Zellic

Econia

Smart Contract Security Assessment

January 5, 2023

Prepared for:

Econia Labs

Prepared by:

Filippo Cremonese and Aaron Esau

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Econia Labs from November 30th to December 23rd, 2022. During this engagement, Zellic reviewed Econia's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are user assets deposited to Econia safe from on-chain attacks?
- Are the econia and integrator fees safe from on-chain attacks?
- Is the order book secured against denial-of-service (DOS) attacks?
- Is the matching engine fair to all market participants?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- problems due to improper key custody or off-chain access control
- issues stemming from code or infrastructure outside of the assessment scope

1.3 Results

During our assessment on the scoped Econia contracts, we discovered three findings. One issue is classified as critical. Of the remaining two findings, one is of low impact, and the other is informational in nature. The critical severity issue allowed an attacker to evict all orders from one or both sides of the order book of any market, opening multiple profitable attack scenarios depending on the application built on the order book.

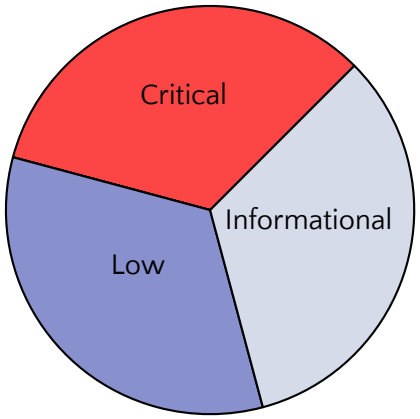
Additionally, Zellic recorded its notes and observations from the assessment for Econia Labs's benefit in the Discussion section (4) at the end of the document.

We want to praise the Econia development team for maintaining a very high quality codebase, with clear code, extensive testing, and outstanding documentation as well as for their responsiveness during the engagement, which significantly eased the audit

process.

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	0
Medium	0
Low	1
Informational	1



2 Introduction

2.1 About Econia

Econia is a fully on-chain order book for the Aptos blockchain.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign

it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Econia Modules

Repository	https://github.com/econia-labs/econia
Versions	`c2c6129940d9b458231a5325fd6ed8f634b6541d`
Programs	• econia::market • econia::avl_queue • econia::registry • econia::resource_account • econia::user • econia::incentives • econia::assets • econia::tablist
Type	Move
Platform	Aptos

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of seven person-weeks. The assessment was conducted over the course of three and a half calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Filippo Cremonese, Engineer
fcremo@zellic.io

Aaron Esau, Engineer
aaron@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

November 30, 2022	Kick-off call
November 30, 2022	Start of primary review period
December 23, 2022	End of primary review period

3 Detailed Findings

3.1 Adversarial order eviction

- **Target**: `econia::avl_queue`
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

Econia's order book is built on an AVL queue. To avoid allowing the data structure to grow too large (incurring excessive gas costs for insertions and deletions), Econia evicts the order with the lowest price-time priority if the AVL queue tree exceeds a critical height.

Critical height checking and eviction occur when inserting a new node using the `insert_check_eviction` or `insert_evict_tail` functions. When placing an order, Econia uses the `insert_check_eviction` function to update the AVL queue, then cancels any evicted orders:

```
// Get new AVL queue access key, evictee access key, and evictee
// value by attempting to insert for given critical height.
let (avlq_access_key, evictee_access_key, evictee_value) =
    avl_queue::insert_check_eviction(
        orders_ref_mut, price, order, critical_height);
// [...]

if (evictee_access_key == NIL) { // If no eviction required:
    // Destroy empty evictee value option.
    option::destroy_none(evictee_value);
} else { // If had to evict order at AVL queue tail:
    // Unpack evicted order, storing fields for event.
    let Order{size, price, user, custodian_id, order_access_key} =
        option::destroy_some(evictee_value);
    // Get price of cancelled order.
    let price_cancel = evictee_access_key & HI_PRICE;
    // Cancel order user-side, storing its market order ID.
    let market_order_id_cancel = user::cancel_order_internal(
        user, market_id, custodian_id, side, price_cancel,
        order_access_key, (NIL as u128));
```

```
    // Emit a maker evict event.
    event::emit_event(&mut order_book_ref_mut.maker_events, MakerEvent{
        market_id, side, market_order_id: market_order_id_cancel, user,
        custodian_id, type: EVICT, size, price});
};
```

The protocol does not take a fee when a user places an order, and orders can be cancelled within the same transaction.

Impact

An attacker can cause legitimate orders to be evicted from the structure, effectively cancelling them. Aptos maximum gas limit allows an attacker to perform the attack in one single transaction, without any risk for the attacker's assets (temporarily required to be deposited to Econia to place the malicious orders).

Aside from the DOS threat for any protocol built on Econia, the vulnerability can have further impacts depending on the protocol. Consider the following examples of protocol types and how they may be impacted by this vulnerability:

- **Decentralized token exchanges:** Attackers can use this vulnerability to manipulate the order book to evict all orders on one side of the book. They could then place orders at arbitrary prices, allowing them to profit from buying or selling assets at an artificial price that does not reflect the market value from unsuspecting users and bots. This would also impact the trading strategies of users who might not expect their orders to be cancelled.
- **Decentralized margin trading protocols:** Attackers can exploit this vulnerability to influence the price of one or more assets, causing margin positions to be liquidated. This scenario is plausible if the margin trading protocol infers asset pricing from order book entries (e.g., using BID or ASK price or mid-market rate) or by looking at the price of recent trade events.
- **Decentralized derivatives markets:** Attackers can place orders with a higher price-time priority than what the derivatives traders have set, allowing them to take advantage of the traders' positions and force them to liquidate at a loss. This can happen if the derivatives market uses the mid-market rate on Econia as the price source, or alternatively, the last trades. This gives the attacker access to the funds that the derivatives trader has deposited in the protocol.
- **Decentralized lending protocols:** Attackers can use this vulnerability to manipulate the order book, allowing them to borrow funds for a lower interest rate than what is actually available on the market. This can happen if the lending protocol uses the mid-market rate on Econia as the price source, or alternatively, the last

trades. This gives them an unfair advantage over legitimate borrowers, allowing them to borrow funds at a much lower rate and thus allowing them to steal funds from the protocol.

Reproduction steps

To perform an attack, an attacker may use the following steps:

1. Place enough limit orders with a higher price-time priority than the target transaction(s) on the same side (BID/ASK), storing each resulting order ID.

