# Peer-to-Peer Affine Commitment using Bitcoin

Karl Crary Michael J. Sullivan

Carnegie Mellon University

**Abstract**

The power of linear and affine logic lies in their ability to model state change. However, in a trustless, peer-to-peer setting, it is difficult to force principals to commit to state changes. We show how to solve the peer-to-peer affine commitment problem using a generalization of Bitcoin in which transactions deal in types rather than numbers. This has applications to proof-carrying authorization and mechanically executable contracts. Importantly, our system can be—and is—implemented on top of the existing Bitcoin network, so there is no need to recruit computing power to a new protocol.

## 1 Introduction

The power of linear and affine logic [9] lies in their ability to model state change. Linear and affine assumptions are treated as scarce resources: a linear assumption must be consumed exactly once, while an affine assumption must be consumed at most once. Thus, an action that consumes a linear or affine resource represents a change in the state of the world, from a state containing the resource to a new state containing whatever was produced in its stead.

For example, the rule:

$$\texttt{bread} \otimes \texttt{ham} \multimap \texttt{ham sandwich}$$

models the state change that takes place when bread and ham are combined to produce a ham sandwich. The bread and ham are gone, replaced by the sandwich. Less whimsically, the rule:

$$\forall i.\, \texttt{counter}(i) \multimap \texttt{counter}(i+1)$$

models the state change that takes place when a counter is incremented. After the state change, the counter no longer contains $i$, it contains only $i+1$.

One important application of linear/affine logic, of particular interest to us here, is in proof-carrying authorization [1, 3, 7], wherein linearity/affinity allows the creation of single-use authorization credentials.

**Proof-carrying authorization** Proof-carrying authorization is a logic based on an affirmation modality $\langle K \rangle P$, which is read "the principal $K$ says $P$". Of interest are propositions like $\langle \texttt{Alice} \rangle\, \texttt{may-read}(\texttt{Bob}, \texttt{file})$. If Bob is able to prove the proposition, and Alice owns `file`, then Bob is authorized to read `file`.

Alice may affirm any statement of the form $\langle \texttt{Alice} \rangle P$ that she chooses. She may affirm $\langle \texttt{Alice} \rangle\, \texttt{may-read}(K_i, \texttt{file})$ for various $K_i$, thereby achieving the same functionality as a conventional access-control list. Or, she may affirm a more general policy, such as:

$$\langle \texttt{Alice} \rangle \forall K.\, \langle \texttt{Registrar} \rangle\, \texttt{in-Alice's-class}(K) \supset \texttt{may-read}(K, \texttt{file})$$

to say that anyone in Alice's class (according to the registrar) may read the file. Authorization then becomes a matter of logical inference.

With a *persistent*[1] statement $\langle \texttt{Alice} \rangle \texttt{may-write}(\texttt{Bob}, \texttt{homework})$, Bob can turn in his homework as many times as he likes. But perhaps Alice wants to grant Bob that privilege only once. She can do so by affirming the proposition as a *linear* or *affine* resource. That way the authorization resource is consumed when Bob uses it, thereby preventing him from using it again.

In a closed setting (that is, in a system made up of one or more trusted sites), proof-carrying authorization is just a matter of proof checking. The system keeps track of all known propositions. It will not add assumptions unless they are either proven or affirmed by the appropriate party, and it deletes linear/affine resources when they are consumed.

In an open setting the problem is more complicated, at least if we wish to be able to move authorizations between sites that do not trust each other. The proofs themselves are trust-free; they can be moved and checked anywhere without trusting anyone. Persistent affirmations can also be made mobile by backing them with digital signatures. Linear/affine resources, however, pose a problem: if one is passed through an untrusted site, there is no way to know that it was not copied.

**Commitment** The essential requirement is a mechanism by which principals can globally and irreversibly[2] commit to a state change. The problem with copying linear/affine credentials is it allows a principal to perform a state change while retaining access to the old state, thereby effectively allowing him to undo the state change by using the copy.

One way to solve the commitment problem is using a trusted third-party. A trusted third-party can hold all the linear/affine credentials and thus be the keeper of all the state. Whenever Bob wishes to exercise a linear/affine credential, the site on which he does it contacts the keeper,

[1] We use the word "persistent" for normal (non-linear, non-affine) assumptions that may be used many times or not at all.

[2] To be sure, linear and affine formalisms often contain rules that can invert the state transitions induced by other rules. For example, an uneaten ham sandwich can be dismantled into bread and ham. But that would be an inverse state change, not a *per se* reversion to the previous state. An analogous distinction is between deleting a keystroke in a text editor using the backspace key, versus using the undo key.

which verifies to Bob's site that Bob possesses the credential and then deletes or transforms it as appropriate.

A trusted third-party removes the need for all sites to trust each other, but it still requires at least one agent to gain the trust of everyone. It also creates a single point of failure: the entire system grinds to a halt if the keeper goes down, and the entire system is compromised if the keeper is compromised.

Thus, we prefer a peer-to-peer solution to commitment, one that is trust-free and fully distributed. Linear/affine credentials should be held collectively by the network, no site should need to trust any other site, and no single failure (nor even a moderate number of failures) should compromise the system or bring it to a halt. For technical and philosophical reasons (Section 4), we formulate our solution for affine logic, not linear logic.

**Bitcoin** A similar need for peer-to-peer commitment arises in crypto-currencies such as Bitcoin [16] and its cousins. Implementing a crypto-currency is almost easy: the issuer can mint coins by digitally signing certificates, and the current owner of a coin can spend it by signing it over to someone else. The problem with the simple scheme is it does not prevent the owner of a coin from spending it multiple times. For a crypto-currency to work, it needs a way for participants to commit to spending a coin.

Bitcoin implements commitment by maintaining a global ledger containing every Bitcoin transaction. When someone wishes to spend a coin, he enters the transaction into the ledger, and the payee waits to dispense goods or services until he sees the payment appear in the ledger.

For our purposes, the exact mechanism by which Bitcoin maintains its ledger—called the *blockchain*—is not crucial, but it is helpful to understand the full picture:

1. The blockchain consists of a set of blocks, each one of which aggregates a number of transactions. Each block contains a cryptographic hash of the previous block [10], thereby turning the set into a tree.

2. In order for the blockchain to provide a commitment mechanism, we need it to be list, not a tree. Otherwise, a state change could be reversed by hopping to an alternate branch of the tree. Therein lies the main contribution of Bitcoin.

3. Parties are incentivized to create new blocks (called "mining", by analogy to miners of precious metals) by the privilege to generate new bitcoins and collect transaction fees. However, the Bitcoin history is defined to be longest branch in the tree, so there is no incentive to work on an inferior branch.

4. In order to create a new block, its creator must solve a problem[3] that is expensive to solve, but easy to verify. This makes the time to create a block much greater than the time needed to disseminate a block and check its correctness. Thus, when a new block is announced, a miner's incentive is always to restart work on a successor to the new block, rather than wasting effort on what has become an inferior branch.

5. In order to reverse a transaction, an attacker would need to create a new block without it, and then outpace the rest of the network to create enough subsequent blocks to make his branch the longest. As new blocks follow a transaction's block, his likelihood of success drops exponentially—assuming that most of the network's computational power is controlled by honest participants.

6. Thus, once a transaction has several subsequent blocks (usually taken as five), it may be considered irreversible. We will say such a transaction is *confirmed.* This takes roughly an hour.[4]

From a type-theoretic perspective, what Bitcoin provides is a commitment mechanism for a single homogeneous resource (currency). In this paper we show how to generalize Bitcoin from a currency into a general-purpose affine-commitment mechanism. We also present an implementation of our system, called Typecoin, that runs on top of the existing Bitcoin architecture. Thus, Typecoin already has all the commitment power of Bitcoin, without any need to recruit participants to a new protocol.

