{M}$-trees of depth n and functions $f : \mathcal{B}_n \to \overline{\mathfrak{A}}$. The definition is by recursion on n.[30]

[27] See `hlist_hashroot` in `MTrees.v`.
[28] See `subqh_In_hlist` in `Mtrees.v`.
[29] See `mtree_hashroot` in `MTrees.v`.
[30] See `mtree_approx_fun_p` in `MTrees.v`.

Note that when $n = 0$, $f : \mathcal{B}_0 \to \overline{\mathfrak{A}}$ is a function from the singleton set $\mathcal{B}_0 = \{\cdot\}$ to $\overline{\mathfrak{A}}$. Moreover, $\mathcal{M}$ of depth 0 is an $\mathcal{H}$-list. Hence we can define $\mathcal{M} \blacktriangleright f$ to hold if $\mathcal{M} \blacktriangleright f(\cdot)$ holds.

Next let $\mathcal{M}$ be an $\mathcal{M}$-tree of depth $n+1$ and $f : \mathcal{B}_{n+1} \to \overline{\mathfrak{A}}$. Define $f_0 : \mathcal{B}_n \to \overline{\mathfrak{A}}$ and $f_1 : \mathcal{B}_n \to \overline{\mathfrak{A}}$ to be $f_0(\mathfrak{b}) = f(0\mathfrak{b})$ and $f_1(\mathfrak{b}) = f(1\mathfrak{b})$. To define $\mathcal{M} \blacktriangleright f$ we consider two cases.

1. If $\mathcal{M}$ is $(\mathcal{M}_0, \mathcal{M}_1)$, then $\mathcal{M} \blacktriangleright f$ holds if $\mathcal{M}_0 \blacktriangleright f_0$ and $\mathcal{M}_1 \blacktriangleright f_1$.
2. If $\mathcal{M}$ is a hash value h, then $\mathcal{M} \blacktriangleright f$ holds if there exist $\mathcal{M}$-trees $\mathcal{M}_0$ and $\mathcal{M}_1$ of depth n such that $h = \ulcorner \mathbf{R}\mathcal{M}_0, \mathbf{R}\mathcal{M}_1 \urcorner^{\perp}$, $\mathcal{M}_0 \blacktriangleright f_0$ and $\mathcal{M}_1 \blacktriangleright f_1$.

We say $\mathcal{M}$ is a *valid* $\mathcal{M}$-tree of depth 160 if there is a valid ledger function f such that $\mathcal{M} \blacktriangleright f$.[31]

For an $\mathcal{M}$-tree $\mathcal{M}$ of depth n and a bit sequence $\mathfrak{a} \in \mathcal{B}_n$ we define when $\mathcal{M}$ *supports* $\mathfrak{a}$ by recursion on n:[32]

- If $n = 0$, then $\mathcal{M}$ supports $\mathfrak{a}$.
- For $n+1$ we distinguish four cases:
  1. If $\mathcal{M}$ is $\perp$, then $\mathcal{M}$ supports $\mathfrak{a}$.
  2. If $\mathcal{M}$ is a hash value (not including $\perp$), then $\mathcal{M}$ does not support $\mathfrak{a}$.
  3. If $\mathcal{M}$ is $(\mathcal{M}_0, \mathcal{M}_1)$ and $\mathfrak{a}$ is $0\mathfrak{b}$, then $\mathcal{M}$ supports $\mathfrak{a}$ if $\mathcal{M}_0$ supports $\mathfrak{b}$.
  4. If $\mathcal{M}$ is $(\mathcal{M}_0, \mathcal{M}_1)$ and $\mathfrak{a}$ is $1\mathfrak{b}$, then $\mathcal{M}$ supports $\mathfrak{a}$ if $\mathcal{M}_1$ supports $\mathfrak{b}$.

When $n = 160$ this defines when an $\mathcal{M}$-tree of depth 160 *supports an address* α.

In a similar way, we define when an $\mathcal{M}$-tree of depth 160 *supports an asset* a *at an address* α by generalizing and defining when an $\mathcal{M}$-tree of depth n *supports an asset* a *at* $\mathfrak{a} \in \mathcal{B}_n$.[33]

- If $n = 0$, then $\mathcal{M}$ (an $\mathcal{H}$-list) supports a at $\mathfrak{a}$ if $a \in \mathcal{M}$.
- For $n+1$ we distinguish three cases:
  1. If $\mathcal{M}$ is an extended hash value, then $\mathcal{M}$ does not support a at $\mathfrak{a}$.
  2. If $\mathcal{M}$ is $(\mathcal{M}_0, \mathcal{M}_1)$ and $\mathfrak{a}$ is $0\mathfrak{b}$, then $\mathcal{M}$ supports a at $\mathfrak{a}$ if $\mathcal{M}_0$ supports a at $\mathfrak{b}$.
  3. If $\mathcal{M}$ is $(\mathcal{M}_0, \mathcal{M}_1)$ and $\mathfrak{a}$ is $1\mathfrak{b}$, then $\mathcal{M}$ supports a at $\mathfrak{a}$ if $\mathcal{M}_1$ supports a at $\mathfrak{b}$.

