Abstract Ledger Specifications

Polina Vinogradova[1], IOG folk, and Univerity of Edinburgh folk

[1] IOG, firstname.lastname@iohk.io
[2] University of Edinburgh, orestis.melkonian@ed.ac.uk, wadler@inf.ed.ac.uk

Abstract. Ensuring deterministic behaviour in distributed blockchain ledger design matters to end users because it allows for locally predictable fees and smart contract evaluation outcomes. In this work we begin by establishing a formal definition of an abstract ledger together with a ledger update procedure. We use this model as a basis for formalizing and studying several definitions of the collection of properties colloquially classified as determinism. We investigate the relations between these properties, and the implications each of these has for design and static analysis of ledgers, giving simple but illustrative examples of ledgers with and without these properties. We discuss how these ideas can be applied to realistic, deployable ledger design and architecture.

Keywords: blockchain · formal methods · ledger

1 The Abstract Ledger Model

Each instance of $\mathcal{LS}$ (ledger specification, or ledger) requires the structure given in Figure 1 to be specified.
We use the $A_{\mathsf{Err}} = A \uplus \mathsf{Err}$ to represent a non-A error term in addition to terms of the A type.
We omit the subscript $_L$ whenever L is implied, eg. State for State_L. We also use the shorthand here

$$(s\ ds) := \mathsf{validUpdate}_L\ s\ ds$$

Polina: Need better notation than this one (above)

Given L, we denote the set of dependent pairs, together with the Err state, by $\mathsf{ValSt}_{\mathsf{Err}}$, where

- the first term of the dependent pair is a state $s : \mathsf{State}_L$, and
- the second is a proof that there exists a list of transactions that, applied in sequence to the initial state, produce the state s. We call such a list the *trace* of s.

We call $s : \mathsf{State}_L$ a *valid state* whenever $(s, pf) : \mathsf{ValSt}$ for some pf specifying a trace of s. The state s may have more than one trace.

$\mathcal{LS}$ Type Accessors

Tx_L	$\in$	Set	The set of transactions
State_L	$\in$	Set	The set of ledger states

$\mathcal{LS}$ functions

update_L	$\in$	$\mathsf{Tx}_L \to \mathsf{State}_L \to (\mathsf{State}_L)_{\mathsf{Err}}$	Updates state with a given transaction
$\mathsf{initState}_L$	$\in$	State_L	Gives the initial state of the specification

Fig. 1: Ledger Specification for a ledger $L \in \mathcal{LS}$

Proof-constructing function

$$\mathsf{proveVl}_L \in \mathsf{ValSt}_L \to (\mathsf{Tx}_L)^* \to$$

State Computation

$$\mathsf{updateErr}_L \in \mathsf{Tx}_L \to (\mathsf{State}_L)_{\mathsf{Err}} \to (\mathsf{State}_L)_{\mathsf{Err}}$$
$$\mathsf{updateErr}_L\ tx\ s = \begin{cases} \mathsf{Err} & \text{if } s = \mathsf{Err} \\ \mathsf{update}\ tx\ s & \text{otherwise} \end{cases}$$

Updates a error state

$$\mathsf{computeState}_L \in \mathsf{State}_L \to (\mathsf{Tx}_L)^* \to (\mathsf{State}_L)_{\mathsf{Err}}$$
$$\mathsf{computeState}_L\ initS\ txs = \mathsf{foldl}\ \mathsf{updateErr}_L\ initS\ txs$$

Applies a list of transactions to the initial state

$$\mathsf{ValSt}_L \in \mathsf{Set}$$
$$\mathsf{ValSt}_L = \{\ s \mid \mathsf{validState}_L\ s\ \}_{\mathsf{Err}}$$

Set of all valid states of L

$$\mathsf{validUpdate}_L \in \mathsf{ValSt}_L \to (\mathsf{Tx}_L)^* \to \mathsf{ValSt}_L$$
$$\mathsf{validUpdate}_L\ s\ txs =$$
$$\begin{cases} \mathsf{Err} & \text{if } s = \mathsf{Err} \\ (\mathsf{computeState}_L\ s\ txs,\ \mathsf{proveVl}_L\ s\ txs) & \text{otherwise} \end{cases}$$

A valid state is updated to another valid state or Err

Validity of ledgers, transactions, and states

$$\mathsf{validTx}_L \in \mathsf{Tx}_L \to \mathsf{State}_L \to \mathsf{Bool}$$
$$\mathsf{validTx}_L\ tx\ s = \mathsf{update}_L\ tx\ s \neq \mathsf{Err}$$

Transaction updates a state non-trivially

$$\mathsf{valid}_L \in (\mathsf{Tx}_L)^* \to \mathsf{Bool}$$
$$\mathsf{valid}_L\ txs = \mathsf{computeState}_L\ txs \neq \mathsf{Err}$$

Computed state is non-trivial

$$\mathsf{validState}_L \in \mathsf{State}_L \to \mathsf{Prop}$$
$$\mathsf{validState}_L\ s = \exists\ txs \in (\mathsf{Tx}_L)^*,\ s = \mathsf{computeState}_L\ (\mathsf{initState}_L)\ txs$$

Given state is non-trivial

Fig. 2: $\mathcal{LS}$ auxiliary functions and definitions

$$(\mathsf{validState}_L\ s) \vee (\mathsf{computeState}_L\ s\ txs = \mathsf{Err})$$

Polina: Proof of $\mathsf{proveVl}_L\ s\ txs$ goes here. See Coq formalization.

