Keywords
linear logic, blockchain, types, Curry-Howard

ABSTRACT
We present an interpretation of classical linear logic in terms of operations on the blockchain.

Linear types can change the blockchain!

L.G. Meredith
CSO, Synereo
greg@synereo.com

1. BACKGROUND AND MOTIVATION

Anyone who understands the current economic, sociological, and technological situation is likely to be very excited by what the blockchain technology promises. Anyone who has actually had to work with the blockchain in real situations with mission-critical exchanges on the line is very likely to be motivated to find a more scalable and reliable architecture for the blockchain. This paper takes a few key steps towards finding a way to explain and test a hypothesis that linear proofs provide the basis for a much more scalable architecture for the blockchain. For background on what is meant here by linear proofs, [5] interprets them in terms of games, while [3] interprets them in terms of traditional computational calculi like the lambda calculus.

A linear proof is a formal structure representing a proof of a formula in linear logic [6]. The Curry-Howard isomorphism [13] tells us that formulae are types (as in data types in a programming language), and that proofs are programs. This is a very broad and deep idea. In the 90's, for example, Abramsky extended it to proofs as processes [4], which Wadler was only very recently able to realize as a correspondence between linear proofs and π-calculus processes [14]. In this context it means that linear proofs provide a representation of both data (blocks) and program (executable transactions) that gives several advantages over the current choices made by the blockchain.

The blockchain is a great example of data that is also program; it's a giant ledger spread out over the Internet, that's made of a bunch of distributed, but interacting servers [11]. To become more scalable, and reliable, both ledger and servers will need certain characteristics of data/program that have to do with a property called compositionality. Scalability is always all about being able to build composite systems from components. For example, if we can prove that sections of the blockchain can be safely isolated from other sections, for example, if all blocks necessary to prove that Alice has sufficient funds to send M btc to Betty, can be isolated from the blocks necessary to prove that Alfred has sufficient funds to send N btc to Bob, then Alice and Betty, and Alfred and Bob can safely work with projections of the blockchain, and thus complete their transactions, not only in isolation of each other, but without the onerous need to sync the entire blockchain.

One analogy is the use of separation logic (a child of linear logic) [12] to prove things about the structure of the heap which can, in turn, be used to guarantee that two threads can operate at the same time safely. The blockchain is like the heap. The Alice - Betty and Alfred - Bob transactions are like the two threads. A proof that the heap is of the form $H_1 \otimes H_2$ together with a proof that $T_1 : H_1 \to H_1'$, $T_2 : H_2 \to H_2'$ constitutes a proof that $T_1 \otimes T_2$ (thread T_1 running concurrently with T_2) operate effectively in isolation and thus safely. Likewise, a proof that the blockchain is of the form $B_1 \otimes B_2$ (none of the transactions in B_1 connect to addresses in B_2 and vice versa), together with a proof that $\mathsf{AliceBetty} : B_1 \to B_1'$ (this txn uses only addresses in B_1), and $\mathsf{AlfredBob} : B_2 \to B_2'$ (this txn uses only addresses in B_2) constitutes a proof that $\mathsf{AliceBetty} \otimes \mathsf{AlfredBob}$ can operate with isolated projections of the blockchain.

If the blockchain is built using the primitives of linear logic, it becomes easier and easier to construct these proofs, but also to construct the blockchain, itself, in terms of smaller blockchains.

2. INTERPRETING LINEAR PROOFS AS OPERATIONS ON THE BLOCKCHAIN

Here's the most basic interpretation.

$$\vdash 1blkchnaddr : \underbrace{A \otimes \ldots \otimes A}_{M}$$

is a statement that there are M A's available at the address, $1blkchnaddr$. A's can be any resource, BTC's, AMP's, DogeCoin, etc.

$$\vdash txn : \underbrace{A \otimes \ldots \otimes A}_{M} \multimap \underbrace{B \otimes \ldots \otimes B}_{N}$$

is a statement that txn will generate N B's if provided M A's.

Terminologically, we say that $1blkchnaddr$ is a *witness* or a *proof* of $A \otimes \ldots \otimes A$, and similarly, that txn is a witness or proof of $A \otimes \ldots \otimes A \multimap B \otimes \ldots \otimes B$. Given two such proofs we can use the cut rule of linear logic to produce a proof

$$\vdash txn(1blkchnaddr) : B \otimes \ldots \otimes B$$

where $txn(1blkchnaddr)$ is a new address in the blockchain constructed from the information in txn together with $1blkchnaddr$. This should look remarkably like function application, because it is.

