Gnosis Mercury Documentation

Release 0.2.2

Gnosis

Apr 18, 2019

Contents:

1 Prediction Market Contracts **3**

2 License **5**

2.1 Security and Liability 5

3 Indices and tables **23**

GNOSIS

CHAPTER 1

Prediction Market Contracts

Experimental smart contracts for prediction markets.

→ Online Documentation

CHAPTER 2

License

All smart contracts are released under the LGPL 3.0 license.

2.1 Security and Liability

All contracts are **WITHOUT ANY WARRANTY**; *without even* the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

2.1.1 Developer Guide

> **Warning:** This document refers to an experimental prediction market framework under active development. Some things may change. You may also be interested in the predecessor of this work: the original Gnosis prediction market contracts.

Prerequisites

Usage of this smart contract system requires some proficiency in Solidity.

Additionally, this guide will assume a Truffle based setup. Client-side code samples will be written in JavaScript assuming the presence of a web3.js instance and various TruffleContract wrappers.

The current state of this smart contract system may be found on Github.

Installation

Via NPM

This developmental framework may be installed from Github through NPM by running the following:

```
npm i '@gnosis.pm/hg-contracts'
```

Preparing a Condition

Before predictive assets can exist in the system, a *condition* must be prepared. A condition is a question to be answered in the future by a specific oracle in a particular manner. The following function may be used to prepare a condition:

function **prepareCondition** (*address oracle*, *bytes32 questionId*, *uint outcomeSlotCount*) *external*

This function prepares a condition by initializing a payout vector associated with the condition.

Parameters

- **oracle** – The account assigned to report the result for the prepared condition.
- **questionId** – An identifier for the question to be answered by the oracle.
- **outcomeSlotCount** – The number of outcome slots which should be used for this condition. Must not exceed 256.

Note: It is up to the consumer of the contract to interpret the question ID correctly. For example, a client may interpret the question ID as an IPFS hash which can be used to retrieve a document specifying the question more fully. The meaning of the question ID is left up to clients.

If the function succeeds, the following event will be emitted, signifying the preparation of a condition:

event **ConditionPreparation** (*bytes32 indexed conditionId*, *address indexed oracle*, *bytes32 indexed questionId*, *uint outcomeSlotCount*)

Emitted upon the successful preparation of a condition.

Parameters

- **conditionId** – The condition's ID. This ID may be derived from the other three parameters via `keccak256(abi.encodePacked(oracle, questionId, outcomeSlotCount))`.
- **oracle** – The account assigned to report the result for the prepared condition.
- **questionId** – An identifier for the question to be answered by the oracle.
- **outcomeSlotCount** – The number of outcome slots which should be used for this condition. Must not exceed 256.

Note: The condition ID is different from the question ID, and their distinction is important.

The successful preparation of a condition also initializes the following state variable:

mapping (bytes32 => uint[]) *public* **payoutNumerators**

Mapping key is an condition ID. Value represents numerators of the payout vector associated with the condition. This array is initialized with a length equal to the outcome slot count.

To determine if, given a condition's ID, a condition has been prepared, or to find out a condition's outcome slot count, use the following accessor:

function **getOutcomeSlotCount** (*bytes32 conditionId*) *external*

Gets the outcome slot count of a condition.

Parameters

- **conditionId** – ID of the condition.

Return Number of outcome slots associated with a condition, or zero if condition has not been prepared yet.

The resultant payout vector of a condition contains a predetermined number of *outcome slots*. The entries of this vector are reported by the oracle, and their values sum up to one. This payout vector may be interpreted as the oracle's answer to the question posed in the condition.

A Categorical Example

Let's consider a question where only one out of multiple choices may be chosen:

> Who out of the following will be chosen?
>
> - Alice
> - Bob
> - Carol

Through some commonly agreed upon mechanism, the detailed description for this question becomes strongly associated with a 32 byte question ID: `0xabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc1234`

Let's also suppose we trust the oracle with address `0x1337aBcdef1337abCdEf1337ABcDeF1337AbcDeF` to deliver the answer for this question.

To prepare this condition, the following code gets run:

```
await predictionMarketSystem.prepareCondition(
    '0x1337aBcdef1337abCdEf1337ABcDeF1337AbcDeF',
    '0xabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc1234',
    3
)
```

The condition ID may also be determined from the parameters via:

```
web3.utils.soliditySha3({
    t: 'address',
    v: '0x1337aBcdef1337abCdEf1337ABcDeF1337AbcDeF'
}, {
    t: 'bytes32',
    v: '0xabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc1234'
}, {
    t: 'uint',
    v: 3
})
```

This yields a condition ID of `0x67eb23e8932765c1d7a094838c928476df8c50d1d3898f278ef1fb2a62afab63`.