   The order size must be valid, but it can be any amount. Each order price must be unique; a new price level must exist for each order. The price must not cross the spread, since the order has to be posted on the book.

   The maximum number of orders required to evict any other order is 2,048 given the critical tree height `CRITICAL_HEIGHT` of 10 at the time of the audit and given that every illegitimate order has a unique price level. Fewer orders may be required when the order book contains legitimate orders at different price levels with a higher price-time priority than the target order.

   Note that this attack may be funded by a flash loan if the attacker does not have sufficient funds to place the malicious orders. Though a flash loan may take a fee, the profit of an attack using this vulnerability will likely exceed any flash loan fee.

2. Cancel all stored order IDs of illegitimate orders. The attacker's funds will be returned without any fee being charged. Limit orders evicted in step 1 remain cancelled.

We note that the worst case scenario from an attacker's perspective is evicting all orders from one side of a very liquid market, where the spread is likely negligible and there's a high concentration of assets near it. It might not be possible to post 2,048 orders at different price levels without crossing the spread in order to evict all orders from one side. This does not prevent the attack, but it will require widening the spread by filling orders on one or both sides of the book, costing some capital. The majority of the capital can likely be recovered, as the orders filled by the attacker are priced near the "correct" market rate of the asset.

Demonstrative test

To demonstrate an attack, we provided the following proof of concept to Econia Labs:

```
#[test(account = @simulation_account)]
fun test_can_cancel_legitimate_order(account: &signer)
acquires OrderBooks, Orders {
    // initialize markets, users, and an integrator.
    let (user_0, user_1) = init_markets_users_integrator_test();
    let user_2 = account::create_account_for_test(@user_2);
    user::register_market_account<BC, QC>(
        &user_2, MARKET_ID_COIN, NO_CUSTODIAN);


    // setup test
    let (taker_divisor, integrator_divisor) =
        (incentives::get_taker_fee_divisor(),
         incentives::get_fee_share_divisor(INTEGRATOR_TIER));
    let price = integrator_divisor * taker_divisor;
    let initial_amount_bc = HI_64/2;
    let initial_amount_qc = HI_64/2;
    user::deposit_coins<BC>(@user_0, MARKET_ID_COIN, NO_CUSTODIAN,
                            assets::mint_test(initial_amount_bc));
    user::deposit_coins<QC>(@user_0, MARKET_ID_COIN, NO_CUSTODIAN,
                            assets::mint_test(initial_amount_qc));
    user::deposit_coins<BC>(@user_1, MARKET_ID_COIN, NO_CUSTODIAN,
                            assets::mint_test(initial_amount_bc));
    user::deposit_coins<QC>(@user_1, MARKET_ID_COIN, NO_CUSTODIAN,
                            assets::mint_test(initial_amount_qc));
    user::deposit_coins<BC>(@user_2, MARKET_ID_COIN, NO_CUSTODIAN,
                            assets::mint_test(initial_amount_bc));
    user::deposit_coins<QC>(@user_2, MARKET_ID_COIN, NO_CUSTODIAN,
                            assets::mint_test(initial_amount_qc));


    // #1: place limit order ASK size*4
    debug::print<u8>(&1);
    let (order_id, _, _, _) = place_limit_order_user<BC, QC>(
        &user_0, MARKET_ID_COIN, @integrator, ASK, MIN_SIZE_COIN*4, price,
        POST_OR_ABORT);
    debug::print(&std::bcs::to_bytes<u128>(&order_id));


    // #2: place limit order BID size (fulfills immediately)
    debug::print<u8>(&2);
```

```
    let (order_id, base_traded, quote_traded, fees) =
    place_limit_order_user<BC, QC>(
        &user_1, MARKET_ID_COIN, @integrator, BID, MIN_SIZE_COIN*1, price,
        FILL_OR_ABORT);
    debug::print(&std::bcs::to_bytes<u128>(&order_id));
    debug::print(&base_traded);
    debug::print(&quote_traded);
    debug::print(&fees);


    // #3: spam orders
    debug::print<u8>(&3);
    let n_orders = 2048;
    let i: u64 = 0;
    let ids: vector<u128> = vector::empty();
    while (i < n_orders) {
        let (order_id, _, _, _) = place_limit_order_user<BC, QC>(
            &user_1, MARKET_ID_COIN, @integrator, ASK, MIN_SIZE_COIN,
    price-i-1,
            POST_OR_ABORT);
        debug::print(&std::bcs::to_bytes<u128>(&order_id));
        vector::push_back(&mut ids, order_id);

        i = i + 1;
    };

    i = 0;
    while (i < n_orders) {
        let order_id = vector::pop_back(&mut ids);
        cancel_order_user(&user_1, MARKET_ID_COIN, ASK, order_id);
        i = i + 1;
    };


    // #4: place market order BUY
    debug::print<u8>(&4);
    let (base_traded, quote_traded, fees) = place_market_order_user<BC,
    QC>(
        &user_2, MARKET_ID_COIN, @integrator, BUY, 0, MAX_POSSIBLE,
        0, MAX_POSSIBLE, price*200);
    debug::print(&base_traded); // should be 0 if #1 was cancelled
    debug::print(&quote_traded); // should be 0 ^
```

```
    debug::print(&fees); // should be 0 ^


    // #5: verify there are no ASK orders left since #1 was evicted
    index_orders_sdk(account, MARKET_ID_COIN); // Index orders.
    let orders = borrow_global<Orders>(@simulation_account);
    assert!(vector::length(&orders.asks) == 0, 0);
}
```

The test places a legitimate order, then places 2048 illegitimate orders, cancels all illegitimate orders, then verifies that the legitimate order was also cancelled.

Recommendations

There are a few potential approaches to lower the risk of attack, though none of the following strategies is a complete solution.

Impose a minimum order size and tick size

This would help deter adversarial behavior by requiring more capital to perform the attack. This approach has the advantage of being relatively straightforward to implement.

The approach does not eliminate the vulnerability for a well-funded attacker (possibly financed via a flash loan); the profits may easily exceed the cost, especially since the attacker recovers their funds when cancelling their order.

Disallowing immediate cancellation of orders

This would require storing a sequence number in each user's order, and it could be imposed either for the current block number (one-block delay required to cancel) or by an account sequence number (a minimum of one transaction delay required to cancel). This strategy has the advantage of making the attack riskier and more costly, as the attacker would now have to wait before being able to cancel the orders and recover their funds; during the delay period, bots may fulfill the high price-level orders.

