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 Introduction

In the context of blockchain transaction processing and smart contract execution, determinism is usually taken to mean something like "the ability to predict locally, before subimitting a transaction, the on-chain result of processing a that transaction, including its scripts". This is an important aspect of ledger design because users care about being able to accurately predict fees they will be charged, the rewards they will receive from staking, the outcomes of smart contract executions, etc. before posting transactions. Transaction processing and smart contract execution on a variety of ledgers can be informally characterized as either more or less determnistic according to this conception of determinism, see in [**?**].

In order to formalize a single definition of determinism that can be used to characterize a variety of ledgers across different platforms with distinct design choices, we first present an abstract ledger model which captures the architectural core shared by a number of prominent platforms [4] [5] [**?**] [6]. This core can be summarized as follows : a ledger is a state transition system, with transactions (or blocks) as the the only transitions, some of which are invalid.

With this simplified functional view of ledgers, we formalize several interpretations of the informal definition of determinism, and argue that the only source of non-determinism is the impossibility of predicting the exact on-chain state their transaction will be applied to.

2 The Abstract Ledger Model

Each instance of $\mathcal{LS}$ (ledger specification type) requires the structure given in Figure 1 to be defined.

We denote by A_{Err} a type constructed from A that is identical to

$$A^{?} \;=\; A \uplus \diamond$$

We use the $A_{\mathsf{Err}} \;=\; A \uplus \mathsf{Err}$ to highlight that there was an error, rather than a valid value $\diamond$.

Proof-constructing function

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

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

Proof of $\mathsf{proveVl}_L\ s\ txs$ goes here.

$\mathcal{LS}$ Type Accessors

$$\begin{array}{rcll} \mathsf{Tx}_L & \in & \mathsf{Type} & \text{Accessor for the type of the transactions} \\ \mathsf{State}_L & \in & \mathsf{Type} & \text{Accessor for the type of the ledger state} \end{array}$$

$\mathcal{LS}$ functions

$$\begin{array}{rcll} \mathsf{update}_L & \in & \mathsf{Tx}_L \to \mathsf{State}_L \to (\mathsf{State}_L)_{\mathsf{Err}} & \text{Updates state with a given transaction} \\ \mathsf{initState}_L & \in & \mathsf{State}_L & \text{Gives the initial state of the specification} \end{array}$$

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

3 Categories of Ledgers

3.1 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.

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 map valid states to valid states, Err to Err, transactions to transactions, and preserve update_L and $\mathsf{initState}_L$, so

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

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

4 Deterministic Ledgers

Determinism in the context of ledgers and ledger state updates refers to the idea that a user has no control over the order in which submitted transactions will be applied to a ledger state, what the ledger state will be when their transactions finally get applied, or even the precise specification of the update that is being computed. In some cases, randomness, or differences in rounding or different machines, or certain kinds of data access can be a source of indeterminism.

Here we attempt to make precise the sort of determinism that assumes that the function update does not itself vary at all, as it would in the case of using different rounding strategies for computing update $tx\ s$ on different machines. Investigating this source of indeterminism will be part of future work.

Here we give two definitions of determinism and compare them.

4.1 Order-determinism

The definition in Figure 3 is given in terms of the traces leading to a particular ledger state. Specifically, it is concerned with the independence of the ledger state from the order of the transactions in the trace that has lead to any given state.

Polina: Randomness as a source of determinism can be accounted for in this definition - reorder the transactions and you get a different outcome randomly, however, things like mismatched rounding do not - this is a consensus-level issue?

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

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

4.2 Stateful-determinism

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)

The definition in Figure 4 is operational, or local. It describes the requirement that given any valid ledger state, the change in the ledger state resulting from the application of a transaction to one state is, in some sense, the same as the change resulting from the application of that transaction to a different ledger state, given that in both cases, the transaction is valid.

Order-determinism constraint on a ledger specification

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

Fig. 3: Order-determinism

Arbitrary state-determinism constraint on a ledger specification

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

Fig. 4: State-determinism

We will later specify what exactly is the function

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

This function must express *the changes that applying an update txs to the ledger state s makes*. 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.

