Towards Verifiable-by-Design Smart Contracts: A Declarative Limit Order Books Implementation

Srisht Fateh Singh
Electrical & Computer Engineering
University of Toronto
Toronto, Canada
srishtfateh.singh@mail.utoronto.ca

Jeffrey Klinck
Electrical & Computer Engineering
University of Toronto
Toronto, Canada
jeffrey.klinck@mail.utoronto.ca

Zissis Poulos
School of Information Technology
York University
Toronto, Canada
zpoulos@yorku.ca

Andreas Veneris
Electrical & Computer Engineering
University of Toronto
Toronto, Canada
veneris@eecg.toronto.edu

Mohammad Fawaz
Independent Researcher
Toronto, Canada
mohammadfawaz89@gmail.com

Simon Roberts
Essential
Montreal, Canada
simon.roberts@essentialcontributions.com

***Abstract*—We present a declarative approach to on-chain limit order books (LOBs) that prioritizes formal verification over raw throughput. Unlike automated market makers, LOBs offer granular control and capital efficiency but are difficult to verify when implemented imperatively in Solidity. Using `Pint`, a declarative domain-specific language, we encode LOB matching logic, price-time priority, partial fills, and asset conservation, as first-order constraints. Off-chain solvers compute valid state transitions, while the blockchain performs lightweight constraint verification. We implement eight LOB predicates and evaluate performance using real-world transaction traces. Our declarative LOBs achieve 141 predicates/s for simple operations and 11 predicates/s for complex settlement with 1,000 accounts. Performance correlates strongly with state access patterns rather than constraint complexity. Critically, our approach eliminates verification challenges that make imperative smart contracts hard to formally verify, such as unbounded loops, recursion, cross-contract/function calls, and complex control flow. This enables correctness-by-construction through constraint satisfaction, removing the need to prove implementation conformance to specifications. This work demonstrates the first practical evidence that declarative LOBs achieve reasonable performance while providing superior verification guarantees for DeFi protocols.**

***Index Terms*—Limit Orderbooks, Declarative, Blockchain, Smart Contract, DeFi, Pint**

I. Introduction

The introduction of smart contract blockchains, such as Ethereum [1], in the last decade has led to the rise and adoption of decentralized finance (DeFi), where traditional financial intermediaries are replaced with autonomous and public protocols. Prominent DeFi protocols routinely manage and safeguard assets worth hundreds of millions of dollars, yet high-value smart contract failures remain endemic in the DeFi ecosystem. Between 2018 and 2025, cumulative losses exceeding $3 billion have been caused by smart contract logic bugs, oracle manipulation, and re-entrancy attacks [2]–[4]. To this extent, regulators now flag smart contract correctness as a critical barrier to institutional adoption. Institutions such as the Bank for International Settlements and the International Organization of Securities Commissions call for *formally demonstrable safeguards* before large asset managers can participate in DeFi [5], [6]. Thus, in the DeFi space, smart contract correctness and verifiability have now been elevated to first-order design goals rather than optional retroactive audits.

A fundamental application in any financial market is the ability to exchange one asset for another at the underlying exchange rate. In DeFi markets, this is achieved using one of the two protocol categories: automated market makers (AMMs) and limit orderbooks (LOBs). AMMs dominate today's on-chain trading volume by achieving simplicity of the underlying smart contract design at the expense of forgoing the trader's expressiveness and precision that are common in conventional exchanges [7], [8]. LOBs, on the other hand, are the canonical engine of traditional markets and provide the expressiveness missing in AMMs [9]–[11]. However, as expressive trading primitives amplify the need for correctness and verifiability, previous on-chain LOB attempts were thwarted by both execution cost and the difficulty of auditing program execution [12], [13]. Consequently, no fully on-chain LOB today offers the same formal safety guarantees as simpler DeFi primitives.

We approach the above problem from a *declarative* perspective. In a declarative smart contract design, the developer specifies constraints that must hold between the current state and the new state post-transaction. On the other hand, off-chain agents, known as *solvers*, search for compliant state transitions and propose them to blockchain validators. These validators merely check the satisfaction of constraints using the proposed new state values. Languages such as FINDEL and DECON have shown that many financial contracts naturally fit this paradigm [14], [15]. However, they have not tackled the full complexity of an LOB exchange.

To close this gap, we leverage a new declarative blockchain architecture introduced by *Essential* that supports constraint-based execution through a custom virtual machine and domain-specific language (DSL) called *Pint* [16]. In this model, smart contracts define permissible state transitions through logical *predicates*. We design a constraint system for LOB settlement in which fundamental properties such as order validity, price-time priority, and order matching consistency are enforced as declarative rules. Execution is delegated to off-chain solvers that submit candidate state updates, *e.g.*, matched trades. Therefore, a single predicate can confirm dozens of order

matches, while the on-chain verifier validates those predicates. The underlying virtual machine guarantees termination and supports efficient constraint evaluation, sidestepping the gas-metered, step-by-step execution model of traditional imperative virtual machines such as the EVM.

As part of our design, we represent resting trade orders using sorted linked lists indexed by price level, enabling efficient off-chain traversal and order matching while preserving deterministic ordering for on-chain verification. This decoupling allows the order state to reside off-chain (*e.g.,* in mempools) and excludes adversarial executions by construction, thereby guaranteeing correctness by design. While our implementation targets the `Essential` chain and its declarative virtual machine, the underlying programming logic is execution-agnostic. It enables LOB semantics to be expressed and verified via constraints. The declarative abstraction that we introduce is portable in principle and could be instantiated in any execution environment that supports predicate-based block validation. `Essential` serves here as a reference implementation.

We implement eight LOB predicates whose constraint design enforces LOB specifications and benchmark these predicates on `Essential`'s virtual machine using real-world LOB transaction traces. Our results show that single-threaded execution achieves throughput of 141 predicates/s for the simplest operation, `deposit`, and 11 predicates/s for the most complex, `settle_market_orders`[1] when the address space comprises $1,000$ accounts. Throughput decreases non-linearly with state size and correlates strongly with storage access patterns, while the number of logical constraints shows minimal impact on performance. Importantly, our declarative approach eliminates several verification challenges that plague imperative smart contracts, including unbounded loops, recursion, and cross-contract/function calls, which render formal verification computationally expensive or undecidable. This design achieves correctness-by-construction at modest throughput cost, providing the first practical evidence that purely declarative, constraint-verified orderbooks can deliver reasonable performance while offering superior verification guarantees compared to traditional imperative implementations.