## 2 Transactions

A Bitcoin transaction consists of a set of inputs and a set of outputs. Each input and output has a bitcoin amount, and each input gives the identifier of a specific transaction-output that it spends:

$$\left.\begin{array}{r} I_1 : a_1 \\ \vdots \\ I_m : a_m \end{array}\right\} \xrightarrow{\text{in}} \quad \xrightarrow{\text{out}} \left\{\begin{array}{l} b_1 \\ \vdots \\ b_n \end{array}\right.$$

Additionally (not shown), each output lists a public key needed to spend that output, and each input provides a digital signature. In order for a transaction to be valid (a prerequisite for inclusion in the blockchain):

1. The sum of the outputs must equal the sum of the inputs (minus a transaction fee that we will neglect).

2. Each input amount must be equal to the output amount it identifies.

3. All the inputs must identify distinct unspent outputs.

4. All of the inputs' digital signatures must be valid signatures of the full transaction for the public key of the output being spent.

The first condition is the one we wish to generalize. Instead of making each "amount" a number, we make it a type. We also include a proof term $M$ in the transaction.

$$\left.\begin{array}{r} I_1 : A_1 \\ \vdots \\ I_m : A_m \end{array}\right\} \xrightarrow{\text{in}} M \xrightarrow{\text{out}} \left\{\begin{array}{l} B_1 \\ \vdots \\ B_n \end{array}\right.$$

Then, instead of requiring the inputs and outputs to agree arithmetically:

$$a_1 + \cdots + a_m = b_1 + \cdots + b_n$$

[3] In particular, the block's cryptographic hash, viewed as an integer, must be less than a given target. The miner can change the hash by altering a nonce, but no strategy for hitting the target better than brute force is known.

[4] Bitcoin dynamically adjusts the mining difficulty so that new blocks are always generated approximately every ten minutes, even as the computational power of the network changes.

we require them to agree type-theoretically:[5]

$$\vdash M : (A_1 \otimes \cdots \otimes A_m) \multimap (B_1 \otimes \cdots \otimes B_n)$$

Thus, if Alice has the private key to unlock a transaction output having type $A$, she can either pass it along to Bob, or (if she can prove $A \multimap B$) convert it to a $B$, or both. But once she does any of these, and the transaction is confirmed, her action is irreversible. The transaction output she used is now spent, and cannot be spent again.

**Usage** Observe that we can specialize Typecoin back to a crypto-currency by dealing with a single homogeneous resource type. For example, the arithmetic equation $2 + 2 = 1+3$ becomes the affine implication $((\texttt{coin} \otimes \texttt{coin}) \otimes (\texttt{coin} \otimes \texttt{coin})) \multimap (\texttt{coin} \otimes (\texttt{coin} \otimes \texttt{coin} \otimes \texttt{coin}))$. For large amounts this quickly becomes unwieldy, so a more practical encoding uses an indexed type $\texttt{coin}(n)$, with rules $\texttt{coin}(m + n) \multimap (\texttt{coin}(m) \otimes \texttt{coin}(n))$ and vice versa.

However, we can also use Typecoin to express more complex systems. Returning to our proof-carrying authorization example from earlier, suppose Alice wants to give Bob a single-use credential to turn in his homework. Alice does not sign a persistent statement $\langle\texttt{Alice}\rangle\texttt{may-write}(\texttt{Bob}, \texttt{homework})$, because that would allow Bob to hand in his homework as many times as he chooses. Instead, Alice creates a transaction that outputs that same proposition as an affine resource.

Alice may generate $\langle\texttt{Alice}\rangle P$ for any $P$, from nothing, by signing the transaction that it will be created in. (Signing the transaction prevents an attacker from replaying the affine resource as part of a different transaction. An attacker cannot replay the entire transaction either, because every transaction has at least one input. In a replayed transaction that input would already be spent, so the replay would be invalid.)

Alice directs the output of her transaction to Bob. (More precisely, she locks the output using Bob's public key.) Bob could then pass it on to someone else, but he has no reason to do so since $\texttt{may-write}(\texttt{Bob}, x)$ is worthless to anyone but Bob.

When Bob is ready to turn in his homework, he must show to the file system that he is expending his writing credential for this particular write. One simple protocol is as follows: Bob submits the write to the file system, which replies with a nonce $n$. Bob then submits a Typecoin transaction that alters his credential to include the nonce:

$$\begin{aligned}&\texttt{may-write}(\texttt{Bob}, \texttt{homework})\\ &\multimap \texttt{may-write-this}(\texttt{Bob}, \texttt{homework}, n)\end{aligned}$$

Once the filesystem sees the nonce in a confirmed transaction, it recognizes that Bob has committed to the write, so it performs it.

In other circumstances, it might make sense for a resource to be transferable. Consider:

$$\langle\texttt{ACM}\rangle \forall K.\, \texttt{may-read}(K, \texttt{TOPLAS})$$

This credential can be used by anyone, by filling in the principal $K$. The holder of such a credential could exercise it by instantiating $K$ with himself, or he could transfer it to someone else, possibly in exchange for other affine resources.

The connectives of affine logic create other interesting possibilities. For example, external choice allows the resource's holder to choose between multiple options, as in:

$$\langle\texttt{ACM}\rangle \forall K.\, (\texttt{may-read}(K, \texttt{TOPLAS}) \,\&\, \texttt{may-read}(K, \texttt{TOCL}))$$

## 3 Implementation

We implement Typecoin by overlaying Typecoin transactions atop Bitcoin transactions. Thus, each input and output has a bitcoin amount and a type:

$$\left.\begin{array}{r} I_1 : a_1/A_1 \\ \vdots \\ I_m : a_m/A_m \end{array}\right\} \xrightarrow{\text{in}} M \xrightarrow{\text{out}} \left\{\begin{array}{l} b_1/B_1 \\ \vdots \\ b_n/B_n \end{array}\right.$$

The Bitcoin protocol checks that each input is unspent, and the digital signatures are valid (*i.e.,* the third and fourth conditions from Section 2). The Bitcoin protocol also checks that the bitcoin amounts agree (*i.e.,* the first and second conditions). Naturally, it does not check that the types agree, as it knows nothing about them.

Thus, every transaction-output ("txout" in Bitcoin nomenclature) carries both a bitcoin amount and a type. Txouts that do not arise from valid Typecoin transactions are taken to have the trivial type 1. In a typical Typecoin transaction, all the bitcoin amounts will be very small.

The full Typecoin transaction (including inputs, outputs, a proof term, and other material) is cryptographically hashed and embedded into its corresponding Bitcoin transaction. When combined with Bitcoin's requirement that no txout be spent more than once, this provides a commitment mechanism: An affine resource represented by a txout can be spent only once, and the manner in which it was spent is irreversibly fixed by publishing its hash.

As noted, Bitcoin cannot be induced to type-check Typecoin transactions. Indeed, even if it could, it would be wrong for the network at large to have to assume that cost, and it would open the door to a denial-of-service attack.

Instead, type-checking is performed by the interested parties, outside the Bitcoin mechanism. When Bob tries to turn in his homework, he identifies to the filesystem a txout (say $I$) that he claims has the type $\texttt{may-write-this}(\texttt{Bob}, \texttt{homework}, n)$. To substantiate his claim, he provides the Typecoin transaction $T_I$ that outputs $I$, as well as $\mathfrak{T}$, the set of all Typecoin transactions upstream of $T_I$.

The type-checker then checks that $I$'s type is as claimed, and checks, for each $T \in \mathfrak{T}$, that:

1. The hash of $T$ agrees with the hash embedded in its corresponding Bitcoin transaction.
2. $T$ type-checks.
3. The type of each input of $T$ agrees with the type of the output it spends.

Thus, the affine invariant (resources are spent at most once) is enforced in two different ways: within transactions by the Typecoin type-checker, and between transactions by the Bitcoin invariant that no txout is spent multiple times.

Our reference implementation of Typecoin is written in Standard ML, and includes a new Standard ML implementation of Bitcoin.

[5] In linear/affine logic, $A \otimes B$ is "simultaneous conjunction," representing the combination of $A$ and $B$; and $A \multimap B$ is linear/affine implication, representing a function that consumes an $A$ to produce a $B$.

### 3.1 Typecoin and Bitcoins

Since non-Typecoin txouts are taken to have type 1, we can inject extra bitcoins into a transaction by adding an extra input of type 1. Such trivial inputs are type-theoretically irrelevant, but they can still be useful. For example, a transaction might operate at both levels at once, in order to exchange an affine resource for bitcoins. Trivial inputs can also be used to bring a transaction into balance, or to pay the Bitcoin transaction fee.