Later, if the oracle `0x1337aBcdef1337abCdEf1337ABcDeF1337AbcDeF` makes a report that the payout vector for the condition is `[0, 1, 0]`, the oracle essentially states that Bob was chosen, as the outcome slot associated with Bob would receive all of the payout.

A Scalar Example

Let us now consider a question where the answer may lie in a range:

> What will the score be? [0, 1000]

Let's say the question ID for this question is `0x777def777def777def777def777def777def777def777def777def777de` and that we trust the oracle `0xCafEBAbECAFEbAbEcaFEbabECAfebAbEcAFEBaBe` to deliver the results for this question.

To prepare this condition, the following code gets run:

```
await predictionMarketSystem.prepareCondition(
    '0xCafEBAbECAFEbAbEcaFEbabECAfebAbEcAFEBaBe',
    '0x777def777def777def777def777def777def777def777def777def777def7890',
    2
)
```

The condition ID for this condition can be calculated as `0x3bdb7de3d0860745c0cac9c1dcc8e0d9cb7d33e6a899c2c29834`

In this case, the condition was created with two slots: one which represents the low end of the range (0) and another which represents the high end (1000). The slots' reported payout values should indicate how close the answer was to these endpoints. For example, if the oracle `0xCafEBAbECAFEbAbEcaFEbabECAfebAbEcAFEBaBe` makes a report that the payout vector is `[9/10, 1/10]`, then the oracle essentially states that the score was 100, as the slot corresponding to the low end is worth nine times what the slot corresponding with the high end is worth, meaning the score should be nine times closer to 0 than it is close to 1000. Likewise, if the payout vector is reported to be `[0, 1]`, then the oracle is saying that the score was *at least* 1000.

Outcome Collections

The main concept for understanding the mechanics of this system is that of a *position*. We will build to this concept from conditions and outcome slots, and then demonstrate the use of this concept.

However, before we can talk about positions, we first have to talk about *outcome collections*, which may be defined like so:

> A nonempty proper subset of a condition's outcome slots which represents the sum total of all the contained slots' payout values.

Categorical Example Featuring Alice, Bob, and Carol

We'll denote the outcome slots for Alice, Bob, and Carol as `A`, `B`, and `C` respectively.

A valid outcome collection may be `(A|B)`. In this example, this outcome collection represents the eventuality in which either Alice or Bob is chosen. Note that for a categorical condition, the payout vector which the oracle reports will eventually contain a one in exactly one of the three slots, so the sum of the values in Alice's and Bob's slots is one precisely when either Alice or Bob is chosen, and zero otherwise.

`(C)` by itself is also a valid outcome collection, and this simply represents the case where Carol is chosen.

`()` is an invalid outcome collection, as it is empty. Empty outcome collections do not make sense, as they would essentially represent no eventuality and have no value no matter what happens.

Conversely, `(A|B|C)` is also an invalid outcome collection, as it is not a proper subset. Outcome collections consisting of all the outcome slots for a condition also do not make sense, as they would simply represent any eventuality, and should be equivalent to whatever was used to collateralize these outcome collections.

Finally, outcome slots from different conditions (e.g. `(A|X)`) cannot be composed in a single outcome collection.

Index Set Representation and Identifier Derivation

A outcome collection may be represented by an a condition and an *index set*. This is a 256 bit array which denotes which outcome slots are present in a outcome collection. For example, the value `3 == 0b011` corresponds to the outcome collection `(A|B)`, whereas the value `4 == 0b100` corresponds to `(C)`. Note that the indices start at the lowest bit in a `uint`.

A outcome collection may be identified with a 32 byte value called a *collection identifier*. In order to calculate the collection ID for `(A|B)`, simply hash the condition ID and the index set:

```
web3.utils.soliditySha3({
    // See section "A Categorical Example" for derivation of this condition ID
    t: 'bytes32',
    v: '0x67eb23e8932765c1d7a094838c928476df8c50d1d3898f278ef1fb2a62afab63'
}, {
    t: 'uint',
    v: 0b011 // Binary Number literals supported in newer versions of JavaScript
})
```

This results in a collection ID of `0x52ff54f0f5616e34a2d4f56fb68ab4cc636bf0d92111de74d1ec99040a8da118`.

We may also combine collection IDs for outcome collections for different conditions by adding their values modulo 2^256 (equivalently, by adding their values and then taking the lowest 256 bits).

To illustrate, let's denote the slots for range ends 0 and 1000 from our scalar condition example as `LO` and `HI`. We can find the collection ID for `(LO)` to be `0xd79c1d3f71f6c9d998353ba2a848e596f0c6c1a9f6fa633f2c9ec65aaa097cdc`.