Though, this approach does not eliminate the attack vector because an attacker can still exploit the vulnerability over multiple transactions. Additionally, this strategy may be problematic for market makers, as it would prevent them from quickly cancelling orders.

Increase the critical height for eviction

This would make it so that clearing out the order book would take more than 16k deletions from the AVL queue, which cannot be done in a single transaction given the current maximum per-transaction gas limit on Aptos. This strategy has the advantage of being relatively straightforward to implement and could potentially deter a malicious actor.

However, it may lead to higher gas costs because Econia must traverse a tree with potentially more price levels. Increasing the maximum size of the order book may also introduce a DOS attack vector where an attacker places many small orders to cause the AVL queue to grow to the point where it is not practical to place orders because of gas fees (or where it is impossible because of the per-transaction gas limit).

Remediation

Econia Labs acknowledged this finding and created a GitHub issue to discuss remediations.

They provided the following response to our proof of concept:

> When we were designing the AVL queue we imposed the critical height constraints basically to prevent the data structure from getting too large: the more tree nodes, the more it costs to insert or delete. Hence the eviction schema, which is supposed to solve a different DOS attack vector: placing a bunch of small orders that grow the tree grows too large and eats up gas. Per the eviction approach, if someone places an order far away from the spread, they risk getting evicted if someone comes along with a better order. But as you demonstrated, this approach could lead to adversarial behavior.

The critical height of the AVL queue (`econia::market::CRITICAL_HEIGHT`) was increased from 10 to 18 in commit `9b3cada1`. This makes it harder for an attacker to exploit this issue without risk, but it does not completely eliminate the issue.

The increased critical height requires 16,383 orders to be inserted in order to guarantee fully evicting all the orders on one side of the book (per the calculations provided by the Econia team). This is impossible to accomplish in a single transaction due to the current max compute limit in place on Aptos.

However, the attack is still viable if an attacker accepts the risk of some of their malicious orders being filled. Note that the capital cost of inserting many orders is not necessarily high and varies on the minimum order size for the market and the price level at which the orders are inserted.

Additionally, an attacker might not be interested in evicting one side of the order book as a whole but might still find advantageous to evict the tail of the orders. In general, an attacker has the ability to choose a price cutoff and evict all orders that have a price that is worse (lower or higher, depending on the side being attacked) by posting malicious orders with a better price. This price does not have to be the best price for the chosen side of the order book, but it can instead be in between other better priced orders (which will not be evicted) and the victim ones. This makes it possible to evict the majority of the orders on the book without significant risk even without doing so in a single transaction, since the malicious orders will not be filled unless all the other better priced ones are filled first.

Econia Labs provided the following notes:

> We have added a section to our documentation about the topic and how to avoid making erroneous assumptions as an integrating protocol: https://econia.dev/overview/orders#adversarial-considerations

> We have started looking into a B+ tree (per discussions with @fcremo) that is unaffected by eviction behavior, and are considering it as an upgradeable feature pending more research: https://github.com/econia-labs/econia/issues/62

3.2 Cancelling nonexistent market order IDs aborts

- **Target**: `econia::market`
- **Category**: Coding Mistakes
- **Likelihood**: Informational
- **Severity**: Informational
- **Impact**: Informational

Description

Cancelling a market order ID that does not exist in the AVL queue ungracefully aborts with the following error:

```
native fun borrow_box_mut<K: copy + drop, V, B>(table: &mut Table<K, V>,
    key: K): &mut Box<V>;
           ^^^^^^^^^^^^^
           |
           Test was not expected to abort but it aborted with 25863 here
           In this function in 0x1::table
```

To reproduce this issue, use the following test:

```
#[test]
fun test_nonexistent_market_order_id()
acquires OrderBooks {
    let (_, user_1) = init_markets_users_integrator_test();
    let nonexistent_market_order_id = 0xdeadbeef;
    cancel_order_user(&user_1, MARKET_ID_COIN, ASK,
    nonexistent_market_order_id);
}
```

The function call chain to the offending `borrow_box_mut` call is

- `market::cancel_order`
- `avl_queue::remove`
- `avl_queue::remove_list_node`
- `avl_queue::remove_list_node_update_edges`

The `borrow_mut` line below causes the transaction to abort:

```
} else { // If node was not list head:
    // Mutably borrow last list node.
```

```
    let list_node_ref_mut = table_with_length::borrow_mut(
        list_nodes_ref_mut, last_node_id);
```

Impact

It may be more difficult for developers building on Econia to debug code cancelling a nonexistent market order ID.

Recommendations

Assert that the market order ID exists, or otherwise, gracefully exit if the node is not found in the AVL queue.

Remediation

Econia remediated the issue in commit `7549fef`.

3.3 Incorrect check for minimum order size

- **Target**: `econia::market::place_limit_order`
- **Category**: Coding Mistakes
- **Severity**: Low
- **Likelihood**: High
- **Impact**: Low

Description

After performing the order book match to attempt to fill a new limit order, the `place_limit_order` function returns early without placing an order for the remaining unfilled size only if the order is of type `IMMEDIATE_OR_CANCEL` or if the remaining order size is zero.

This is incorrect, as the remaining order size can be lower than the minimum order size configured for the market.

```
fun place_limit_order<
    BaseType,
    QuoteType,
>(
    user_address: address,
    market_id: u64,
    custodian_id: u64,
    integrator: address,
    side: bool,
    size: u64,
    price: u64,
    restriction: u8,
    critical_height: u8
): (
    u128,
    u64,
    u64,
    u64
) acquires OrderBooks {
    // [SNIP] removed a major part of the code for legibility, including
    many checks

    // Order crosses spread if an ask and would trail behind bids
    // AVL queue head, or if a bid and would trail behind asks AVL
    // queue head.
    let crosses_spread = if (side == ASK)
```

```
        !avl_queue::would_update_head(&order_book_ref_mut.bids, price)
    else
        !avl_queue::would_update_head(&order_book_ref_mut.asks, price);

    // [SNIP] removed a major part of the code for legibility, including
    many checks

    if (crosses_spread) { // If order price crosses spread:

        // [SNIP] fill order as much as the other side of the book allows

    }; // Done with optional matching as a taker across the spread.

    // Return without market order ID if no size left to fill.
    if ((restriction == IMMEDIATE_OR_CANCEL) || (size == 0))
        return ((NIL as u128), base_traded, quote_traded, fees);
    // [ ... ]
```

This issue was independently reported to Econia while our audit was ongoing (source).