Conversely, nothing prevents the owner of a txout from spending it in a non-Typecoin transaction. The owner is then, in essence, cracking a resource open to recover the bitcoins inside. This will be a common cleanup operation. For example, Bob has no more use for his nonce-infused credential once he has turned in his homework, so he has every reason to turn it back into bitcoins.

### 3.2 Batch Mode

A significant weakness of Typecoin is its latency. A Bitcoin transaction takes about an hour to be confirmed, which makes it impractical for many uses. Certainly we could not base a filesystem on a mechanism that requires an hour to deliver an access permission.

A similar issue stems from Bitcoin's transaction fees. A typical transaction fee is 0.0005 bitcoin, which, as of mid-April 2015, is about 11¢ US. This is a small amount in absolute terms, but in any kind of automated application it would add up quickly.

To resolve these problems, Typecoin can be operated in batch mode. In batch mode, a trusted third-party maintains a credential server that holds Typecoin resources on behalf of other principals. When principals wish to conduct a batch-mode transaction, they notify the server, which records the transaction but does not submit it to the network.

A principal may also withdraw a resource from the server—sending it to his own public key—and only then does the server submit a transaction to the network. The server batches together all the transactions upstream of the resource in question, routing that resource to its owner's key and the rest back to its own key. (This will likely be a large Typecoin transaction, but the Bitcoin network sees only its hash.) Conversely, a principal can deposit a resource at the server by sending it to the server's public key.

When an interested party (such as the fileserver) wants to check the validity of a claimed resource, she queries the batch-mode server, which answers based on its own records, if it holds the resource, or on the blockchain if it does not.

Note that batch mode does not compromise the trustlessness of the network. No one ever needs to use a batch-mode server, batch mode only exploits trust relationships that happen to exist already. For instance, universities or companies might choose to operate batch-mode servers for their respective institutions. Indeed, the Typecoin client itself can be viewed as a very small batch-mode server, trusted by only one person.

### 3.3 Metadata in Bitcoin

One messy detail of the implementation pertains to the Typecoin transaction hash that must be embedded into the Bitcoin transaction. Unfortunately, Bitcoin transactions do not have a metadata field, so we have to be creative.

One easy way to introduce metadata into a Bitcoin transaction would be to add a bogus output whose "public key" is not actually a public key at all, but the desired metadata. A bitcoin amount would have to be sent to the non-key, and that amount would be unrecoverable, but the amount could be kept very small.

However, that strategy cannot be considered, because it would have a severe consequence on Bitcoin itself. Any Bitcoin node that verifies transactions' validity must be able to tell whether a particular txout has been spent already, and this requires maintaining a table of all unspent txouts. Unrecoverable txouts mean permanent deadweight in the table.

At the time of this writing, the unspent-txout table is about one-quarter gigabyte (as the most popular Bitcoin implementation represents it). This already poses a long-term challenge for Bitcoin's scalability and adding an uncollectable entry for each Typecoin transaction would only exacerbate the problem.

Another potential strategy is to use Bitcoin's scripting language [4], which allows txouts to define unlocking rules more interesting than the typical signed-by-a-given-key rule. The scripting language is a stack machine reminiscent of Forth [18], and provides ample opportunity to embed extraneous data.

Alas, this does not work either. For various reasons, particularly concerns that novel scripts could be used for denial-of-service attacks, the Bitcoin network makes most scripts unavailable for normal use. A very small number of script schemas are deemed to be *standard,* and most Bitcoin nodes will not forward transactions that use non-standard scripts. Thus, while non-standard scripts are legal when they appear in blocks, participants cannot get non-standard scripts into a block unless they control a miner.

To embed metadata, our implementation uses a standard script schema called "$m$-of-$n$" [5]. In an $m$-of-$n$ script, the output specifies $n$ public keys, and the input must provide signatures for $m$ of them. This is intended, in its 2-of-3 form, for two-party escrow contracts, wherein the signatures are those of the two parties and an escrow agent. When no dispute arises, the parties unlock the payment together, but when a dispute does arise, the escrow agent sides with one party or the other.

We use $m$-of-$n$ scripts in their 1-of-2 form. One of the public keys is the actual public key, the other "public key" is the desired metadata. Since the output can be unlocked by satisfying just one of the two keys (the real one), the output can be spent, and its entry in the unspent-txout table can be garbage-collected.

## 4 The Typecoin Logic

The syntax of the Typecoin logic is given in Figure 1. The two main classes of interest are types, inhabited by index terms, and propositions, inhabited by proof terms.

Propositions are the main topic of discourse. We support the connectives of dual intuitionistic linear logic [2] except $\top$ (which is meaningless in affine logic), universal and existential quantification, an affirmation modality [8], and one other form (receipts) relevant to building transactions. As we have seen already, the logic is dependently typed. Atomic propositions have the form $c\, m_1 \cdots m_i$, for index terms $m_1, \ldots, m_i$.

| | | | | |
|---|---|---|---|---|
| *primitives* | number | $n$ | ::= | $0 \mid 1 \mid 2 \mid \cdots$ |
| | principal | $K$ | ::= | $\cdots$ |
| | transaction id | *txid* | ::= | $\cdots$ |
| | digital sig | *sig* | ::= | $\cdots$ |
| *consts* | local constant | $\ell$ | ::= | $\cdots$ |
| | reference | $r$ | ::= | $\mathsf{this} \mid \mathit{txid}$ |
| | global constant | $c$ | ::= | $r.\ell$ |
| *LF* | kind | $k$ | ::= | $\mathsf{type} \mid \mathsf{prop} \mid \Pi u{:}\tau.k$ |
| | type family | $\tau$ | ::= | $c \mid \tau\, m \mid \Pi u{:}\tau.\tau \mid \mathsf{principal} \mid \mathsf{nat}$ |
| | index term | $m$ | ::= | $u \mid c \mid \lambda u{:}\tau.m \mid m\, m \mid K \mid n$ |
| *propositions* | proposition | $A$ | ::= | $\tau$ |
| | | | $\mid$ | $A \multimap A \mid A \,\&\, A \mid A \otimes A$ |
| | | | $\mid$ | $A \oplus A \mid 0 \mid 1 \mid !A$ |
| | | | $\mid$ | $\forall u{:}\tau.A \mid \exists u{:}\tau.A$ |
| | | | $\mid$ | $\langle m \rangle A \mid \mathsf{receipt}(A/m \twoheadrightarrow m)$ |
| *proof terms* | proof term | $M$ | ::= | $x \mid c$ |
| | | | $\mid$ | $\ldots \mathit{standard\ affine\ logic} \ldots$ |
| | | | $\mid$ | $\mathsf{sayreturn}_m(M)$ |
| | | | $\mid$ | $\mathsf{saybind}\ x \leftarrow M\ \mathsf{in}\ M$ |
| | | | $\mid$ | $\mathsf{assert}(K, A, \mathit{sig})$ |
| | | | $\mid$ | $\mathsf{assert!}(K, A, \mathit{sig})$ |
| *bases* | sort | $s$ | ::= | $k \mid \tau \mid A$ |
| | basis | $\Sigma$ | ::= | $\epsilon \mid \Sigma, c : s$ |
| *transactions* | input | $\iota$ | ::= | $\mathit{txid}.n \rightarrowtail A/n$ |
| | inputs | $\vec{\iota}$ | ::= | $\iota, \ldots, \iota$ |
| | output | $\omega$ | ::= | $A/n \twoheadrightarrow K$ |
| | outputs | $\vec{\omega}$ | ::= | $\omega, \ldots, \omega$ |
| | transaction | $T$ | ::= | $(\Sigma, A, \vec{\iota}, \vec{\omega}, M)$ |

Figure 1: Syntax

For maximum generality, we follow Simmons [20] and use LF [11] for our index terms. Using LF, one can define whatever language of discourse one requires. Because of the important role played in Typecoin by affirmation and digital signatures, it is convenient to isolate two particular LF types (principal and nat) for special treatment elsewhere. The type principal is inhabited by principal literals $K$, which we take to be cryptographic hashes of public keys,[6] and the type nat is inhabited by natural numbers. Note that since we identify principals with public keys, Typecoin is open-ended with respect to the addition of new principals.