The paper is organized as follows: Section II discusses related work, Section III provide background on the `Pint` language, Section IV decribes the underlying LOB model used in this work, along with the model properties, Section V describes the smart contract design and specifications enforcing the model properties, Section VI benchmarks the performance of the `Pint` smart contracts on `Essential`'s virtual machine, and lastly, the paper is concluded in Section VII.

II. Related Work

Early decentralized exchanges such as EtherDelta, 0x, and IDEX stored orders on Ethereum but matched them off-chain, re-introducing trust assumptions and limiting composability [12]. Subsequent designs moved execution off chain with on-chain verification. Optimistic rollups (e.g., Arbitrum) formalized the pattern of off-chain computation plus on-chain fraud/validity checks [17]. Layer-two (L2) designs exhibit distinct security/latency trade-offs: channels require online counterparties and locked collateral, plasma/commit chains weaken data availability via operator assumptions, whereas rollups inherit L1 security, incurring challenge delays for optimistic rollups or prover overhead for validity rollups [18]. Production DEXs then adopted validity rollups. For instance, Loopring uses zkSNARK-based proofs within a zk-rollup to attest correct matching and settlement [19], while dYdX v3 employs zkSTARK proofs and an off-chain order book and matcher [20], [21]. Despite these gains, ordering remains a critical concern. Transaction sequencing can create extractable value and fairness violations [22] and most rollups still rely on a centralized sequencer whose ordering policy is not publicly enforceable, motivating ongoing work on decentralized sequencing [23]. An alternative line of work relies on high-throughput L1s. Serum, for example, operates a fully on-chain LOB on Solana by exploiting sub-second blocks, yet it offers no static correctness guarantee beyond traditional code review [24]. Across all these architectures, a general, developer-level method for proving LOB invariants is missing.

AMMs circumvent explicit matching by enforcing a single algebraic invariant. Concentrated-liquidity variants, including Uniswap v3 [25] and Curve v2 [26], allow liquidity to be placed in price bands, partially emulating order-book depth. Nevertheless they lack discrete order types and time priority, and formal reasoning is limited to arithmetic soundness.

Declarative and logic-based smart-contract languages seek stronger assurances. FINDEL encodes derivative payoffs as pure expressions [14], and DECON models token contracts as Datalog-style rules [15]. Both ultimately compile to Solidity, so the resulting bytecode must still be audited. Industrial tools such as Certora translate each control-flow path in a Solidity function into a *proof obligation*, a logical formula that must be satisfied by an SMT solver [27]. Consequently, the number and size of these obligations grow with every additional loop iteration and branch in the code, so verification effort scales roughly with loop depth and branching complexity rather than with the simpler, post-state constraint we check in our declarative design. Our work differs by keeping execution and specification unified: order-matching logic is expressed as first-order constraints whose on-chain verification cost is constant in the number of matched orders.

III. `Pint` Overview

We implement the LOB as a declarative smart contract in `Pint` [16], a custom declarative programming language with its own virtual machine runtime. In the following sections, we first give a brief overview of the `Pint` suite, followed by the motivation for choosing `Pint`.

A. `Pint` *Declarative Language*

`Pint` is an open-source, content-addressable language for declarative programming, a programming paradigm that expresses computational logic without specifying control flow. Therefore, declarative smart contracts focus on what the program should accomplish, rather than how to accomplish it. On the other hand, imperative programs explicitly define the sequence of operations and control flow structures, such as loops and conditional statements, that dictate how computational goals are achieved. This fundamental difference makes

[1]This is amortized throughput where each call processes one order.

```
1  // addresses mapped to internal balances
2  storage {
3      balances: (b256 => int),
4  }
5  // predicate arguments
6  predicate deposit(addr: b256, amount: int)
7  {
8    // mutable state
9    let bal: int = mut storage::balances[addr];
10   // constraint 1
11   constraint amount >= 0;
12   // constraint 2
13   constraint (bal == nil && bal0' == amount) ||  (bal'
         == bal0 + amount);
14 }
```

Listing 1: Illustration of a simple Pint declarative contract.

declarative contracts more amenable to formal verification and parallel execution. `Pint` contracts consist of three components *viz. Storage*, *Predicates*, and *Solution*.

Storage: This defines all persistent variables in the smart contract, that is, variables that reside in the chain's persistent state. These include hash mappings and common data types such as 64-bit integers (`int`), booleans (`bool`), and addresses (represented as `[int; 4]` and aliased as `b256`). Lines 2–4 of Listing 1 show an example illustrating the storage element of the address–balance mapping.

Predicates: These are statically declared smart contract components analogous to function declarations in imperative smart contracts. However, instead of explicitly defining the state transition logic, predicates consist of the following subcomponents:

- **Address:** Each predicate is content-addressable with a unique 256-bit predicate address.
- **Mutable State:** A subset of contract storage variables that can be assigned new values during predicate execution. These variables must be explicitly declared within the predicate.
- **Arguments:** Similar to function parameters in imperative smart contracts, these are input values to a predicate that serve as local variables. These values may be used to define the predicate's logic, including constraints.
- **Constraints:** These are the central components of a predicate that define logical conditions. They constrain current state values, post-state values, and local variables, including predicate arguments. These constraints *must* be satisfied for a proposed post-state to be considered valid.

Lines 6–14 of Listing 1 show an example of a simple `deposit` predicate that increases the balance of the specified address by `amount`. The predicate consists of two arguments (Line 6), one mutable state (Line 9), and two constraints. The first constraint checks for a positive value of the `amount` argument (Line 11). The second constraint checks for the appropriate post-value of the user's balance (Line 13-14).

Solution: Each block in `Pint`-based blockchains, such as `Essential`, consists of a single state transition with corresponding pre-state and post-state values. In the below, pre-state variables are represented without prime notation, while post-state variables use prime notation (*e.g.*, state variable v becomes v' in the post-state). Agents known as *solvers* propose solutions that satisfy predicate constraints.