Notice that now we can see, recursively, what a proof of a statement like $\vdash 1blkchnaddr : A \otimes \ldots \otimes A$ looks like. In most cases it will be a proof made from a previous application of the cut rule. This tree of cuts will trace all the way back to some genesis block – which is the only other way to have a proof of a statement like $\vdash 1blkchnaddr : A \otimes \ldots \otimes A$.

Now, where does the return address associated with $txn(1blkchnaddr)$ come from? To see this we have to look into the mechanics of $\multimap$ (called linear implication or, more affectionately, lollipop).

$$A \multimap B = A^{\perp} \parr B$$

Linear implication is decomposed much like classical implication in terms of a negation ($A^{\perp}$) and a disjunctive connective ($A \parr B$). It is literally an expression capturing the sentiment we need A to get B, or B comes with a cost of A. The use of a proof rule for these kinds of links looks like

$$\frac{\vdash \Gamma, 1blkchnaddr : A^{\perp}, 2blkchnaddr : B}{\vdash \Gamma, 1blkchnaddr \parr 2blkchnaddr : A^{\perp} \parr B}$$

where we snuck in the rest of the blockchain as Γ. As we saw above, $A^{\perp} \parr B = A \multimap B$; so, we can write

$$\frac{\vdash \Gamma, 1blkchnaddr : A^{\perp}, 2blkchnaddr : B}{\vdash \Gamma, 1blkchnaddr \multimap 2blkchnaddr : A \multimap B}$$

Now we see that forming a txn comes with the requirement to provide an address where A's will be sent and an address where B's will be received. To complete the picture, applying the cut rule will create the txn that links an address, say $3blkchnaddr$ with an A in it, to $1blkchnaddr$, resulting in $\vdash \Gamma, 2blkchnaddr : B$. Expanding on these intuitions, we can see that the rules of classical linear logic correspond exactly to a specification of operations on the blockchain.

2.1 Linear Sequents

In more detail, a proof rule in linear logic is usually written in terms of a transformation,

$$\frac{S_1, S_2, \ldots, S_N}{S}$$

taking *sequent*s $S_1, \ldots, S_N$ to a *sequent* S, where a sequent is of the form

$$\vdash \Gamma, t_1 : A_1, \ldots, t_N : A_N$$

A sequent is really just a statement about what is distributed where in an instance of the blockchain.

- $t_1, \ldots, t_N$ are either addresses or programs that take addresses as input; they constitute the "focus" of the proof rule, or where the action is going to happen.
- $A_1, \ldots, A_N$ are (types built from) the different types of coin
- Γ is the rest of the blockchain – it is necessary to establish the distribution of resources we see at $t_1, \ldots, t_N$, but it's not the focus of the operation of the proof rule.

Putting it all together, a proof rule of the form

$$\frac{S_1}{S_2}$$

is then a statement about how the blockchain in state S_1 goes to a blockchain in state S_2. If you think about it, that's just what we need to reason about transactions. In a transaction where Alice sends Betty N coin, we can think of the transaction as a rule that takes a blockchain in a state where Alice has N btc to a blockchain in a state where Betty has N btc.

2.2 The Multiplicatives

Linear logic, however, allows to build bigger blockchains from smaller ones, and manages the dependency and information flow so that everything remains consistent. Here's an example. The proof rule for the tensor $A \otimes B$ looks like this

$$\frac{\vdash \Gamma, t : A, \vdash \Delta, u : B}{\vdash \Gamma, \Delta, t \otimes u : A \otimes B}$$

It says that if you have one blockchain, $\vdash \Gamma, t : A$, and another completely independent blockchain, with a totally separate address space, $\vdash \Delta, u : B$, then you can make a new one

$$\vdash \Gamma, \Delta, t \otimes u : A \otimes B$$

in which you just combine all the data of assignments of addresses to resources in G and H in one big blockchain, G, H, and you can make a kind of composite address (or program), $t \otimes u$, at which can be found the combined $A \otimes B$ resource.