Since the form of atomic propositions is identical to the form of atomic types, it is convenient for us to view atomic propositions as type families whose kind is prop rather than type. Since we omit family-level lambda abstractions (following Harper and Pfenning [12]), it is easy to show that the addition of a new kind does not affect the existing LF metatheory.

**Proof Terms** Most of the proof terms are the standard proof terms of affine logic. In addition, there are four forms for affirmation. Affirmation forms a monad, with unit sayreturn and bind saybind. The unit expresses that every principal affirms everything provable. The bind allows us, given a proof of $\langle K \rangle A$, to assume $A$, but only to prove a proposition of the form $\langle K \rangle B$.

In addition to the monad forms, there are two primitive affirmations that allow $K$ to affirm any statement he desires. Both $\mathsf{assert}(K, A, \mathit{sig})$ and $\mathsf{assert!}(K, A, \mathit{sig})$ prove the proposition $\langle K \rangle A$; they differ in what the digital signature *sig* signs. In the former, *sig* signs essentially the entire transaction in which it appears;[7] in the latter, *sig* signs only the proposition $A$. The former is intended for affine affirmations (*e.g.,* $\langle \texttt{Alice} \rangle \texttt{may-write}(\texttt{Bob}, \texttt{homework})$), so it cannot be lifted out of its transaction. The latter is intended for persistent affirmations, so it can.

**Receipts** Suppose ACM wants to grant access to read TOPLAS in exchange for a coupon. Then ACM can issue the offer:

$$!\langle \texttt{ACM} \rangle (\texttt{coupon} \multimap \forall K.\, \texttt{may-read}(K, \texttt{TOPLAS}))$$

When the reader exercises this offer, the coupon is destroyed and the reader gains access to TOPLAS. But suppose that the coupon is valuable for some reason, so ACM wishes to recover it rather than destroy it.

Receipts record the fact that a payment is being made to some principal, turning that fact into a resource that can be demanded as part of an offer. By demanding a receipt, a principal requires that the corresponding payment is made. For each of the transaction's outputs $\omega$, the transaction gets $\mathsf{receipt}(\omega)$ as an additional input.

For example, if ACM wishes to recover the coupon, it would say:

$$!\langle \texttt{ACM} \rangle\, (\mathsf{receipt}(\texttt{coupon} \twoheadrightarrow \texttt{ACM}) \multimap \forall K.\, \texttt{may-read}(K, \texttt{TOPLAS}))$$

In order to read TOPLAS, the customer must obtain a $\mathsf{receipt}(\texttt{coupon} \twoheadrightarrow \texttt{ACM})$, and to do so the customer must send the coupon to ACM.

The general form of this is $\mathsf{receipt}(A \twoheadrightarrow K)$, indicating that a resource of type $A$ has been sent to principal $K$. Another form, $\mathsf{receipt}(n \twoheadrightarrow K)$ indicates that $n$ bitcoins have been sent to $K$. These forms can be combined into $\mathsf{receipt}(A/n \twoheadrightarrow K)$, to indicate a resource of type $A$ and also $n$ bitcoins have been sent to $K$.

**Bases** A basis is a set of constant declarations. Each constant represents a new type family, index term, or proof term. A transaction uses its local basis to define concepts or rules relevant to its transaction. (In the logical frameworks literature, a basis is usually called a signature, but we wish to avoid the unfortunate terminological collision with digital signatures.) The *global basis* is the local basis appended to the bases of all previous transactions.

Every constant is relative to a reference to the transaction in which the constant originated. Since a transaction's identifier is not known in advance, constants local to the transaction are identified using a special local reference, this. Once the transaction enters the blockchain, all its declarations are added to the global basis, with this replaced by the transaction's identifier.

Naturally, a transaction's local basis may only declare local constants (that is, constants of the form $\mathsf{this}.\ell$). Furthermore, each constant's sort (*i.e.,* kind, type, or proposition)

[6] We use hashes, rather than raw keys, because this is standard practice in Bitcoin.

[7] The proof term need not be signed, and indeed cannot be, since it contains the signatures.

must be restricted so that no transaction can make declarations that change the meanings of non-local constants.[8] This check, called the *freshness check,* requires that any *restricted form* must appear on the left-hand-side of a lolli or universal quantifier. Thus, restricted forms can be consumed but not produced. Restricted forms include non-local constants, the proposition 0, affirmations, and receipts.

**Transactions** A transaction consists of five components: its local basis, a proposition called its affine grant, inputs, outputs, and a proof term. The local basis makes persistent definitions that any subsequent transaction may reference. The affine grant creates affine resources for use within the transaction. Like the local basis, it must pass the freshness check.

Each input $\mathit{txid}.n \rightarrowtail A/n$ specifies that resources typed $A$ and $n$ bitcoins are taken in from the $n$th output of *txid*. (The digital signature used to unlock the input is not represented in the formalism.) Each output $A/n$ specifies that resources typed $A$ and $n$ bitcoins are sent to the principal $K$.

Finally, the proof term $M$ must prove that the transaction balances. It shows that the outputs can be produced using the affine grant, the inputs, and receipts for the outputs. More precisely, if $A$ tensors the inputs together, $B$ tensors the outputs together, $R$ tensors the receipts together, $C$ is the affine grant, and $\Sigma$ is the local basis, then:

$$\Sigma_{\text{global}}, \Sigma \vdash M : (C \otimes A \otimes R) \multimap B$$

**Affinity** We designed Typecoin to implement affine commitment, rather than linear commitment, for several reasons. We could have based the system on linear logic, but it still would have admitted several ways in which resources could be destroyed. The easiest is to declare constants with type $A \multimap 1$ in the local basis. This is legal, since 1 is not a restricted form. (Even if it were, there would be several other slightly-less-easy ways to destroy a resource.)

Moreover, even if the logic were formulated in some different fashion that made it impossible to destroy a resource *per se,* it would still be possible to make a resource permanently unusable by storing it in a txout and discarding its private key.

Since it seems awkward to be affine only through a clumsy idiom, we have elected to embrace affinity and have formulated our system to admit weakening.

## 5 Expiration and Revocation

An important financial contract is the option, which allows the holder to purchase a commodity at a given price, or not, until the option expires. As we have seen, much of this contract is already expressible:

$$\mathsf{receipt}(\texttt{payment} \twoheadrightarrow \texttt{Alice}) \multimap \texttt{commodity}$$

However, this fails to express that the offer expires at some given time. To account for expiration, we add a new form of proposition, as shown in Figure 2. The conditional $\mathsf{if}(\varphi, A)$ can be converted to $A$, provided the condition $\varphi$ holds.

[8] Even if the other transaction was authored by the same principal, others might have come to rely on its definitions. For example, the other transaction might define a contract.

| | | | |
|---|---|---|---|
| condition | $\varphi$ | ::= | $\mathsf{before}(m) \mid \mathsf{spent}(\mathit{txid}.n)$ |
| | | $\mid$ | $\mathsf{true} \mid \varphi \wedge \varphi \mid \neg\varphi$ |
| proposition | $A$ | ::= | $\cdots \mid \mathsf{if}(\varphi, A)$ |
| proof term | $M$ | ::= | $\cdots$ |
| | | $\mid$ | $\mathsf{ifreturn}_\varphi(M)$ |
| | | $\mid$ | $\mathsf{ifbind}\ x \leftarrow M\ \mathsf{in}\ M$ |
| | | $\mid$ | $\mathsf{ifweaken}_\varphi(M)$ |
| | | $\mid$ | $\mathsf{if/say}(M)$ |

Figure 2: Conditionals

By choosing the condition $\varphi$ to be $\mathsf{before}(t)$, we can express an expiring option:

$$\mathsf{receipt}(\texttt{payment} \twoheadrightarrow \texttt{Alice}) \multimap \mathsf{if}(\mathsf{before}(t), \texttt{commodity})$$

Until time $t$, the holder can pay Alice to obtain the conditional, then discharge the conditional to obtain the commodity. After time $t$, the conditional $\mathsf{if}(\mathsf{before}(t), \texttt{commodity})$ becomes worthless, so the option becomes worthless.

Note that it is important that the condition appear beneath the lolli, not above it. In the incorrect alternative:

$$\mathsf{if}(\mathsf{before}(t), \mathsf{receipt}(\texttt{payment} \twoheadrightarrow \texttt{Alice}) \multimap \texttt{commodity})$$

the holder can discharge the condition before $t$, and then hold a non-expiring option indefinitely.