A solution with n predicate arguments and m mutable states consists of the following fields:

$$\{predAddr,\ (a_1, a_2, \ldots, a_n),\ (k_1, v_1'),\ (k_2, v_2') \ldots (k_m, v_m')\}$$

Here, $predAddr$ represents the predicate address, a_i denotes the i-th predicate argument, k_i represents the key of a mutable storage variable, and v_i' represents the corresponding post-state value. For example, in Listing 1, a solver produces a solution containing: the predicate address, predicate arguments (user address `addr` and deposit amount `amount`), and the key-value pair for the user's updated balance (`balances[addr]`'s key and its new value).

B. Motivation for `Pint`

Although there are numerous smart contract programming languages, we chose `Pint` because it offers several key advantages over alternatives.

Formal Verification: `Pint` smart contracts are statically deterministic at compile time, unlike Solidity contracts [28], which exhibit dynamic, runtime-dependent behavior. This static nature makes `Pint` contracts more amenable to formal verification methods, particularly for cross-contract interactions. This is discussed in detail in Section VI-C. Furthermore, since contracts consist of constraint sets, they are inherently well-suited for formal verification techniques.

Enhanced Security: Historically, reentrancy attacks on Solidity smart contracts have caused losses exceeding hundreds of millions of dollars. `Pint`'s declarative structure makes reentrancy attacks impossible. Additionally, all state variables within a predicate are immutable by default, unless explicitly declared mutable. This restriction on state access significantly reduces contract vulnerability to attacks exploiting unexpected state modifications.

Optimized for Parallelization: `Pint` is specifically optimized for declarative programming, unlike general-purpose languages such as Solidity. Notably, `Pint` can parallely validate multiple predicates with a common state transition. In contrast, functions in imperative languages, which serve as the equivalent of predicates, cannot be executed in parallel due to their sequential control flow requirements.

IV. Limit Orderbook Model

This section describes the underlying model for limit orderbooks upon which our smart contract design is based. We present both the underlying state representation of an LOB and the properties that must be satisfied during state transitions.

A. State Representation

We consider a limit orderbook for the token pair $token\ 0$, $token\ 1$, where $token\ 0$ represents the principal asset (such as BTC or ETH) and $token\ 1$ represents the numeraire (such as USDC). Our model consists of the following state components, including derived states represented as function mappings.

User Balances: We consider a set of user addresses $\mathbb{A}$ and denote an individual address as $a \in \mathbb{A}$. For each address a, we represent its $token\ 0$ and $token\ 1$ balances using functions $B_0(a)$ and $B_1(a)$, respectively, where both balances are positive integers.

Limit Orders: We represent limit orders, which are quotes to buy or sell $token\ 0$ for $token\ 1$, as tuples (p, amt, t) where p denotes the order price, amt denotes the unfilled order amount, and t denotes the order's posting time, all represented as integers. A limit order quoting a buy is referred to as a *bid*, while one quoting a sell is referred to as an *ask*. An order is considered *active* if its posting time t is less than the current time.

Order Indexing: We index the set of all limit orders using indices i and j. The i^{th} bid is represented using function $Bid(i)$, which maps i to the tuple (p_i, amt_i, t_i). Similarly, the j^{th} ask is represented using function $Ask(j)$, which maps j to tuple (p_j, amt_j, t_j).

B. Model Properties

The following properties must be enforced during state transitions to ensure correct orderbook behavior.

Property 1 (Token Balance Invariant)**.** *The total amounts of $token\ 0$ and $token\ 1$ across all users cannot increase during a valid state transition. This ensures that tokens are not created during trading operations. Mathematically, this constraint on balances B_0 and B_1 for $tokens\ 0$ and 1, respectively, is expressed as:*

$$\begin{aligned} \sum_{a \in \mathbb{A}} B_0(a) &\geq \sum_{a \in \mathbb{A}} B'_0(a) \\ \sum_{a \in \mathbb{A}} B_1(a) &\geq \sum_{a \in \mathbb{A}} B'_1(a). \end{aligned} \tag{1}$$

Property 2 (Order Settlement Invariant)**.** *This property ensures that when orders are executed, user balances and order amounts change consistently, preserving the conservation of tokens during trades.*

For a bid order at index i executed by address a:

$$\begin{aligned} B_0(a) + amt_i &= B'_0(a) + amt'_i \\ B_1(a) - p_i \cdot amt_i &= B'_1(a) - p_i \cdot amt'_i \end{aligned}$$

This means the user gains $token\ 0$ tokens and loses $token\ 1$ tokens proportional to the filled amount.

Similarly, for an ask order at index j by address a:

$$\begin{aligned} B_0(a) - amt_j &= B'_0(a) - amt'_j \\ B_1(a) + p_j \cdot amt_j &= B'_1(a) + p_j \cdot amt'_j \end{aligned}$$

This means the user loses $token\ 0$ tokens and gains $token\ 1$ tokens proportional to the filled amount.

Property 3 (Price Priority)**.** *This property ensures that limit orders with better prices are executed with priority over those with worse prices. Formally:*

- *If a bid order with index i is executed, then in the post-state, there is no active bid order with index j that has a higher price ($p_j > p_i$) and a positive amount ($amt'_j > 0$).*
- *If an ask order with index i is executed, then in the post-state, there is no active ask order with index j that has a lower price ($p_j < p_i$) and a positive amount ($amt'_j > 0$).*

Property 4 (Time Priority)**.** *This property ensures that when multiple orders have the same price, the order that arrived first is prioritized. Formally, if a limit order (either bid or ask) with index i is executed, then there does not exist an order with index j in the post-state that has a positive amount ($amt'_j > 0$), the same price as order i ($p_i = p_j$), and arrived before order i ($t_j < t_i$).*

V. Declarative Limit Orderbook Design

In this section, we describe the design of `Pint` smart contracts for the LOB model presented above. We begin with a design overview, followed by contract specifications based on LOB model properties, and conclude with detailed predicate designs.