Errors in ledger updates.

Polina: Important, cant compare reorderings without them.

2 Categories of Ledgers

2.1 Categories of ledgers

There are different ways to consider categories of ledgers.

Single-ledger category SLC_L**.** The first category we look at is, for a given ledger $L \in \mathcal{LS}$ the category with a single object, ValSt_L, and lists of transactions Tx_L^*. The empty list is the identity function in the category, and composition of functions is list concatenation. To calculate the result of applying a list of transactions to a valid state, $\mathsf{validUpdate}_L$ is used.

Given any transaction type Tx_{list}, we can define SLC_{list} by

$$\mathsf{State}_{list} := [\mathsf{Tx}_{list}]$$

$$\mathsf{initState}_{list} := []$$

$$\mathsf{update}\ s\ tx\ :=\ s + +tx$$

We can then define a free-forgetful adjunction for any category SLC_L where $\mathsf{Tx}_L = \mathsf{Tx}_{list}$.

Polina: specify this! is this even right?

Category of all ledgers LED**.** We may also consider the category of all ledgers, where each object is given by a ledger specification $L \in \mathcal{LS}$, ie.

$$(\mathsf{Tx}_L,\ \mathsf{State}_L,\ \mathsf{update}_L,\ \mathsf{initState}_L)$$

And the maps between ledgers $\mathsf{f} \in \mathsf{hom}\ (L,\ K),\ L,\ K \in \mathcal{LS}$ are ones that and preserve update_L and $\mathsf{initState}_L$, so

$$\mathsf{f}\ (\mathsf{update}_L\ tx_L\ s_L)\ =\ \mathsf{update}_K\ (\mathsf{f}\ tx_L)\ (\mathsf{f}\ s_L)$$

$$\mathsf{f}\ \mathsf{initState}_L\ =\ \mathsf{initState}_K$$

The initial object in this category is the ledger I, which has one valid state $initS$, and $\mathsf{Tx}_I := \{\ \}$. This is also the final object.

Polina: is this right?

3 Deterministic Ledgers

3.1 Stateful-determinism

We define a constraint in Figure 3, and say that a ledger which satisfies this constraint is *statefully deterministic*. The stateful determinism constraint requires that given any two valid ledger states $s,\ s'$, and a transaction tx valid in both s and s', applying tx to either state will result in the same changes to the state. We compute using the function

$$\Delta\ \in\ (\mathsf{Tx})^* \to \mathsf{ValSt} \to \mathsf{Diff}$$

Note that it is the changes made by a transaction that must be the same – the output states resulting from applying these changes may themselves be distinct if $s \neq s'$. We will later specify what exactly is the function that returns the ledger changes made by a sequence of transactions, and what is the type Diff representing the changes.

Polina: make a note here about only the transaction body making the changes, which is a part of the tx that, if kept constant, makes the same s' from every s regardless of the changes to the rest of the tx (and produces Err in the same cases)

What the state-determinism constraint says is that the update (or, change set) to the state of a state-deterministic ledger is specified uniquely by the transaction list txs that is being applied, and is independent of the state to which it is applied in the case that the update is valid in that state.

Arbitrary state-determinism constraint on a ledger specification

$$\begin{aligned}\mathsf{stateDetConstraint} \;=\; & \forall (s\ s' \in \mathsf{ValSt})\ (tx \in \mathsf{Tx}),\\ & \mathsf{update}\ tx\ s \neq \mathsf{Err} \neq \mathsf{update}\ tx\ s\\ & \Rightarrow\ \Delta\ [tx]\ s \;=\; \Delta\ [tx]\ s'\end{aligned}$$

Fig. 3: State-determinism

A definition of Δ making the constraint trivially true is easy to give. However, in order for Δ to carry the intended meaning, the ledger itself must admit a certain kind of structure, with which Δ indeed reduces to a function that simply discards its second argument and returns its first.

In the general case of an arbitrary L, the definition of Δ may not have this property. In fact, it should be very dependent on the specific data structures of Tx and State.

3.2 Order-determinism

The differences between any two ledgers states $s,\ s'$ can be characterized fully by a sequence of operations applied the trace of s such that the output sequence is a trace of the state s'. These operations are reordering, dropping, and insertion of transactions.

The conception of determinism, which we call *order-determinism* (OD), given Figure 4, directly addresses the state differences caused by reordering only. That is, if two valid states are generated by sequences of transactions that are permutations of each other, OD constrains them to be the same state. Therefore, any additional transaction applied to either state will also result in the same state, and the same changes, trivially satisfying the state-determinism constraint

$$(\mathsf{initState}\ txs) \;=\; (\mathsf{initState}\ txs') \;\Rightarrow\; \Delta\ [tx]\ (\mathsf{initState}\ txs) \;=\; \Delta\ [tx]\ (\mathsf{initState}\ txs')$$

Order-determinism constraint on a ledger specification

$$\begin{aligned}\mathsf{orderDetConstraint} \;=\; & \forall\ (s \in \mathsf{ValSt})\ (txs \in \mathsf{Tx}^*),\\ & txs' \in \mathsf{Permutation}\ txs,\\ & \mathsf{validUpdate}\ s\ txs \neq \mathsf{Err} \neq \mathsf{validUpdate}\ s\ txs'\\ & \Rightarrow\ \mathsf{validUpdate}\ s\ txs \;=\; \mathsf{validUpdate}\ s\ txs'\end{aligned}$$

Fig. 4: Order-determinism

Note here that, unlike in permutations groups, which are generated by all pairwise permutations, it does not suffice for the OD constraint to require equality of state for only pairwise swaps of transactions. This is because we only require the output state to be the same for two transaction orderings if both produce a valid state. This relation is not transitive. We can easily imagine a ledger where the transaction sequences $[a, b, c, d]$ and $[c, a, d, b]$ produce valid but not equal states, and all other permutations of a, b, c, d produce Err. This ledger would still satisfy a pairwise constraint, but not full OD.