Now, comparison of the par ($A \parr B$) rule, which establishes transaction links, is even more illuminating.

$$\frac{\vdash \Gamma, t : A, u : B}{\vdash \Gamma, t \parr u : A \parr B}$$

This rule insists that the transaction link, $t \parr u$, is made in the same piece of the blockchain, Γ.

The piece of the puzzle that interprets commitment to and execution of transactions is the cut rule. If $1blkchnaddr \multimap 2blkchnaddr$ is a transaction waiting to happen, so to speak, $txn(3blkchnaddr, 1blkchnaddr - o2blkchnaddr)$ is the commitment to carry out the txn against the blockchain. Likewise, cut-elimination, also called proof-normalization, which corresponds to computation, via Curry-Howard, constitutes the execution of the transaction on the blockchain that results in the assignment $\vdash \Gamma, 2blkchnaddr : B$ after execution. Someone familiar with functional programming might interpret

$txn(3blkchnaddr, 1blkchnaddr \multimap 2blkchnaddr)$

as

$apply(1blkchnaddr \multimap 2blkchnaddr, 3blkchnaddr)$

making the correspondence to function application, and the correspondence between proof normalization and β-reduction explicit.

The fragment of linear logic that includes, $A^{\perp}$, $A \otimes B$, $A \parr B$, $A \multimap B$, is called the multiplicative fragment of linear logic, or MLL. It talks about the basics of transactions, loading up addresses with resources and establishing dependencies between addresses, essentially recording transaction history. However, it does so in a way that keeps track of how the blockchain itself is segmented. This allows us to determine things like how much of the blockchain do i have to see in order to safely conduct this transaction, or can i conduct this transaction without needing visibility into that region of the blockchain.

2.3 The Additives

Linear logic also enjoys another fragment, called the additives. This aspect of the logic is all about conditionals and contingencies, this or that, but not both. The linear logic connective called 'with', and denoted $A \,\&\, B$, collects options together into a menu for subsequent selection by interaction with choices indicated by the linear connective 'plus', $A + B$. In symbols,

$$\frac{\vdash \Gamma, t : A, u : B}{\vdash \Gamma, t \,\&\, u : A \,\&\, B}$$

while

$$\frac{\vdash \Gamma, t : A}{\vdash inl(t) : A + B}$$

and

$$\frac{\vdash \Gamma, u : B}{\vdash inr(u) : A + B}$$

If during a more complex transaction $t \,\&\, u$ gets tied to $inl(t')$, via $txn(t \,\&\, u, inl(t'))$, then this will reduce to a transaction of the form $txn(t, t')$. On the other hand, $txn(t \,\&\, u, inr(u'))$ will reduce to a transaction of the form $txn(u, u')$.

2.4 The Exponentials

The fragment of linear logic that includes the multiplicative and additive connectives is called MALL. The remaining connectives are called the exponentials, $?A$, and $!A$. They denote copyable, non-conserved resources. When we write $\vdash \Gamma, t :!A$, we are saying that you can get as many A's from the address (or program) t as you want. Thus, unlike currency, that address is linked to a copyable resource like a document, or a jpeg, or audio file, or ... that can be shared widely. When we write $\vdash \Gamma, t :?A$, we are saying that you can put as many A's into the address (or program) t as you want. You can think of it as a place to store A's, or discard them.

What's critically important about the use of the exponentials is that they mark resources that ought not to stay on the blockchain. They indicate content and content types that can be better served by a different kind of content delivery network. This is another important function in helping with a scalable blockchain – use blockchain technology where it makes sense and use other means where it doesn't.

Taken all together, we have an interpretation of full classical linear logic in terms of operations on the blockchain.

3. CONCLUSIONS AND FUTURE WORK

We have developed a view of full classical linear logic in terms of operations against the blockchain. The view we have been developing not only extends to provide a meaningful interpretation of full classical linear logic to natural and intuitive operations on the blockchain, it also extends and expands how we think about the blockchain and what transactions on it are. Additionally, it provides guarantees, mathematical certainties about the correctness of transactions structured and executed this way. In particular, notice that we focused mostly on the connectives governing A's and B's (the resources to be found at addresses or programs). We didn't really talk about the structure of t's and u's. These provide us with a simple and intuitive syntax for transactions. Of equal importance, these transactions are *typed* programs. When we write $\vdash \Gamma, t : A$, we are not only saying something about the resources produced or manipulated by t, we are saying something about how t can be used, and in what blockchain context we can expect t to perform correctly.