**Conditions** Conditions are built from $\mathsf{true}$, conjunction, negation, and two primitive conditions. As we have seen, the primitive condition $\mathsf{before}(t)$ expresses expiration. A second primitive condition, $\mathsf{spent}(\mathit{txid}.n)$, expresses that the $n$th output of transaction *txid* has been spent.

The latter is useful in negated form to express revocation. If Alice controls the txout $I$, she can make a revocable offer by conditioning it on $I$ being unspent:

$$\mathsf{receipt}(\texttt{payment} \twoheadrightarrow \texttt{Alice}) \multimap \mathsf{if}(\neg\mathsf{spent}(I), \texttt{commodity})$$

Alice can revoke the offer at any time (with about fifteen minutes average latency), simply by spending $I$.

Other conditions could also be added. The essential property of all conditions $\varphi$ is that there be unambiguous evidence of the truth or falsity of $\varphi$ for any particular transaction in the blockchain. Each block includes a timestamp that can be used to determine the transaction's time. To show that a txout is spent, one can point to an earlier transaction that spent it. To show a txout is unspent, one can point out a later transaction that spent it, or observe that it is still unspent. (Recall that Bitcoin maintains a table of all unspent txouts.)

**The conditional monad** As we have seen, the propositions $A \multimap \mathsf{if}(\varphi, B)$ and $\mathsf{if}(\varphi, A \multimap B)$ must not be equivalent. This precludes interpreting $\mathsf{if}(\varphi, A)$ as something like $\varphi \multimap A$. In fact, conditionals form a monad. We may view $\mathsf{if}(\varphi, A)$ as an effectful operation: it produces $A$ only after checking the current state of the world to ensure that $\varphi$ holds.

Thus we manipulate conditionals using the unit $\mathsf{ifreturn}_\varphi$ (which allows any $A$ to be weakened to $\mathsf{if}(\varphi, A)$) and bind $\mathsf{ifbind}$. We may also weaken $\mathsf{if}(\varphi', A)$ to $\mathsf{if}(\varphi, A)$ using $\mathsf{ifweaken}_\varphi$, provided $\varphi$ implies $\varphi'$.

Since we have two monads, we include a commutation operation $\mathsf{if/say}$, which takes $\langle K \rangle \mathsf{if}(\varphi, A)$ to $\mathsf{if}(\varphi, \langle K \rangle A)$. The opposite direction (which might be called $\mathsf{say/if}$) is semantically dubious and we do not include it.

**Discharge** The trickiest aspect of conditionals is the means by which they are discharged. At first glance, it seems appealing to add a primitive $\mathsf{discharge} : \mathsf{if}(\varphi, A) \multimap A$ that may be used only in transactions in which $\varphi$ holds. But this would be a misstep similar to interpreting $\mathsf{if}(\varphi, A)$ as $\varphi \multimap A$.

Suppose $M : A \multimap \mathsf{if}(\varphi, B)$. Then, using $\mathsf{discharge}$, we could defeat the conditional:

$$\lambda x{:}A.\mathsf{discharge}(M\,x) : A \multimap B$$

We could similarly defeat a conditional that appeared under additive conjunction (*i.e.,* &, which might be used to require that a choice be made by a particular time) or an exponential (*i.e.,* !). This illustrates that conditionals should be discharged only at the top level.

Therefore, we have no explicit discharge operation at all. Instead, discharge is done implicitly at the top-level. Transactions use the monad forms to build their outputs in conditional form. (If multiple conditions are in play, transactions use weakening to move to the conditions' conjunction.) The transaction's top-level proof term then must have the type:

$$(C \otimes A \otimes R) \multimap \mathsf{if}(\varphi, B)$$

for some $\varphi$, and the transaction is valid only if $\varphi$ holds.

Since conditions are volatile properties, batch-mode servers must write transactions discharging anything other than $\mathsf{true}$ through to the blockchain.

**Fallback transactions** Using conditionals can be risky. If a transaction's top-level condition is false when it goes into the blockchain, the transaction is invalid. However, the transaction's inputs are already spent (Bitcoin doesn't know if the transaction is valid or not) and cannot be recovered. Thus, an invalid transaction spoils its inputs, which might be valuable.

The delay in getting a transaction into the blockchain is unpredictable, so even if a transaction is valid when submitted, it might still be invalid by the time it enters the blockchain. Moreover, revocations can take place at any time without warning, so any transaction that discharges a revocable resource could turn out to be invalid.

To address this problem, Typecoin allows users to submit a list of fallback transactions. If the primary transaction turns out to be invalid, the first valid fallback transaction is used instead. A typical fallback transaction simply returns all inputs to their original owners, which keeps the inputs from being destroyed but otherwise accomplishes nothing.

All the transactions in the list must map onto the same Bitcoin transaction. This means that they must agree on the input txouts, the output principals, and the input and output Bitcoin amounts.

The fact that all transactions in the list must agree on the output Bitcoin amounts is unfortunate, because it means that a fallback transaction cannot recover payment made on an expired or revoked contract. In cases where the payment is substantial, the likelihood of expiration/revocation is non-negligible, and the payee cannot be trusted to issue a refund, the payer ought to use escrow (Section 7).

## 6 Example

We present a basis that gives defines a currency that we will call "newcoins" as a concrete demonstration of the system. before. The basis defines a `coin` proposition, indexed by natural numbers, and rules that allow the merging and splitting of coins.

$$\begin{array}{rcl}
\texttt{coin} & : & \mathsf{nat} \to \mathsf{prop} \\
\texttt{merge} & : & \forall N, M, P : \mathsf{nat}.\, (\exists x{:}\texttt{plus}\, N M P.\, 1) \multimap \\
 & & \texttt{coin}\, N \otimes \texttt{coin}\, M \multimap \texttt{coin}\, P \\
\texttt{split} & : & \forall N, M, P : \mathsf{nat}.\, (\exists x{:}\texttt{plus}\, N M P.\, 1) \multimap \\
 & & \texttt{coin}\, P \multimap \texttt{coin}\, N \otimes \texttt{coin}\, M
\end{array}$$

This example depends on the existence of LF type constant $\texttt{plus} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \to \mathsf{type}$, where $\texttt{plus}\, N\, M\, P$ is the type of proofs that $N + M = P$.

The proposition $(\exists x{:}\texttt{plus}\, N M P.\, 1)$ makes use of a somewhat unusual idiom: it has no interesting resource content, but serves to require that $\texttt{plus}\, N M P$ is inhabited.

The remaining major question about the `coin` example is how to introduce money into circulation. The principal (which we will refer to as "the bank") who publishes the basis defining coins can also define how to create new coins. One powerful tool for this is the affine grant. The bank, for example, could make the money supply fixed, by creating a `coin` 1000000000 or the like, and giving it to themselves. This will give the bank a large amount of money to distribute, but it will not be able to print new money.

Alternatively, the bank could include the resource[9] $!(\forall n{:}\mathsf{nat}.\texttt{coin}\, n)$ in the affine grant and hang on to it, thus giving itself the equivalent of a printing press. (More whimsically, the bank could simply give itself $!\texttt{coin}\, 1$.) Creating persistent resources in the affine grant is an important idiom for modeling things that are unrestricted in how much they can be used but are restricted in who they can be used by. If $\forall n{:}\mathsf{nat}.\texttt{coin}\, n$ instead appeared in the basis, then anyone could print arbitrary amounts of money!

Another approach is to empower some specific principal to trigger the action with affine affirmations:

$$\begin{array}{rcl}
\texttt{print} & : & \mathsf{nat} \to \mathsf{prop} \\
\texttt{issue} & : & \forall N{:}\texttt{number}.\, \langle \texttt{Bank} \rangle (\texttt{print}\, N) \multimap \texttt{coin}\, N
\end{array}$$

Now the bank no longer needs to thread a private printing press through all of its transactions: it simply signs an affine affirmation and then immediately uses it to trigger the `issue` rule. The bank also now has additional flexibility to sign new rules for proving $\langle \texttt{Bank} \rangle (\texttt{print}\, N)$ resources, which we explore in the next section.