Moreover, it does not make sense to drop the "both sequence orderings do not produce an error" precondition, since real ledgers do, in fact, enforce partial orders on transactions dependency (e.g., a ledger update would fail if a transaction spends money from an account before another transaction first pays into it). It is also not possible to ignore failing transactions in this definition of determinism since dropping transaction from one of the sequences means that we now constraining sequences containing different transactions (e.g. a, b, c, d and a, d) to producing the same output.

Polina: can we somehow give similar characterizations that take into account inserting and dropping transactions as well??

Polina: A key goal of this work is to give a local, as opposed to trace-based characterization of determinism, so we want to connect state-determinism with order-determinism, and find conditions under what ledgers are deterministic (without relying on analyzing all traces, if possible)

3.3 Derivative structure in single-ledger categories

A theory of changes and program derivatives is presented in [2]. We use this theory to make precise the definition of Δ, and to describe the structure characteristic of state-deterministic ledgers that admit a trivial Δ.

This work gives a framework for defining *derivatives*, where, for a given input, a derivative maps changes to that input directly to changes in the output. This approach adheres to some of the key ideas of the usual notion of differentiation of functions, while adapting them to programs. Other abstract notions of differentiation, such as presented in [3], impose additional constraints on the functions being differentiated, such as the definition of addition of functions. The theory of changes outlined in [2] draws inspiration from an abstract notion of derivation to deal with changes to data structures.

One noteworthy difference between the differentiation presented in the two papers is the type of the derivative function. A differential category derivative of a function $f : X \to Y$ has the type

$$\mathsf{D}[f] : X \times X \to Y$$

While the type of a derivative of a program $p : A \to B$ is

$$\mathsf{diff}\ p : A \to \mathsf{Diff}\ A \to \mathsf{Diff}\ B$$

The reason for this is that to define a coherent and applicable theory of changes to a data structure, we want to have a special type $\mathsf{Diff}\ A$ for each structure A that can express changes to A. Additional structure required for functional differentiation allows for a theory wherein the changes to a term of a type can be expressed by another term of that same type.

Now that we gave some justification for discussing the type of changes to a ledger state, we can explore what that looks like. The change type operator, Diff, introduced in [2], along with its behaviour and the constraints on it, is specified in Figure 5. Here we tailor the definition to a single ledger specification, so that it only describes the change type of the ledger state type for a given ledger specification L.

In the work mentioned above, all functions are total, as they are applied only to the domains (sets) in which they are valid. However, in our model, the partiality of transaction application to the ledger state object is integral to the discussion of determinism. Disallowing the application of certain updates to certain states is what allow us to guarantee deterministic updates in the valid update cases — so, we must be able to reason about the error cases.

Moreover, we want to be able to discuss what it means for a particular category of valid ledger states and maps between them to enjoy differential structure even if applying certain changes to a given state results in a failed (Err) update.

For this reason, we additionally make the change that only some changes are permitted, i.e. the ones that lead to a valid ledger state, so that applyDiff returns a error-type.

Now, we want to introduce the idea of taking and evaluating derivatives in a single-ledger category for a given ledger $L \in \mathcal{LS}$, see Figure 6.

Polina: derivativeConstraint is wrong! we want to say something like : for any interleaving of txs and ds changes, the equality must hold - how can we express this categorically?

3.4 Instantiating derivative structure

Polina: evalDer definition might also be wrong here w.r.t. Err

We give one way to instantiate the data-differentiable structure in a single-ledger category SLC in Figure 7, which we will denote DCat.

This instantiation of the Diff structure is common to all SLC categories, and trivially satisfies the constraints in Figure 5. The type representing changes to the (valid or Err) ledger state, Diff, is $(\mathsf{Tx})^*$ because lists of

Diff *Type Accessors*

$$\mathsf{Diff} \in \mathsf{Type} \quad \text{The change type of the ledger state}$$

Diff *functions*

$$\mathsf{applyDiff} \in \mathsf{State} \to \mathsf{Diff} \to \mathsf{State} \quad \text{Apply a set of changes to a given state}$$

$$\mathsf{extend} \in \mathsf{Diff} \to \mathsf{Diff} \to \mathsf{Diff} \quad \text{Compose sets of changes}$$

$$\mathsf{zero} \in \mathsf{Diff} \quad \text{No-changes term}$$

Diff *constraints*

$\mathsf{zeroChanges} \in \forall\, s, \mathsf{applyDiff}\ s\ \mathsf{zero} = s$
Applying the zero change set results in no changes

$\mathsf{applyExtend} \in \forall\, s\ txs1\ txs2,\ \mathsf{validUpdate}\ s\ txs1 \neq \mathsf{Err} \neq \mathsf{validUpdate}\ s\ txs2,$
$\mathsf{applyDiff}\ s\ (\mathsf{extend}\ txs2\ txs1) = \mathsf{applyDiff}(\mathsf{applyDiff}\ s\ txs1)\ txs2$
If both change sets are valid, composing them gives the same result as applying them in sequence