Understood this way, the blockchain interpretation gives new meaning and perspective on some theorems from the linear logic literature. In particular, it is well established that there is a natural notion of execution of t's. That is, when thought of as programs, we know how to run them. When they are well typed, that is, if we have established $\vdash t : A$, then t is *terminating*. That's a theorem from [3]. What this means for the blockchain is that proof terms and their linear connectives provide a scripting language for transactions that, on the one hand, provides termination for all well typed scripts, and on the other is highly expressive. Further, if it turns out that this scripting language is not expressive enough, then there is a natural extension of proof terms via a correspondence between linear proof terms and π-calculus processes that we mentioned at the top of these notes.

proof term	blockchain meaning
address	address
$t \otimes u$	isolated concurrent transactions
$t \parr u$	interacting or linked concurrent transactions
$t \,\&\, u$	menu of transaction options
$inl(t), inr(u)$	transaction option selection
$!t$	copyable resource server
$?t$	copyable resource storage
$txn(t, u)$	joined transactions

This correspondence is not just useful for extending a scripting language for blockchain transactions. It turns out the π-calculus the premier formalism for specifying, reasoning about, and executing protocols in distributed systems [10] [9] [2] [1] [7] [8]. Since one of the real values of the blockchain is the fact that it is a distributed means to conduct transactions, the need to tie this formalism to one for specifying protocols in distributed systems is plain.

3.1 Proof-of-work

The glaring lacunae in this discussion is, of course, the relationship to proof-of-work. Consider the following example. Suppose C_1 and C_2 are blockchains both of height N.

$$C_1 = B_{1N} \leftarrow B_{1N-1} \leftarrow \ldots \leftarrow B_{10}$$
$$C_2 = B_{2N} \leftarrow B_{2N-1} \leftarrow \ldots \leftarrow B_{20}$$

We can define

$$C_1 \otimes C_2 = (B_{1N} \otimes B_{2N}) \leftarrow (B_{1N-1} \otimes B_{2N-1}) \leftarrow \ldots \leftarrow (B_{10} \otimes B_{20})$$

Note that it is insufficient merely to guarantee for $B \otimes B'$ that all the transactions in B are isolated from the transactions in B'. The counterexample is

$$C_1 = Block\{1AliceAddr \xrightarrow{5btc} 1AllanAddr\}$$
$$\leftarrow Block\{1BobAddr \xrightarrow{7btc} 1BettyAddr\}$$
$$C_2 = Block\{1BobAddr \xrightarrow{7btc} 1BettyAddr\}$$
$$\leftarrow Block\{1AliceAddr \xrightarrow{5btc} 1AllanAddr\}$$

Clearly B_{11} is isolated from B_{21}, and B_{10} is isolated from B_{20}; but, B_{20} is not isolated from B_{11}, and B_{10} is not isolated from B_{21}. As a result, the spends in the earlier blocks could impact the spends in the later blocks.

Instead, the entire address space of C_1 must be isolated from C_2. In this case the network of servers, N_1, that maintain C_1 can be safely combined with the network of servers, N_2, that maintain C_2, and we can safely define the composite chain as above. The proof-of-work protocol organizing N_1 is completely separate from that in N_2. They do not interact. Yet, it is safe to combine the chains using a glorified zip function. In this example,

$$C_1 = Block\{1AliceAddr \xrightarrow{5btc} 1AllanAddr\}$$
$$\leftarrow Block\{2BobAddr \xrightarrow{7btc} 2BettyAddr\}$$
$$C2 = Block\{1BobAddr \xrightarrow{7btc} 1BettyAddr\}$$
$$\leftarrow Block\{2AliceAddr \xrightarrow{5btc} 2AllanAddr\}$$