A definition of Δ_L making the constraint trivially true is easy to give. However, in order for Δ_L to carry the intended meaning, the ledger itself must admit a certain kind of structure, with which Δ_L 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_L may not have this property. In fact, it should be very dependent on the specific data structures of Tx_L and State_L. A theory of changes and program derivatives is presented in [1]. We use this theory to make precise the definition of Δ_L, and to describe the structure characteristic of state-deterministic ledgers that admit a trivial Δ_L.

4.3 Derivative structure in single-ledger categories

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 [2], impose additional constraints on the functions being differentiated, such as the definition of addition of functions. The theory of changes outlined in [1] 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 [1], 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.

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.

Diff *Type Accessors*

Spec_D	$\in$	Type	Accessor for the type of the ledger specification
State_D	$\in$	Type	Accessor for the type of the valid ledger state, $\mathsf{State}_D = \mathsf{ValSt}_{\mathsf{Spec}_D}$
Diff_D	$\in$	Type	Accessor for the change type of the ledger state

Diff *functions*

applyDiff	$\in$	$\mathsf{State}_D \to \mathsf{Diff}_D \to \mathsf{State}_D$	Apply a set of changes to a given state
extend	$\in$	$\mathsf{Diff}_D \to \mathsf{Diff}_D \to \mathsf{Diff}_D$	Compose sets of changes
zero	$\in$	Diff_D	No-changes term

Diff *constraints*

zeroChanges $\in \forall\, s, \mathsf{applyDiff}\ s\ \mathsf{zero} = s$

Applying the zero change set results in no changes

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 $D \in$ Diff

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: Should evalDer ever produce an Err ?? txs (ds s) = (f s ds txs) (txs s)

4.4 Instantiating derivative structure

We give one way to instantiate the data-differentiable structure in a single-ledger category SLC_L in Figure 7, which we will denote $\mathsf{DCat}_{constDer\_L}$.

This instantiation of the Diff structure is common to all SLC_L categories, and trivially satisfies the constraints in Figure 5. The type representing changes to the (valid or Err) ledger state, Diff_L, is $(\mathsf{Tx}_L)^*$ because lists of transactions is exactly the data structure that represents changes to the state. Applying changes is therefore represented by transaction application $\mathsf{validUpdate}_L$. Extending one change set with another is concatenation of the two transaction lists, and the empty change set is the nil list.

The $\mathsf{DCat}_{constDer\_L}$ 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

DCat *Type Accessors*

$$\begin{aligned}
\mathsf{Spec}_{CD} &\in \mathsf{Type} && \text{Accessor for the type of the ledger specification}\\
\mathsf{State}_{CD} &\in \mathsf{Type} && \text{Accessor for the type of the valid ledger state, } \mathsf{State}_{CD} = \mathsf{ValSt}_{\mathsf{Spec}_{CD}}\\
\mathsf{Diff}_{CD} &\in \mathsf{Type} && \text{Accessor for the Diff type of the ledger state}\\
\mathsf{DerType}_{CD} &\in \mathsf{Type} && \text{Accessor for the type representing derivative functions}
\end{aligned}$$

DCat *functions*

$$\begin{aligned}
&\mathsf{takeDer} \in (\mathsf{Tx}_{\mathsf{Spec}_CD})^* \to \mathsf{DerType}_{CD} && \text{Take the derivative of a transaction}\\
&\mathsf{evalDer} \in \mathsf{DerType}_{CD} \to \mathsf{State}_{CD} \to \mathsf{Diff}_D \to \mathsf{Diff}_D && \text{Calculate the diff value of a derivative at } (s,\ ds)
\end{aligned}$$

DCat *constraints*

$$\begin{aligned}
&\mathsf{derivativeConstraint} \in \forall\ s\ ds\ txs,\\
&\quad \mathsf{validUpdate}_L\ (\mathsf{applyDiff}\ ds\ s)\ txs \neq \mathsf{Err},\\
&\quad \mathsf{applyDiff}\ (\mathsf{evalDer}\ (\mathsf{takeDer}\ txs)\ s\ ds)\ (\mathsf{validUpdate}_L\ s\ txs) \neq \mathsf{Err},\\
&\quad \mathsf{validUpdate}_L\ (\mathsf{applyDiff}\ ds\ s)\ txs =\\
&\qquad \mathsf{applyDiff}\ (\mathsf{evalDer}\ (\mathsf{takeDer}\ txs)\ s\ ds)\ (\mathsf{validUpdate}_L\ s\ txs)
\end{aligned}$$

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

Data-differentiable structure

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

DCat *structure instantiation* $\mathsf{DCat}_{constDer\_L}$

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

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

will now discuss the relation of the specification we give, the satisfiability of the derivative constraint, and both versions of the determinism definition.