A. Limit Orderbook Design Overview

To enforce Properties 3 and 4, we maintain submitted bid and ask orders as two separate linked lists ordered by price-time priority. Although insertion and deletion in a linked list requires $\mathcal{O}(n)$ operations (where n is the list size), verification of these operations requires only $\mathcal{O}(1)$ operations. Since declarative contracts only validate proposed solutions rather than computing them, the execution cost of insertion and deletion predicates is $\mathcal{O}(1)$. However, the computational cost for off-chain solvers to generate solutions remains $\mathcal{O}(n)$.

Beyond insertion and deletion, we implement *limit order settlement*, where solvers match and settle groups of bid orders with groups of ask orders. In this settlement predicate, the solver provides the order indices to be matched as predicate arguments. The predicate's constraints validate Properties 1 and 2 from Section IV-B. Additionally, the solver can post a single bid and ask order as part of the arguments, enabling the solver to act as a market maker, as illustrated in Figure 1.

Figure 1a shows the non-overlapping arrangement of bid and ask limit orders at block b, where the market price lies between the highest bid price and the lowest ask price. Figure 1b illustrates the subsequent block $b + 1$, where the market price increases above the lowest ask price. In this scenario, the solver posts a bid order that matches the lowest ask order and simultaneously settles both orders.

Finally, we implement market orders, which are similar to limit orders but lack a specified execution price and require immediate settlement. To execute a bid (ask) market order predicate, the solver provides the corresponding ask (bid) limit orders from the linked list as predicate arguments.

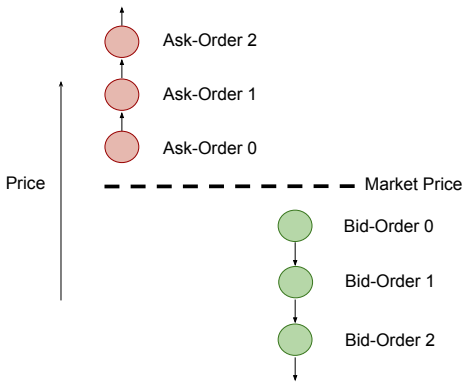
B. LOB Smart Contract Specifications

We outline the specifications of the LOB smart contract, followed by predicate designs that include constraints to enforce these specifications.

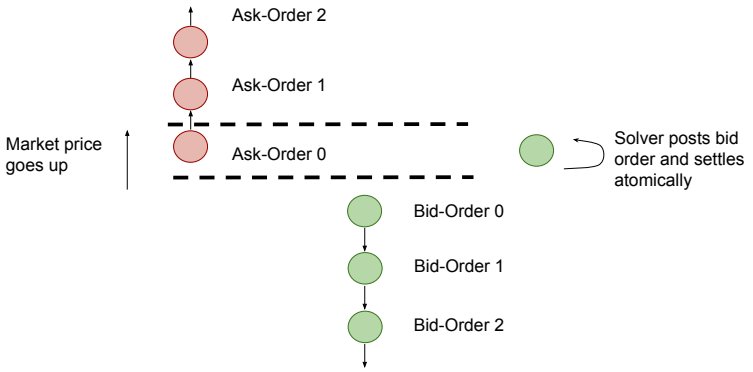
Specification 1 (Token Pair Uniqueness)**.** *The contract is uniquely defined for a single pair of tokens $(token\ 0, token\ 1)$ and does not support trading of other token pairs.*

Specification 2 (Custodial Balance Management)**.** *The smart contract maintains custody of all $token\ 0$ and $token\ 1$ tokens involved in trading. Users must deposit tokens into the contract before trading and can withdraw their remaining balances after trading activities conclude.*

Specification 3 (Price-Time Priority Ordering)**.** *Limit orders are stored as two separate linked lists (bids and asks), each ordered by price-time priority:*



(a) Bid and Ask orders maintained as linked lists.



(b) When the market price rises, the solver posts a new order that settles immediately in the same block.

Fig. 1: Design overview of limit orderbooks on declarative smart contracts.

- ***Bid list****: Each bid order points to the subsequent bid with lower price. The highest-priced bid is at the head of the list.*
- ***Ask list****: Each ask order points to the subsequent ask with a higher price. The lowest-priced ask is at the head of the list.*
- ***Time priority****: For orders with identical prices, earlier-arriving orders point to later-arriving orders.*

Specification 4 (Settlement Invariants)**.** *During the settlement of multiple bid and ask orders, the following must hold:*

- *Token balance invariant (Property 1).*
- *Order settlement invariant (Property 2).*
- *Price-time priority is respected (Properties 3 and 4).*

Specification 5 (Solver Market Making)**.** *During settlement execution, solvers may post exactly one bid order and one ask order that are settled alongside existing orders.*

Specification 6 (Market Order Execution)**.** *Market orders are executed as follows:*

- *Bid market orders match with the highest-priority (lowest-priced) ask orders*
- *Ask market orders match with the highest-priority (highest-priced) bid orders*
- *Multiple market orders in the same block receive identical average execution prices*
- *All settlement invariants (Property 1 and 2) are enforced*

Specification 7 (Order Lifecycle)**.** *Orders can be inserted, partially filled, completely filled, or cancelled. Partially filled orders retain their original priority position with updated amounts.*

C. LOB Smart Contract Design

In this section, we describe the design of the declarative smart contract that implements the specifications outlined above for limit orderbooks. We detail the key components of the declarative contract structure: the storage state and the predicates, including their arguments, mutable state, and constraints.

1) Storage State: To implement Specification 1, the smart contract operates exclusively on a unique pair of tokens, $token\ 0$ and $token\ 1$, per contract instance.

```
1  predicate deposit(amount0: int, amount1: int, addr:
       b256){
2      let bal0: int = mut storage::balances_0[addr];
3      let bal1: int = mut storage::balances_1[addr];
4      constraint amount0 >= 0;
5      constraint amount1 >= 0;
6      constraint (bal0 == nil && bal0' == amount0) ||
7      (bal0' == bal0 + amount0);
8      constraint (bal1 == nil && bal1' == amount1) ||
9      (bal1' == bal1 + amount1);
10 }
```

Listing 2: Simplified design of the deposit predicate.

For Specification 2, the contract maintains two mappings: `balance_0` and `balance_1`, which map user addresses to their internal balances of $token\ 0$ and $token\ 1$, respectively. Users must deposit these tokens before interacting with the orderbook contract and can withdraw their remaining tokens after completing their interactions, as detailed in the predicate design section.