The address spaces of these chains are completely isolated (often written $addresses(C_1)\#addresses(C_2)$). We are free to calculate

$$C_1 \otimes C_2 = Block\{1AliceAddr \xrightarrow{5btc} 1AllanAddr\}$$
$$\otimes Block\{1BobAddr \xrightarrow{7btc} 1BettyAddr\}$$
$$\leftarrow Block\{2BobAddr \xrightarrow{7btc} 2BettyAddr\}$$
$$\otimes Block\{2AliceAddr \xrightarrow{5btc} 2AllanAddr\}$$
$$= Block\{1AliceAddr \xrightarrow{5btc} 1AllanAddr;$$
$$1BobAddr \xrightarrow{7btc} 1BettyAddr\}$$
$$\leftarrow Block\{2BobAddr \xrightarrow{7btc} 2BettyAddr$$
$$;2AliceAddr \xrightarrow{5btc} 2AllanAddr\}$$

The ordering of transactions provided by the two independently executing proof-of-work protocols is combined in a completely safe.

Note that there are at least two possible interpretations of $C_1 * C_2$. One is that the requirement is to verify that $addresses(C_1)\#addresses(C_2)$. Another is to ensure this is the case by rewiring the transactions. Under this latter interpretation even the counterexample becomes safe

$$C_1 \otimes C_2 = Block\{1AliceAddr \xrightarrow{5btc} 1AllanAddr\}$$
$$\otimes Block\{1BobAddr \xrightarrow{7btc} 1BettyAddr\}$$
$$\leftarrow Block\{1BobAddr \xrightarrow{7btc} 01BettyAddr\}$$
$$\otimes Block\{1AliceAddr \xrightarrow{5btc} 11AllanAddr\}$$
$$= Block\{01AliceAddr \xrightarrow{5btc} 01AllanAddr;$$
$$11BobAddr \xrightarrow{7btc} 11BettyAddr\}$$
$$\leftarrow Block\{01BobAddr \xrightarrow{7btc} 01BettyAddr;$$
$$11AliceAddr \xrightarrow{5btc} 11AllanAddr\}$$

There is much more to be said, but that must be left to future work!

Acknowledgments. We would like to acknowledge Vlad Zamfir for some thoughtful and stimulating conversation about the blockchain protocol.

4. REFERENCES

[1] Martín Abadi, Bruno Blanchet, and Cédric Fournet, *Just fast keying in the pi calculus*, ACM Trans. Inf. Syst. Secur. **10** (2007), no. 3.

[2] Martín Abadi, Ricardo Corin, and Cédric Fournet, *Computational secrecy by typing for the pi calculus*, Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings (Naoki Kobayashi, ed.), Lecture Notes in Computer Science, vol. 4279, Springer, 2006, pp. 253–269.

[3] Samson Abramsky, *Computational interpretations of linear logic*, Theor. Comput. Sci. **111** (1993), no. 1&2, 3–57.

[4] ———, *Proofs as processes*, Theor. Comput. Sci. **135** (1994), no. 1, 5–9.

[5] Samson Abramsky and Paul-André Melliès, *Concurrent games and full completeness*, 14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999, IEEE Computer Society, 1999, pp. 431–442.

[6] Jean-Yves Girard, *Linear logic*, Theor. Comput. Sci. **50** (1987), 1–102.

[7] Andrew D. Gordon, *Provable implementations of security protocols*, 21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15

August 2006, Seattle, WA, USA, Proceedings, IEEE Computer Society, 2006, pp. 345–346.

[8] Steve Kremer and Mark Ryan, *Analysis of an electronic voting protocol in the applied pi calculus*, Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Shmuel Sagiv, ed.), Lecture Notes in Computer Science, vol. 3444, Springer, 2005, pp. 186–200.

[9] Robin Milner, *Functions as processes*, Mathematical Structures in Computer Science **2** (1992), no. 2, 119–141.

[10] ———, *The polyadic π-calculus: A tutorial*, Logic and Algebra of Specification **Springer-Verlag** (1993).

[11] Satoshi Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, Bitcoin.org (2008).

[12] John C. Reynolds, *An overview of separation logic*, Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions (Bertrand Meyer and Jim Woodcock, eds.), Lecture Notes in Computer Science, vol. 4171, Springer, 2005, pp. 460–469.

[13] Morten Heine B. Sørensen and Pawel Urzyczyn, *Lectures on the curry-howard isomorphism*, 1998.

[14] Philip Wadler, *Propositions as sessions*, J. Funct. Program. **24** (2014), no. 2-3, 384–418.