4.5 Δ and derivation structure specification in ledgers

In this section we present and justify our choice of Δ_L definition, as well as use of $\mathsf{DCat}_{constDer\_L}$ structure.

The idea of derivation, as presented in [1], 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*.

Polina: this needs more explanation about the connection between Delta and the derivative constraint and evalDer being a proj function

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

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

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

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

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

We can prove that if a ledger L is order-deterministic, it admits data-differentiable category structure with constant derivaties.

Theorem 1. *Suppose L is a ledger, and* $\mathsf{DCat}_{constDer\_L}$ *structure is as specified in Figure* **??**. *Then,*

$$\mathsf{orderDetConstraint}_L\ \Rightarrow\ \mathsf{SLC}_L\ \in\ \mathsf{DCat}_{constDer\_L}$$

Proof. For an arbitrary $s,\ txs,\ ds$, we can instantiate the variables in the order-determinism definition in Figure 3 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 .

We can also prove that assuming a leger admits constant derivaties, it is state-deterministic.

Theorem 2. *Suppose for a given ledger L with* DCat *structure instantiated as in Figure 7, the following hold :*

(i)

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

(ii) $\mathsf{derivativeConstraint}_L$

(iii)

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

It follows that $\mathsf{stateDetConstraint}_L$ *holds.*

Proof. To prove this, observe that in a category SLC_L that admits $\mathsf{DCat}_{constDer\_L}$ structure as specified in Figure 7, we get that for any $s,\ s'$ and tx valid in s and s',

$$\Delta_L\ [tx]\ s\ =\ \mathsf{evalDer}\ []\ s\ [tx]\ =\ [tx]\ =\ \mathsf{evalDer}\ []\ s'\ [tx])\ \Delta_L\ [tx]\ s'$$

Which proves the result.

If constant derivatives are admissible in a ledger L, we can conclude that it is an order-deterministic ledger as well.

Theorem 3. *Suppose for a given ledger L with* DCat *structure instantiated as in Figure 7, the following hold :*

(i)

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

(ii) $\mathsf{derivativeConstraint}_L$

It follows that $\mathsf{orderDetConstraint}_L$ *holds.*

We use shorthand here

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

$$\mathsf{evalDer}\ txs1\ s\ tx\ :=\ \mathsf{evalDer}\ (\mathsf{takeDer}\ txs1)\ s\ tx$$

Proof. Suppose we have a ledger where defining $\mathsf{DCat}_{constDer\_L}$ structure as in 7 satisfies the derivativeConstraint.
We use strong induction on the length of $txs\ \in\ [\mathsf{Tx}]$ to prove that for any s,

$$\forall\ txs'\ \in\ \mathsf{Permutation}\ txs,\ (s\ txs)\ \neq\ \mathsf{Err}\ \neq\ (s\ txs')\ \Rightarrow\ (s\ txs)\ =\ (s\ txs')$$

Base case, $txs = []$ is trivial. Induction step : suppose that for some n, for any txs with $\mathsf{len}\ txs\ \leq\ n$, the above claim is holds. We want to show that it holds for $n+1$.
We can say that for any decomposition $txs\ =\ txs1\ +\!+\ txs2$, for some two lists $txs1,\ txs2$,

$$(s\ tx)\ txs\ =\ ((s\ tx)\ txs1)\ txs2$$

And, if

$$((s\ tx)\ txs1)\ txs2\ \neq\ \mathsf{Err}\ \neq\ ((s\ txs1)\ (\mathsf{evalDer}\ txs1\ s\ tx))\ txs2$$