Fig. 5: Specification for a change type $D \in \mathsf{Diff}$

DCat *Type Accessors*

$$\mathsf{DerType} \in \mathsf{Type} \quad \text{The type representing derivative functions}$$

DCat *functions*

$$\mathsf{takeDer} \in (\mathsf{Tx})^* \to \mathsf{DerType} \quad \text{Take the derivative of a transaction}$$

$$\mathsf{evalDer} \in \mathsf{DerType} \to \mathsf{State} \to \mathsf{Diff} \to \mathsf{Diff} \quad \text{Calculate the diff value of a derivative at } (s,\ ds)$$

DCat *constraints*

$\mathsf{derivativeConstraint} \in \forall\, s\ ds\ txs,$
$\mathsf{validUpdate}\ (\mathsf{applyDiff}\ ds\ s)\ txs \neq \mathsf{Err},$
$\mathsf{applyDiff}\ (\mathsf{evalDer}\ (\mathsf{takeDer}\ txs)\ s\ ds)\ (\mathsf{validUpdate}\ s\ txs) \neq \mathsf{Err},$
$\mathsf{validUpdate}\ (\mathsf{applyDiff}\ ds\ s)\ txs =$
$\quad \mathsf{applyDiff}\ (\mathsf{evalDer}\ (\mathsf{takeDer}\ txs)\ s\ ds)\ (\mathsf{validUpdate}\ s\ txs)$
If both change sets are valid, composing them gives the same result as applying them in sequence

Fig. 6: Structure $DC \in \mathsf{DCat}$ for a data-differentiable category

transactions is exactly the data structure that represents changes to the state. Applying changes is therefore represented by transaction application validUpdate. Extending one change set with another is concatenation of the two transaction lists, and the empty change set is the nil list.

The DCat structure we specified, however, is not necessarily commonly admissible as a DCat instantiation in all ledgers, as it does not guarantee that the derivativeConstraint will always be satisfied. We will now discuss the relation of the specification we give, the satisfiability of the derivative constraint, and both versions of the determinism definition.

3.5 Δ and derivation structure specification in ledgers

In this section we present and justify our choice of Δ definition, as well as use of DCat structure.

The idea of derivation, as presented in [2], is that, given a function f and an input s to that function, takes a change set ds to another change set ds' such that the changes ds' are consistent with the changes of the original ds, but done after f has been applied — to the new state output by $f\ s$.

Recall that, intuitively, the definition of *state-determinism* conveys that the changes a transaction makes *are specified entirely within the transaction itself*.

Data-differentiable structure

$$\begin{array}{lcl} \mathsf{Diff} & := & (\mathsf{Tx})^* \\ \mathsf{applyDiff} & := & \mathsf{flip\ validUpdate} \\ \mathsf{extend} & := & \mathsf{flip}\ (++) \\ \mathsf{zero} & := & [] \end{array}$$

DCat *structure instantiation* DCat

$$\begin{array}{lcl} \mathsf{DerType} & := & \diamond \\ \mathsf{takeDer}\ txs & := & \diamond \\ \mathsf{evalDer}\ txs\ s\ ds & := & \begin{cases} ds & \text{if validUpdate } s\ ds \neq \mathsf{Err} \\ \mathsf{Err} & \text{otherwise} \end{cases} \end{array}$$

Fig. 7: Instantiation of data-differentiable structure in SLC

Polina: this needs more explanation about the connection between Delta and the derivative constraint and evalDer being a proj function AND figure out what to do about Err

The derivative constraint then reduces to, for all $s,\ txs,\ ds$,

$$\text{(a)}: \quad \mathsf{validUpdate}\ (\mathsf{validUpdate}\ ds\ s)\ txs\ =\ \mathsf{validUpdate}\ ds\ (\mathsf{validUpdate}\ s\ txs)$$

whenever Err is not produced by any computation. In particular,

$$\forall\ s,\ tx,\ \mathsf{evalDer}\ (\mathsf{takeDer}\ [])\ s\ [tx]\ =\ [tx]$$

3.6 Order-determinism, state-determinism, and derivation.

Polina: The proof in this section does not work without a good definition of differentiation for ledgers with Err outputs. We want to be able to prove that if a ledger L is order-deterministic, it admits data-differentiable category structure with constant derivaties, and maybe even IFF

Polina: In this section, we want to specify for what kinds of ledgers do change sets correspond exactly to the lists of transactions (i.e. the derivatives should be constant)? transactions i think always map onto change sets, but we want the change set to be represented exactly by the transaction, independent of the state they are applied to

Theorem 1. *Suppose L is a ledger, and* DCat *structure is as specified in Figure* **??**. *Then,*

$$\mathsf{orderDetConstraint}\ \Rightarrow\ \mathsf{SLC}\ \in\ \mathsf{DCat}$$

Proof. For an arbitrary $s,\ txs,\ ds$, we can instantiate the variables in the order-determinism definition in Figure 4 by

$$\text{(b)}: \quad \mathsf{orderDetConstraint}\ s\ (\mathsf{exted}\ txs\ ds)\ (\mathsf{extend}\ ds\ txs)$$

Since

$$\mathsf{extend}\ ds\ txs\ \in\ \mathsf{Permutation}\ (\mathsf{extend}\ txs\ ds)$$

Whenever (b) holds for $s,\ txs,\ ds$, it immediately follows that so does derivativeConstraint .