Let $\mathcal{M}$ be an $\mathcal{M}$-tree of depth 160 and $\tau = (\iota, o)$ be a transaction.

We say $\mathcal{M}$ *can support* τ if two conditions hold:[34]

[31] See `mtree_valid_` and `mtree_valid` in `MTrees.v`.
[32] See `mtree_supports_addr` in `MTrees.v`.
[33] See `mtree_supports_asset` in `MTrees.v`.
[34] See `mtree_can_support_tx` in `MTrees.v`.

1. For each $(\alpha, \omega, u) \in o$ $\mathcal{M}$ supports α.

2. For each $(\alpha, h) \in \iota$ there is an asset a supported by $\mathcal{M}$ at α with asset identifier h.

In order to say the transaction is supported we need to also check that the total value of the assets spent correspond to the new assets that will be created, up to fees and rewards. As before the total value being created can be easily computed from the output list o since the preassets are explicit in o as $\overleftarrow{\Sigma}(o)$. The total value being spent is not explicit in ι since we only have the asset identifiers. Recall that with ledger functions f we defined a relation $\vec{\Sigma}_f(\iota, t)$ meaning t is the total value of the input list ι according to f. We will follow the same strategy to define a relation $\vec{\Sigma}_{\mathcal{M}}(\iota, t)$ meaning t is the total value of the input list ι according to $\mathcal{M}$. The relation is defined inductively.[35]

- $\vec{\Sigma}_{\mathcal{M}}(\cdot, 0)$ – no inputs have zero value.

- If $\vec{\Sigma}_{\mathcal{M}}(\iota, t)$, a is an asset supported by $\mathcal{M}$ at α and a has identifier h and value u, then $\vec{\Sigma}_{\mathcal{M}}((\alpha, h) :: \iota, u + t)$.

- If $\vec{\Sigma}_{\mathcal{M}}(\iota, t)$, a is an asset supported by $\mathcal{M}$ at α and a has identifier h and no value, then $\vec{\Sigma}_{\mathcal{M}}((\alpha, h) :: \iota, t)$.

Let $\mathcal{M}$ be an $\mathcal{M}$-tree of depth 160 and $\tau = (\iota, o)$ be a transaction. We say $\mathcal{M}$ *supports* τ *with fees* q *and reward* r if the following conditions hold:[36]

1. For each $(\alpha, \omega, u) \in o$ $\mathcal{M}$ supports α.

2. There is an integer t such that $\vec{\Sigma}_{\mathcal{M}}(\iota, t)$ and $t + r = \overleftarrow{\Sigma}(o) + q$.

It is easy to prove that if there is a t such that $\vec{\Sigma}_{\mathcal{M}}(\iota, t)$, then for every input (α, h) in ι there is an asset a supported by $\mathcal{M}$ such that a has asset identifier h. Consequently if $\mathcal{M}$ supports τ with fees q and reward r, then $\mathcal{M}$ can support τ.[37]

Suppose $\mathcal{M}$ can support τ and f is a semivalid ledger function such that $\mathcal{M} \blacktriangleright f$. Then $\mathcal{M}$ supports τ if and only if f supports τ (with the same fees and rewards). That is, so long as $\mathcal{M}$ can support τ we have enough information in $\mathcal{M}$ to determine if the ledger function it approximates supports τ. This equivalence can be refined into the following two results.[38]

Theorem 7.1. *Let* $\mathcal{M}$ *be an* $\mathcal{M}$*-tree of depth* 160 *and* f *be a semivalid ledger function such that* $\mathcal{M} \blacktriangleright f$*. If* $\mathcal{M}$ *supports* τ *with fees* q *and reward* r*, then* f *supports* τ *with fees* q *and reward* r*.*

[35] See `mtree_asset_value_in` in `MTrees.v`.
[36] See `mtree_supports_tx` in `MTrees.v`.
[37] See `mtree_supports_tx_can_support` in `MTrees.v`.
[38] See `mtree_supports_tx_statefun` and `mtree_supports_tx_statefun_conv` in `MTrees.v`.

Theorem 7.2. *Let $\mathcal{M}$ be an $\mathcal{M}$-tree of depth* 160 *and f be a semivalid ledger function such that $\mathcal{M} \blacktriangleright f$. Suppose $\mathcal{M}$ can support τ. If f supports τ with fees q and reward r, then $\mathcal{M}$ supports τ with fees q and reward r.*

We use $[\mathcal{M}]^\tau$ to denote the result of transforming an $\mathcal{M}$-tree $\mathcal{M}$ of depth 160 using a transaction τ. As above this can be defined by appropriately generalizing and using recursion on n. We omit the details.[39]

Part of the transformation of $\mathcal{M}$ involves removing spent assets from certain asset lists. As a result we may be left with a subtree of the form $(\bot, \bot)$ even though this is equivalent to the simpler empty tree $\bot$. We can *normalize* an $\mathcal{M}$ by recursively simplifying subtrees $(\bot, \bot)$ to be $\bot$.[40] We use $\langle \mathcal{M} \rangle$ to denote the result of normalizing $\mathcal{M}$.