We get that, by the derivativeConstraint,

$$...\ =\ ((s\ txs1)\ (\mathsf{evalDer}\ txs1\ s\ tx))\ txs2\ =\ ((s\ txs1)\ tx)\ txs2$$

Now, $(s\ txs1)$ is independent of the order of $txs1$ by the strong induction hypothesis, as is $((s\ txs1)\ tx)\ txs2$ on the order of $txs2$ (or $txs1$), and $(s\ txs)$ on the order of txs.
We can conclude that inserting tx anywhere in the list txs gives the same state $((s\ txs1)\ tx)\ txs2$ or Err. That is, any permutation of the list of length $n+1$, $txs1\ +\!+\ [tx]\ +\!+\ txs2$, will result in the same state, or an Err as well. This proves the result for permutations of lists of arbitrary length.

So, if we choose to define the ledger changes Δ as in 3, a ledger is both state- and order-deterministic exactly when it admits the following definition of a derivative of a single transaction (such that it satisfies the derivativeConstraint) :

$$\mathsf{evalDer}\ ls\ s\ [tx]\ =\ [tx]$$

Polina: is it true that evalDer indep. of function =¿ it is indep. of state too, and vice-versa?

4.7 Examples.

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*,*

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

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

(i) *UTxO ledger*. A basic UTxO ledger $UTxO$ with the constraint in its update_{UTxO} function requiring that $\mathsf{nonEmpty}\ (\mathsf{inputs}\ tx)$ provides replay protection since each output can only be consumed once, preventing any other (or the same) transaction consuming that output again, and therefore from being reapplied. A UTxO ledger is also deterministic. Note that we assume here that there are no hash collissions. A hash collission possibility undermines both determinism and replay protection.

(ii) *Account ledger with value proofs.* A ledger AP where the state consists of a map $\mathsf{State}_{AP} := \mathsf{AccID} \mapsto \mathsf{Assets}$, and $\mathsf{Tx}_{AP} := (\mathsf{State}_{AP}, \mathsf{State}_{AP})$. 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,

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

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,

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

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

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 the APB 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.

5 Update-deterministic ledgers

Determinism, both state- and order- (SD and OD, for short), specifies what it means for a ledger to behave predictably under an update caused by a transaction in the face of a lack of information about what the state might be at any given time, or what transactions happen to have been processed before the transaction doing the update in question. This approach to specifying determinism focuses on allowing only certain transactions to be processed.

Here we propose another angle from which to investigate determinism, which instead focuses on the properties of the update function itself, as well as the state and transaction structure.

A ledger L is *without inverses* whenever

$$\forall\, lts\ s,\ (s\ lts)\ =\ s\ \Leftrightarrow\ lts\ =\ []$$

If a ledger has replay protection, it is without inverses. Replay protection precludes inverses, since having inverses implies that applying a list of transactions is possible more than once,

$$(s\ lts\ lts)\ =\ (s\ lts)\ =\ s$$

A ledger L is *update-deterministic* (or, UD) whenever it has *consistent transaction application*, defined in Figure 8

$$\begin{aligned}\mathsf{updateDetConstraint}_L\ =\ & \forall\, s\ tx\ tx',\\ & (\mathsf{Err}\ \neq\ (s\ tx)\ =\ (s\ tx')\ \neq\ \mathsf{Err}\ \Rightarrow\\ & \forall\, s',\ (s'\ tx)\ \neq\ \mathsf{Err}\ \neq\ (s'\ tx')\\ & \quad \Rightarrow\ (s'\ tx)\ =\ (s'\ tx'))\end{aligned}$$

Fig. 8: Consistent transaction application

Theorem 4. *If a ledger L is order-deterministic, it is update-deterministic.*

Proof.

Polina: Prove this! I think it's true

UD is a weaker condition. It captures the idea that while the transaction order cannot be changed, there are no additional transaction application computations done using the order (see example (i) and (ii) below). In a sense, some order-dependent internal indexing takes place when a transaction is applied. If this indexing is not later inspected by other computations on-chain, there is no issue. However, building applications than do inspect this indexing data off-chain is bad.