To implement Specification 3, the contract stores two key mappings: `bid_list` and `ask_list`. These mappings associate each order index i with a tuple (p_i, amt_i, j), where p_i represents the order's price, amt_i represents the unfilled amount, and j represents the index of the next order in the linked list structure. Additionally, the contract maintains `first_bid_index` and `first_ask_index` to identify the head of each linked list. The specific ordering of these lists to enforce price-time priority is detailed in the following section.

2) Predicate Design: The smart contract consists of eight predicates:

`deposit`: This predicate addresses specifications 1 and 2, and is used to deposit either token into the contract. Listing 2 shows the simplified design. Its arguments are the depositor's address and the deposit amounts of $token\ 0$ and $token\ 1$, respectively. The mutable state comprises only the two balance slots of the sender (lines 2–3), while all other storage remains immutable. Constraints enforce *(i)* positive deposit amounts (lines 4–5) and (ii) the correct update of the pre- and post-state balances (line $6-9$), optionally including a cross-call to a token contract in a full implementation.

`withdraw`: Complementing `deposit`, this predicate lets a user withdraw tokens and also satisfies specifications 1 and 2. The

```
1  predicate add_bid(lead_key: int, trail_key: int,
       new_order: order, new_index: int){
2      //mutate storage
3      let new_slot = mut storage::bid_list[new_index];
4      let lead_order_next: int =  mut storage::bid_list[
           lead_key].next_key;
5      let first_index: int = mut storage::
           first_bid_index;
6      ...
7      //verify solver gave empty slot
8      constraint new_slot == nil && new_slot' ==
           new_order;
9      //update leading order
10     constraint lead_order_next == trail_key;
11     constraint lead_order_next' == new_index;
12     //verify order info
13     ...
14     //verify the user has enough balance
15     ...
16     //verify and update the first order pointer
17     if (lead_key == 0) {
18         constraint first_index' == new_index;
19         constraint first_index == trail_key;
20     }else{// first order index is not changing
21         constraint first_index' == first_index;
22     }
23     // verify correct price-time priority order
24     constraint new_order.price <= lead_order_price;
25     constraint new_order.price > trail_order_price;
26 }
```

Listing 3: Simplified design of the add_bid predicate.

arguments and mutable state mirror those of `deposit`, however, the balance constraints now *decrease* the post-state balances, and ensures the user has sufficient funds before withdrawal.

`add_bid`: Listing 3 implements the logic for inserting a new bid order while preserving price-time priority (Specification 3). The predicate receives the keys of the leading and trailing orders (`lead_key` and `trail_key`), the new order tuple, and its storage index as arguments. Mutable state includes the candidate slot for the new order, the `next_key` of the leading order, and the global pointer `first_bid_index`. Constraints *(i)* guarantee the chosen slot is empty (line 8), *(ii)* patch the linked list by redirecting the leading order's `next_key` to point to the new order (lines 10-11), *(iii)* perform sanity checks on the new order and verify the order owner has sufficient balance (lines 13-15), *(iv)* update `first_bid_index` if inserting at the head (lines 17-22), and *(v)* verify the new order's price maintains price-time priority relative to its neighbors (lines 24-25). The strict inequality in line 25 enforces time priority among orders with identical prices, *i.e.* $p_{\text{lead}} \geq p_{\text{new}} > p_{\text{trail}}$.

`remove_bid`: Listing 4 removes an existing bid order identified by `mid_index`. The mutable state includes the order being removed, the `next_key` of the leading order, and the `first_bid_index` pointer. Constraints ensure *(i)* the target slot is cleared in the post-state (line 7), *(ii)* the linked list is properly spliced by connecting the leading order directly to the trailing order (lines 9-10), *(iii)* the removed order's neighbors match expected values (lines 13), and *(iv)* the head pointer is updated when removing the first order (lines 15-20).

`add_ask`: Symmetric to `add_bid`, this predicate inserts a new ask order into `ask_list`, enforcing ascending price-time priority for asks. Its structure, mutability pattern, and constraint set mirror those in Listing 3, with the change that prices are compared in the opposite direction ($p_{\text{lead}} \leq p_{\text{new}} < p_{\text{trail}}$).

```
1  predicate remove_bid(lead_key: int, trail_key: int,
       mid_index: int){
2      //mutate storage
3      let mid_slot = mut storage::bid_list[mid_index];
4      let lead_order_next: int =  mut storage::bid_list[
           lead_key].next_key;
5      let first_index: int = mut storage::
           first_bid_index;
6      //verify solver updated the middle order to zero
7      constraint mid_slot' == nil;
8      //update leading order
9      constraint lead_order_next' == trail_key;
10     constraint lead_order_next == mid_index;
11     //verify order info
12     constraint mid_index > 0; //0 index is nil order
13     constraint mid_slot.next_key == trail_key;
14     //verify and update the first order pointer
15     if (lead_key == 0) {
16         constraint first_index == mid_index;
17         constraint first_index' == trail_key;
18     }else{// first order index is not changing
19         constraint first_index' == first_index;
20     }
21 }
```

Listing 4: Simplified design of the remove_bid predicate.

`remove_ask`: This predicate is the ask-side counterpart of `remove_bid`. It unlinks an ask order from the linked list, updates `first_ask_index` when required, and validates that all pointer rewrites, including first index pointer are correct.

`settle_limit_orders`: This predicate, as shown in listing 5, matches and settles a batch of highest-priority (highest-priced) bid orders with highest-priority (lowest-priced) ask orders until the terminal indices provided by the solver are reached. Its arguments include the unfilled remainders of terminal orders (`last_amnt_bid`, `last_amnt_ask`), their indices (`last_bid_id`, `last_ask_id`), arrays of fully-matched order indices (`bid_ids`, `ask_ids`), and two virtual bridging orders provided by the solver (`solver_orders`), addressing Specification 5.

The mutable state comprises every order slot referenced in the two ID arrays (set to `nil` once cleared), the two partially filled terminal orders (whose `max_amnt` fields are reduced by the remainder arguments), and the solver's and order owners' $token$ 0 and $token$ 1 balances. The predicate instantiates four internal variables that count the total inflow and outflow of each token.