The following result holds.[41]

Theorem 7.3. *Let $\mathcal{M}$ be an $\mathcal{M}$-tree of depth* 160*, f be a valid ledger function and τ be a transaction. Suppose $\mathcal{M}$ supports τ. If $\mathcal{M} \blacktriangleright f$, then $[\mathcal{M}]^\tau \blacktriangleright [f]^\tau$ and $\langle [\mathcal{M}]^\tau \rangle \blacktriangleright [f]^\tau$.*

Next we would like a result similar to Theorem 5.2 for $\mathcal{M}$-trees. To state this result we need to know what it means for $\overline{a}$ to be a list of total assets of $\mathcal{M}$. In the case of a ledger function f we could simply concatenate all the asset lists at the leaves (though this operation is somewhat impractical). An $\mathcal{M}$-tree $\mathcal{M}$ will often omit leaves and even when the leaf is there the $\mathcal{H}$-list may not list all the assets. Nevertheless we can say an $\mathcal{H}$-list $\mathcal{H}$ *has total assets* $\overline{a}$ if $\mathcal{M} \blacktriangleright \overline{a}$ and extend this to say an $\mathcal{M}$-tree $\mathcal{M}$ of depth $n+1$ *has total assets* $\overline{a}$ if either[42]

- $\mathcal{M}$ is $(\mathcal{M}_0, \mathcal{M}_1)$ and there are asset lists $\overline{b}$ and $\overline{c}$ such that $\mathcal{M}_0$ has total assets $\overline{b}$, $\mathcal{M}_1$ has total assets $\overline{c}$ and $\overline{a}$ is the concatenation of $\overline{b}$ and $\overline{c}$.
- $\mathcal{M}$ is an extended hash value h and there are $\mathcal{M}$-trees $\mathcal{M}_0$ and $\mathcal{M}_1$ of depth n and asset lists $\overline{b}$ and $\overline{c}$ such that h is $\ulcorner \mathbf{R}\mathcal{M}_0, \mathbf{R}\mathcal{M}_1 \urcorner^\bot$, $\mathcal{M}_0$ has total assets $\overline{b}$, $\mathcal{M}_1$ has total assets $\overline{c}$ and $\overline{a}$ is the concatenation of $\overline{b}$ and $\overline{c}$.

The sum of the values of the total assets of an $\mathcal{M}$-tree before being transformed and normalized corresponds to the sum of the values after the transformation and normalization, in accordance with fees and rewards.[43]

Theorem 7.4. *Let $\mathcal{M}$ be a valid $\mathcal{M}$-tree and τ be a valid transaction such that $\mathcal{M}$ supports τ with fee q and reward r. Suppose $\overline{a}$ lists the total assets of $\mathcal{M}$ and $\overline{b}$ lists the total assets of $\langle [\mathcal{M}]^\tau \rangle$. Let t be the sum of the values of $\overline{a}$ and t' be the sum of the values of $\overline{b}$. We have $t' + q = t + r$.*

[39] See `tx_mtree_trans_` and `tx_mtree_trans` in `MTrees.v`.
[40] See `normalize_mtree` in `MTrees.v`.
[41] See `mtree_approx_trans` and `mtree_normal_approx_trans` in `MTrees.v`.
[42] See `mtree_totalassets` in `MTrees.v`.
[43] See `mtree_normalize_tx_asset_value_sum` in `MTrees.v`.

8 Compact Trees

In this section we consider a different representation we call *compact trees*. The intention is that compact trees are a more efficient representation than $\mathcal{M}$-trees, though this could only be tested by an implementation. There is an argument that it is better to have a slightly less efficient representation (e.g., $\mathcal{M}$-trees) rather than making the representation increasingly complicated. The more complex the system is, the more difficult it is for people to understand. On the other hand, the formal development (followed properly) ensures there will be no bugs.

Unlike $\mathcal{M}$-trees we expect all compact trees to be nonempty. When we explicitly need to consider an "empty" tree we will again use $\perp$ and speak of *extended compact trees*.

We define the notion of a *compact tree of depth* n by recursion on n.[44]

- If $n = 0$, then a compact tree of depth n is an $\mathcal{H}$-list other than the empty $\mathcal{H}$-list.
- A compact tree of depth $n + 1$ has one of 5 possible shapes:
  1. a pair $(\mathfrak{a}, \mathcal{H})$ where $\mathfrak{a}$ is an $n + 1$-bit sequence and $\mathcal{H}$ is a nonempty $\mathcal{H}$-list,
  2. a hash value (not including $\perp$),
  3. a compact tree $\mathcal{C}_0$ of depth n along with a tag indicating it is the left child,
  4. a compact tree $\mathcal{C}_1$ of depth n along with a tag indicating it is the right child,
  5. or a pair $(\mathcal{C}_0, \mathcal{C}_1)$ of compact trees of depth n.