The combined collection ID for `(A|B)&(LO)` can be calculated via:

```
'0x' + BigInt.asUintN(256,
    0x52ff54f0f5616e34a2d4f56fb68ab4cc636bf0d92111de74d1ec99040a8da118n +
    0xd79c1d3f71f6c9d998353ba2a848e596f0c6c1a9f6fa633f2c9ec65aaa097cdcn
).toString(16)
```

Note: BigInt is used here for the calculation, though BN.js or BigNumber.js should both also suffice.

This calculation yields the value `0x2a9b72306758380e3b0a31125ed39a635432b283180c41b3fe8b5f5eb4971df4`.

Defining Positions

In order to define a position, we first need to designate a collateral token. This token must be an ERC20 token which exists on the same chain as the PredictionMarketSystem instance.

Then we need at least one condition with a outcome collection, though a position may refer to multiple conditions each with an associated outcome collection. Positions become valuable precisely when *all* of its constituent outcome collections are valuable. More explicitly, the value of a position is a *product* of the values of those outcome collections composing the position.

With these ingredients, position identifiers can also be calculated by hashing the address of the collateral token and the combined collection ID of all the outcome collections in the position. We say positions are *deeper* if they contain more conditions and outcome collections, and *shallower* if they contain less.

As an example, let's suppose that there is an ERC20 token called DollaCoin which exists at the address `0xD011ad011ad011AD011ad011Ad011Ad011Ad011A`, and it is used as collateral for some positions. We will denote this token with `$`.

We may calculate the position ID for the position `$:(A|B)` via:

```
web3.utils.soliditySha3({
    t: 'address',
    v: '0xD011ad011ad011AD011ad011Ad011Ad011Ad011A'
}, {
    t: 'bytes32',
    v: '0x52ff54f0f5616e34a2d4f56fb68ab4cc636bf0d92111de74d1ec99040a8da118'
})
```

The ID for `$:(A|B)` turns out to be `0x6147e75d1048cea497aeee64d1a4777e286764ded497e545e88efc165c9fc4f0`

Similarly, the ID for `$:(LO)` can be found to be `0xfdad82d898904026ae6c01a5800c0a8ee9ada7e7862f9bb6428b6f81e` and `$:(A|B)&(LO)` has an ID of `0xcc77e750b61d29e158aa3193faa3673b2686ba9f6a16f51b5cdbea2a4f694be0`.

All the positions backed by DollaCoin which depend on the example categorical condition and the example scalar condition form a DAG (directed acyclic graph):

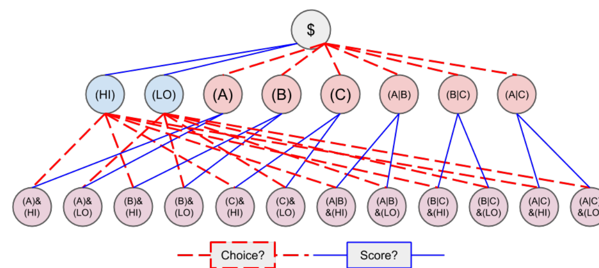


Fig. 1: Graph of all positions backed by `$` which are contingent on either or both of the example conditions.

Splitting and Merging Positions

Once conditions have been prepared, stake in positions contingent on these conditions may be obtained. Furthermore, this stake must be backed by collateral held by the system. In order to ensure this is the case, stake in shallow positions may only be created directly by sending collateral to the system for the system to hold, and stake in deeper positions may only be created by destroying stake in shallower positions. Any of these is referred to as *splitting a position*, and is done through the following function:

function **splitPosition**(*IERC20 collateralToken*, *bytes32 parentCollectionId*, *bytes32 conditionId*, *uint[ ]external calldata partition*, *uint amount*)

This function splits a position. If splitting from the collateral, this contract will attempt to transfer *amount* collateral from the message sender to itself. Otherwise, this contract will burn *amount* stake held by the message sender in the position being split. Regardless, if successful, *amount* stake will be minted in the split target positions. If any of the transfers, mints, or burns fail, the transaction will revert. The transaction will also revert if the given partition is trivial, invalid, or refers to more slots than the condition is prepared with.

Parameters

- **collateralToken** – The address of the positions' backing collateral token.
- **parentCollectionId** – The ID of the outcome collections common to the position being split and the split target positions. May be null, in which only the collateral is shared.
- **conditionId** – The ID of the condition to split on.
- **partition** – An array of disjoint index sets representing a nontrivial partition of the outcome slots of the given condition.
- **amount** – The amount of collateral or stake to split.