### 6.1 Further Development

Using receipts and conditionals, we can extend the newcoin example in several interesting ways. Using time-based conditionals, we can encode an independent central banker who is appointed for a set term:[10]

$$\begin{array}{rcl}
\texttt{appoint} & : & \mathsf{principal} \to \mathsf{time} \to \mathsf{prop} \\
\texttt{is_banker} & : & \mathsf{principal} \to \mathsf{time} \to \mathsf{prop} \\
\texttt{confirm} & : & \forall K{:}\mathsf{principal}.\, \forall t{:}\mathsf{time}. \\
 & & \langle \texttt{President} \rangle (\texttt{appoint}\, K\, t) \multimap \texttt{is_banker}\, K\, t
\end{array}$$

[9] The proposition $!A$ is the linear/affine exponential, representing as many copies of $A$ as desired.

[10] The type $\mathsf{time}$ is actually just $\mathsf{nat}$; we give it a different name for its use in timestamps in the interest of clarity.

We represent the banker with a proposition $\texttt{is\_banker}\, K\, t$, which states that principal $K$ is the banker until time $t$. Here, a banker is appointed simply through the choice of a principal called `President`, although one can imagine a more complicated system requiring confirmation from a number of other principals.

$$\begin{array}{lcl} \texttt{print} & : & \mathsf{nat} \to \mathsf{prop} \\ \texttt{issue} & : & \forall K{:}\mathsf{principal}.\, \forall t{:}\mathsf{time}.\forall N{:}\mathsf{nat}. \\ & & \quad \texttt{is\_banker}\, K\, t \\ & & \quad \multimap \langle K \rangle (\texttt{print}\, N) \\ & & \quad \multimap \mathsf{if}(\mathsf{before}(t), \texttt{coin}\, N) \end{array}$$

Here, we use conditional propositions in order to allow the banker to print money, but only during the banker's term. An affirmation from the banker ordering the creation of $N$ newcoins can be converted to $N$ newcoins, but only if it is extracted before the banker's term expires.

The combination of receipts, affirmations, and conditions give the banker a great deal of flexibility in how to issue newcoins. For example, the banker may want to introduce newcoins into the market by purchasing bitcoins with them. One way for the bank to do this is to simply have traditional transactions with people wishing to buy newcoins: the customer sends bitcoins, the bank sends back newcoins. We can do better, however, and almost fully represent the offer in our logic.

To do this, the banker can sign a proposition that allows you to turn a receipt proving that you have sent $N_{btc}$ bitcoins to some bank-controlled address $D$ into an order to print $N_{nc}$ newcoins. So that the banker can cancel the offer and adjust prices in response to market conditions, it makes the order conditional on a txout $R$ being unspent.

$$\begin{array}{l} \langle \texttt{Banker} \rangle (\mathsf{receipt}(N_{btc} \twoheadrightarrow D) \\ \qquad \multimap \mathsf{if}(\neg \mathsf{spent}(R), \texttt{print}\, N_{nc})) \end{array}$$

By publishing a signature of this proposition, the banker enables users to incorporate the proposition into their own proof terms.

If we have a receipt bound to $r$, the affirmation bound to $p$, and a proof of $\texttt{is\_banker banker}\, T$ bound to $b$, we can produce $\texttt{coin}\, N_{nc}$ with the proof term in Figure 3.

As usual, let is a derived form built from lambda and application. The most subtle part of this example is the handling of the various monadic constructs. By using the order that the banker published, we can derive $\langle \texttt{Banker} \rangle \mathsf{if}(\neg \mathsf{spent}(R), \texttt{print}\, N_{nc})$ and must use $\mathsf{if}/\mathsf{say}$ to commute that to $\mathsf{if}(\neg \mathsf{spent}(R), \langle \texttt{Banker} \rangle (\texttt{print}\, N_{nc}))$. We use ifweaken twice in order to merge together the separate constraints.

Observe that newcoin buyers make Bitcoin payments based on a revocable offer. If the bank cannot be trusted to make refunds to principals who attempt to buy just as the offer is revoked, those buyers could lose the purchase price. We address this problem in the next section.

## 7 Open Transactions and Type-Checking Escrow

Suppose Alice wishes to award a prize to the first person to solve a puzzle. Simply announcing $!(\texttt{solution} \multimap \texttt{prize})$ won't do, because that would award the prize to everyone who solves the puzzle. The competition can easily be done on a batch-mode server, but that would require all participants to trust the server.

Half of the solution is to use an *open transaction,* which is a transaction with holes that anyone can fill in. Alice signs and issues an open transaction that takes in *solution* and `prize` and outputs them. On the input side, she specifies her txout carrying the prize, but she leaves blank the txout carrying the solution. On the output side she sends the solution to her own public key, but leaves blank the public key receiving the prize. Bob then can fill in the two blanks to receive the prize. Observe that the transaction is only valid if his txout really does have the solution and her txout is still unspent.

By themselves, open transactions don't solve the problem: Nothing keeps Alice from reneging and taking the prize herself. Worse, we can't implement open transactions on Bitcoin, because Bitcoin does not (and ought not) know anything about Typecoin type checking.

The second half of the solution is to use a type-checking escrow agent. Alice sends her prize to a public key controlled by Charlie, an escrow agent. She also issues an open transaction, as above, except that the prize comes from Charlie's txout, not hers. Charlie's policy is to sign any instance of the transaction that type checks. In order to claim the prize, Bob fills in the transaction as before, and then sends it to Charlie. Charlie finds that the instance type-checks, so he signs it and sends it back to Bob, who uses it to claim the prize.

By itself, this is no more robust than using a batch-mode server, since all participants need to trust the escrow agent. However, we can lessen the need for trust by sending the prize to several escrow agents at once, using an $m$-of-$n$ script (recall Section 3.3). For example, using a 2-of-3 script, participants can tolerate one of the three agents becoming compromised.

Another application of this technique is for redeeming Typecoin assets for bitcoins. Suppose the banker wants to back his currency by making an executable promise to buy newcoins for bitcoins at a certain rate. The banker sends his bitcoins to a pool of escrow agents, and issues an open transaction that takes in the bitcoins and a newcoin, destroys the newcoin, sends the appropriate number of bitcoins to the customer, and sends the rest back to the escrow agents.

A more complex application is for contracts that time out if not completed by a deadline. As we have seen, these are easy, using expiration, provided one can generate the asset in question. One simply sends a contract $\texttt{receipt-for-stuff} \multimap \mathsf{if}(\mathsf{before}(t), \texttt{coin}\, 1)$ and the contract spoils when time expires. This is fine for the banker, but unacceptable for anyone else (say Alice), because it doesn't allow Alice to recover the coin if the contract spoils.

Instead, Alice sends a contract $\texttt{receipt-for-stuff} \multimap \mathsf{if}(\mathsf{before}(t), \texttt{token-for-coin})$, sends the newcoin to the escrow agents, and issues an open transaction that trades the token for the newcoin. She also creates a rule that allows her to create a token once time expires. Using that token, she can cash in her own open transaction to recover the newcoin.

## 8 Related Work

The strategy of using Bitcoin to track ownership of virtual property other than bitcoins, by overlaying txouts with additional meaning invisible to the Bitcoin network, was first employed in *colored coins* [19, 15]. In colored coins, a txout is said to represent an asset (colloquially called a color)

$$\begin{array}{l}\mathsf{let}\ x : \langle \mathtt{Banker}\rangle \mathsf{if}(\neg \mathsf{spent}(R), \mathtt{print}\ N_{nc}) \leftarrow (\mathsf{saybind}\ f \leftarrow p\ \mathsf{in}\ \mathsf{sayreturn}(\mathtt{Banker}, f\, r))\ \mathsf{in} \\ \mathsf{let}\ y : \mathsf{if}(\neg \mathsf{spent}(R), \langle \mathtt{Banker}\rangle(\mathtt{print}\ N_{nc})) \leftarrow \mathsf{if/say}(x)\ \mathsf{in} \\ \mathsf{ifbind}\ z : \langle \mathtt{Banker}\rangle(\mathtt{print}\ N_{nc}) \leftarrow \mathsf{ifweaken}_{\neg \mathsf{spent}(R) \wedge \mathsf{before}(T)}(y)\ \mathsf{in} \\ \mathsf{ifweaken}_{\neg \mathsf{spent}(R) \wedge \mathsf{before}(T)}(\mathtt{issue}\ \mathtt{Banker}\ T\ N_{nc}\ b\ z)\end{array}$$

Figure 3: Proof term for purchasing newcoins

in much the same way as in Typecoin txouts are said to represent affine resources.