We let $\hat{\mathcal{C}}$ be the $\mathcal{M}$-tree corresponding to $\mathcal{C}$ given by recursion over the depth as follows.[45]

- If $n = 0$, then $\mathcal{C}$ is a nonempty $\mathcal{H}$-list and $\hat{\mathcal{C}}$ is this $\mathcal{H}$-list.
- Suppose $\mathcal{C}$ is a compact tree of depth $n + 1$. We consider 5 cases.
  1. If $\mathcal{C}$ is $(\mathfrak{a}, \mathcal{H})$, then $\hat{\mathcal{C}}$ is the $\mathcal{M}$-tree of depth $n + 1$ where the leaf at $\mathfrak{a}$ is the (nonempty) $\mathcal{H}$-list $\mathcal{H}$ and every other leaf is empty.
  2. If $\mathcal{C}$ is the hash value h, then $\hat{\mathcal{C}}$ is the same hash value.
  3. If $\mathcal{C}$ is $\mathcal{C}_0$ with a tag indicating this is the left child, then $\hat{\mathcal{C}}$ is $(\hat{\mathcal{C}_0}, \perp)$.
  4. If $\mathcal{C}$ is $\mathcal{C}_1$ with a tag indicating this is the right child, then $\hat{\mathcal{C}}$ is $(\perp, \hat{\mathcal{C}_1})$.
  5. If $\mathcal{C}$ is $(\mathcal{C}_0, \mathcal{C}_1)$, then $\hat{\mathcal{C}}$ is $(\hat{\mathcal{C}_0}, \hat{\mathcal{C}_1})$.

[44] See `ctree` in `CTrees.v`.
[45] See `ctree_mtree` in `CTrees.v`.

If we speak of extended compact trees, we take $\hat{\perp}$ to be the $\mathcal{M}$-tree $\perp$ (at the corresponding depth).

We can also define the *hash root* $\mathbf{R}\mathcal{C}$ of a compact tree $\mathcal{C}$ of depth n (again by recursion on n). We omit the definition, but note that the definition is such that $\mathbf{R}\mathcal{C}$ is the same as $\mathbf{R}\hat{\mathcal{C}}$. That is, the hash root of a compact tree is the same as the hash root of the $\mathcal{M}$-tree it represents.

Let $\mathcal{C}$ be an extended compact tree of depth 160 and f be a ledger function. We would like to define the relation $\mathcal{C} \blacktriangleright f$. Since this relation is only of mathematical interest (i.e., there is no intention to implement a function testing $\blacktriangleright$), we can easily define the concept by falling back on the $\mathcal{M}$-tree $\hat{\mathcal{C}}$. That is, we say $\mathcal{C} \blacktriangleright f$ holds if $\hat{\mathcal{C}} \blacktriangleright f$.[46] We also say $\mathcal{C}$ is a *valid compact tree* if $\hat{\mathcal{C}}$ is a valid $\mathcal{M}$-tree.[47]

Let $\mathcal{C}$ be a compact tree of depth 160 and τ be a transaction. Just as with $\mathcal{M}$-trees we wish to define when $\mathcal{C}$ can support τ, and when $\mathcal{C}$ supports τ with fee q and reward r. We could again fall back on the $\mathcal{M}$-tree $\hat{\mathcal{C}}$ and the definitions in the previous section. However, in this case it is clear that we would need a practical implementation of functions testing whether $\mathcal{C}$ can support or does support τ. Instead of using $\hat{\mathcal{C}}$, we follow the same strategy used in Section 7 to define when $\mathcal{C}$ *can support* τ and to define when $\mathcal{C}$ *supports* τ *with fee* q *and reward* r. We omit the details here.[48] One can then prove, for example, that $\mathcal{C}$ supports τ with fee q and reward r if and only if $\hat{\mathcal{C}}$ supports τ with fee q and reward r.[49] Such an equivalence allows one to lift the results for $\mathcal{M}$-trees to corresponding results for compact trees. We give two such results explicitly here.[50]

Theorem 8.1. *Let $\mathcal{C}$ be a compact tree of depth* 160 *and f be a semivalid ledger function such that $\mathcal{C} \blacktriangleright f$. If $\mathcal{C}$ supports τ with fees q and reward r, then f supports τ with fees q and reward r.*

Theorem 8.2. *Let $\mathcal{C}$ be a compact tree of depth* 160 *and f be a semivalid ledger function such that $\mathcal{C} \blacktriangleright f$. Suppose $\mathcal{C}$ can support τ. If f supports τ with fees q and reward r, then $\mathcal{C}$ supports τ with fees q and reward r.*