If this transaction does not revert, the following event will be emitted:

event **PositionSplit** (*address indexed stakeholder*, *IERC20 collateralToken*, *bytes32 indexed parentCollectionId*, *bytes32 indexed conditionId*, *uint[] partition*, *uint amount*)
Emitted when a position is successfully split.

To decipher this function, let's consider what would be considered a valid split, and what would be invalid:

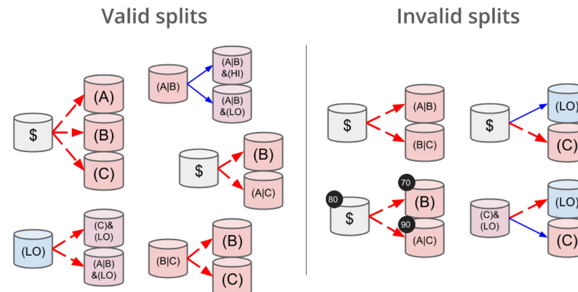


Fig. 2: Details for some of these scenarios will follow

Basic Splits

Collateral `$` can be split into outcome tokens in positions `$:(A)`, `$:(B)`, and `$:(C)`. To do so, use the following code:

```
const amount = 1e18 // could be any amount

// user must allow predictionMarketSystem to
// spend amount of DollaCoin, e.g. through
// await dollaCoin.approve(predictionMarketSystem.address, amount)

await predictionMarketSystem.splitPosition(
    // This is just DollaCoin's address
    '0xD011ad011ad011AD011ad011Ad011Ad011Ad011A',
    // For splitting from collateral, pass bytes32(0)
    '0x00',
    // "Choice" condition ID:
    // see A Categorical Example for derivation
    '0x67eb23e8932765c1d7a094838c928476df8c50d1d3898f278ef1fb2a62afab63',
    // Each element of this partition is an index set:
    // see Outcome Collections for explanation
    [0b001, 0b010, 0b100],
    // Amount of collateral token to submit for holding
    // in exchange for minting the same amount of
    // outcome token in each of the target positions
    amount,
)
```

The effect of this transaction is to transfer `amount` DollaCoin from the message sender to the `predictionMarketSystem` to hold, and to mint `amount` of outcome token for the following positions:

Symbol	Position ID
`$:(A)`	`0x8c12fa3bb72c9c455acd4d6034989ec0ce9188afd7c89c8c42d064ed7fe5a9d8`
`$:(B)`	`0x21aec03d8dfd8b5f0a2750718fe491e439f3625816e383b66a05cabd56624b4c`
`$:(C)`	`0x8085f7c500098412ff2fc701a74174527e7b39a2b923cd0bca6ad2d5f7fa348d`

Note: The previous example, where collateral was split into shallow positions containing collections with one slot each, is similar to `Event.buyAllOutcomes` from v1.

The set of `(A)`, `(B)`, and `(C)` is not the only nontrivial partition of outcome slots for the example categorical condition. For example, the set `(B)` (with index set `0b010`) and `(A|C)` (with index set `0b101`) also partitions these outcome slots, and consequently, splitting from `$` to `$:(B)` and `$:(A|C)` is also valid and can be done with the following code:

```
await predictionMarketSystem.splitPosition(
    '0xD011ad011ad011AD011ad011Ad011Ad011Ad011A',
    '0x00',
    '0x67eb23e8932765c1d7a094838c928476df8c50d1d3898f278ef1fb2a62afab63',
    // This partition differs from the previous example
    [0b010, 0b101],
    amount,
)
```

This transaction also transfers `amount` DollaCoin from the message sender to the `predictionMarketSystem` to hold, but it mints `amount` of outcome token for the following positions instead:

Symbol	Position ID
`$:(B)`	`0x21aec03d8dfd8b5f0a2750718fe491e439f3625816e383b66a05cabd56624b4c`
`$:(A|C)`	`0xb33b3d0035913315b76e85842f682920f78b32c43c7175768c4c67e3f31e6413`

Warning: If non-disjoint index sets are supplied to `splitPosition`, the transaction will revert.

Partitions must be valid partitions. For example, you can't split `$` to `$:(A|B)` and `$:(B|C)` because `(A|B)` (`0b011`) and `(B|C)` (`0b110`) share outcome slot `B` (`0b010`).