Assets in the colored-coin system are issued in fungible units, such as shares of a stock, or units of an alternative crypto-currency. (Non-fungible assets can be implemented by issuing just a single unit.) Each txout carries a certain number of units of the asset.

Unlike Typecoin, a colored-coin transaction does not include a proof term that dictates how the assets/colors propagate from inputs to outputs. Instead, propagation is defined by a collection of rules, based on the order and bitcoin amounts of the inputs and outputs. These rules are flexible enough to allow the splitting and merging of assets, and to allow different assets to be traded in a single transaction.

However, colored coins do not provide the general expressive power of affine authorization logic. For instance, they provide no mechanism for state transitions.

There has been substantial interest [14] in formal languages for specifying contracts and (to varying degrees) executing them. Particularly influential in the type theory community is Peyton Jones, *et al.* [17], who devised a collection of Haskell combinators for specifying contracts and a semantics for determining their value. Szabo [21] surveys issues and strategies for executing contracts on the Internet.

Bitcoin is primarily used for processing simple payments, but it also has a variety of facilities for implementing contracts [13]. Our open transactions are inspired by and generalize Bitcoin's SIGHASH rules, which erase parts of a transaction before checking its signatures, thereby allowing those parts to be altered.

Much of Bitcoin's expressive power for contracts is subsumed by our affine logic, but it also supports some idioms that we cannot express in our logic. For example, to implement a contract that can be reversed if not completed by a deadline, Typecoin requires a type-checking escrow agent, but Bitcoin can do it natively.

A recent alternative is the nascent Ethereum system [22]. Ethereum can be viewed as a variant of Bitcoin with a much more expressive scripting language for implementing smart contracts. To prevent denial-of-service attacks, code running on Ethereum carries "fuel" that is consumed as the code executes. Exhausting its fuel is one way that an Ethereum program may halt abnormally (the others being various runtime type errors).

We conjecture that Ethereum's scripting language can be encoded in LF, the logical framework on which Typecoin is built. If so, Typecoin's expressive power theoretically subsumes Ethereum, although Ethereum might well offer better performance for applications it does support. Moreover, Ethereum mandates a new infrastructure; it does not build on the existing Bitcoin infrastructure as Typecoin does.

Bowers *et al.* [6] describe another method to implement affine credentials in a proof-carrying authorization system that avoids the need for a *single* centralized trusted system for tracking resources by allowing affirmations to be associated with a *ratifier* that will track linear use for the resource. Linearity of resource use in a proof is then enforced by a *ratification* step in which the ratifiers of each resource the proof consumes must sign off on the proof using a fair contract signing algorithm.


### Acknowledgements

We gratefully acknowledge Robert Harper, Robert Simmons, and Glenn Willen for very helpful discussions and suggestions.


## A Judgements and Selected Rules

The Typecoin judgements use four different kinds of contexts: LF contexts ($\Psi$), persistent proof term contexts ($\Gamma$), affine proof term contexts ($\Delta$), and condition contexts ($\Phi$):

$$\begin{array}{lllll}\text{LF context} & \Psi & ::= & \cdot \mid \Psi, u{:}\tau \\ \text{proof term context} & \Gamma, \Delta & ::= & \cdot \mid \Gamma, x{:}A \\ \text{condition context} & \Phi & ::= & \cdot \mid \Phi, \varphi\end{array}$$

All contexts except LF contexts are taken to be unordered. In addition, most of the judgements use a basis ($\Sigma$) to resolve constants. There are thirteen judgement forms in Typecoin:

$$\begin{array}{ll}\Sigma \vdash \Sigma'\ \mathsf{ok} & \text{basis formation} \\ \Sigma; \Psi \vdash k\ \mathsf{kind} & \text{LF kind formation} \\ \Sigma; \Psi \vdash \tau : k & \text{LF type family formation} \\ \Sigma; \Psi \vdash m : \tau & \text{LF term typing} \\ \Sigma; \Psi \vdash A\ \mathsf{prop} & \text{proposition formation} \\ \Sigma; \Psi \vdash \varphi\ \mathsf{cond} & \text{condition formation} \\ T; \Sigma; \Psi; \Gamma; \Delta \vdash M : A & \text{proof term typing} \\ \mathfrak{T}; \Sigma \vdash T\ \mathsf{ok} & \text{transaction formation} \\ \mathfrak{T} : \Sigma & \text{chain formation} \\ \Phi \supset \Phi' & \text{condition entailment} \\ A\ \mathsf{fresh} & \text{proposition freshness} \\ \tau\ \mathsf{fresh} & \text{type family freshness} \\ \Sigma\ \mathsf{fresh} & \text{basis freshness}\end{array}$$

The proof term typing judgement, $T; \Sigma; \Psi; \Gamma; \Delta \vdash M : A$ has a lot of moving parts. It expresses that the proof term $M$ proves $A$ under the LF context $\Psi$, the persistent context $\Gamma$, the linear context $\Delta$, the basis $\Sigma$, as part of the transaction $T$. The transaction $T$ needs to be part of the judgement because linear affirmations must be signed relative to the transaction, in order to prevent replay attacks on it. In the interest of brevity, we nearly always omit the $T$.

$$\boxed{T; \Sigma; \Psi; \Gamma; \Delta \vdash M : A}$$

$$\frac{\Sigma; \Psi \vdash m : \mathsf{principal} \quad \Sigma; \Psi; \Gamma; \Delta \vdash M : A}{\Sigma; \Psi; \Gamma; \Delta \vdash \mathsf{sayreturn}_m M : \langle m \rangle A}$$

$$\frac{\Sigma;\Psi;\Gamma;\Delta \vdash M_1 : \langle m\rangle A \quad \Sigma;\Psi;\Gamma;\Delta', x{:}A \vdash M_2 : \langle m\rangle B}{\Sigma;\Psi;\Gamma;(\Delta,\Delta') \vdash \mathsf{saybind}\ x \leftarrow M_1\ \mathsf{in}\ M_2 : \langle m\rangle B}$$

$$\frac{T = (\Sigma', C, \vec{\iota}, \vec{\omega}, M) \qquad \Sigma;\Psi \vdash A\,\mathsf{prop} \qquad \begin{array}{c}sig \text{ is a signature by } K \text{ of } A, \Sigma', C, \vec{\iota}, \vec{\omega}\end{array}}{T;\Sigma;\Psi;\Gamma;\Delta \vdash \mathsf{assert}(K, A, sig) : \langle K\rangle A}$$

$$\frac{sig \text{ is a signature by } K \text{ of } A \quad \Sigma;\Psi \vdash A\,\mathsf{prop}}{\Sigma;\Psi;\Gamma;\Delta \vdash \mathsf{assert!}(K, A, sig) : \langle K\rangle A}$$

$$\frac{\Psi \vdash \varphi\ \mathsf{cond} \quad \Sigma;\Psi;\Gamma;\Delta \vdash M : A}{\Sigma;\Psi;\Gamma;\Delta \vdash \mathsf{ifreturn}_\varphi(M) : \mathsf{if}(\varphi, A)}$$

$$\frac{\Sigma;\Psi;\Gamma;\Delta \vdash M_1 : \mathsf{if}(\varphi, A) \quad \Sigma;\Psi;\Gamma;\Delta', x{:}A \vdash M_2 : \mathsf{if}(\varphi, B)}{\Sigma;\Psi;\Gamma;(\Delta,\Delta') \vdash \mathsf{ifbind}\ x \leftarrow M_1\ \mathsf{in}\ M_2 : \mathsf{if}(\varphi, B)}$$

$$\frac{\Psi \vdash \varphi\ \mathsf{cond} \quad \varphi \supset \varphi' \quad \Sigma;\Psi;\Gamma;\Delta \vdash M : \mathsf{if}(\varphi', A)}{\Sigma;\Psi;\Gamma;\Delta \vdash \mathsf{ifweaken}_\varphi(M) : \mathsf{if}(\varphi, A)}$$

$$\frac{\Sigma;\Psi;\Gamma;\Delta \vdash M : \langle m\rangle \mathsf{if}(\varphi, A)}{\Sigma;\Psi;\Gamma;\Delta \vdash \mathsf{if/say}(M) : \mathsf{if}(\varphi, \langle m\rangle A)}$$