Impact

Order books are likely to eventually get orders posted for less than the minimum configured order size.

Recommendations

Return early without posting an order if the remaining unfilled size is less than the minimum size configured for the market.

Remediation

Econia remediated the issue in commit `562bfea`.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Function inlining

Parts of the Econia codebase – especially the AVL queue module – are written without isolating trivial functionality in separate functions. This is a conscious choice by the developers made to save the cost of call operations, which is deemed too high, making them accept the tradeoff between runtime efficiency and code legibility. The Econia team has already opened an issue about this on the Aptos issue tracker, which has not received attention (source).

This is not entirely a fundamental limitation of the Aptos/Move cost model. Even without changing it, it is clear that the codebase would massively benefit in terms of legibility and maintainability if Move supported macros or an `always_inline` annotation for functions; in agreement with the team, we investigated the feasibility of implementing the `always_inline` feature in the Move compiler, with the ideal goal of upstreaming it for the benefit of the whole community. We set out to answer two main questions:

- How open is Move to community contributions?
- How can inlining be implemented?

How open is Move to community contributions?

This question is fundamental, as the Econia team would refrain from using a forked version of the compiler. We found some PRs, which lead us to believe that the team managing Move is open in principle to accepting community contributions adding language features. These are a couple of examples of PRs that touch some areas related to the work that would be required to implement the inlining feature:

- (https://github.com/move-language/move/pull/714) - propagates all the way to the build artifacts function attributes. We would probably need to introduce a function attribute `always_inline`; note that there is no need to have the attribute in the final artifact, but this shows PRs related to this part of the code can be accepted.
- (https://github.com/move-language/move/pull/420) - implements a new attribute.

We note that the PRs above are simpler than what we would aim for, and we could

not find community PRs making significant changes to the compiler passes.

How can inlining be implemented?

The Move compiler is structured as a pipeline with the following steps:

- parse: parses the textual source into an AST
- expand: filters test-only and verification-only functions depending on the build type; expands module aliases
- naming: performs name resolution
- typing: applies type inference and performs type checking
- lower HLIR: lowers from HLIR to CFGIR and applies optimizations on the CFG (constant folding, inlining BBs with just one predecessor, elimination of unneeded locals)
- lower CFGIR (`to_bytecode`): emits the bytecode understood by the Move VM

Adding support for an `always_inline` attribute requires modifying the parser so that the new keyword would be recognized and implementing a compiler pass that performs the inlining. While modifying the parser is trivial, actually inlining function bodies appears to be harder than we would have expected based on experience with the design of popular compilers such as LLVM.

In our opinion, there are two places where applying inlining would make the most sense:

Inlining right after parsing: This approach should be possible, but not ideal nor trivial, since for example function calls introduce a new scope; all variables declared in the inlined function's body would need to be redeclared along with their usages. In addition, "passing" arguments and return values efficiently could be difficult, as the Move compiler seems to be lacking most of the optimizations that are normally found in general-purpose compilers.

Inlining as a CFGIR optimization pass: This is where we would intuitively like to implement the feature. However, CFGIR optimization passes run on individual functions without access to the surrounding context. This means it is not possible to access the definition or the code of the functions that are called by the `Commands` (the name for instructions in the Move compiler) contained in the various basic blocks. This makes it impossible to inline the code without also changing how the CFGIR pass as a whole is designed.

4.2 Integrator address provided by the user

When an order is filled, the taker is charged a fee. A portion of the taker fee is shared with the integrator that facilitated the trade. The integrator is identified by their address, provided as an argument to the functions for placing orders. This parameter is not validated; therefore, users are trusted not to alter it. This has two implications:

- A user could register themselves as an integrator (the first tier being free) and recoup a part of the fees paid for the orders they take.
- The integrator has no way to guarantee the orders it facilitates will collect fees, as the integrator address is freely specified by the signing user.

We do not consider this a security issue that affects Econia directly but rather a design choice that could affect the incentives for end users and integrators. We note that savvy users do not have to use an integrator at all, as Econia books are freely available to anyone and are not tied to any specific integrator.

4.3 Registering duplicate markets by inverting base/quote

Note that after registering a market with a given base coin and quote coin pair, it is possible to register a market given the same two coins by inverting them (i.e., make the quote coin the base coin and vice versa).

The following test — which should be run in `econia::market` — demonstrates that this is possible:

```
#[test]
fun test_register_markets_duplicate_inverted()
acquires OrderBooks {
    init_test(); // Init for testing.
    let fee = incentives::get_market_registration_fee();
    let user = account::create_account_for_test(@user);
    coin::register<UC>(&user); // Register user coin store.
    coin::deposit<UC>(@user, assets::mint_test(fee*2));

    register_market_base_coin_from_coinstore<BC, QC, UC>(
        &user, LOT_SIZE_COIN, TICK_SIZE_COIN, MIN_SIZE_COIN);
    register_market_base_coin_from_coinstore<QC, BC, UC>(
        &user, LOT_SIZE_COIN, TICK_SIZE_COIN, MIN_SIZE_COIN);
}
```

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm. Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

It's important to note that the Econia test suite includes both explicit and implicit tests. While some function behaviors may not have explicit tests written for them, they may still be checked through internal calls to other functions that do have explicit tests. For example, a function may not have a specific test for coin type mismatches, but it does make an internal call to a function that verifies coin type compatibility and does have an explicit test.

Therefore, the absence of a test coverage check mark in the report does not necessarily mean that a function was not tested at all. Instead, it just indicates that there is no explicit test written specifically for that function.

5.1 Module: `econia::avl_queue`

The AVL queue is a data structure that is a hybrid of an AVL tree and a queue. In Econia, it is used for storing limit orders in an efficient manner.