This definition is also representative of the idea that state-determinism attempted to capture (but ended up defaulting to OD), that a list of transactions makes the same changes to the ledger no matter what state they are applied to.

Polina: Give example of pointer addresses on Cardano. The process of building the map of pointer addresses as transactions update the delegations map could itself be UD (it is just an internal indexing process), but we notice that in fact, reward calculations are done using this pointer address map, so UD is broken.

Examples :

(i) $\mathsf{Tx}\ =\ \mathbb{N}$, $\mathsf{State}\ =\ [\mathbb{N}]$, and the update function appends the number in a tx to the list. This ledger is update-deterministic.

(ii) $\mathsf{Tx}\ =\ \mathbb{N}$, $\mathsf{State}\ =\ n\ \in\ \mathbb{N}^{*}$, where the update function includes the number in the tx in the multiset ($...^{*}$ is the notation for multiset). This ledger is OD

(iii) $\mathsf{Tx}\ =\ \mathsf{Bool}$, $\mathsf{State}\ =\ \mathsf{Bool}$, where the tx flips the bit in the state if they match. This ledger is not deterministic.

(iv) $\mathsf{Tx}\ =\ \mathsf{Bool}$, $\mathsf{State}\ =\ \mathsf{Bool}$, where the update function is XOR. This ledger is OD (and so, probably UD), but has inverses.

5.1 UD and derivatives

Recall that in Section 4.6 we discussed the relation between derivation in the ledger and OD. We can also make a claims about UD and derivation.

Claim : A derivative of a UD ledger must be independent of the state, but can still depend on the set of changes ds applied before or after the transaction list txs, of which we are calculating the derivative. Recall the constraint on valid derivatives (using shorthand notation) :

$$\mathsf{derivativeConstraint} \in \forall\, s\ ds\ txs,$$

$$((s\ ds)\ txs) \neq \mathsf{Err} \neq ((s\ txs)\ (txs\ s\ ds)') \Rightarrow ((s\ ds)\ txs) = ((s\ txs)\ (txs\ s\ ds)')$$

The $\mathsf{updateDetConstraint}_L$ then tells us that

$$\forall\, s_1\ s_2,\ (s\ (txs\ +\!+\ (txs\ s_1\ ds)')) = (s\ (txs\ +\!+\ (txs\ s_2\ ds)'))$$

And therefore the derivative $(txs\ s\ ds)'$ must not be dependent on s.

Polina: can we say anything about this in the other direction? ie. derivative is independed of s implies ... ?

5.2 Implementation of an OD update function

Polina: clean this up!!

This section we consider only ledgers that can be represented as partial products, ie. products $P \times Q$ that have functions into them from some type R

$$\langle\ \mathsf{p},\ \mathsf{q}\ \rangle\ :\ R \to P \times Q$$

return Err whenever *either* p *or* q returns Err, as well as possibly in other cases.

For example, the UTxO set is (maybe?) isomorphic to a (probably finite) product of Maybe-outputs, where each element of the product object looks like $(o_1\ ,\ ....,\ o_k)$, where each spot in the tuple corresponds to an input.

We conjecture that ledgers that have a partial product decomposition such that the update function can be expressed as a *projection* are OD, so that

$$\mathsf{update} = \pi_{TxS}$$

Note that this does not mean that this is simply a projection onto the second coordinate. It rather means that all data needed to construct the new state is stored in either the old state or transaction, and no computation outside of projection is allowed.

Projections commute (except when Err occurs).

If we have transaction

$$tx = (inputs = (txid, 1),\ (txid, 2);\ outputs = (txout_1,\ txout_2))$$

6 Threads

We can identify a particular kind of a lens-like structure that arises from a ledger specification, which we call a *thread*, and denote $T \in \mathcal{T}$. Here we will abuse notation to denote both the thread and the type which the lens-like structure is viewing as T.