5. APPENDIX: A TERMINATING SCRIPTING LANGUAGE

In the main body of the paper we presented what amounts to the high level intuitions. In this appendix we present enough of the details that a reader skilled in the art could implement the proposal to test it for themselves. This presentation follows Abramksy's proof expressions from [3] very closely.

5.1 Syntax

$p, q ::= (e_1, \ldots, e_m)\{t_1;\ \ldots;\ t_n\}$	programs
$e ::= \mathsf{satoshi} \mid \ldots \mid \mathsf{ampere}$	currency units
$\mid x$	address
$\mid e * e$	isolation
$\mid e \# e$	connection
$\mid e \multimap e$	obligation
$\mid \mathsf{choose}(x_1, \ldots, x_n)\{p; q\}$	menu
$\mid \mathsf{inl}(e) \mid \mathsf{inr}(e)$	selection
$\mid ?e \mid \_$	storage, disposal
$\mid e@e$	contraction
$\mid !(x_1, \ldots, x_n)\{p\}$	replication
$t ::= \mathsf{txn}(e_1, e_2)$	transaction

Discussion. $e_1 \multimap e_2$ is really just convenient syntactic sugar for $e_1^{\perp} \# e_2$, where $e^{\perp}$ is identity on addresses, but changes the *polarity* of the type and otherwise operates as

$$(e_1 * e_2)^{\perp} = e_1^{\perp} \# e_2^{\perp}$$
$$(e_1 \# e_2)^{\perp} = e_1^{\perp} * e_2^{\perp}$$

5.1.1 Interpretation

Programs p and q represent blockchain states. For $p = (e_1, \ldots, e_m)\{t_1;\ \ldots;\ t_n\}$, the e's represent resources available on the blockchain p, while the t's represent transactions in progress. For example, if we write $M \cdot \mathsf{satoshi}$ for $\underbrace{\mathsf{satoshi} * \ldots * \mathsf{satoshi}}_{M}$, then

$$(1blkchnaddr)\{\mathsf{txn}(1blkchnaddr, M \cdot \mathsf{satoshi})\}$$

represents the genesis block where $1blkchnaddr$ has been assigned M satoshi's. At the other end of the spectrum,

$$(1blkchnaddr)\{\mathsf{txn}(1blkchnaddr, \_)\}$$

represents burning the assets sent to $1blkchnaddr$.

At this level of abstraction modeled by the operational semantics in the next section, addresses are more closely aligned with transaction inputs in blockchain transactions. Thus, the genesis block is more accurately represented as

$$\begin{array}{l}(addr_1 * \ldots * addr_M)\{\\ \quad \mathsf{txn}(addr_1, \mathsf{satoshi});\\ \quad \ldots;\\ \quad \mathsf{txn}(addr_M, \mathsf{satoshi})\\ \}\end{array}$$

which for future reference we'll write genesis. Similarly, the second example is more accurately written as

$$(addr_1 * \ldots * addr_M)\{\mathsf{txn}(addr_1, \_); \ldots; \mathsf{txn}(addr_M, \_)\}$$

we'll write as burn in the sequel.

5.2 Operational Semantics

In what follows we use the notational conventions:

- $\vec{e}$ is a list of e's of length $|\vec{e}|$; likewise $\vec{t}$ is list of t's.
- $\mathsf{txn}(\vec{e}, \vec{e'}) = \mathsf{txn}(e_1, e_1'); \ \ldots; \mathsf{txn}(e_n, e_n')$ assuming $|\vec{e}| = |\vec{e'}|$
- we have operations, $(-)^l : \mathsf{Addr} \to \mathsf{Addr}$, $(-)^r : \mathsf{Addr} \to \mathsf{Addr}$ such that given an address x, x^l, x^r are distinct from x and each other; these operations extend uniquely to p, e, and t in the obvious manner.