Splits to Deeper Positions

It's also possible to split from a position, burning outcome tokens in that position in order to acquire outcome tokens in deeper positions. For example, you can split `$:(A|B)` to target `$:(A|B)&(LO)` and `$:(A|B)&(HI)`:

```
await predictionMarketSystem.splitPosition(
    // Note that we're still supplying the same collateral token
    // even though we're going two levels deep.
    '0xD011ad011ad011AD011ad011Ad011Ad011Ad011A',
    // Here, instead of just supplying 32 zero bytes, we supply
    // the collection ID for (A|B).
    // This is NOT the position ID for $:(A|B)!
    '0x52ff54f0f5616e34a2d4f56fb68ab4cc636bf0d92111de74d1ec99040a8da118',
    // This is the condition ID for the example scalar condition
    '0x3bdb7de3d0860745c0cac9c1dcc8e0d9cb7d33e6a899c2c298343ccedf1d66cf',
    // This is the only partition that makes sense
    // for conditions with only two outcome slots
    [0b01, 0b10],
    amount,
)
```

This transaction burns `amount` of outcome token in position `$:(A|B)` (position ID `0x6147e75d1048cea497aeee64d1a4777e286764ded497e545e88efc165c9fc4f0`) in order to

mint `amount` of outcome token in the following positions:

Symbol	Position ID
`$:(A|B)&(LO)`	`0xcc77e750b61d29e158aa3193faa3673b2686ba9f6a16f51b5cdbea2a4f694be0`
`$:(A|B)&(HI)`	`0xbacf3ddf0474d567cd254ea0674fe52ab20a3e2ebca00ec71a846f3c48c5de9d`

Splits on Partial Partitions

Supplying a partition which does not cover the set of all outcome slots for a condition, but instead some outcome collection, is also possible. For example, it is possible to split `$:(B|C)` (position ID `0x5d06cd85e2ff915efab0e7881432b1c93b3e543c5538d952591197b3893f5ce3`) to `$:(B)` and `$:(C)`:

```
await predictionMarketSystem.splitPosition(
    '0xD011ad011ad011AD011ad011Ad011Ad011Ad011A',
    // Note that we also supply zeroes here, as the only aspect shared
    // between $:(B|C), $:(B) and $:(C) is the collateral token
    '0x00',
    '0x67eb23e8932765c1d7a094838c928476df8c50d1d3898f278ef1fb2a62afab63',
    // This partition does not cover the first outcome slot
    [0b010, 0b100],
    amount,
)
```

Merging Positions

Merging positions does precisely the opposite of what splitting a position does. It burns outcome tokens in the deeper positions to either mint outcome tokens in a shallower position or send collateral to the message sender:

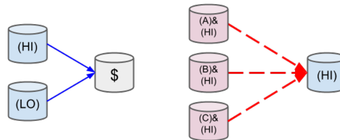


Fig. 3: Splitting positions, except with the arrows turned around.

To merge positions, use the following function:

function **mergePositions**(*IERC20 collateralToken*, *bytes32 parentCollectionId*, *bytes32 conditionId*,*external uint[] calldata partition*, *uint amount*)

If successful, the function will emit this event:

event **PositionsMerge**(*address indexed stakeholder*, *IERC20 collateralToken*, *bytes32 indexed parentCollectionId*, *bytes32 indexed conditionId*, *uint[] partition*, *uint amount*)

Emitted when positions are successfully merged.

Note: This generalizes `sellAllOutcomes` from v1 like `splitPosition` generalizes `buyAllOutcomes`.

Querying and Transferring Stake

Outcome tokens in positions are not ERC20 tokens, but rather part of an ERC1155 multitoken.

In addition to a holder address, each token is indexed by an ID in this standard. In particular, position IDs are used to index outcome tokens. This is reflected in the balance querying function:

function **balanceOf** (*address owner*, *uint256 positionId*) *external*

To transfer outcome tokens, the following functions may be used, as per ERC1155:

function **safeTransferFrom** (*address from*, *address to*, *uint256 positionId*, *uint256 value*, *bytes data*) *external*

function **safeBatchTransferFrom** (*address from*, *address to*, *uint256[] positionIds*, *uint256[] values*, *bytes data*) *external*

function **safeMulticastTransferFrom** (*address[] from*, *address[] to*, *uint256[] positionIds*, *uint256[] values*, *bytes data*) *external*

Approving an operator account to transfer outcome tokens on your behalf may also be done via:

function **setApprovalForAll** (*address operator*, *bool approved*) *external*

Querying the status of approval can be done with:

function **isApprovedForAll** (*address owner*, *address operator*) *external*

Redeeming Positions

Before this is possible, the payout vector must be set by the oracle:

function **receiveResult** (*bytes32 questionId*, *bytes calldata result*) *external*

Called by the oracle for reporting results of conditions. Will set the payout vector for the condition with the ID `keccak256(abi.encodePacked(oracle, questionId, outcomeSlotCount))`, where oracle is the message sender, questionId is one of the parameters of this function, and outcomeSlotCount is derived from result, which is the result of serializing 32-byte EVM words representing payoutNumerators for each outcome slot of the condition.

