Design and Formalization of Blockchain Oracles

Giselle Reis[1], Bruno Woltzenlogel Paleo[2] and Mohammad Shaheer[1]

[1] Carnegie Mellon University in Qatar
[2] Djed Alliance

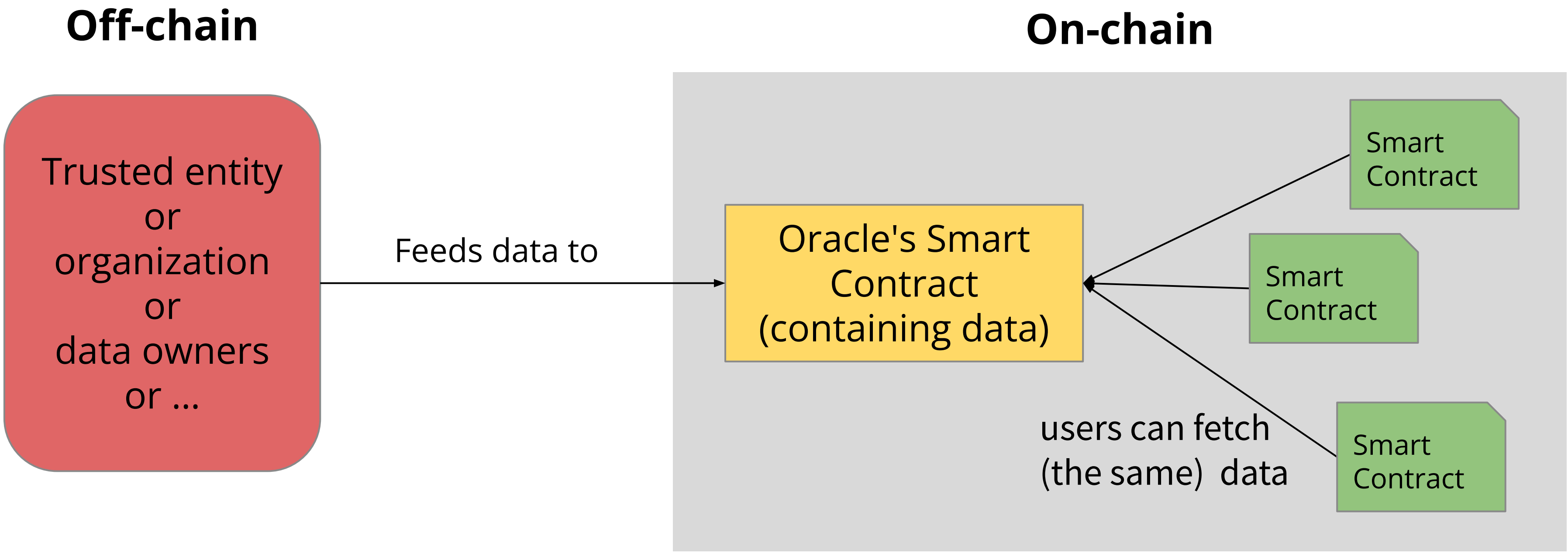
Preliminaries

- **Blockchain**
  - A distributed database for computations.
- **Smart Contracts**
  - Programs uploaded to the blockchain that can be executed by users through *transactions* (function calls).
  - Written in the blockchain language (e.g. Solidity), executed using the blockchain virtual machine.
  - Stateful.

Oracles

- Smart contracts can only interact with on-chain data.
- Need external data to be useful.
  - E.g. transaction may depend on the value of a commodity or result of a game).
- Hence, **oracles**
  - Framework for bringing external data onto the chain.
  - Ensures consistency for all on-chain users (all access the same data).
  - Needs to be *trusted*.
  - Smart-contract (to be queried and store data)