The constraints first perform sanity checks on every supplied ID, then enforce two token balance-conservation equalities that include the solver's bridging orders, according to Inequality (1) (line $10-11$). Lastly, the solver's post-state balances are updated accordingly and constrained to remain non-negative (line $13-14$).

`settle_market_orders`: This predicate simultaneously clears a batch of market orders against the current highest-priority limit orders at solver-quoted average prices. Its arguments include the same four terminal parameters as the above predicate, arrays of incoming market orders (`bid_mos`, `ask_mos`), which are a tuple of user address and order size, the IDs of limit orders that are fully filled during the process (`bid_ids`, `ask_ids`), and the solver's average-price quotation for each side (`avg_price_bids` and `avg_price_asks`).

The mutable state touches every order object referenced by any of those arrays, the global head pointers of both

```
predicate settle(last_amnt_bid: int, last_amnt_ask: int, last_bid_id: int, last_ask_id: int, bid_ids: int[], ask_ids: int[], solver_orders: order[2]){
    //mutable storage
    ...
    // perform sanity checks on bid and ask orders
    ...
    let sum_all_zero_bids: int = ...;
    let sum_all_one_bids: int = ...;
    let sum_all_zero_asks: int = ...;
    let sum_all_one_asks: int = ...;
    constraint -sum_all_zero_bids + sum_all_zero_asks - solver_orders[0].max_amnt * solver_orders[0].price + solver_orders[1].max_amnt * solver_orders[1].price <= 0;
    constraint sum_all_one_bids - sum_all_one_asks + solver_orders[0].max_amnt - solver_orders[1].max_amnt <= 0;
    //Verifying order settlement invariant for solver
    constraint solver_bal0' == solver_bal0 - solver_max_amnt0 * solver_price0 + solver_max_amnt1 * solver_price1;
    constraint solver_bal1' == solver_bal1 + solver_max_amnt0 - solver_max_amnt1;
    constraint solver_bal0' >= 0;
    constraint solver_bal1' >= 0;
}
```

Listing 5: Simplified design of the settle predicate.

```
predicate settle_market_orders(last_amnt_bid: int, last_amnt_ask: int, last_bid_id: int, last_ask_id: int, bid_mos: market_order[], ask_mos: market_order[], bid_ids: int[], ask_ids: int[], avg_price_bids: int, avg_price_asks: int){
    // sum all market orders
    let sum_bid_market_orders: int = ...;
    let sum_ask_market_orders: int = ...;
    //sum all limit orders and verify updated balances
    ...
    let sum_all_zero_bids: int = ...;
    let sum_all_one_bids: int = ...;
    let sum_all_zero_asks: int = ...;
    let sum_all_one_asks: int = ...;
    // verify average prices given by solver
    constraint sum_all_zero_bids/sum_all_one_bids == avg_price_bids;
    constraint sum_all_zero_asks/sum_all_one_asks == avg_price_asks;
    // verify market orders are getting fair price
    constraint sum_ask_market_orders == sum_all_one_bids;
    constraint sum_bid_market_orders == sum_all_one_asks;
    //pro-rata market orders distribution
    @distribute_market_orders_bids(avg_price_asks; ~bid_market_orders);
    @distribute_market_orders_asks(avg_price_bids; ~ask_market_orders);
}
```

Listing 6: Simplified design of settle_market_orders.

books, and the user's balances. In addition to the above predicate, it declares two local variables `sum_bid_market_orders` and `sum_ask_market_orders` that aggregate the sizes of the market-order arrays (lines $3-4$).

Its constraints *(i)* verify that the solver's quoted average prices correctly reflect the weighted average of matched limit orders (lines 12-13), *(ii)* ensure total market order volume matches total limit order volume (lines 15-16), and *(iii)* distribute fills pro-rata using the `@distribute_market_orders_*` macros at the quoted average price. Note that the above constraints on ensuring the correct average price, equal market and limit order sizes, and distribution at average prices implicitly enforce the token balance-conservation invariant (1). Overall, the predicate implements the market order specification (6).

VI. Evaluation

In this section, we evaluate and benchmark the performance of the `Essential` virtual machine that executes the limit order-book smart contracts. `Essential` provides an open-source Rust language suite for executing `Pint` declarative smart contracts. Our experiments attempt to answer the following questions: *i)* What is the LOB throughput? *ii)* What is the state requirement, and how does throughput depend on it? *iii)* How do constraints scale, and what is their effect on throughput?

A. Methodology

All experiments are run on a 12-core Apple M2 Pro (ARM64) machine with 16 GB RAM and a 1 TB SSD, running macOS 15.5 (Darwin 24.5). All code is compiled with Rust 1.85 using the default `--release` profile. The implementations of the pint source code and Rust-based solver are publicly available. [2]

We fund 10k addresses with 10^{12} units of each token to create the experimental trace. For benchmarking the `deposit` and `withdraw` predicates, we average the measurements over this set of addresses. For benchmarking the remaining predicates *viz.*, `add_bid`, `remove_bid`, `add_ask`, `remove_ask`, `settle_limit_orders`, and `settle_market_orders`, we generate an input trace from real-world LOB data for the ETH–USDC pair on the Kraken exchange [29] as follows.

Every $\Delta = 15$ seconds, we first fetch the mid-price, defined as the average of the highest bid and lowest ask. Using this price, we settle bid orders priced above it, ask orders priced below it, and any solver orders supplying the remaining counter-orders, as illustrated in Figure 1. We settle a batch of 10 bid and ask orders per predicate. In the second step, we retrieve the number of market orders from the exchange and settle the same number of market orders in the smart contract. In the third step, we fetch a snapshot of the limit-order book (LOB) containing the bid and ask lists. We bin these limit orders by aggregating orders within a $0.1 price range and compare the current snapshot with the previous one. Depending on whether the volume in a bin increases or decreases, we add or remove limit orders, respectively, in the linked list. In each round, we call each of the six predicates and amortize the measurements over the entire run for that predicate.