Parameters

- **questionId** – The question ID the oracle is answering for
- **result** – The oracle's answer

This will emit the following event:

event **ConditionResolution** (*bytes32 indexed conditionId*, *address indexed oracle*, *bytes32 indexed questionId*, *uint outcomeSlotCount*, *uint[] payoutNumerators*)

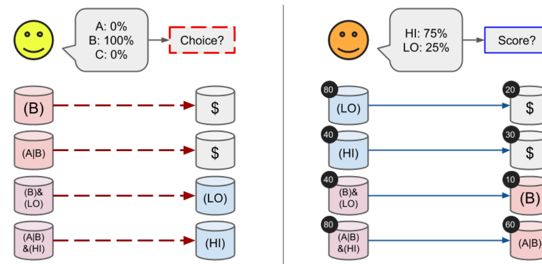
Then positions containing this condition can be redeemed via:

function **redeemPositions** (*IERC20 collateralToken*, *bytes32 parentCollectionId*, *bytes32 conditionId*, *uint[] calldata indexSets*) *external*

This will trigger the following event:

event **PayoutRedemption** (*address indexed redeemer*, *IERC20 indexed collateralToken*, *bytes32 indexed parentCollectionId*, *bytes32 conditionId*, *uint[] indexSets*, *uint payout*)

Also look at this chart:



2.1.2 Changes from Version 1.x

Documentation for the first version of the prediction market contract framework can be found here, or via the menu on the sidebar.

Modular vs Monolithic

Modular smart contracts composed v1 of the framework. Various factories for the smart contracts which were integral parts of the system had canonical locations on chain, but were not strictly necessary for using the framework.

In contrast, this version collects all core functionality and storage requirements of prediction markets into a single monolithic instance. Although the property of modularity typically is desired in software projects, on the chain it also comes with the overhead of message passing between contract instances and duplicate code deployments. Consequently, a monolithic design can produce significant gas savings for users of the system.

Furthermore, this aggregation of prediction market data allows for advanced on-chain support of conditional markets.

From Events to Conditions

In v1, the contracts used the term *event* to refer to future events to be resolved by an oracle. These events were represented by contract instances initialized with a nonstandard oracle contract instance and a collateral token address.

Each oracle contract instance was expected to report on a single specific future event. Once the report is ready, the event contract must be notified to pull the result from the oracle.

Furthermore, since event contract instances are tied to single collateral token addresses, a new event contract instance would have to be deployed if support for a different type of collateral token is desired.

The nomenclature clashed with Solidity's use of `event`, which indicated the definition of a class of EVM logs consumers of contracts could expect to be emitted during transactions with the contract.

In v2, future events to be resolved by an oracle are referred to as *conditions*. These conditions are defined by an oracle account, a question ID, and a outcome slot count.

Any account, whether externally owned or a contract, may act as an oracle. The oracle simply reports the result of the specific question about a future event, which should be retrievable from the question ID, to the monolith.

Because there's no reference to any collateral token in the specification of a condition, the report for the condition applies for all concievable collateral tokens.

Conditional Market Support

In v1, conditional markets may be set up by creating events which uses the outcome tokens of another event as collateral tokens, creating a deeper set of outcome tokens.

For example, let's suppose there are two oracles which report on the following questions:

1. Which **choice** out of Alice, Bob, and Carol will be made?
2. Will the **score** be high or low?

There are two ways to create outcome tokens backed by a collateral token denoted as `$`, where the value of these outcome tokens depend on *both* of the reports of these oracles on their respective assigned questions:

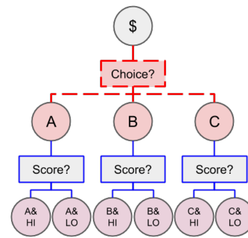


Fig. 4: **Choice**, then **Score**

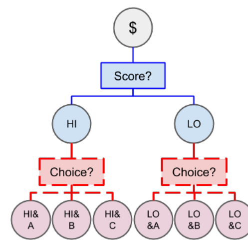


Fig. 5: **Score**, then **Choice**

Although the outcome tokens in the second layer may have the same value in collateral under the same conditions, they are in reality separate entities:

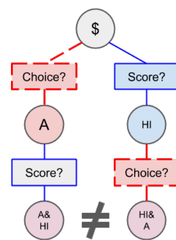


Fig. 6: These tokens should be the same, but aren't.