Function: `insert`

The `insert` function inserts a key-value pair into an AVL queue. Internally, it uses `insert_check_head_tail` to update the AVL queue state.

```
public fun insert<V>(
    avlq_ref_mut: &mut AVLqueue<V>,
    key: u64,
    value: V
)
```

Branches and code coverage

Intended branches:

- Inserts the key-value pair into the AVL queue.
  ☑ Test coverage

Negative behavior:

- The `key` cannot be higher than `HI_INSERTION_KEY`, which is the maximum value of a 32-bit int (leaving other bits free).
  ☑ Negative test

Inputs

- `avlq_ref_mut`:
  - **Control**: None; value is passed from caller when inserting order.
  - **Authorization**: None.
  - **Impact**: The key-value pair will be inserted in this AVL queue.
- `key`:
  - **Control**: Arbitrary, but must be valid.
  - **Authorization**: None.
  - **Impact**: The key that will be inserted.
- `value`:
  - **Control**: Arbitrary, but the use depends on the caller.
  - **Authorization**: None.
  - **Impact**: The value that will be inserted.

Function: `insert_check_eviction`

The `insert_check_eviction` function attempts to insert a key-value pair into the AVL queue, and evicts the tail node if the queue is above a certain critical height or if the maximum number of list nodes have been allocated. It does not guarantee that the height will remain below the critical height post-insertion, nor does it check the number of active tree nodes. Internally, it calls the `insert` function to insert the key-value pair.

```
public fun insert_check_eviction<V>(
    avlq_ref_mut: &mut AVLqueue<V>,
    key: u64,
    value: V,
    critical_height: u8
)
```

Branches and code coverage

Intended branches:

- Inserts the key-value pair into a non-empty AVL queue without evicting.
  ☑ Test coverage
- Inserts the key-value pair into an empty AVL queue without evicting.
  ☑ Test coverage
- Inserts the key-value pair into the AVL queue, evicting the tail.
  ☑ Test coverage
- Attempts to insert the key-value pair into the AVL queue, but insertion is invalid (attempting to insert the tail or critical height exceeded).
  ☑ Test coverage

Negative behavior:

- Attempts to insert the key-value pair with an invalid height (exceeds max height).
  ☑ Test coverage

Inputs

- `avlq_ref_mut`:
  - **Control**: None; value is passed from caller when inserting order.
  - **Authorization**: Must have mutable reference to an AVL queue.
  - **Impact**: The key-value pair will be inserted in this AVL queue.
- `key`:
  - **Control**: Full, but must be valid.
  - **Authorization**: N/A.
  - **Impact**: The key of the key-value pair being inserted.
- `value`:
  - **Control**: Full.
  - **Authorization**: N/A.
  - **Impact**: The value of the key-value pair being inserted.
- `critical_height`:
  - **Control**: Must be less than or equal to the maximum tree height.
  - **Authorization**: N/A.
  - **Impact**: Determines whether or not height-driven eviction is required.

Function: `insert_check_eviction`

This function inserts a key-value insertion pair and evicts the AVL queue tail.

```
public fun insert_evict_tail<V>(
    avlq_ref_mut: &mut AVLqueue<V>,
    key: u64,
    value: V
)
```

Branches and code coverage

Intended branches:

- Inserts the key-value pair into the AVL queue, evicting the tail.
  ☑ Test coverage

Negative behavior:

- Evicting the tail of an empty AVL queue.
  ☑ Test coverage
- Evicting the tail when the key-value pair would become the new tail.
  ☑ Test coverage

Inputs

- `avlq_ref_mut`:
  - **Control**: None; value is passed from caller when inserting order.
  - **Authorization**: Must have mutable reference to an AVL queue.
  - **Impact**: The key-value pair will be inserted in this AVL queue.
- `key`:
  - **Control**: Full, but must be valid.
  - **Authorization**: N/A.
  - **Impact**: The key of the key-value pair being inserted.
- `value`:
  - **Control**: Full.
  - **Authorization**: N/A.
  - **Impact**: The value of the key-value pair being inserted.

Function: `new`

Returns a new AVL queue, optionally allocating inactive nodes.

```
public fun new<V: store>(
    sort_order: bool,
    n_inactive_tree_nodes: u64,
    n_inactive_list_nodes: u64,
)
```

Branches and code coverage

Intended branches:

- Inserts the key-value pair into the AVL queue, evicting the tail.
  ☑ Test coverage

Negative behavior:

- Expects `n_inactive_tree_nodes` to be less than or equal to `N_NODES_MAX`; aborts with `E_TOO_MANY_TREE_NODES` on failure.
  ☑ Test coverage
- Expects `n_inactive_list_nodes` to be less than or equal to `N_NODES_MAX`; aborts with `E_TOO_MANY_LIST_NODES` on failure.
  ☑ Test coverage

Inputs

- `sort_order`:
  - **Control**: The sort order must be either `ASCENDING` or `DESCENDING`
  - **Authorization**: N/A
  - **Impact**: Determines the order in which to store the entries in the AVL queue.
- `n_inactive_tree_nodes`:
  - **Control**: Must be less than or equal to `N_NODES_MAX`
  - **Authorization**: N/A
  - **Impact**: Specifies the number of inactive tree nodes to allocate.
- `n_inactive_list_nodes`:
  - **Control**: Must be less than or equal to `N_NODES_MAX`
  - **Authorization**: N/A
  - **Impact**: Specifies the number of inactive list nodes to allocate.

Function: `remove`

The `pop_head` and `pop_tail` functions internally call the `remove` function, which is also publicly callable.

The `remove` function removes a node having a given access key from an AVL queue and returns its insertion value. It updates the AVL queue's head, tail, and root fields as needed. It assumes the provided access key corresponds to a valid list node in the given AVL queue. Internally, it calls the `remove_tree_node` function.

```
public fun pop_head<V>(
    avlq_ref_mut: &mut AVLqueue<V>
)
public fun pop_tail<V>(
    avlq_ref_mut: &mut AVLqueue<V>
)
public fun remove<V>(
    avlq_ref_mut: &mut AVLqueue<V>,
    access_key: u64
)
```

Branches and code coverage

Intended branches:

- Remove the node with the given access key.
  ☑ Test coverage

Negative behavior:

- Removing a node with an access key that does not correspond to a valid node in the given AVL queue.
  ☐ Test coverage
- Removing an insertion value from the head or tail of an AVL queue should abort if empty.
  ☐ Test coverage

Inputs

- `avlq_ref_mut`:
  - **Control**: None.

  - **Authorization**: A mutable reference to an AVL queue.
  - **Impact**: Used to modify the AVL queue.
- `access_key`:
  - **Control**: None.
  - **Authorization**: N/A.
  - **Impact**: Used to identify the list node to be removed from the AVL queue.

5.2 Module: `econia::incentives`

Function: `upgrade_integrator_fee_store`

This function can be used by integrators to upgrade their tier for a given market.