B. Result Summary

Smart-Contract Throughput: Figure 2 plots the single-thread validation throughput measurements for the LOB predicates when the address space equals 1k, 4k, 7k, and 10k, respectively. For every predicate, the throughput decreases non-linearly as the state size increases, however, the same degradation trend is observed for all the predicates. The simplest predicate, `deposit`, achieves 141.64 predicates/s at the 1k state size and drops to 43.99, 26.52, and 18.71 for 4k, 7k,

[2]The code can be found at: https://github.com/fateh321/Declarative-LOB.

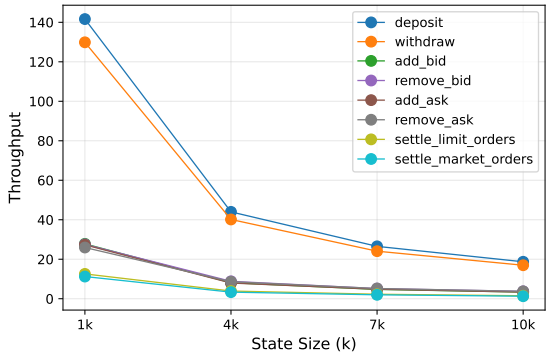


Fig. 2: Throughput degradation vs. increasing state size.

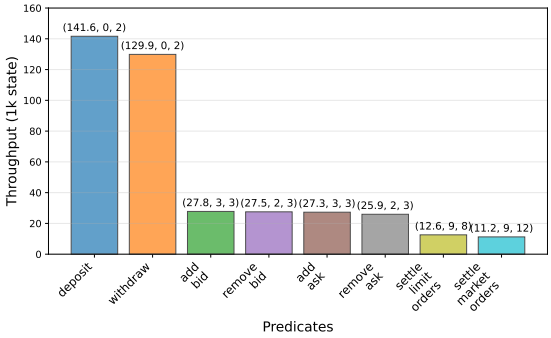


Fig. 3: Monotonic throughput decline vs. state access.

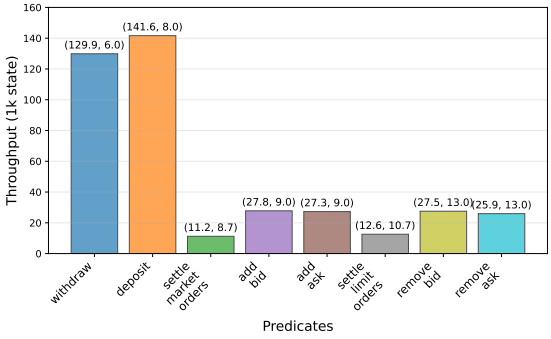


Fig. 4: Weak throughput correlation vs. constraint count.

and 10k addresses, respectively. Conversely, the most complex predicate, `settle_market_orders`, reaches only 11.23, 3.33, 1.98, and 1.27 predicates per second per order. This trend is further explained by the relationship between throughput and state access, which is discussed next.

Predicate State Access: Figure 3 plots the throughput at 1k state size along with the number of state writes (mutable accesses) and reads per predicate, shown in parentheses. Simple predicates such as `deposit` and `withdraw` touch only two state slots, whereas `settle_limit_orders` and `settle_market_orders` perform, on average, 16.8 and 20.6 read–write accesses per order, respectively. Across all predicates, validation throughput decreases monotonically as total state accesses increase, confirming that state access is the primary bottleneck.

Predicate Constraints: Figure 4 plots throughput along with the number of logical constraints that each predicate evaluates, in parentheses. Unlike state size or access, this metric shows little correlation with validation throughput.

TABLE I: Verification Complexity Features.

Operation	**Imperative LOB (Solidity)**	**Declarative LOB (Pint)**
Unbounded Loops	✓	×
Recursion	✓	×
Cross-function Calls	✓	×
External Contract Calls	✓	×
Dynamic Memory Access	✓	×
Dynamic Gas Calculation	✓	×
Conditional Statements	✓	✓
Exception Handling	✓	✓
Time-dependent Behavior	✓	✓

For example, `deposit` executes eight constraints yet attains the highest throughput, whereas `add_bid` executes roughly the same number (nine) but runs four times slower, and `settle_limit_orders` averages 10.7 constraints while still being an order of magnitude slower. These observations confirm that throughput is dominated by on-chain state access rather than constraint complexity, with implications for designing scalable declarative smart contracts. Note that these measurements represent single-threaded performance on the `Essential` VM. Parallel execution could yield higher throughput.

C. Program Analysis Cost

Formal correctness of smart contracts can be broken down into two distinct stages. The first stage converts domain requirements into a set of formal invariants (*e.g.,* price–time priority and balance conservation for an LOB). This specification step is paradigm-agnostic; both imperative and declarative designs must first state *what* must hold true. The second stage proves that the executable contract never violates those invariants. This is where the two paradigms diverge. In an imperative Solidity implementation the developer leverages program features such as unbounded loops, external contract calls, dynamically allocated memory, and gas-dependent control flow, which significantly hinder verification efforts. Formal methods like SMT solvers and model checkers struggle with the undecidable nature of features like recursion and unbounded iteration [30]. Consequently, industry tools often impose constraints, such as bounding loops or simplifying conditionals, to make verification tractable. These compromises may lead to incomplete, inaccurate, or overly conservative results, especially in complex DeFi protocols. On the other hand, a declarative smart contract eliminates this second stage: the constraints written in the specification constitute the executable contract itself. Off-chain solvers propose state transitions, and the blockchain checks a single first-order formula whose size is independent of internal loops or data-structure traversals. The costly second stage disappears and verification reduces to constraint satisfaction at runtime.

Table I demonstrates that `Pint` eliminates most computationally expensive verification challenges present in Solidity. Most notably, `Pint`'s declarative nature removes unbounded loops, recursion, cross-function and external contract calls (both statically verifiable), dynamic memory access, and dynamic gas calculations that render verification undecidable or compu-

tationally intractable. However, both languages face challenges with conditional statements (though `Pint`'s conditionals are statically bounded and lack complex control flow compared to Solidity's dynamic branching), exception handling, and time-dependent behavior, indicating that some verification complexity is inherent to smart contract logic rather than the programming paradigm. The elimination of six major complexity sources significantly reduces the verification burden, enabling correctness-by-construction through constraint satisfaction.