3.7 Examples.

Polina: Example (iv) should work if we had a proper definition of differentiation for ledgers that output Err - we want a definition that lets us reason about such cases.

In some of the following examples, we use an orthogonal but complementary context to determinism — the notion of *replay protection*.

Definition 1. *In a ledger with* replay protection*,*

$$\forall\, lstx,\ tx,\ tx \in lstx \Rightarrow \mathsf{validUpdate}\ \mathsf{initState}\ (lstx +\!+ [tx]) = \mathsf{Err}$$

This is a valuable property to have to avoid adversarial or accidental replaying of transactions.

(ii) *Account ledger with value proofs.* A ledger AP where the state consists of a map $\mathsf{State} := \mathsf{AccID} \mapsto \mathsf{Assets}$, and $\mathsf{Tx} := (\mathsf{State}, \mathsf{State})$. If for a given state and transaction s, tx, $\mathsf{fst}\ tx \subseteq s$, the update function outputs $(s \setminus \mathsf{fst}\ tx) \cup (\mathsf{snd}\ tx)$. Presumably, there will also be a preservation of value condition imposed on the asset total.
This ledger provides no replay protection — one can easily submit transactions that keep adding and removing the same key-value pairs.
Note here that a regular account-based ledger is also deterministic, but not when smart contracts or case analysis is added (eg. the update allows for transfer for a quantity q *or* any amount under q if q is not available). This value-proof feature makes it possible to enforce determinism even in such conditions of multiple control flow branches. Account-based ledgers also, unlike the UTxO model, do not offer a simple, space-efficient solution to replay protection. Ethereum does offer it, but it is somewhat convoluted, see EIP-155.

(iii) *Account ledger with value proofs and replay protection*. It is easy to add replay protection to the previous example. For example, by extending the data stored in the state,

$$\mathsf{State} := ([\mathsf{Tx}], \mathsf{AccID} \mapsto \mathsf{Assets})$$

The update function will then have an additional check that a given $tx \notin \mathsf{txList}\ s$.

(iv) *Non-deterministic differentiable ledger.* It is possible to have a ledger that is differentiable, has replay protection, but is not deterministic. Let us extend the above state with a boolean,

$$\mathsf{State} := (\mathsf{Bool}, [\mathsf{Tx}], \mathsf{AccID} \mapsto \mathsf{Assets})$$

$$\mathsf{Tx} := ((\mathsf{Bool}, \mathsf{Bool}), (\mathsf{State}, \mathsf{State}))$$

And specify $\mathsf{update}\ s = s'$ in a way that the boolean of the state s is updated in s' whenever the first boolean in the transaction matches the one in the state :

$$\mathsf{bool}\ s' := \begin{cases} \mathsf{snd}\ (\mathsf{bool}\ tx) & \text{if } \mathsf{fst}\ (\mathsf{bool}\ tx) == \mathsf{bool}\ s \\ \mathsf{bool}\ s & \text{otherwise} \end{cases}$$

We can define differentiation for such a ledger in a way that allows switching the order of application of functions ds and txs by using re-interpreting the changes txs makes when ds is applied first so that the derivativeConstraint is satisfied :

$$\mathsf{evalDer}\ txs\ s\ ds := ds +\!+ [txFlip]$$

where

$$txFlip = ((\mathsf{bool}\ (s\ txs), \mathsf{bool}\ (s\ ds\ txs)), \diamond)$$

The evalDer function cannot be defined to be a projection of the third coordinate, as this does not satisfy the constraint. It is straightforward to change the update function of this ledger such that a valid derivative function can be a simple projection, and the ledger - deterministic. The update function must produce an Err instead of allowing the state update to take place even if $\mathsf{fst}\ (\mathsf{bool}\ tx) == \mathsf{bool}\ s$ does not hold.

4 Threads

So far, we have not identified any structure in the underlying state types of ledgers. To study specific parts of a ledger, we introduce a concept we call a *thread*, and denote $T \in \mathcal{T}$. More specifically,

Definition 2. *A* thread $T \in \mathcal{T}$ *in a ledger specification* L *is a tuple of*

(i) an underlying type T,
(ii) a function

$$\mathsf{proj}_T \ \in \ \mathsf{State} \to T$$

That such that it has a right inverse inj_T *:*

$$\mathsf{proj}_T \ \circ \ \mathsf{inj}_T \ = \ \mathsf{id}_T$$

(iii) an update function $\mathsf{updateT}_T$ *is such that for all* tx, s,

$$\mathsf{updateT}_T \ tx \ s \ (\mathsf{proj}_T \ s) = \mathsf{Err} \ \Rightarrow \ \mathsf{update} \ tx \ s \ = \ \mathsf{Err}$$

And if $\mathsf{update} \ tx \ s \ \neq \ \mathsf{Err}$,

$$\mathsf{updateT}_T \ tx \ s \ (\mathsf{proj}_T \ s) = \mathsf{proj}_T \ (\mathsf{update} \ tx \ s)$$

Note that the projection function is not from a valid state, but rather from an arbitrary state.

Relation between updating the thread and ledger states.