Oracles – overview



Oracles – considerations

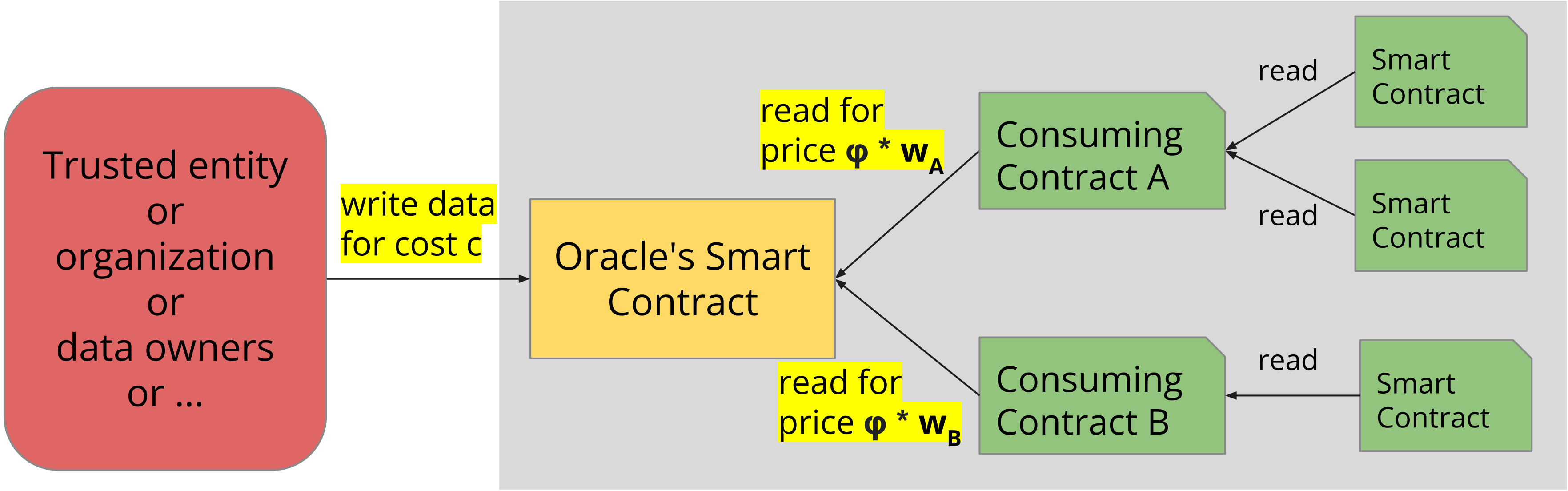
- There is no free lunch and there is no free oracle.
  - Data may have a price
  - Running functions on-chain has an associated cost (gas fee)
  - Operational costs (machines, people, electricity, etc.)
- Current solutions:
  - Ad-hoc, no agreed shared principles or guarantees
  - Documentation inexistent or not detailed enough (white papers)
- Our solution: **formalized and verified oracle protocol!**

Step 1: define goals and protocol

Oracle – goals

- The oracle protocol should fulfil the following goals:
  - **G1: Sustainable:** Oracle costs are covered by the fee charged.
  - **G2: Fairness:**
    a) Consumers should pay only once for the same data point.
    b) Consumers should be charged proportionally to the benefit obtained.