There is another entry point for reaching this function, `upgrade_integrator_fee_store_via_coinstore`, which withdraws the coins needed for the upgrade from the integrator coin store before calling `upgrade_integrator_fee_store`.

```
public fun upgrade_integrator_fee_store<
    QuoteCoinType,
    UtilityCoinType
>(
    integrator: &signer,
    market_id: u64,
    new_tier: u8,
    utility_coins: coin::Coin<UtilityCoinType>
)
```

Branches and code coverage

Intended branches:

- Deposit utility coins to Econia resource account, store the new tier level
  ☑ Test coverage

Negative behavior:

- tier level too high
  ☑ Negative test
- tier level lower than current one
  ☐ Negative test

- insufficient utility coins provided
  - ☑ Negative test
- signer is not an integrator (not handled explicitly, will abort in `get_cost_to_upgrade_integrator_fee_store`)
  - ☐ Negative test
- invalid market ID (not handled explicitly, will abort in `get_cost_to_upgrade_integrator_fee_store`)
  - ☐ Negative test
- invalid quote coin type (not handled explicitly, will abort in `get_cost_to_upgrade_integrator_fee_store`)
  - ☐ Negative test
- invalid utility coin type
  - ☑ Negative test (indirectly, tested for `deposit_utility_coins_verified`)

Inputs

- `integrator`:
  - **Control**: None, it is the signer of the transaction
  - **Authorization**: N/A
  - **Impact**: The integrator upgrading their tier
- `market_id`:
  - **Control**: Arbitrary, but must be valid
  - **Authorization**: Must be a market on which the integrator has a fee store
  - **Impact**: Identifies the market
- `new_tier`:
  - **Control**: Must be a valid tier higher than the current one
  - **Authorization**: None
  - **Impact**: Tier the integrator will be upgraded to
- `utility_coins`:
  - **Control**: Must be enough to cover the upgrade
  - **Authorization**: None
  - **Impact**: Coins to cover the upgrade fee

Function: `withdraw_integrator_fee`

This function can be used to withdraw integrator fees for a given market; integrator fees come from taking a share of the taker fees on orders coming from the integrator.

It can be reached also through the related function `withdraw_integrator_fees_via_coinstores`, which uses the integrator coinstore to withdraw the utility coins to pay for

the withdrawal fee and to deposit the integrator fees.

```
public fun withdraw_integrator_fees<
    QuoteCoinType,
    UtilityCoinType
>(
    integrator: &signer,
    market_id: u64,
    utility_coins: coin::Coin<UtilityCoinType>
)
```

Branches and code coverage

Intended branches:

- deposits the utility coins for the withdrawal fee, withdraws all the coins in the integrator fee coin store for the market
  ☑ Test coverage

Negative behavior:

- invalid market ID (not handled, will abort)
  ☐ Negative test
- incorrect type for utility coins
  ☑ Negative test (covered by tests for `deposit_utility_coins_verified`)
- insufficient amount of utility coins
  ☐ Negative test
- incorrect type for quote coins (not handled, will abort)
  ☐ Negative test

Inputs

- `integrator`:
  - **Control**: None
  - **Authorization**: It is the signer of the transaction
  - **Impact**: Integrator withdrawing the fees
- `market_id`:
  - **Control**: Must be a market where the integrator is registered
  - **Authorization**: None

  - **Impact**: Identifies the market from which the fees are withdrawn
- `utility_coins`:
  - **Control**: Must be enough to cover the withdrawal fee
  - **Authorization**: None
  - **Impact**: Used to pay the withdrawal fee

Function: `withdraw_econia_fees_internal`

This function withdraws taker fees collected by the order book and can be reached from four public entry points:

- `withdraw_econia_fees`
- `withdraw_econia_fees_all`
- `withdraw_econia_fees_all_to_coin_store`
- `withdraw_econia_fees_to_coin_store`

As the names imply, the entry points allow to withdraw some specific amount or all the fees collected for a given market, either by directly returning the extracted coins or by depositing them to a coin store.

```
fun withdraw_econia_fees_internal<QuoteCoinType>(
    account: &signer,
    market_id: u64,
    all: bool,
    amount: u64
)
```

Branches and code coverage

Intended branches:

- `all` is true: borrow the fee store for the fee account address, take the coins for the market ID given as argument, and call `coin::extract_all` to return them all
  ☑ Test coverage (the public callers are tested)
- `all` is false: borrow the fee store for the fee account address, take the coins for the market ID given as argument, and call `coin::extract` to return just the specified `amount`
  ☑ Test coverage (the public callers are tested)

Negative behavior:

- signer is not Econia
  - ☑ Negative test (covered by tests for the public callers, e.g., `test_withdraw_econia_fees_not_econia`)
- invalid market ID
  - ☐ Negative test

Inputs

- `account`:
  - **Control**: Must be the Econia hardcoded account
  - **Authorization**: Must sign the transaction
  - **Impact**: N/A
- `market_id`:
  - **Control**: Must be a valid market (not explicitly checked, will abort if invalid)
  - **Authorization**: None
  - **Impact**: Specifies which market fees to withdraw
- `all`:
  - **Control**: The callers control this
  - **Authorization**: N/A
  - **Impact**: If true, withdraw all fee balance, otherwise use `amount`
- `amount`:
  - **Control**: Arbitrary, but must be less than or equal to the available fee balance
  - **Authorization**: None
  - **Impact**: If `all` is false, withdraw this specific amount

Function: `withdraw_utility_coins_internal`

This function withdraws the utility coins paid as fees to Econia to perform certain operations, like registering a market.