We also need to define $[\mathcal{C}]^\tau$, i.e., the result of transforming a compact tree by a transaction. Since assets will be deleted and others will be added, it is clear that we cannot ignore the empty tree. Hence we define $[\mathcal{C}]^\tau$ for extended compact trees $\mathcal{C}$ (so $\mathcal{C}$ may be $\perp$). The definition of $[\mathcal{C}]^\tau$ is tedious and we omit the full details here.[51] Technically the information in τ is split into the relevant parts of the input and output for the part of the tree under consideration. At each stage if it is clear there is no input or output affecting this part of the tree, then the tree is returned unchanged. Suppose we are considering a compact

[46] See `octree_approx_fun_p` in `CTrees.v`.
[47] See `ctree_valid` in `CTrees.v`.
[48] See `ctree_can_support_tx` and `ctree_supports_tx` in `CTrees.v`.
[49] See `octree_mtree_supports_tx` and `mtree_ctree_supports_tx` in `CTrees.v`.
[50] See `ctree_supports_tx_statefun` and `ctree_supports_tx_statefun_conv` in `CTrees.v`.
[51] See `tx_octree_trans_` and `tx_octree_trans` in `CTrees.v`.

tree $(\mathcal{C}_0, \mathcal{C}_1)$ of depth $n+1$ and there is some input or output which will affect this tree. We would make an appropriate recursive call to obtain two extended compact trees $\mathcal{C}_0'$ and $\mathcal{C}_1'$. We then must consider four subcases. If both $\mathcal{C}_0'$ and $\mathcal{C}_1'$ are $\perp$, then $\perp$ is returned. If only $\mathcal{C}_1'$ is $\perp$, then the compact tree $\mathcal{C}_0'$ with a tag indicating it is the left child is returned. The case where only $\mathcal{C}_0'$ is $\perp$ analogous. Finally if neither is $\perp$, then the compact tree $(\mathcal{C}_0', \mathcal{C}_1')$ is returned. In essence the normalization operation from $\mathcal{M}$-trees must be inlined for compact trees. This is somewhat painful in the formalization, but is more realistic. There is no reason why transforming a tree and normalizing a tree should require two passes through the tree.

The following result is provable.[52]

Theorem 8.3. *Let $\mathcal{C}$ be an extended compact tree of depth* 160 *and f be a valid ledger function. Suppose $\mathcal{C}$ supports τ. If $\mathcal{C} \blacktriangleright f$, then $[\mathcal{C}]^\tau \blacktriangleright [f]^\tau$.*

As before we also have a result about the sum of the values of all the assets before and after a transaction has been used to transform a compact tree.[53] A list of the total assets of $\mathcal{C}$ is simply a list of the total assets of the $\mathcal{M}$-tree $\hat{\mathcal{C}}$.

Theorem 8.4. *Let $\mathcal{C}$ be a valid compact tree and τ be a valid transaction such that $\mathcal{C}$ supports τ with fee q and reward r. Suppose $\overline{a}$ lists the total assets of $\mathcal{C}$ and $\overline{b}$ lists the total assets of $[\mathcal{C}]^\tau$. Let t be the sum of the values of $\overline{a}$ and t' be the sum of the values of $\overline{b}$. We have $t' + q = t + r$.*

Finally we discuss a way to extend a compact tree by grafting on subtrees. To motivate this recall that blocks will be separated into headers and deltas. Assuming the cryptocurrency is proof of stake, the header should contain a compact tree with just enough information to verify the claimed stake of the forger. The delta, on the other hand, must have a compact tree with enough information to support all the transactions in the block. To avoid repeating some of the information in both the header and the delta, we give a way to specify in the delta how to extend the compact tree in the header to be a more informative compact tree.

A *graft* G is a list of $(h, n, \mathcal{C})$ where h is a hash value,[54] n is an integer and $\mathcal{C}$ is a compact tree of depth n. A graft G is *valid* if for each $(h, n, \mathcal{C})$, the hash root $\mathbf{R}\mathcal{C}$ of $\mathcal{C}$ is h.

Given a graft G and a compact tree $\mathcal{C}$ of depth n, we can obtain a new compact tree $\mathcal{C}^G$ of depth n by a simple recursion.[55] The idea is that when $\mathcal{C}$ is a hash value h where $(h, n, \mathcal{C}')$ is in G, then we replace the hash value with $\mathcal{C}'$.

We have the following result.[56]

Theorem 8.5. *Suppose G is a graft, $\mathcal{C}$ is a compact tree of depth n and $f : \mathcal{B}_n \to \overline{\mathfrak{A}}$ is a function. If G is a valid graft and $\mathcal{C} \blacktriangleright f$, then $\mathcal{C}^G \blacktriangleright f$.*

[52] See `octree_approx_trans` in `CTrees.v`.
[53] See `octree_tx_asset_value_sum` in `CTrees.v`.
[54] See `cgraft` in `CTreeGrafting.v`.
[55] See `ctree_cgraft` in `CTreeGrafting.v`.
[56] See `ctree_approx_fun_cgraft_valid` in `CTreeGrafting.v`.