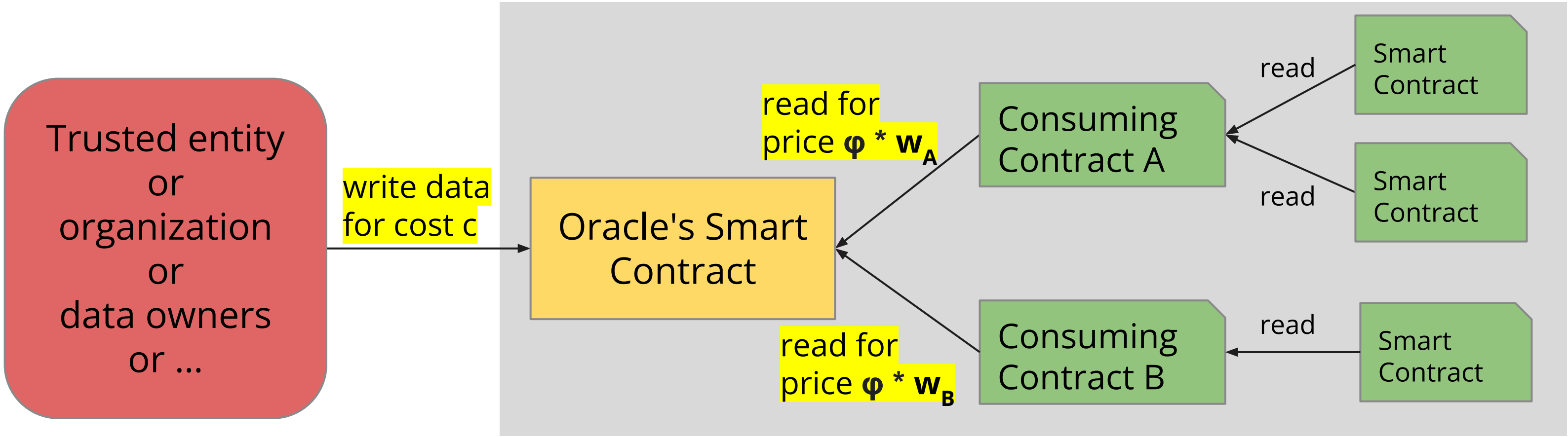
Protocol



- Reads and writes are calls to the oracle's smart contract functions.
- φ is called the **base fee**
- Oracle's smart contract stores C and sets φ, w_A and w_B
- Consumers pay with existing **credit**.

Satisfying Goals

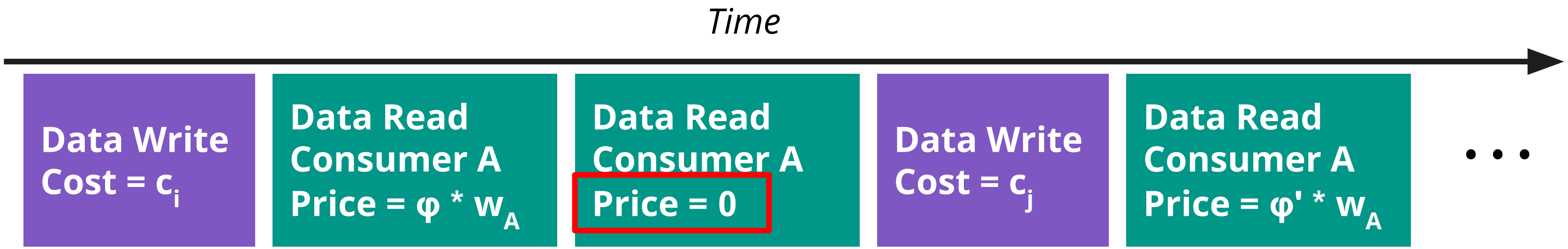
G2b. *Consumers should be charged proportionally to the benefit obtained.*



- Consumers' price is determined by their weight, stored in the oracle's contract.
- Ideally, $w_A > w_B$
- Currently not formally verified.

Satisfying Goals

G2a. *Consumers should pay only once for the same data point.*



- Depends on the implementation of data read function.
  - Function requirement, formally checked.
- Requires bookkeeping of the times of latest writes and reads per consumer.

Satisfying Goals

G1. *Oracle costs are covered by the fee charged.*

- Unpredictable!
- Base fee **φ** is adjusted to cushion big profits or losses.
- Depends on the implementation of the function that adjusts the base fee.
- Requires bookkeeping of accumulated cost and revenue.
- Formal verification is hard because of unpredicatbility…
  - Proved that cost = revenue only under very strong assumptions.