In v2, because all core prediction market data is held in a single contract, and because conditions are not tied to a specific collateral token, this discrepancy may be addressed. The situation in v2 looks more like this:

It can be seen that the outcome tokens from v1 which were different are now the same in v2:

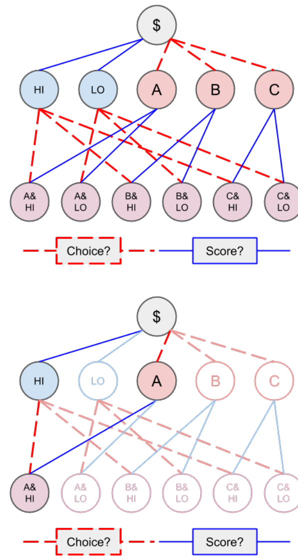


Fig. 7: Contrast this with v1.

Peripheral Contracts

Markets and MarketMakers from v1 are considered not core functionality, and will be moved into dependant packages. The mechanism for CentralizedOracles can now be implemented by regular externally owned accounts. Similarly, MajorityOracles can now be implemented by your preferred multisignature wallets. The FutarchyOracle mechanism is planned to be ported to the new contract system in a separate dependent package. There are no plans to continue support of the Campaign mechanism.

2.1.3 Contributing

The source for the contracts can be found on Github.

To set up for contributing, first install requirements with NPM:

```
npm install
```

Then, set up Git hooks to ensure commits pass the linters:

```
npm run setup-githooks
```

Tip: Many of the following commands simply wrap corresponding Truffle commands.

Testing and Linting

The test suite may be run using:

```
npm test
```

In order to run a subset of test cases which match a regular expression, the `TEST_GREP` environment variable may be used:

```
TEST_GREP='obtainable conditionIds' npm test
```

The JS test files may be linted via:

```
npm run lint
```

Contracts may also be linted via:

```
npm run lint-contracts
```

Development commands

To compile all the contracts, obtaining build artifacts containing each containing their respective contract's ABI and bytecode, use the following command:

```
npm run compile
```

Running the migrations, deploying the contracts onto a chain and recording the contract's deployed location in the build artifact can also be done:

```
npm run migrate
```

Dropping into a Truffle develop session can be done via:

```
npm run develop
```

Network Information

Showing the deployed addresses of all contracts on all networks can be done via:

```
npm run networks
```

Extra command line options for the underlying Truffle command can be passed down through NPM by preceding the options list with `--`. For example, in order to purge the build artifacts of any unnamed network information, you can run:

```
npm run networks -- --clean
```

To take network info from `networks.json` and inject it into the build artifacts, you can run:

```
npm run injectnetinfo
```

If you instead wish to extract all network information from the build artifacts into `networks.json`, run:

```
npm run extractnetinfo
```

Warning: Extracting network info will overwrite `networks.json`.

Building the Documentation

(Will install Sphinx and Solidity Domain for Sphinx):

```
cd docs
pip install -r requirements.txt
make html
```

Contributors

- Stefan George (Georgi87)
- Martin Koeppelmann (koeppelmann)
- Alan Lu (cag)
- Roland Kofler (rolandkofler)
- Collin Chin (collinc97)
- Christopher Gewecke (cgewecke)
- Anton V Shtylman (InfiniteStyles)
- Billy Rennekamp (okwme)
- Denis Granha (denisgranha)
- Alex Beregszaszi (axic)

2.1.4 Glossary

Condition

A question that a specific oracle reports on with a preset number of outcome slots. Analogous to events from PM contracts v1.

For example, a condition with a categorical outcome, that is, one of N outcomes, may have N outcome slots, where the resolution of the condition sets one of the outcome slots to receive the full payout.

Another example: a condition with a scalar outcome, that is an outcome X in some range [A, B], may have two outcome slots which correspond to the ends of the range A and B. Both slots are set to receive a proportion of the payout according to how close the outcome X is to A or B.

Identified by keccak256(oracle . questionId . outcomeSlotCount)

Outcome Slot Defines the redemption rate of Outcome Tokens. Outcome Tokens convert to a proportion of collateral depending on the outcome resolution of a set of conditions.

Outcome Slots can either be unresolved (when the condition hasn't been reported on) or resolved (after condition resolution).

Index Set A bit array that represents a subset of Outcome Slots in one condition.

Example: Condition1 has Outcome Slots: A,B,C. It would have 7 possible indexSets: A, B, C, A|B, A|C, B|C, A|B,C

Partition A specific way to separate the subsets of the Outcome Slots in a condition, using a combination of indexSets.

Oracle The account which can report the results on the condition.

Outcome Resolution The process in which an oracle reports results for the Outcome Slots in a condition, setting the Outcome Slot value for each of the condition's Outcome Slots.

Position

A set of conditions, along with a non-empty proper subset of Outcome Slots for each condition (represents a combination of one or many Outcome Slots from multiple conditions) represented as a DAG (Directed Acyclic Graph) and tied to a specific stakeholder, Collateral Token, and amount of Outcome Tokens.