TRANSACTION
$$\mathsf{txn}(e_1, x); \mathsf{txn}(x, e_2) \to \mathsf{txn}(e_1, e_2)$$

PAIR
$$\mathsf{txn}(e_1 * e_1', e_2 \# e_2') \to \mathsf{txn}(e_1, e_2); \mathsf{txn}(e_1', e_2')$$

LEFT
$$\mathsf{txn}(\mathsf{choose}(x, \vec{x})\{(e, \vec{e})\{\vec{t}\}; q\}, \mathsf{inl}(e')) \to \mathsf{txn}(e, e'); \vec{t}; \mathsf{txn}(\vec{x}, \vec{e})$$

RIGHT
$$\mathsf{txn}(\mathsf{choose}(x, \vec{x})\{p; (e, \vec{e})\{\vec{t}\}\}, \mathsf{inr}(e')) \to \mathsf{txn}(e, e'); \vec{t}; \mathsf{txn}(\vec{x}, \vec{e})$$

READ
$$\mathsf{txn}(!(\vec{x})\{(e, \vec{e})\{\vec{t}\}\}, ?e') \to \mathsf{txn}(e, e'); \mathsf{txn}(\vec{x}, \vec{e})$$

DISPOSE
$$\mathsf{txn}(!(\vec{x})\{p\}, \_) \to \mathsf{txn}(\vec{x}, \_)$$

COPY
$$\mathsf{txn}(!(\vec{x})\{p\}, e_1 @ e_2) \to \mathsf{txn}(\vec{x}, x^l @ x^r); \mathsf{txn}(!(\vec{x})\{p\}^l, e_1); \mathsf{txn}(!(\vec{x})\{p\}^r, e_2)$$

5.2.1 Interpretation

The operational semantics should be viewed as the specification of an abstract machine that needs no other registers than the program itself. Let's look at an example in some detail.

Executing a transaction amounts to joining to expressions, e_1 and e_2 in $\mathsf{txn}(e_1, e_2)$. Thus, to send $I < M$ $\mathsf{satoshi}$ to $bddr_1 * \ldots * bddr_I$, in the context of the genesis block, first we have to turn the genesis block into an expression.

$$\mathsf{choose}(1 spndaddr)\{\mathsf{genesis}; \mathsf{burn}\}$$

Next, we form a spend expression $bddr_1 * \ldots * bddr_I \multimap addr_{I+1} \# addr_M$ which will consume I $\mathsf{satoshi}$ from the genesis block addresses $addr_1$ through $addr_I$, and deposit them in $addr_1$ through $addr_I$.

Now, we can create a transaction that selects the genesis block from the menu of blockchain states via

$$\begin{array}{l}\mathsf{txn}(\\ \quad \mathsf{choose}(1 spndaddr)\{\mathsf{genesis}; \mathsf{burn}\},\\ \quad \mathsf{inl}(bddr_1 * \ldots * bddr_I \multimap addr_{I+1} \# addr_M)\\ )\end{array}$$

Using the operational semantics we see that this reduces to

$$\begin{array}{l}\mathsf{txn}(\\ \quad addr_1 * \ldots * addr_M,\\ \quad bddr_1 * \ldots * bddr_I \multimap addr_{I+1} \# addr_M\\ );\end{array}$$

which then reduces to

$$\begin{array}{l}\mathsf{txn}(addr_1, bddr_1);\\ \ldots;\\ \mathsf{txn}(addr_I, bddr_I);\\ \mathsf{txn}(addr_{I+1}, addr_{I+1});\\ \ldots;\\ \mathsf{txn}(addr_M, addr_M);\\ \mathsf{txn}(addr_1, \mathsf{satoshi});\\ \ldots;\\ \mathsf{txn}(addr_M, \mathsf{satoshi});\end{array}$$

which then reduces to

$$\begin{array}{l}\mathsf{txn}(bddr_1, \mathsf{satoshi});\\ \ldots;\\ \mathsf{txn}(bddr_I, \mathsf{satoshi});\\ \mathsf{txn}(addr_{I+1}, \mathsf{satoshi});\\ \ldots;\\ \mathsf{txn}(addr_M, \mathsf{satoshi});\end{array}$$

This can be seen as a ledger-like representation assigning satoshi's to addresses.