Step 2: implement and prove

Formalization

- Oracle smart contract was implemented in **Solidity** (object oriented PL) https://github.com/DjedAlliance/Oracle-Solidity/tree/cmu-qatar
- Oracle smart contract + state were implemented in **Coq** (inherently functional) https://github.com/DjedAlliance/Oracle-FormalMethods
- **Goals:**
  - Coq and Solidity code should be as close as possible (adequacy)
  - Properties should be easy enough to prove (depends on representation)

Formalization – Contracts as Records

```
contract Oracle is MultiOwnable {
    // # Parameters
    string public description;

    // # State Variables
    uint private data;
    mapping(address => uint) private credit;

    event DataWritten(uint data, uint cost);

    function writeData(uint _data, uint cost) external onlyOwner
    {
        writes += 1;
        data = _data;
        totalCost += cost;
        latestCost = cost;
        time += 1;
        latestWrite = time;
        emit DataWritten(data, cost);
    }
}
```

```
Record State :=
{
    oracleState      : OracleState;
    oracleParameters : OracleParameters;
    trace            : Trace
}.

Record OracleParameters :=
{
    description   : string;
    ...
}.

Record OracleState :=
{
    data            : float;
    totalCredit     : nat;
    ...
}.

Inductive Event : Type :=
    | DataWritten (newData : float) (newCost : nat) (caller : address)
    ...

Definition write_data (state : State) (...) : State :=

Definition Trace : Type := list (Event).
```

Formalization – Contracts as Records

```
contract Oracle is MultiOwnable {
    // # Parameters
    string public description;

    // # State Variables
    uint private data;
    mapping(address => uint) private credit;

    event DataWritten(uint data, uint cost);

    function writeData(uint _data, uint cost) external onlyOwner
    {
        writes += 1;
        data = _data;
        totalCost += cost;
        latestCost = cost;
        time += 1;
        latestWrite = time;
        emit DataWritten(data, cost);
    }
}
```

```
Record State :=
{
    oracleState      : OracleState;
    oracleParameters : OracleParameters;
    trace            : Trace
}.

Record OracleParameters :=
{
    description   : string;
    ...
}.

Record OracleState :=
{
    data           : float;
    totalCredit    : nat;
    ...
}.

Inductive Event : Type :=
    | DataWritten (newData : float) (newCost : nat) (caller : address)
    ...

Definition write_data (state : State) (...) : State :=

Definition Trace : Type := list (Event).
```

Formalization – Contracts as Records

```
contract Oracle is MultiOwnable {
    // # Parameters
    string public description;

    // # State Variables
    uint private data;
    mapping(address => uint) private credit;

    event DataWritten(uint data, uint cost);

    function writeData(uint _data, uint cost) external onlyOwner
    {
        writes += 1;
        data = _data;
        totalCost += cost;
        latestCost = cost;
        time += 1;
        latestWrite = time;
        emit DataWritten(data, cost);
    }
}
```

```
Record State :=
{
    oracleState      : OracleState;
    oracleParameters : OracleParameters;
    trace            : Trace
}.

Record OracleParameters :=
{
    description   : string;
    ...
}.

Record OracleState :=
{
    data          : float;
    totalCredit   : nat;
    ...
}.

Inductive Event : Type :=
    | DataWritten (newData : float) (newCost : nat) (caller : address)
    ...

Definition write_data (state : State) (...) : State :=

Definition Trace : Type := list (Event).
```

Formalization – Contracts as Records

```
contract Oracle is MultiOwnable {
    // # Parameters
    string public description;

    // # State Variables
    uint private data;
    mapping(address => uint) private credit;

    event DataWritten(uint data, uint cost);

    function writeData(uint _data, uint cost) external onlyOwner
    {
        writes += 1;
        data = _data;
        totalCost += cost;
        latestCost = cost;
        time += 1;
        latestWrite = time;
        emit DataWritten(data, cost);
    }
}
```

```
Record State :=
{
    oracleState      : OracleState;
    oracleParameters : OracleParameters;
    trace            : Trace
}.

Record OracleParameters :=
{
    description   : string;
    ...
}.

Record OracleState :=
{
    data          : float;
    totalCredit   : nat;
    ...
}.

Inductive Event : Type :=
    | DataWritten (newData : float) (newCost : nat) (caller : address)
    ...

Definition write_data (state : State) (...) : State :=

Definition Trace : Type := list (Event).
```