Essentially this guarantees that when we graft onto a compact tree approximating f, then we obtain a compact tree approximating the same f.

9 Blocks and Chains

We now have enough infrastructure to specify block headers, block deltas, blocks, block header chains and block chains. At this point we will commit to a proof of stake consensus system, though there is nothing to rule out other possibilities. In this section we will present definitions and results in a way that is somewhat simplified from the Coq formalization. The Coq formalization includes some dependencies intended to stand in for proof of stake information (current target, time stamp, checking a hit, and so on). We ignore most of this information here. The reader can find the full details in the Coq code.[57]

The blocks will contain enough information to approximate the current ledger, transform this approximation using the transactions in the block and hence compute the ledger root of the transformed ledger. This new ledger root is simply a single hash value which is stored in the block header and summarizes the new state of the ledger after the block has been processed. We say a hash value $\mathfrak{n}$ is a *valid ledger root* if $\mathfrak{n}$ is valid as a compact tree of depth 160.

A *block header* is a tuple

$$(h, \mathfrak{n}, u, \alpha, k, \sigma, \mathcal{C})$$

where h is an extended hash value (the hash of the previous block or $\perp$ for the genesis block), $\mathfrak{n}$ is a hash value (the hash root of the new ledger), u is the value of the stake of the forger of the block header, α is the address where the stake is held, k is the asset identifier of the stake, σ is a signature for the block by α and $\mathcal{C}$ is a compact tree of depth 160. A *block delta* is a tuple

$$(q, o, G, \overline{\tau})$$

where q is an integer (the total fees in the block), o is the output list for the coinstake transaction, G is a graft and $\overline{\tau}$ a list of signed transactions. A *block* is a pair (Γ, Δ) where Γ is a block header and Δ is a block delta.[58]

Let $\mathfrak{h}$ be an integer and $\mathfrak{p}$ be a hash value. We say a block header

$$\Gamma = (h, \mathfrak{n}, u, \alpha, k, \sigma, \mathcal{C})$$

is *valid at height* $\mathfrak{h}$ *with previous ledger root* $\mathfrak{p}$ if the following conditions hold:[59]

1. The signature σ is a valid signature using the private key for α of the hash of the data in $(h, \mathfrak{n}, u, \alpha, k)$.

2. The hash root $\mathbf{R}\mathcal{C}$ is $\mathfrak{p}$. (That is, the approximation $\mathcal{C}$ of the previous ledger is legitimate.)

[57] See `Blocks.v`.
[58] See `BlockHeader`, `BlockDelta` and `Block` in `Blocks.v`.
[59] See `valid_BlockHeader` in `Blocks.v`.

3. There is an obligation $\omega = (\overline{\alpha}, m, n)$ such that $\mathcal{C}$ supports the asset (k, ω, u) at α and $n > \mathfrak{h} + 1000$.[60]

Let $\mathfrak{B} = (\Gamma, \Delta)$ be a block where

$$\Gamma = (h, \mathfrak{n}, u, \alpha, k, \sigma, \mathcal{C})$$

and

$$\Delta = (q, o, G, (((\iota_1, o_1), \sigma_1) \cdots ((\iota_l, o_l), \sigma_l))))).$$

We define the *compact tree of* $\mathfrak{B}$, denoted by $\mathcal{C}_{\mathfrak{B}}$, to be $\mathcal{C}^G$ (the compact tree in the header grafted with the graft in the delta). We define the *transaction of* $\mathfrak{B}$, denoted by $\tau_{\mathfrak{B}}$, to be (ι', o') where

$$\iota' = (h, \mathfrak{n}, n) :: \iota_1 + \cdots + \iota_l$$

and

$$o' = o + o_1 + \cdots + o_l$$

where $+$ denotes list concatentation. In words, $\tau_{\mathfrak{B}}$ is formed by combining all the inputs (including the stake asset) to be spent by the block and all the outputs to be created by the block. A natural question is whether or not this transaction is supported by the approximation of the state of the ledger $\mathcal{C}^G$ given in the block. We will define when a block is valid with enough conditions to assume the transaction is supported by $\mathcal{C}^G$, assuming $\mathcal{C}^G$ is a valid compact tree. We say the block $\mathfrak{B}$ is *valid at height* $\mathfrak{h}$ *with previous ledger root* $\mathfrak{p}$ *and reward* r if the following conditions hold:[61]

1. The block header Γ is valid at height $\mathfrak{h}$ with previous ledger root $\mathfrak{p}$.
2. There is an output of the form (α, ω, u) where (k, ω, u) is the staked asset,[62] and $n > \mathfrak{h} + 1000$ for every other output $(\beta, (\overline{\alpha}, m, n), v) \in o$.[63]
3. For every $(\beta, \omega, v) \in o$ the compact tree $\mathcal{C}^G$ supports the address β.
4. The total value of the preassets listed in o is $u + r + q$. (That is, the value of the output of the coinstake transaction will be the input stake plus the reward plus the fees paid by the transactions in the block.)
5. There are no duplicate signed transactions in the list

$$((\iota_1, o_1), \sigma_1) \cdots ((\iota_l, o_l), \sigma_l).$$

[60]This last condition is that a node is only allowed to use stake to forge if there is an obligation preventing it from being spent for the next 1000 blocks. This is intended to make certain kinds of attacks on proof of stake networks more difficult.