Polina: This structure is more like a ledger state with with a pointer to the object of interest, eg. State + "Script Purpose"

Given a ledger specification $L = \mathsf{Spec}_T$, a type $T = \mathsf{Ttype}_T^?$, and any surjective function $\mathsf{proj}_T \in \mathsf{ValSt}_L \to T$, we can consider the following categories :

Projections

$$\begin{aligned} \pi_{TxS} &: (\mathsf{Tx},\ \mathsf{State}) \to \mathsf{State} \\ &: \text{projects data that will be in the state after applying } \mathsf{update} \end{aligned}$$

UTxO example

$txid_1$	$txid_2$	$txid_3$	...all possible $txid$s...	$txid_k$
v_1	$\diamond$	v_3	...values corresponding to above $txid$s...	$\diamond$

View of UTxO by tx

$txid_1$	$txid_2\ (=\ (txid, 1))$	$txid_3\ =(\ (txin_1))$	...all possible $txid$s...	$txid_k\ (=\ (txid, 2))$
v_1	$txout_1$	v_3	...values corresponding to above $txid$s...	$txout_2$

Fig. 9: Transaction and state composition of OD ledgers

(i) State_L, The one-object category representing all (valid) ledger states in a given ledger specification. The maps are given by transactions $(\mathsf{Tx}_L)^*$, so that they are all of the form $\mathsf{computeState}_L\ txs \in \mathsf{ValSt}_L \to \mathsf{ValSt}_L$.

(ii) T, The one-object category where each $v \in T$ is an image of some $s \in S$. and the hom-set of $T \to T$ is given by pairs of type $(\mathsf{ValSt}_L,\ (\mathsf{Tx}_L)^*)$. The update function used to apply such a pair $(s,\ txs)$ as a function to T is given by folding $\mathsf{updateT}$ (see Figure 10) over txs at s. That is, homs on T are specified by both the transactions being applied to the ledger, as well as the ledger state itself.

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

We can trivially specify a lens-like relation between these categories using the proj_T and $\mathsf{updateT}_T$ functions.

$$\mathsf{Lens}\ \mathsf{ValSt}_L\ T$$

$\mathcal{T}$ Type Accessors

Spec_T	$\in$	$\mathcal{LS}$	Accessor for the type of the underlying ledger specification
Ttype_T	$\in$	Type	Accessor for the type of the thread

$\mathcal{T}$ functions

$\mathsf{proj}_T \in \mathsf{ValSt}_{\mathsf{Spec\_}T} \to \mathsf{Ttype}_T^?$ Projects to the thread type from the ledger state

$\mathcal{T}$ constraints

$$\mathsf{updateT}_T\ \in\ \mathsf{Tx}_{\mathsf{Spec\_}T} \to \mathsf{ValSt}_{\mathsf{Spec\_}T} \to \mathsf{Ttype}_T^? \to (\mathsf{Ttype}_T^?)_{\mathsf{Err}}$$

$$\mathsf{updateT}_T\ tx\ s\ t\ :=\ \begin{cases} \mathsf{Err} & \text{if } \mathsf{updateErr}_{\mathsf{Spec\_}T}\ tx\ s = \mathsf{Err} \vee \mathsf{proj}_T\ s\ \neq\ t \\ \mathsf{proj}_T\ (\mathsf{updateErr}_{\mathsf{Spec\_}T}\ tx\ s) & \text{otherwise} \end{cases}$$

Fig. 10: Thread $\mathcal{T}$ structure

We can come at the idea of defining threads from another direction. In practice, the idea of threads is meant to represent some permanent or temporary part of the data structure that is State_L, and the evolution thereof as a result of applying incoming transactions to the state. We can try to, therefore, represent a part of the state using product (in the categorical sense) decomposition.

Suppose $T \in \mathcal{T}$ (and $T = \mathsf{Ttype}_T$). We might contemplate whether $L = \mathsf{Spec}_T$ can be represented by $T \times P$ for some P with a corresponding surjective function $\mathsf{proj}_P \in L \to P$. This sort of approach yields a trivial way of defining a standard factorization of update functions $L \to L$, for each $tx \in \mathsf{Tx}_L$, we can define this factorization by

$$\mathsf{updateErr}_L\ tx\ s = \langle\ \mathsf{updateT}_T\ tx\ s\ (\mathsf{proj}_T\ s),\ \mathsf{updateT}_T\ tx\ s\ (\mathsf{proj}_P\ s)\ \rangle$$

However, this trivial observation does not give us much insight into ledger structure. What we want is to, instead of starting with a complete ledger specification, build one up from its thread components. That is, to have a way to define the updates of each thread individually, and then build up the total ledger state update from the collection of individual update functions.