Formalization – Contracts as Records

```
contract Oracle is MultiOwnable {
    // # Parameters
    string public description;

    // # State Variables
    uint private data;
    mapping(address => uint) private credit;

    event DataWritten(uint data, uint cost);

    function writeData(uint _data, uint cost) external onlyOwner
    {
        writes += 1;
        data = _data;
        totalCost += cost;
        latestCost = cost;
        time += 1;
        latestWrite = time;
        emit DataWritten(data, cost);
    }
}
```

```
Record State :=
{
    oracleState      : OracleState;
    oracleParameters : OracleParameters;
    trace            : Trace
}.

Record OracleParameters :=
{
    description   : string;
    ...
}.

Record OracleState :=
{
    data          : float;
    totalCredit   : nat;
    ...
}.

Inductive Event : Type :=
    | DataWritten (newData : float) (newCost : nat) (caller : address)
    ...

Definition write_data (state : State) (...) : State :=

Definition Trace : Type := list (Event).
```

Formalization – Contracts as Records

```
contract Oracle is MultiOwnable {
    // # Parameters
    string public description;

    // # State Variables
    uint private data;
    mapping(address => uint) private credit;

    event DataWritten(uint data, uint cost);

    function writeData(uint _data, uint cost) external onlyOwner
    {
        writes += 1;
        data = _data;
        totalCost += cost;
        latestCost = cost;
        time += 1;
        latestWrite = time;
        emit DataWritten(data, cost);
    }
}
```

```
Record State :=
{
    oracleState      : OracleState;
    oracleParameters : OracleParameters;
    trace            : Trace
}.

Record OracleParameters :=
{
    description   : string;
    ...
}.

Record OracleState :=
{
    data           : float;
    totalCredit    : nat;
    ...
}.

Inductive Event : Type :=
    | DataWritten (newData : float) (newCost : nat) (caller : address)
    ...

Definition write_data (state : State) (...) : State :=

Definition Trace : Type := list (Event).
```

Formalization – Contracts as Records

```
contract Oracle is MultiOwnable {
    // # Parameters
    string public description;

    // # State Variables
    uint private data;
    mapping(address => uint) private credit;

    event DataWritten(uint data, uint cost);

    function writeData(uint _data, uint cost) external onlyOwner
    {
        writes += 1;
        data = _data;
        totalCost += cost;
        latestCost = cost;
        time += 1;
        latestWrite = time;
        emit DataWritten(data, cost);
    }
}
```

```
Record State :=
{
    oracleState      : OracleState;
    oracleParameters : OracleParameters;
    trace            : Trace
}.

Record OracleParameters :=
{
    description   : string;
    ...
}.

Record OracleState :=
{
    data           : float;
    totalCredit    : nat;
    ...
}.

Inductive Event : Type :=
    | DataWritten (newData : float) (newCost : nat) (caller : address)
    ...

Definition write_data (state : State) (...) : State :=

Definition Trace : Type := list (Event).
```

Formalization – Functions as Definitions

```
function writeData(uint _data, uint cost) external onlyOwner
{
    writes += 1;
    data = _data;
    totalCost += cost;
    latestCost = cost;
    time += 1;
    latestWrite = time;
    emit DataWritten(data, cost);
}
```

```
Definition write_data (state : State)
                      (newData : float)
                      (newCost : nat)
                      (caller : address) : State :=
    let oldOracleState := state.(oracleState) in
    let oldOracleParams := state.(oracleParameters) in
    let oldTrace := state.(trace) in

    if compare_address oldOracleParams.(owner) caller
    then
        let newOracleState := Build_OracleState
                                  (oldOracleState.(writes) + 1)
                                  newData
                                  (oldOracleState.(totalCost) + newCost)
                                  newCost
                                  (oldOracleState.(time) + 1)
                                  (oldOracleState.(time) + 1)
                                  ...
        in
        let newEvent := DataWritten newData newCost caller in
          Build_State newOracleState oldOracleParams (oldTrace ++ (newEvent :: nil))
    else
        state.
```