It can be reached through one of the following entry points:

- `withdraw_utility_coins`
- `withdraw_utility_coins_all`
- `withdraw_utility_coins_all_to_coin_store`
- `withdraw_utility_coins_to_coin_store`

As the names imply, the four entry points allow to withdraw a specified amount or all the utility coins, by returning them directly or by depositing them to a coin store.

```
fun withdraw_utility_coins_internal<UtilityCoinType>(
    account: &signer,
    all: bool,
    amount: u64
)
```

Branches and code coverage

Intended branches:

- if `all` is true: get a reference to the coin store, and call `coin::extract_all` to return all the coins
  - ☑ Test coverage
- if `all` is false: get a reference to the coin store, and call `coin::extract` to return the specified `amount` of coins
  - ☑ Test coverage

Negative behavior:

- account is not Econia
  - ☑ Negative test (tested in callers, e.g., `test_withdraw_utility_coins_not_econia`)
- incorrect utility coin type
  - ☐ Negative test
- amount requested too big
  - ☐ Negative test

Inputs

- `account`:
  - **Control**: N/A
  - **Authorization**: Must have signed the transaction and be the hardcoded Econia address
  - **Impact**: Used for authorization
- `all`:
  - **Control**: None, controlled by the callers
  - **Authorization**: None
  - **Impact**: If true, all the utility coins balance is withdrawn
- `amount`:

- **Control**: Arbitrary
- **Authorization**: None
- **Impact**: If `all` is false, this amount of coins is extracted

Function: `set_incentive_parameters`

This private function can be reached from the public entry point `update_incentives`, which just forwards its arguments to the private one.

```
fun set_incentive_parameters<UtilityCoinType>(
    econia: &signer,
    market_registration_fee: u64,
    underwriter_registration_fee: u64,
    custodian_registration_fee: u64,
    taker_fee_divisor: u64,
    integrator_fee_store_tiers_ref: &vector<vector<u64>>,
    updating: bool
)
```

Branches and code coverage

Intended branches:

The function has common code paths to initialize and move to the Econia account an `IncentiveParameters` instance. Additionally,

- if `updating` is true: ensure the number of new tiers is greater or equal to the number of existing tiers
  - ☑ Test coverage (covered by tests for `update_incentives`)

Negative behavior:

- number of new tiers lower than current one
  - ☑ Negative test (covered by tests for the caller, `test_update_incentives_fewer_tiers`)
- `econia` argument is not actually econia
  - ☑ Negative test (covered by tests for `set_incentive_parameters_range_check_inputs`)
- minimum market, underwriter, or custodian registration fees too low
  - ☑ Negative test (covered by tests for `set_incentive_parameters_range_check_inputs`)

- taker fee divisor too low
  - ☑ Negative test (covered by tests for `set_incentive_parameters_range_check_inputs`)
- zero integrator fee tiers
  - ☑ Negative test (covered by tests for `set_incentive_parameters_range_check_inputs`)
- too many integrator fee tiers
  - ☑ Negative test (covered by tests for `set_incentive_parameters_range_check_inputs`)

Inputs

- `econia`:
  - **Control**: None, must be the signer
  - **Authorization**: Must be the Econia address
  - **Impact**: Used for authorization
- `market_registration_fee`:
  - **Control**: Must be higher than `MIN_FEE`
  - **Authorization**: None
  - **Impact**: Market registration fee
- `underwriter_registration_fee`:
  - **Control**: Must be higher than `MIN_FEE`
  - **Authorization**: None
  - **Impact**: Underwriter registration fee
- `custodian_registration_fee`:
  - **Control**: Must be higher than `MIN_FEE`
  - **Authorization**: None
  - **Impact**: Custodian registration fee
- `taker_fee_divisor`:
  - **Control**: Must be greater than `MIN_DIVISOR`
  - **Authorization**: None
  - **Impact**: Taker fee divisor, used to determine market taker fees
- `integrator_fee_store_tiers`:
  - **Control**: Must be a vector of vectors containing three elements each, determining the integrator tier parameters
  - **Authorization**: None
  - **Impact**: Configures the integrator tiers
- `updating`:
  - **Control**: None (set by the caller, always true when coming from `update_in`

centives)
- **Authorization**: N/A
- **Impact**: If `true`, additional checks are performed

5.3 Module: `econia::market`

This module provides entry points for performing all the common market-related operations: placing, changing, and cancelling orders as well as registering a new market.

Many core features are exposed by two different public entry points intended to be used by the user or a designated custodian. The public functions are just simple wrappers that invoke an internal function with appropriately populated parameters. Therefore, in those cases functions are grouped by functionality.

Function: `cancel_order`

The functions `cancel_order_custodian` and `cancel_order_user` invoke `cancel_order`, which can be used to remove one specific active order.

```
fun cancel_order(
    user: address,
    market_id: u64,
    custodian_id: u64,
    side: bool,
    market_order_id: u128
)
```

Branches and code coverage

Intended branches:

- cancel the given order
  ☑ Test coverage

Negative behavior:

- inexistent or inconsistent user, market ID, custodian ID, market order ID
  ☑ Negative test

Inputs

- `user`:
  - **Control**: N/A
  - **Authorization**: Must be a signer, if entering from `cancel_order_user`
  - **Impact**: Address of the user whose orders will be cancelled
- `market_id`:
  - **Control**: Arbitrary, but must be valid
  - **Authorization**: None
  - **Impact**: The orders for this market will be cancelled
- `custodian_id`:
  - **Control**: None
  - **Authorization**: ID of a valid custodian if entering from `cancel_order_custodian`, else `NO_CUSTODIAN`
  - **Impact**: N/A
- `side`:
  - **Control**: BID or ASK
  - **Authorization**: None
  - **Impact**: Determines whether BID or ASK orders will be cancelled
- `market_order_id`:
  - **Control**: Must be an order belonging to the provided user, market, and side
  - **Authorization**: None
  - **Impact**: ID of the order to be cancelled

Function: `cancel_all_orders`

The functions `cancel_all_orders_custodian` and `cancel_all_orders_user` invoke `cancel_all_orders`, which can be used to remove all the ASK or BID orders of the invoking (or delegating) user.

```
fun cancel_all_orders(
    user: address,
    market_id: u64,
    custodian_id: u64,
    side: bool
)
```

Branches and code coverage

Intended branches:

- get the IDs of the active orders, and cancels them individually in a loop by calling `cancel_order`
  ☑ Test coverage

Negative behavior:

- inexistent user, market ID, custodian ID
  ☑ Negative test (indirectly, covered by tests for `cancel_order`)

Inputs

- `user`:
  - **Control**: N/A
  - **Authorization**: Must be a signer, if entering from `cancel_all_orders_user`
  - **Impact**: Address of the user whose orders will be cancelled
- `market_id`:
  - **Control**: Arbitrary, but must be valid
  - **Authorization**: None
  - **Impact**: The orders for this market will be cancelled
- `custodian_id`:
  - **Control**: None
  - **Authorization**: Always `NO_CUSTODIAN` or the ID of a valid custodian capability
  - **Impact**: N/A
- `side`:
  - **Control**: BID or ASK
  - **Authorization**: None
  - **Impact**: Determines whether BID or ASK orders will be cancelled