$\boxed{\mathfrak{T};\Sigma \vdash T\ \mathsf{ok}}$

$$\frac{\begin{array}{l}T = (\Sigma, C, \vec{\iota}, \vec{\omega}, M)\\ \Sigma_{\text{global}} \vdash \Sigma\ \mathsf{ok} \qquad \Sigma\ \mathsf{fresh}\\ (\Sigma_{\text{global}}, \Sigma); \cdot \vdash C\ \mathsf{prop} \qquad C\ \mathsf{fresh}\\ \vec{\iota} = txid_1.n_1 \rightarrowtail A_1/a_1, \ldots, txid_\alpha.n_\alpha \rightarrowtail A_\alpha/a_\alpha\\ \vec{\omega} = B_1/b_1 \twoheadrightarrow K_1, \ldots, B_\beta/b_\beta \twoheadrightarrow K_\beta\\ (\Sigma_{\text{global}}, \Sigma); \cdot \vdash A_i\ \mathsf{prop} \quad (\text{for } i = 1 \ldots \alpha)\\ (\Sigma_{\text{global}}, \Sigma); \cdot \vdash B_i\ \mathsf{prop} \quad (\text{for } i = 1 \ldots \beta)\\ \text{output } n_i \text{ of } txid_i \text{ in } \mathfrak{T} \text{ is } A_i' \text{ and}\\ \qquad A_i = [txid_i/\mathsf{this}]A_i' \quad (\text{for } i = 1 \ldots \alpha)\\ A = A_1 \otimes \cdots \otimes A_\alpha\\ R = \mathsf{receipt}(\omega_1) \otimes \cdots \otimes \mathsf{receipt}(\omega_\beta)\\ B = B_1 \otimes \cdots \otimes B_\beta\\ T; (\Sigma_{\text{global}}, \Sigma); \cdot; \cdot \vdash M : (C \otimes A \otimes R) \multimap \mathsf{if}(\varphi, B)\\ \text{the condition } \varphi \text{ holds}\end{array}}{\mathfrak{T};\Sigma_{\text{global}} \vdash T\ \mathsf{ok}}$$

$\boxed{\mathfrak{T} : \Sigma}$

$$\frac{}{\cdot : \cdot} \qquad \frac{\mathfrak{T} : \Sigma_{\text{global}} \quad \mathfrak{T};\Sigma_{\text{global}} \vdash T\ \mathsf{ok} \quad T = (\Sigma, C, \vec{\iota}, \vec{\omega}, M)}{\mathfrak{T}, txid{:}T : \Sigma_{\text{global}}, [txid/\mathsf{this}]\Sigma}$$

$\boxed{\tau\ \mathsf{fresh} \quad A\ \mathsf{fresh} \quad \Sigma\ \mathsf{fresh}}$

There are no freshness rules for restricted forms (non-local constants, 0, affirmations, receipts).

$$\frac{}{\mathsf{this}.l\ \mathsf{fresh}} \qquad \frac{\tau\ \mathsf{fresh}}{\tau\ m\ \mathsf{fresh}} \qquad \frac{\tau'\ \mathsf{fresh}}{\Pi x{:}\tau.\tau'\ \mathsf{fresh}}$$

$$\frac{B\ \mathsf{fresh}}{A \multimap B\ \mathsf{fresh}} \qquad \frac{A\ \mathsf{fresh}}{\forall u{:}\tau.A\ \mathsf{fresh}} \qquad \frac{\tau\ \mathsf{fresh} \quad A\ \mathsf{fresh}}{\exists u{:}\tau.A\ \mathsf{fresh}}$$

$$\frac{A\ \mathsf{fresh} \quad B\ \mathsf{fresh}}{A\ \&\ B\ \mathsf{fresh}} \qquad \frac{A\ \mathsf{fresh} \quad B\ \mathsf{fresh}}{A \otimes B\ \mathsf{fresh}} \qquad \frac{A\ \mathsf{fresh} \quad B\ \mathsf{fresh}}{A \oplus B\ \mathsf{fresh}}$$

$$\frac{}{1\ \mathsf{fresh}} \qquad \frac{A\ \mathsf{fresh}}{!A\ \mathsf{fresh}}$$

$$\frac{}{\cdot\ \mathsf{fresh}} \qquad \frac{\Sigma\ \mathsf{fresh} \quad s\ \mathsf{fresh}}{\Sigma, \mathsf{this}.\ell{:}s\ \mathsf{fresh}} \qquad \frac{\Sigma\ \mathsf{fresh}}{\Sigma, \mathsf{this}.\ell{:}k\ \mathsf{fresh}}$$

$\boxed{\Phi \supset \Phi'}$

Our condition entailment rules are those of the classical sequent calculus:

$$\frac{}{\Phi, \varphi \supset \varphi, \Phi'} \qquad \frac{t \leq t'}{\Phi, \mathsf{before}(t) \supset \mathsf{before}(t'), \Phi'}$$

$$\frac{\Phi, \varphi_1, \varphi_2 \supset \Phi'}{\Phi, (\varphi_1 \wedge \varphi_2) \supset \Phi'} \qquad \frac{\Phi \supset \varphi_1, \Phi' \quad \Phi \supset \varphi_2, \Phi'}{\Phi \supset (\varphi_1 \wedge \varphi_2), \Phi'}$$

$$\frac{}{\Phi \supset \mathsf{true}, \Phi'} \qquad \frac{\Phi, \varphi \supset \Phi'}{\Phi \supset \neg\varphi, \Phi'} \qquad \frac{\Phi \supset \varphi, \Phi'}{\Phi, \neg\varphi \supset \Phi'}$$

## References


[1] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *ACM Conference on Computer and Communications Security*, 1999.

[2] Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Department of Computer Science, University of Edinburgh, September 1996.

[3] Ljudevit Bauer. *Access Control for the Web via Proof-carrying Authorization.* PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, November 2003.

[4] Bitcoin wiki. Script. Wiki page at `https://en.bitcoin.it/wiki/Script`, 2010.

[5] Bitcoin wiki. BIP 0011: M-of-N standard transactions. Wiki page at `https://en.bitcoin.it/wiki/BIP_0011`, 2011.

[6] Kevin D. Bowers, Lujo Bauer, Deepak Garg, Frank Pfenning, and Michael K. Reiter. Consumable credentials in logic-based access-control systems. In *14th Annual Network and Distributed System Security Symposium*, pages 143–157, San Diego, California, February 2007.

[7] Deepak Garg, Lujo Bauer, Kevin Bowers, Frank Pfenning, and Michael Reiter. A linear logic of authorization and knowledge. In *Computer Security—ESORICS 2006: 11th European Symposium on Research in Computer Security*, volume 4189 of *Lecture Notes in Computer Science*, pages 297–312. Springer, September 2006.

[8] Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic. In *19th IEEE Computer Security Foundations Workshop*, 2006.

[9] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[10] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2), 1991.

[11] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[12] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Transactions on Computational Logic*, 6(1), 2005.

[13] Mike Hearn et al. Contracts. Wiki page at `https://en.bitcoin.it/wiki/Contracts`, 2011.

[14] Tom Hvitved. A survey of formal languages for contracts. In *Formal Language and Analysis of Contract-Oriented Software*, 2010.

[15] Killerstorm. The theory of colored coins. GitHub page at `https://github.com/bitcoinx/colored-coin-tools/wiki/colored_coins_intro`, 2013.

[16] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Disseminated to The Cryptography Mailing List, November 2008.

[17] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering. In *2000 ACM International Conference on Functional Programming*, Montreal, September 2000.

[18] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. The evolution of Forth. *SIGPLAN Notices*, March 1993.

[19] Meni Rosenfeld. Overview of colored coins. Available at `https://bitcoil.co.il/BitcoinX.pdf`, December 2012.

[20] Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, November 2012.

[21] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), September 1997.

[22] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Available online at `http://gavwood.com/Paper.pdf`, 2014.