In (iii), the implication is one direction — if the $\mathsf{updateT}_T$ does not produce an error for a given s, tx, the update function can still produce an error applied to the same transaction and state. That is, intuitively, a thread update function only cares that the transaction is attempting to update the thread data badly, and not about what it does to the rest of the ledger.

Since a Err in the output of the $\mathsf{updateT}_T$ function implies in an error in the update function applied to the same state and transaction, $\mathsf{updateT}_T$ is said to *emit constraints* on the ledger update function.

Given any type T and the proj_T function, the function updateT, which applies a transaction-state pair to a $t \in T$, can be defined in terms of the update function by specifying that, unconditionally,

$$\mathsf{updateT}_T \ tx \ s \ (\mathsf{proj}_T \ s) = \mathsf{proj}_T \ (\mathsf{update} \ tx \ s)$$

One-object categories reperesenting threads.

Recall the one-object category than the one-object category SLC_L, with lists of transactions as the functions. We can construct a related one-object category $\mathcal{T}$ for any thread T as follows :

- the object T_{val} is the image of ValSt under the proj_T function,
- the hom-set of $T_{val} \to T_{val}$ is given by the set of pairs $(s, \ txs) : (\mathsf{ValSt}_L, \ (\mathsf{Tx}_L)^*)$
- the update function used to apply a list of transactions $(s, \ txs)$ is given by folding updateT over txs at s

For $\mathsf{proj}_T = \mathsf{id}_{\mathsf{State}}$, the $\mathcal{T}$ is the same as SLC_L.

Polina: Can we do pointed pairs here (i.e. (t, s)) instead as objects, rather than just the thread?

(i) A *non-persistent* thread $T \ \cong \ T'$? is one that may take on an $\diamond$ value in some valid ledger states of a ledger specification L, i.e.

$$\exists \ s \ \in \ \mathsf{ValSt}, \ \mathsf{proj}_T \ s \ = \ \diamond$$

A thread is called *persistent* otherwise.

(ii) A *single-state* thread T is one where the projection function, for some pair of ledger states, outputs exactly one non-$\diamond$ value.

$$\forall \ s, \ s' \ \in \ \mathsf{ValSt}_L, \ \mathsf{proj}_T \ s \ \neq \ \diamond \ \neq \ \mathsf{proj}_T \ s'$$

$$\Rightarrow \ \mathsf{proj}_T \ s \ = \ \mathsf{proj}_T \ s'$$

Order-determinism for a thread T in ledger L

$$\begin{aligned}\forall\ (s\ \in\ \mathsf{ValSt})\ (txs\ \in\ \mathsf{Tx}^*),\\ &txs'\ \in\ \mathsf{Permutation}\ txs,\\ &(s\ txs)\ \neq\ \mathsf{Err}\ \neq\ (s\ txs')\\ &\Rightarrow\ \mathsf{proj}_T\ (s\ txs)\ =\ \mathsf{proj}_T\ (s\ txs')\end{aligned}$$

Fig. 8: Order-determinism for a single thread

Deterministic threads

Polina: Does this work?

Order-determinism for a single thread is defined in Figure 8.
Since a ledger is itself a thread, the following lemma follows immediately :

Lemma 1. *A ledger is OD if and only if all of its threads are.*

A (probably stronger) notion of ledger determinism expressed via threads, see Figure 9. This notion of determinism for threads implies thread order-determinism.

Determinism for a thread T in ledger L

$$\begin{aligned}\forall\ (s\ s'\ \in\ \mathsf{ValSt})\ (tx\ \in\ \mathsf{Tx}),\\ &(s\ tx)\ \neq\ \mathsf{Err}\ \neq\ (s'\ tx),\\ &\mathsf{proj}_T\ s\ =\ \mathsf{proj}_T\ s'\\ &\Rightarrow\ \mathsf{proj}_T\ (s\ tx)\ =\ \mathsf{proj}_T\ (s'\ tx)\end{aligned}$$

Fig. 9: Determinism for a single thread

4.1 Thread examples.

Binary-valued ledgers L and K.
Consider two ledgers, L and K, such that

$$\mathsf{State}_L = \mathsf{Tx}_L = \{(a,\ b)\ \mid\ a,\ b\ \in\ \{0,\ 1\}\ \} = \mathsf{State}_K = \mathsf{Tx}_K$$

and $\mathsf{initState}_L = \mathsf{initState}_K = (0,\ 0)$, with update functions as follows :

$$(L)\ \ ((a,b)\ (t_1,t_2))\ =\ (a\ \mathsf{XOR}\ t_1,\ t_2)$$

$$(K)\ \ ((a,b)\ (t_1,t_2))\ =\ \begin{cases}(a\ \mathsf{XOR}\ t_1,\ t_2) & \text{if } b = t_2\\ \mathsf{Err} & \text{otherwise}\end{cases}$$

The ledger L is not order deterministic, since applying the transaction list $[(0,1);(0,0)]$ produces a different state than $[(0,0);(0,1)]$:

$$((0,0)\ (0,1)\ (0,0)) = (0,0) \neq (0,1) = ((0,0)\ (0,1)\ (0,0))$$

Each term of the tuple is updated using only data from the current state of that term, and does not inspect the other term. We can express this in terms of threads, where $T := \{0,\ 1\}$, and $\mathsf{proj}_T := \pi_0$,

$$\mathsf{updateT}\ (a,b)\ a'\ (t_1,\ t_2)\ =\ \begin{cases}a\ \mathsf{XOR}\ t_1 & \text{if } a = a'\\ \mathsf{Err} & \text{otherwise}\end{cases}$$

The first coordinate is updated using the XOR function, which is commutative. The second coordinate also forms a thread, with $\mathsf{proj}_{T'} = \pi_1$, but it is not deterministic.