For example, if a UTxO set T is a thread in a ledger that, say, also has another thread which is a record P containing some protocol parameters, we want to specify how to update the protocol parameters, and how to update the UTxO set, using separate update functions.

That is, the update functions $\mathsf{updateT}_T$ and $\mathsf{updateT}_P$ are now defined not via the $\mathsf{updateErr}_L$ function, but the other way around. We have the update specifications for T and P, but now we want to put them together into $\mathsf{updateErr}_L$. It quickly becomes evident that considering

$$\mathsf{totalProd}_{T\times P}\ s\ =\ \langle\ \mathsf{updateT}_T\ tx\ s\ (\mathsf{proj}_T\ s),\ \mathsf{updateT}_P\ tx\ s\ (\mathsf{proj}_P\ s)\rangle$$

to have the usual properties of a product of functions cannot work, as an update failure in either T or P should result in Err, not distinct values $(\_,\ \mathsf{Err})$ and $(\mathsf{Err},\ \_)$.

The reason for this is that the collection of states we are investigating properties of is not $T \times P$, but $\mathsf{ValSt}_{T\times P}$. This type includes only the valid ledger states and Err, and so is not a product type.

If we define the domain, for any thread X of L and update corresponding to any tx, by

$$(\mathsf{dom}_X\ (tx,\ s))\ t\ =\ \begin{cases} t & \text{if } \mathsf{updateT}_X\ tx\ s\ t\ =\ t' \text{ for some } t' \neq \mathsf{Err} \\ \mathsf{Err} & \text{otherwise} \end{cases}$$

We can use a parial products construction (see [3]) to specify the behaviour that we want. Defining the partial product to have the required properties in the cited work,

$$\mathsf{partialProd}_{T\times P}\ tx\ \in\ \mathsf{ValSt}_{T\times P}\ \to\ \mathsf{ValSt}_{T\times P}$$

$$\mathsf{partialProd}_{T\times P}\ tx\ s\ =\ \begin{cases} \langle\ \mathsf{updateT}_T\ tx\ s\ (\mathsf{proj}_T\ s), \mathsf{updateT}_P\ tx\ s\ (\mathsf{proj}_P\ s)\ \rangle \\ \quad \text{if } \mathsf{updateT}_T\ tx\ s\ (\mathsf{proj}_T\ s)\ \neq\ \mathsf{Err}\ \neq\ \mathsf{updateT}_P\ tx\ s\ (\mathsf{proj}_P\ s) \\ \mathsf{Err} \\ \quad \text{otherwise} \end{cases}$$

so that

$$s\ \in\ \mathsf{dom}\ (\mathsf{partialProd}_{T\times P}\ tx)\ \Leftrightarrow$$

$$\mathsf{proj}_T\ s\ \in\ \mathsf{dom}\ (\mathsf{updateT}_T\ tx\ s)\ \wedge$$

$$\mathsf{proj}_P\ s\ \in\ \mathsf{dom}\ (\mathsf{updateT}_P\ tx\ s)$$

which is the constraint which a construction must satisfy to constitute a partial product.

With this standardized, product-like way to combine two update functions of threads T and P, we can focus explicitly on defining these updates individually. That is to say, while each of the update functions (for T and for P) may itself depend not only on the state of the thread and the transaction tx being applied as an update, but on the state of the ledger itself, we have a way to specify the resulting update for all of L using only these two updates. We can give an alternate definition of a thread,

Theorem 5. *A* thread $T \in \mathcal{T}$ *in a ledger spec* $L = \mathsf{Spec}_T$ *is a tuple of*

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

$$\mathsf{proj}_T\ \in\ \mathsf{ValSt}_{\mathsf{Spec\_}T} \to \mathsf{Ttype}_T^?$$

(iii) an update function $\mathsf{updateT'}_T$ *such that (see Figure 10 for the definition of* $\mathsf{updateT}_T$*) If for some* tx, s, t, *it holds that* $\mathsf{updateT}_T\ tx\ s\ t \neq \mathsf{Err}$, *then*

$$\mathsf{updateT'}_T\ tx\ s\ t = \mathsf{updateT}_T\ tx\ s\ t$$

We get ?? for some type P,

$$\mathsf{ValSt}_L\ =\ \mathsf{partialProd}_{T\times P}$$

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.