Formalization – Functions as Definitions

```
function writeData(uint _data, uint cost) external onlyOwner
{
    writes += 1;
    data = _data;
    totalCost += cost;
    latestCost = cost;
    time += 1;
    latestWrite = time;
    emit DataWritten(data, cost);
}
```

```
Definition write_data (state : State)
                      (newData : float)
                      (newCost : nat)
                      (caller : address) : State :=
    let oldOracleState := state.(oracleState) in
    let oldOracleParams := state.(oracleParameters) in
    let oldTrace := state.(trace) in

    if compare_address oldOracleParams.(owner) caller
    then
        let newOracleState := Build_OracleState
                                  (oldOracleState.(writes) + 1)
                                  newData
                                  (oldOracleState.(totalCost) + newCost)
                                  newCost
                                  (oldOracleState.(time) + 1)
                                  (oldOracleState.(time) + 1)
                                  ...
        in
        let newEvent := DataWritten newData newCost caller in
          Build_State newOracleState oldOracleParams (oldTrace ++ (newEvent :: nil))
    else
        state.
```

Formalization – Functions as Definitions

```
function writeData(uint _data, uint cost) external onlyOwner
{
    writes += 1;
    data = _data;
    totalCost += cost;
    latestCost = cost;
    time += 1;
    latestWrite = time;
    emit DataWritten(data, cost);
}
```

```
Definition write_data (state : State)
                      (newData : float)
                      (newCost : nat)
                      (caller : address) : State :=
    let oldOracleState := state.(oracleState) in
    let oldOracleParams := state.(oracleParameters) in
    let oldTrace := state.(trace) in

    if compare_address oldOracleParams.(owner) caller
    then
        let newOracleState := Build_OracleState
              (oldOracleState.(writes) + 1)
              newData
              (oldOracleState.(totalCost) + newCost)
              newCost
              (oldOracleState.(time) + 1)
              (oldOracleState.(time) + 1)
              ...
        in
        let newEvent := DataWritten newData newCost caller in
          Build_State newOracleState oldOracleParams (oldTrace ++ (newEvent :: nil))
    else
        state.
```

Formalization – Functions as Definitions

```
function writeData(uint _data, uint cost) external onlyOwner
{
    writes += 1;
    data = _data;
    totalCost += cost;
    latestCost = cost;
    time += 1;
    latestWrite = time;
    emit DataWritten(data, cost);
}
```

```
Definition write_data (state : State)
                      (newData : float)
                      (newCost : nat)
                      (caller : address) : State :=
    let oldOracleState := state.(oracleState) in
    let oldOracleParams := state.(oracleParameters) in
    let oldTrace := state.(trace) in

    if compare_address oldOracleParams.(owner) caller
    then
        let newOracleState := Build_OracleState
            (oldOracleState.(writes) + 1)
            newData
            (oldOracleState.(totalCost) + newCost)
            newCost
            (oldOracleState.(time) + 1)
            (oldOracleState.(time) + 1)
            ...
        in
        let newEvent := DataWritten newData newCost caller in
          Build_State newOracleState oldOracleParams (oldTrace ++ (newEvent :: nil))
    else
        state.
```

Formalization – Functions as Definitions

```
function writeData(uint _data, uint cost) external onlyOwner
{
    writes += 1;
    data = _data;
    totalCost += cost;
    latestCost = cost;
    time += 1;
    latestWrite = time;
    emit DataWritten(data, cost);
}
```

```
Definition write_data (state : State)
                      (newData : float)
                      (newCost : nat)
                      (caller : address) : State :=
    let oldOracleState := state.(oracleState) in
    let oldOracleParams := state.(oracleParameters) in
    let oldTrace := state.(trace) in

    if compare_address oldOracleParams.(owner) caller
    then
        let newOracleState := Build_OracleState
             (oldOracleState.(writes) + 1)
             newData
             (oldOracleState.(totalCost) + newCost)
             newCost
             (oldOracleState.(time) + 1)
             (oldOracleState.(time) + 1)
             ...
        in
        let newEvent := DataWritten newData newCost caller in
          Build_State newOracleState oldOracleParams (oldTrace ++ (newEvent :: nil))
    else
        state.
```