[61]See `valid_Block` in `Blocks.v`.

[62]The staked asset is unchanged. The rewards and fees must be other outputs which are distributed at the discretion of the forger of the block.

[63]This condition prevents newly minted coins from being spent until they have matured 1000 blocks later. It is also an attempt to prevent certain attacks on cryptocurrency networks.

6. For each $i \in \{1, \ldots, l\}$ there is an asset list $\overline{a}$ such that
   (a) the signatures in $((\iota_i, o_i), \sigma_i)$ are valid relative to $\mathfrak{h}$ and $\overline{a}$ and
   (b) for every $(\beta, h') \in \iota_i$ there is an $a \in \overline{a}$ such that the asset identifier for a is h' and $\mathcal{C}^G$ supports a at address β.
7. For each $i \in \{1, \cdots, l\}$ the transaction (ι_i, o_i) is valid.
8. For each $i \in \{1, \cdots, l\}$ $(\alpha, h) \notin \iota_i$. (None of the transactions attempts to spend the asset being staked.)
9. For each $i, j \in \{1, \cdots, l\}$ if $\iota_i \cap \iota_j \neq \emptyset$, then $i = j$. (No two distinct transactions attempt to spend the same asset.)
10. The graft G is valid.
11. For each $i \in \{1, \cdots, l\}$ there is a fee q' such that $\mathcal{C}^G$ supports (ι_i, o_i) with fee q' and reward 0.
12. There is a total value t of the input assets of $\iota_1 + \cdots + \iota_l$ according to $\mathcal{C}^G$ and $t' + q = t$ where t' is the sum of the values of the preassets in $o_1 + \cdots + o_l$. (That is, the total output values plus the total fees equals the total value of the inputs, all ignoring the coinstake transaction.)
13. There is a compact tree $\mathcal{C}'$ of depth 160 such that the hash root $\mathbf{R}\mathcal{C}'$ of $\mathcal{C}'$ is $\mathfrak{n}$ and $[\tau_{\mathfrak{B}}]^{\mathcal{C}^G} = \mathcal{C}'$. (That is, if we transform $\mathcal{C}^G$ by the transaction of the block then we obtain a compact tree with the hash root $\mathfrak{n}$ declared in the block header. This verifies that $\mathfrak{n}$ is the hash root of the ledger after processing the block.)

We have the following result:[64]

Theorem 9.1. *Suppose $\mathfrak{B}$ is a valid block at height $\mathfrak{h}$ with previous ledger root $\mathfrak{p}$ and reward r. If $\mathcal{C}_{\mathfrak{B}}$ is a valid compact tree, then $\mathcal{C}_{\mathfrak{B}}$ supports the transaction $\tau_{\mathfrak{B}}$ with fee 0 and reward r.*

In Theorem 9.1 the reader may be concerned that the result only holds under the assumption that $\mathcal{C}_{\mathfrak{B}}$ is a valid compact tree. One might suspect that if $\mathfrak{B}$ is a valid block, then $\mathcal{C}_{\mathfrak{B}}$ must be a valid compact tree. This is not true. In order to directly check a compact tree is valid, we would need to find a ledger function approximated by the corresponding $\mathcal{M}$-tree. Since the conditions for a block to be valid were chosen to be easily checkable, we have not included conditions which would directly guarantee validity of $\mathcal{C}_{\mathfrak{B}}$. However, as we pass from single blocks to block chains we will know in fact the compact trees $\mathcal{C}_{\mathfrak{B}}$ will be valid for blocks $\mathfrak{B}$ in the block chain (assuming we started from a valid ledger root). In other words, validity of the compact trees will be an invariant indirectly guaranteed by the block chain.

[64] See `tx_of_Block_supported` in `Blocks.v`.

A *block chain of height* $\mathfrak{h}$ is a sequence of $\mathfrak{h}+1$ blocks:[65]

$$\mathfrak{B}_0 \cdots \mathfrak{B}_{\mathfrak{h}}$$

and a *block header chain of height* $\mathfrak{h}$ is a sequence of $\mathfrak{h}+1$ block headers:[66]

$$\Gamma_0 \cdots \Gamma_{\mathfrak{h}}.$$

We say the *last ledger root* of a block chain or block header chain is the new ledger root $\mathfrak{n}$ specified by the last block or block header.