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.

(i) A *temporary* thread T is one that may not appear in some valid ledger states of a ledger specification $\mathsf{Spec}\ T = L$, i.e.

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

A permanent thread is one that is not temporary.

(ii) A *stateful* thread T is one where the projection function, for some pair of ledger states, gives two different non-$\diamond$ outputs.

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

(iii) We say that the domain of a thread T with the update function $\mathsf{updateT}_T$ *constrains* the update of the ledger state when for some tx,

$$\mathsf{dom}_T\ (tx,\ s)\ \neq\ \mathsf{id}_T$$

(iv) Threads T and U in a ledger specification $L\ =\ \mathsf{Spec}_T\ =\ \mathsf{Spec}_U$ are said to be *independent* when any change by a transaction tx to the state of both threads can be instead implemented by two lists of transactions $txs1, txs2$, where

- applying the list $txs1$ to the ledger state updates one of the threads to have the same state as after applying tx to the ledger while keeping the state of the other thread constant, and
- applying the list $txs2$ to the resulting ledger state updates the state of the other thread to be the same as after applying tx.

$$\begin{gathered}\forall\ tx\ s\ \in\ \mathsf{ValSt}_L,\ \ \mathsf{validTx}_L\ tx\ s\ \Rightarrow\ \exists\ txs1\ txs2,\\ \mathsf{proj}_T\ (\mathsf{updateErr}_L\ tx\ s)\ =\ \mathsf{proj}_T\ (\mathsf{computeState}_L\ s\ (txs1 + +txs2))\\ \wedge\ \mathsf{proj}_U\ (\mathsf{updateErr}_L\ tx\ s)\ =\ \mathsf{proj}_U\ (\mathsf{computeState}_L\ s\ (txs1 + +txs2))\\ \wedge\\ \mathsf{proj}_T\ (\mathsf{computeState}_L\ s\ txs1)\ =\ \mathsf{proj}_T\ (\mathsf{updateErr}_L\ tx\ s)\\ \wedge\ \mathsf{proj}_U\ (\mathsf{computeState}_L\ s\ txs1)\ =\ \mathsf{proj}_U\ s\end{gathered}$$

(iv) Threads T and U in a ledger specification $L\ =\ \mathsf{Spec}\ T\ =\ \mathsf{Spec}\ U$ are said to be *separateable by message passing* when there exist threads T' and U' such that

$$\forall\ tx\ s\ \in\ \mathsf{ValSt}_L,\ \ \mathsf{validTx}_L\ tx\ s$$

.....

Polina: when can two dependent threads be made separable?

6.1 Deterministic threads

Polina: Does this work?

See Figures 11, 12. UD defined in 5.

Order-determinism for a thread T in ledger L

$$\forall\,(s \in \mathsf{ValSt}_L)\,(txs \in \mathsf{Tx}_L^*),\quad txs' \in \mathsf{Permutation}\ txs,\ \mathsf{validUpdate}_L\ s\ txs \neq \mathsf{Err} \neq \mathsf{validUpdate}_L\ s\ txs' \Rightarrow \mathsf{proj}_T\ (\mathsf{validUpdate}_L\ s\ txs) = \mathsf{proj}_T\ (\mathsf{validUpdate}_L\ s\ txs')$$

Fig. 11: Order-determinism for a single thread

Update-determinism for a thread T in ledger L

Fig. 12: Consistent transaction application (UD) for a single thread T

References

1. 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
2. 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
3. Cockett, J., Lack, S.: Restriction categories i: categories of partial maps. Theoretical Computer Science **270**(1), 223–259 (2002). https://doi.org/https://doi.org/10.1016/S0304-3975(00)00382-0, https://www.sciencedirect.com/science/article/pii/S0304397500003820
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)