Returns the modified Oracle State

Formalization – Traces

```
Inductive Event : Type :=
    | DataWritten (newData : float) (newCost : nat) (caller : address)
    | DataRead (consumer : address) (weight : nat) (data : float)
    ...
```

Separate event for each **contract function**.

```
Definition execute (state : State) (event : Event) : State :=
    match event with
    ...
```

Simulation

Theorems Proved

Theorem 1: Consumers' credits are always non-negative.

```
Theorem credit_non_negative : forall (event : Event) (state : State),
    credit_non_negative_all_consumers (get_consumers state) ->
    credit_non_negative_all_consumers (get_consumers (execute state event)).
```

- Sanity check.
- 2 helper lemmas

Theorems Proved

Theorem 2: Between any two consecutive 'DataWrite' events, every consumer pays exactly once for obtaining data.

```
Fixpoint all_consumers_pay_once_slice (slice : list (State * Event)) : Prop :=
    match slice with
    | nil => True
    | (state, event) :: slice' =>
        match event with
```

```
Fixpoint all_consumers_pay_once (splitList : list (list (State * Event))) : Prop :=
```

```
Theorem all_consumers_pay_once_proof :
    forall (state : State) (baseFee : nat),
    all_consumers_pay_once (split_trace (get_trace state)
                                        (constructor (get_owner state)
                                                     (get_description state)
                                                     (get_locking_period state)
                                                     baseFee)
                                        nil).
```

- Goal G2a.
- Requires reasoning on the trace: 9 helper lemmas, 9 fixpoint definitions.

Theorems Proved

Theorem 3: The implemented adjustment of the base fee ensures the cost is the same as the revenue (or the base fee is at its max value) if:

- data was read (there was revenue);
- the write cost remained constant;
- the estimated number of reads and writes was correct (predictability).

```
Theorem base_fee_adjusted_single_slice :
    forall (slice : list (State * Event)),
    (reads_more_than_zero (slice) /\ cost_remains_same (slice) /\
     reads_writes_same (slice)    /\ correct_format_slice (slice)) ->
    total_cost_equals_total_revenue (slice) \/ new_base_fee_gt_max_fee (slice).
```

- Goal G1.
- Proof in progress.
- Requires reasoning on the trace: 8 helper lemmas so far.

Lessons Learned

Lessons Learned

🙁 A lot of effort to come up with the right representation.

Lessons Learned

- 🙁 A lot of effort to come up with the right representation.
- 🙁 Manually keeping the state is cumbersome
(also it relies on our understanding of the virtual machine).

Lessons Learned

- 🙁 A lot of effort to come up with the right representation.
- 🙁 Manually keeping the state is cumbersome
  (also it relies on our understanding of the virtual machine).
- 🙁 Formally verifying even simple properties takes a lot of time.

Lessons Learned

- 🙁 A lot of effort to come up with the right representation.
- 🙁 Manually keeping the state is cumbersome
  (also it relies on our understanding of the virtual machine).
- 🙁 Formally verifying even simple properties takes a lot of time.
- 😃 Thinking about properties to be satisfied guides the protocol design.

Lessons Learned

- 🙁 A lot of effort to come up with the right representation.
- 🙁 Manually keeping the state is cumbersome
(also it relies on our understanding of the virtual machine).
- 🙁 Formally verifying even simple properties takes a lot of time.
- 😃 Thinking about properties to be satisfied guides the protocol design.
- 😃 Implementation bugs were detected when developing proofs
(since we are forced to look at *every* case).

Lessons Learned

- 🙁 A lot of effort to come up with the right representation.
- 🙁 Manually keeping the state is cumbersome (also it relies on our understanding of the virtual machine).
- 🙁 Formally verifying even simple properties takes a lot of time.
- 😃 Thinking about properties to be satisfied guides the protocol design.
- 😃 Implementation bugs were detected when developing proofs (since we are forced to look at *every* case).
- 🤔 Formalization has potential (smart contract auditing is a thing!) but with the current state of the tools, it is hard to scale.

Thank you for your attention!
Questions?