Polina: we want to say that the change in the first coordinate is always + t1 mod 2, whereas the change in the second coordinate depends on the current state (i.e. either "flip the bit" or "dont flip the bit", depending on whether b and t2 match), even though the output itself does not depend on the current state

The ledger K also has threads formed by the first and second projection function, however, both are deterministic. The π_0 thread is deterministic as in L, and the updateT function for the projection π_1 either makes no changes, or fails – instead of flipping the bit when $b \neq t_2$ (as in L) .

List-of-transactions ledger SLC_{list}.

Recall the ledger SLC_{list} for a fixed transaction type Tx, where the state is a list of processed transactions. Note that ledger itself is not OD, since reordering transactions produces distinct lists.

Every other ledger T with the same transaction type Tx and initial state i is a (possibly non-persistent) thread in SLC_{list}, where $\mathsf{validUpdate}_T\ i$ is the $\mathsf{proj}_T : [\mathsf{Tx}] \mapsto \mathsf{State}^?$ function, with $\diamond$ representing the Err state.

4.2 Characterization of ledger determinism via threads.

Polina: I would like to find any additional preconditions P1..Pn such that it is possible to say that : order-determinism for a single thread + P1...Pn implies thread determinism in the stronger sense – hopefully, no additional preconditions P1..Pn are necessary

5 Blocks, transactions, and threads

We can use the notion of threads to represent the relationship between blocks and transactions. Let us consider an illustrative example, which we will afterwards generalize to formally state the structure most blockchains are instances of. Consider the types are as specified in Figure 10.

Block example types

$$
\begin{array}{rcll}
\mathsf{Value} & = & \mathbb{Z} & \text{The ledger's currency} \\
\mathsf{Ix} & = & \mathbb{N} & \text{Index type} \\
\mathsf{TxIn} & = & (\mathsf{Tx}, \mathsf{Ix}) & \text{Transaction input} \\
\mathsf{TxOut} & = & \mathsf{Ix} \mapsto \mathsf{Value} & \text{Transaction outputs} \\
\mathsf{UTxO} & = & \mathsf{TxIn} \mapsto \mathsf{Value} & \text{UTxO} \\
\mathsf{UTxOTx} & = & (\mathsf{[TxIn]}, \mathsf{TxOut}, \mathsf{Slot}) & \text{Transaction} \\
\mathsf{Block} & = & (\mathsf{Block}^?, \mathsf{[UTxOTx]}) & \text{Block} \\
\mathsf{ChainState} & = & (\mathsf{Block}^?, \mathsf{Slot}, \mathsf{UTxO}) & \text{Chain state}
\end{array}
$$

Fig. 10: Types of a block-based ledger

In blockchains, as the name suggests, the atomic state update is done by a block rather than a transaction. Blocks contain lists of transactions, as well as other data used to perform the update. Block application updates the *chain state*, which the state updated by transactions is a part of. So, blocks play the role of the $\mathsf{Tx} := \mathsf{Block}$ type in our model, and the State refers to the chain state.

We define the initial state by $\mathsf{initState} := (\diamond, 0,)$, and the block application function is

$$\mathsf{update}\ (b, lstx)\ (b_c, slot, utxo) =$$

$$
\begin{cases}
\mathsf{Err} & \text{if } \neg\ \mathsf{blockChecks}\ (b, lstx)\ (b_c, slot, utxo) \\
((b, lstx), slot + 1, \mathsf{foldl}\ (\mathsf{updateUTxO}\ slot)\ utxo\ lstx) & \text{otherwise}
\end{cases}
$$

Performs the following checks via the blockChecks function, and returns Err either fails :

(i) $b == b_c$, which ensures that the (last) block recorded in the state is the same block to which the new block specifies that it must be attached
(ii) $lstx \neq []$, which ensures that blocks must contain at least one transaction
(ii) $\mathsf{foldl}\ (\mathsf{updateUTxO}\ slot)\ utxo\ lstx \neq \mathsf{Err}$, which verifies that the UTxO set is updated without errors by left-folding updateUTxO over the list of transactions.

The updateUTxO update is done in the usual way, by removing all entries corresponding to transaction inputs, and adding ones corresponding to the outputs. The following checks are performed, and Err is returned by $\mathsf{updateUTxO}\ slot\ (lsin,\ outs,\ lastslot)\ utxo$ if any of them fail :

(i) $lsin \neq \diamond$ ensures the transaction has at least one input

(ii) ensures the current slot is before or equal to the slot up to which the transaction is valid
(iii) $\forall i \in lsin, i \mapsto \_ \in utxo$ ensures all the transaction inputs correspond to existing UTxO entries

(iv) $\Sigma_{\_ \mapsto v \in outs} v \neq \Sigma_{v \in \{v \mid i \in lsin \wedge i \mapsto v \in utxo\}} v$ is the preservation of value condition, which ensure that the total value in the consumed UTxO entries is equal to the value in the entries produced

$$\mathsf{updateUTxO}\ slot\ (lsin,\ outs,\ lastslot)\ utxo\ =$$

$$(utxo\ setminus\ \{i \mapsto \_ \mid i \in lsin\}) \cup \{((lsin,\ outs),\ ix) \mapsto v \mid ix \mapsto v \in outs...\}$$

Note here that we use the full block and transaction data where their hashes would normally be used instead : within keys of UTxO entries, and inside blocks to specify the preceeding block. The reason for this is that properties we discuss here may not hold up without the assumption there will be no hash collisions, and additional machinery is required to reason under this assumption, while using hashes instead of preimages does not appear to provide additional insight into the properties we study here. However, a closer examination of this claim could be part of future work.