Let $\mathfrak{g}$ be a hash value. It is intended that $\mathfrak{g}$ is the ledger root of the initial ledger (seeded with an initial distribution). Let R be a reward function from integers to integers. The intention is that $R(\mathfrak{h})$ is the block reward at height $\mathfrak{h}$. We define when a block chain $\mathfrak{C}$ is *valid from genesis ledger root* $\mathfrak{g}$ *with reward function* R by recursion:[67] If $\mathfrak{C}$ is height 0 and consists only of the genesis block $\mathfrak{B}_0$, then it is valid with genesis ledger root $\mathfrak{g}$ with reward function R if $\mathfrak{B}_0$ is a valid block at height 0 with previous ledger root $\mathfrak{g}$ and reward $R(0)$. Suppose $\mathfrak{C}$ is

$$\mathfrak{B}_0 \cdots \mathfrak{B}_{\mathfrak{h}} \mathfrak{B}_{\mathfrak{h}+1}.$$

Let $\mathfrak{p}$ be the new ledger root in $\mathfrak{B}_{\mathfrak{h}}$. We say $\mathfrak{C}$ is valid from genesis ledger root $\mathfrak{g}$ with reward function R if

$$\mathfrak{B}_0 \cdots \mathfrak{B}_{\mathfrak{h}}$$

is valid from genesis ledger root $\mathfrak{g}$ and reward function R and $\mathfrak{B}_{\mathfrak{h}+1}$ is a valid block at height $\mathfrak{h}+1$ with previous ledger root $\mathfrak{p}$ and reward $R(\mathfrak{h}+1)$.

We have the following result guaranteeing that if we start from a valid genesis ledger root then we can only reach valid ledger roots using a valid block chain.[68]

Theorem 9.2. *Suppose* $\mathfrak{C}$ *is a valid block chain with genesis ledger root* $\mathfrak{g}$ *and reward function* R*. If* $\mathfrak{g}$ *is a valid ledger root, then the last ledger root of* $\mathfrak{C}$ *is a valid ledger root.*

We have the following result ensuring that at height $\mathfrak{h}$ the ledger has precisely the number of currency units (asset values) intended.[69]

Theorem 9.3. *Suppose* $\mathfrak{g}$ *is a valid ledger root and* $\mathfrak{C}$ *is a block chain of height* $\mathfrak{h}$*. Suppose* $\mathfrak{C}$ *is valid from genesis ledger root* $\mathfrak{g}$ *with reward function* R*. Suppose* $\overline{a}$ *is a list of the total assets of* $\mathfrak{g}$ *(as a one node compact tree of depth* 160*). Let* t_0 *be the sum of the value of the assets in* $\overline{a}$ *(i.e., the initial distribution). Let* $\mathfrak{n}$ *be the last ledger root of* $\mathfrak{C}$*. Suppose* $\overline{b}$ *is a list of the total assets of* $\mathfrak{n}$ *(as a one node compact tree of depth* 160*). Let* t *be the sum of the value of the assets in* $\overline{b}$*. We have*

$$t = t_0 + \sum_{i=0}^{\mathfrak{h}} R(i)$$

[65] See `BlockChain` in `Blocks.v`.
[66] See `BlockHeaderChain` in `Blocks.v`.
[67] See `valid_BlockChain` in `Blocks.v`.
[68] See `lastledgerroot_valid` in `Blocks.v`.
[69] See `lastledgerroot_sum_correct` in `Blocks.v`.

10 Conclusion

We have outlined representations of ledgers for which minimal storage is required. This could lead to the design and creation of cryptocurrencies without the problem of "block chain bloat." In particular no historical information is needed beyond having enough headers to be secure that one is not on an attacker's chain. Transactions and blocks can be checked by nodes without keeping details of the current state of the ledger beyond a single hash value representing the current ledger root.

References

[1] andruiman. PoS forging algorithms: formal approach and multibranch forging, 2014.
github.com/ConsensusResearch/articles-papers/blob/master/multibranch/multibranch.pdf

[2] Vitalik Buterin. Proof of Stake: How I Learned to Love Weak Subjectivity, November 2014.
blog.ethereum.org/2014/11/25/proof-stake-learned-love-weak-subjectivity/

[3] Alexander Chepurnoy. Inside a Proof-of-Stake Cryptocurrency Part 3: A Local Ledger, 2014.
chepurnoy.org/blog/2014/11/inside-a-proof-of-stake-cryptocurrency-part-3/

[4] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011.

[5] The Nxt community, Nxt Whitepaper, Revision 4, July 2014.

[6] Sunny King and Scott Nadal. PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake, 2012.

[7] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2012. Version 8.4.

[8] Ralph C. Merkle. Protocols for public key cryptosystems. In *Proc. 1980 Symposium on Security and Privacy*, pages 122–133. IEEE Computer Society, April 1980.

[9] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[10] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2014. www.cis.upenn.edu/ bcpierce/sf.

[11] Bill White. Formal Idealizations of Cryptographic Hashing, 2015.