Representing a specific stakeholders stake in a certain condition(s) Outcome Slots as an ERC1155 token.

A position is made up of:

1. Stakeholder Collection Identifier Condition(s) IndexSet(s) CollateralToken Outcome Tokens

Identified by the hash of a H(Collateral Token, Collection Identifier)

Collection Identifier An identifier used by positions to target Condition(s) and indexSet(s).

Rather than target individual Conditions and IndexSets. The Condition Identifier can identify a DAG (Directed Acyclic Graph) of dependant Condition(s) and indexSet(s).

It is the abstract structure that identifies what Conditions and IndexSets, a position is representing, along with their heirarchy. Without being tied to any specific stakeholder or Collateral Token.

If the parentCollectionId is equal to 0, then it is a Root Position.

Identified by a sum(parentCollectionIdentifier, hash(ConditionIdentifier . indexSet)

Collateral Token An ERC20 token used to create stake in positions.

Outcome Tokens For a given Collection Identifier, a stakeholder may express a belief in what that Collection Identifier of Outcome Slots represents by using a collateral token to create a position from the Collection Identifier and holding Outcome Tokens in that slot.

For non-root positions, redemption will convert Outcome Tokens into stake in a shallower position. For root positions, redemption will convert Outcome Tokens into Collateral Tokens.

Position Depth The number of conditions a position is based off of. Terminology is chosen because positions form a DAG which is very tree-like. Shallow positions have few conditions, and deep positions have many conditions.

Root Position A position based off of only a single condition. Pays out depending on the outcome of the condition. Pays out directly to the Collateral Token

Non-Root Position A position based off of multiple conditions. Pays out depending on all of the outcomes of the multiple conditions. Pays out to a shallower Position.

Atomic Position A position is atomic with respect to a set of conditions if it is contingent on all of the conditions in that set. Pays out to a shallower Position.

Splitting a Position Stakeholders can split a position on an optional collection identifier and a condition.

For Root Positions, a collection identifier is not given (instead it is 0), and the stakeholder transfers an input amount of collateral tokens in order to get an equal amount of outcome tokens in each of the condition's outcome slots.

For Non-Root Positions, a parent Collection Identifier is provided, and the stakeholder transfers an input amount of Outcome Tokens from the Position corresponding to the parent Collection Identifier down to a set of new Non-Root Position(s).

Results in outcome tokens being transferred from the position being split to the positions resulting from the split.

Merging a Position Basically the opposite of splitting a position. Stakeholders can merge a position on an optional Outcome Slot and a Collection Identifier for non-root positions.

For Root Positions, if an Outcome Slot is not given, the stakeholder inputs an equal amount of Outcome Tokens in each of the condition's root Outcome Slots to receive an equal amount of Collateral Tokens.

For Non-Root Positions, a parent Collection Identifier is provided, and the stakeholder transfers an input amount of Outcome Tokens from all the Outcome Slots input in the partition[] either up to a position identified by the parent Collection Identifier or merged into a single Position.

Results in outcome tokens being transferred from the positions being merged to the position resulting from the merge.

Redeeming Positions Redeems (1 - all Index Sets) of Positions that are predicated on a single Condition and collection identifier.

Resulting in either more Outcome Tokens in a shallower position, or a conversion of Outcome Tokens into the Collateral Token, depending on whether it's a Root Position or Non-Root Position.

To redeem a position, you need:

1. The Collateral Token that position is tied to. It's parent positions Collection Identifier (if it has one), otherwise it would be a Root Position, and you would input 0 to receive back Collateral Tokens. The condition you want to redeem. The Index Sets[] you want to redeem.

This will redeem all of the Index Sets[] slots listed in the given condition, for only positions with a parent position that has a Collection Idenfier equal to parentCollectionId.

CHAPTER 3

Indices and tables

- genindex

Index

B

`balanceOf` (*function*), 14

C

`ConditionPreparation` (*event*), 6
`ConditionResolution` (*event*), 14

G

`getOutcomeSlotCount` (*function*), 6

I

`isApprovedForAll` (*function*), 14

M

`mergePositions` (*function*), 13

P

`payoutNumerators` (*statevar*), 6
`PayoutRedemption` (*event*), 14
`PositionsMerge` (*event*), 13
`PositionSplit` (*event*), 11
`prepareCondition` (*function*), 6

R

`receiveResult` (*function*), 14
`redeemPositions` (*function*), 14

S

`safeBatchTransferFrom` (*function*), 14
`safeMulticastTransferFrom` (*function*), 14
`safeTransferFrom` (*function*), 14
`setApprovalForAll` (*function*), 14
`splitPosition` (*function*), 10