VII. Conclusion and Future Work

This paper presents the first practical implementation of a fully declarative limit orderbook that prioritizes formal verification over raw throughput. By leveraging `Pint`'s constraint-based programming paradigm, we shift verification from post-hoc auditing to correctness-by-construction, demonstrating that complex financial protocols can be expressed as first-order constraints while maintaining practical performance levels. The implications extend beyond orderbooks to other financial primitives such as derivatives, lending protocols, and complex DeFi applications where traditional imperative implementations struggle with formal verification. Future work includes: (i) formalizing LOB specifications using first-order logic with automated theorem proving for complete correctness verification, (ii) implementing more complex LOB features such as Immediate-or-Cancel (IOC) and Fill-or-Kill (FOK) orders, stop-limit orders, iceberg liquidity, and fee tiers, though the verification advantages over imperative implementations would persist due to `Pint`'s declarative structure, (iii) comparing declarative constraint verification against zero-knowledge proof approaches where each predicate is implemented as a ZK circuit with solver-generated proofs for state transition verified on-chain, particularly analyzing trade-offs in verification cost, scalability, and implementation complexity, (iv) enhancing throughput through hybrid architectures that store order metadata off-chain while maintaining constraint verification on-chain, and (v) optimizing `Pint`'s virtual machine through parallel constraint evaluation and improved state access mechanisms to reduce the performance gap with imperative implementations.

References

[1] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2014, https://ethereum.org/en/whitepaper/.

[2] S. Werner, D. Perez, F. Tramer, and A. Gervais, "Sok: Decentralized finance (defi)," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2021, pp. 1048–1064.

[3] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (SoK)," in *Proceedings of the 6th International Conference on Principles of Security and Trust (POST)*, ser. Lecture Notes in Computer Science, vol. 10204. Springer, 2017, pp. 164–186.

[4] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais, "Sok: Decentralized finance (defi) attacks," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2444–2461.

[5] International Organization of Securities Commissions, "Policy recommendations for decentralized finance (defi): Consultation report," https://www.iosco.org/library/pubdocs/pdf/IOSCOPD744.pdf, 2023.

[6] Bank for International Settlements, "The financial stability risks of decentralised finance – executive summary," https://www.bis.org/fsi/fsisummaries/defi.pdf, 2023.

[7] J. Milionis, C. C. Moallemi, and T. Roughgarden, "Complexity-approximation trade-offs in exchange mechanisms: Amms vs. lobs," in *Financial Cryptography and Data Security*, 2023.

[8] A. Park, "The conceptual flaws of decentralized automated market making," *SSRN Electronic Journal*, 2023. [Online]. Available: https://ssrn.com/abstract=3805750

[9] T. Foucault, O. Kadan, and E. Kandel, "Limit order book as a market for liquidity," *The Review of Financial Studies*, vol. 18, no. 4, pp. 1171–1217, 2005.

[10] K. Malinova and A. Park, "Subsidizing liquidity: The impact of make/take fees on market quality," *Journal of Finance*, 2014, forthcoming. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=1823600

[11] A. Aidov and A. Lobanova, "The relation between intraday limit order book depth and spread," *International Journal of Financial Studies*, vol. 9, no. 4, p. 60, 2021.

[12] S. Eskandari, S. Moosavi, and J. Clark, "Sok: Transparent dishonesty: Front-running attacks on blockchain," in *Financial Cryptography Workshops*, 2019.

[13] S. A. Moosavi and J. Clark, "Lissy: Experimenting with on-chain order books," *arXiv preprint arXiv:2101.06291*, 2021.

[14] A. Biryukov, D. Khovratovich, and S. Tikhomirov, "Findel: Secure derivative contracts for ethereum," in *Financial Cryptography and Data Security Workshops (WTSC '17)*, ser. Lecture Notes in Computer Science, vol. 10323. Springer, 2017, pp. 453–467.

[15] H. Chen, G. Whitters, M. J. Amiri, Y. Wang, and B. T. Loo, "Declarative smart contracts," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. Association for Computing Machinery, 2022, p. 281–293. [Online]. Available: https://doi.org/10.1145/3540250.3549121

[16] Essential Protocol, "Pint language: Declarative smart contracts on essential," https://essential-contributions.github.io/pint/book/the-book-of-pint.html, 2024.

[17] H. Kalodner, S. Goldfeder, S. M. Weinberg, and E. W. Felten, "Arbitrum: Scalable, private smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 1353–1370.

[18] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, "Sok: Layer-two blockchain protocols." Berlin, Heidelberg: Springer-Verlag, 2020, p. 201–226. [Online]. Available: https://doi.org/10.1007/978-3-030-51280-4_12

[19] L. Protocol, "Loopring 3.0: zkrollup exchange and payment protocol," 2021, technical documentation.

[20] dYdX Foundation, "dydx layer 2 exchange architecture overview," 2022, whitepaper.

[21] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von neumann architecture," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 781–796.

[22] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 910–927.

[23] S. Motepalli, L. Freitas, and B. Livshits, "Sok: Decentralized sequencers for rollups," *arXiv preprint arXiv:2310.03616*, 2023, well-cited preprint. [Online]. Available: https://arxiv.org/abs/2310.03616

[24] D. Shuttleworth, "Serum: A decentralized on-chain central limit order book," https://consensys.io/blog/serum-a-decentralized-on-chain-central-limit-order-book, Feb. 2022.

[25] Uniswap Labs, "Uniswap v3 core whitepaper," 2021, technical whitepaper.

[26] Curve Finance, "Curve v2: Amm for volatile assets," 2022, protocol documentation.

[27] Certora, "Certora prover: Scalable formal verification of smart contracts," https://docs.certora.com/en/latest/prover/introduction.html, 2023.

[28] The Solidity Authors, "*Solidity*: The Smart-Contract Programming Language," https://docs.soliditylang.org/, 2025.

[29] Kraken, "Kraken exchange rest api documentation," https://docs.kraken.com/rest/, 2024.

[30] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in isabelle/hol," in *Proceedings of the 7th ACM SIGPLAN international conference on certified programs and proofs*, 2018, pp. 66–77.