Now, the final piece of the puzzle is that that spend transaction needs to be created in the context of a blockchain state, which constitutes the *resulting* blockchain state. In point of fact, this is a piece of context we elided when we formed the transaction to focus on the reduction. A more complete picture of the execution looks like

$$
\begin{array}{l}
(bddr_1 * \ldots * bddr_I * addr_{I+1} * \ldots * addr_M)\{ \\
\quad \mathsf{txn}(\\
\quad\quad \mathsf{choose}(1 spndaddr)\{\mathsf{genesis}; \mathsf{burn}\}, \\
\quad\quad \mathsf{inl}(bddr_1 * \ldots * bddr_I \multimap addr_{I+1} \# addr_M) \\
\quad) \\
\} \\
\rightarrow * \\
(bddr_1 * \ldots * bddr_I * addr_{I+1} * \ldots * addr_M)\{ \\
\quad \mathsf{txn}(bddr_1, \mathsf{satoshi}); \\
\quad \ldots; \\
\quad \mathsf{txn}(bddr_I, \mathsf{satoshi}); \\
\quad \mathsf{txn}(addr_{I+1}, \mathsf{satoshi}); \\
\quad \ldots; \\
\quad \mathsf{txn}(addr_M, \mathsf{satoshi}); \\
\}
\end{array}
$$

This brings us full circle. At the beginning of the paper we explicitly recognized the blockchain as data that is program. The reduction above provides an explicit model of just this phenomenon. A blockchain state, i.e. a representation of data, is a *program*. The transition from one state to the next is the execution of the program. Any state of the program actually allows a "read back" to a ledger-like representation capturing the distribution of resources to addresses.

5.3 Type assignment

In the main body of the paper we wrote proof rules in terms of sequents. In point of fact, that formalism amounts to a typing discipline on the scripting language presented above. Here we present the details of that typing discipline along with the basic result that all well typed programs are terminating.

$$\textsc{Axiom}\quad \vdash (x : A^{\perp}, x : A)\{\}$$

$$\textsc{Tensor}\quad \frac{\vdash (t : A, \Gamma)\{t\vec{x}n\} \qquad \vdash (u : B, \Delta)\{t\vec{x}n'\}}{\vdash (t * u : A \otimes B, \Gamma, \Delta)\{t\vec{x}n; t\vec{x}n'\}}$$

$$\textsc{Par}\quad \frac{\vdash (t : A, u : B, \Gamma)\{t\vec{x}n\}}{\vdash (t \# u : A \parr B, \Gamma)\{t\vec{x}n\}}$$

$$\textsc{With}\quad \frac{\begin{array}{c}\vdash p \qquad \vdash q \\ p = (t : A, \vec{t} : \vec{G})\{t\vec{x}n\} \qquad q = (u : B, \vec{u} : \vec{G})\{t\vec{x}n'\}\end{array}}{\vdash (\mathsf{choose}(\vec{x} : \vec{G})\{p\}\{q\}) : A \,\&\, B, \vec{x} : \vec{G})\{\}}$$

$$\textsc{Left}\quad \frac{\vdash (t : A, \Gamma)\{t\vec{x}n\}}{\vdash (inl(t) : A + B, \Gamma)\{t\vec{x}n\}}$$

$$\textsc{Right}\quad \frac{\vdash (u : B, \Gamma)\{t\vec{x}n\}}{\vdash (inr(u) : A + B, \Gamma)\{t\vec{x}n\}}$$

$$\textsc{Storage}\quad \frac{\vdash (t : A, \Gamma)\{t\vec{x}n\}}{\vdash (?t : ?A, \Gamma)\{t\vec{x}n\}} \qquad \textsc{Disposal}\quad \frac{\vdash (t : A, \Gamma)\{t\vec{x}n\}}{\vdash (\_ : ?A, \Gamma)\{t\vec{x}n\}}$$

$$\textsc{Contraction}\quad \frac{\vdash (t : ?A, u : ?A, \Gamma)\{t\vec{x}n\}}{\vdash (t@u : ?A, \Gamma)\{t\vec{x}n\}}$$

$$\textsc{Replication}\quad \frac{\begin{array}{c}\vdash p \\ p = (t : A, \vec{t} : ?\vec{G}, \Gamma)\{t\vec{x}n\}\end{array}}{\vdash (!(\vec{x})\{p\} : !A, \vec{x} : ?\vec{G})\{\}}$$

Discussion. As is easily seen, this is merely a transliteration of Abramsky's proof expressions from [3], and as such the scripting language enjoys all the properties of proof expressions. In particular, theorem 7.18 pg 47 tells us that well typed programs terminate.

In a discussion of "smart contract" the types play a specially important role. If programs in this language consitute financial contracts, then the types provide

a means by which parties can probe the contracts for properties above and beyond termination.