We see immediately that the update function of the UTxO set (along with the constraints on it) in this example can be specified by

$$\mathsf{updateUTxOThread}\ lstx\ (\_,\ slot,\ utxo)\ utxo' := \begin{cases} \mathsf{foldl}\ (\mathsf{updateUTxO}\ slot)\ utxo'\ lstx & \text{if } utxo == utxo' \\ \mathsf{Err} & \text{otherwise} \end{cases}$$

Since the thread update is only valid if the argument corresponding to the state of the thread (here, $utxo'$) is equal to the the projection from the chain state to the thread state (here, $utxo$ in the chain state triple).

The slot number is not used in the generation of the updated UTxO set in the case when a UTxO set, and not Err, is returned. However, in order to determine whether a thread update is permitted, the *slot* value, which is not included in the UTxO thread data, is used needed. We will show that it is in fact a constraint on the design of deterministic ledgers that this pattern of thread update be followed, ie. the updated state of a thread can depend on data external to the thread, but only in so far that the update can produce an error for certain values if that data.

We can make this example more general by remaining agnostic of the data types of the chain state. Following this example,

Definition 3. *(i) Given a type* Tx, *a* Block *is a type which admits the following accessors :*

$$\mathsf{prevBlock} \in \mathsf{Block} \to \mathsf{Block}$$

$$\mathsf{txList} \in \mathsf{Block} \to [\mathsf{Tx}]$$

(ii) A ChainState *is a type with the following accessors*

$$\mathsf{lastBlock} \in \mathsf{ChainState} \to \mathsf{Block}$$

$$\mathsf{txState} \in \mathsf{Block} \to \mathsf{TxState}$$

where TxState.

(iv) Given an initial state initState $\in$ Block *and an update procedure* update $\in$ Block $\to$ ChainState $\to$ ChainState, *the tuple* (Block, ChainState, update, initState) *is said to be a* blockchain *if*

(1) it is a ledger specification,

(2) it admits a thread TxState *with the update function defined by*

$$\mathsf{updateLedger}\ lstx\ (led,\ os)\ led' := \begin{cases} \mathsf{foldl}\ (\mathsf{updateLedger}\ os)\ led'\ lstx & \text{if } led == led' \\ \mathsf{Err} & \text{otherwise} \end{cases}$$

and

wherein (updateLedger *os*)

A (Block, ChainState, initState) *and an update procedure*

, as we will see in the next definition, represents the part of the ChainState *that applying transactions updates*

Recall here, again, that our formalism is not probabilistic, so, unlike a realistic deployed blockchain, it does not rely on hash functions having a low probability of collisions. Instead, we use pre-images where hashes would normally be used in practice (eg. as pointers to previous blocks).

we will refer to the *ledger* as the thread in the chain state that can be described as *any data updatable by a transaction*. Formally,

While the above specification is a fairly abstract idea of what a blockchain is, some distributed-consensus append-only ledgers may have a structure which deviates from the specification we present here. However, the [4] [5] [1] [6]

In the case of updating the UTxO set and the protocol parameter record, this situation can manifest itself as, eg. a specifying a UTxO update which disallows some new UTxOs to be added to it if they do not satisfy a constraint that depends on one of the protocol parameters. There might be, for instance, a "minimum value constraint" imposed by the $\mathsf{updateT}_{UTxO}\ tx$ function on any new UTxO being added by a tx such that the value contained in a UTxO entry must be greater than a value indicated in some protocol parameter in minV. The update specification for protocol paramenters may also have some constraints.

Any transaction that is valid (ie. does not produce an Err) must satisfy both the constraints on the UTxO update, as well as those for the protocol parameter updates.

6 Applications to Existing Ledgers

Polina: we need to discuss real stuff here (EUTxO, Cardano, etc)

Pointer addresses ...

If we consider individual stateful smart contracts as threads, we again see that the update of the total UTxO set (or the account-based ledger's state) require that some general rules must be satisfied, while each smart contract will have its own constraints that it may place on the transaction performing its state update.

References

1. Buterin, V.: Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper/ (2014)
2. Cai, Y., Giarrusso, P.G., Rendel, T., Ostermann, K.: A theory of changes for higher-order languages - incrementalizing λ-calculi by static differentiation (2013). https://doi.org/10.48550/ARXIV.1312.0658, https://arxiv.org/abs/1312.0658
3. Cockett, J.R.B., Cruttwell, G.S.H., Gallagher, J.D.: Differential restriction categories (2012). https://doi.org/10.48550/ARXIV.1208.4068, https://arxiv.org/abs/1208.4068

4. Corduan, J., Vinogradova, P., Güdemann, M.: A formal specification of the Cardano ledger. Tech. rep., IOHK (2019), available at https://github.com/input-output-hk/cardano-ledger-specs
5. Goodman, L.: Tezos—a self-amending crypto-ledger white paper (2014)
6. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/en/bitcoin-paper (October 2008)