Function: `change_order_size`

The functions `change_order_size_user` and `change_order_size_custodian` are public entry points for the private function `change_order_size`. The private function changes the size on the order entry appearing on the global order book. It calls `user::change_order_size_internal` to apply the change to the user-side entries.

```
fun change_order_size(
    user: address,
    market_id: u64,
    custodian_id: u64,
    side: bool,
    market_order_id: u128,
    new_size: u64
)
```

Branches and code coverage

Intended branches:

- perform sanity checks, call `user::change_order_size_internal` to change the user-side order entry, change order size on market-wide entry
  - ☑ Test coverage

Negative behavior:

- mismatching user (order not belonging to user)
  - ☑ Negative test
- invalid market ID
  - ☑ Negative test
- invalid order ID (== NIL)
  - ☑ Negative test
- mismatching custodian
  - ☑ Negative test

Inputs

- `user`:
  - **Control**: Arbitrary if coming from `change_order_size_custodian`, but must be the user associated with the order
  - **Authorization**: Signer's address if entering from `change_order_size_user`, otherwise arbitrary but must have authorized the custodian
  - **Impact**: N/A
- `market_id`:
  - **Control**: Must be the market on which the order was placed
  - **Authorization**: The market has to belong to the user

  - **Impact**: N/A
- `custodian_id`:
  - **Control**: None, provided by the callers by unpacking a capability
  - **Authorization**: Must be authorized for the given user
  - **Impact**: N/A
- `side`:
  - **Control**: Can either be BID or ASK
  - **Authorization**: N/A
  - **Impact**: Determines the side on which the order to change resides
- `market_order_id`:
  - **Control**: Arbitrary, but must be an order belonging to the user
  - **Authorization**: Must be an order belonging to the user
  - **Impact**: Determines which order will be changed
- `new_size`:
  - **Control**: Arbitrary, but restricted by user funds
  - **Authorization**: None
  - **Impact**: Determines the new order size

Function: `place_limit_order`

This function is the common private function reachable by `place_limit_order_custodian`, `place_limit_order_user`, and `place_limit_order_user_entry`, responsible for coordinating all that is necessary to place a limit order.

```
fun place_limit_order<
    BaseType,
    QuoteType,
>(
    user_address: address,
    market_id: u64,
    custodian_id: u64,
    integrator: address,
    side: bool,
    size: u64,
    price: u64,
    restriction: u8,
    critical_height: u8
)
```

Branches and code coverage

Intended branches:

- restriction: `NO_RESTRICTION`
  - ☑ Test coverage
- restriction: `FILL_OR_ABORT`
  - ☑ Test coverage
- restriction: `IMMEDIATE_OR_CANCEL`
  - ☑ Test coverage
- restriction: `POST_OR_ABORT`

☑ Test coverage

- order: crosses spread as ask
  - ☑ Test coverage
- order: crosses spread as ask, keeps crossing spread after matching
  - ☑ Test coverage
- order: does not cross spread as ask
  - ☑ Test coverage
- order: crosses spread as bid
  - ☑ Test coverage
- order: crosses spread as bid, keeps crossing spread after matching
  - ☑ Test coverage
- order: does not cross spread as bid

☑ Test coverage

- eviction: required
  - ☑ Test coverage
- eviction: not required
  - ☑ Test coverage (implied by all other tests that do not evict)

Negative behavior:

- Invalid (too big/too small/disallowed) values for size, price, restriction
  - ☑ Negative test
- Invalid type for base or quote assets
  - ☑ Negative test
- `FILL_OR_ABORT` order does not cross spread
  - ☑ Negative test
- `FILL_OR_ABORT` order can only be partially filled
  - ☑ Negative test

- `POST_OR_ABORT` order crosses spread
  ☑ Negative test
- AVLQ full and price too bad to evict another order
  ☑ Negative test

Inputs

- `user_address`:
  - **Control**: Arbitrary if coming from `place_limit_order_custodian`, but must match the custodian ID
  - **Authorization**: Signer, if coming from `place_limit_order_user`
  - **Impact**: User placing the order (directly or through a custodian)
- `market_id`:
  - **Control**: Arbitrary
  - **Authorization**: None
  - **Impact**: Determines the market where the order is to be placed
- `custodian_id`:
  - **Control**: None (provided by the callers by unpacking a capability)
  - **Authorization**: Must belong to a custodian authorized for the user
  - **Impact**: ID associated with the custodian capability
- `integrator`:
  - **Control**: Arbitrary
  - **Authorization**: None
  - **Impact**: ID of the integrator facilitating the order
- `side`:
  - **Control**: Either `BID` or `ASK`
  - **Authorization**: None
  - **Impact**: Determines the side of the book where the order is placed
- `size`:
  - **Control**: Arbitrary
  - **Authorization**: None
  - **Impact**: Determines the size of the order
- `price`:
  - **Control**: Arbitrary
  - **Authorization**: None
  - **Impact**: Determines the price of the order
- `restriction`:
  - **Control**: Either `NO_RESTRICTION`, `FILL_OR_ABORT`, `IMMEDIATE_OR_CANCEL` or `PO`

ST_OR_ABORT
  - **Authorization**: None
  - **Impact**: Determines the restriction to be applied on the inserted order
- `critical_height`:
  - **Control**: None (controlled by callers)
  - **Authorization**: None
  - **Impact**: Determines the max AVLQ height; when reached, lesser priority orders are evicted

Function: `place_market_order`

The public functions `place_market_order_custodian`, `place_market_order_user` and `place_market_order_user_entry` invoke the common private function `place_market_order` to place and fill a market order.

```
fun place_market_order<
    BaseType,
    QuoteType
>(
    user_address: address,
    market_id: u64,
    custodian_id: u64,
    integrator: address,
    direction: bool,
    min_base: u64,
    max_base: u64,
    min_quote: u64,
    max_quote: u64,
    limit_price: u64,
)
```

Branches and code coverage

Intended branches:

- direction: buy
  ☑ Test coverage
- direction: sell

☑ Test coverage

- base: max possible
  - ☑ Test coverage
- base: fixed limit

☑ Test coverage

- quote: max possible
  - ☑ Test coverage
- quote: fixed limit
  - ☑ Test coverage

Negative behavior:

- Invalid base or quote asset type
  - ☑ Negative test
- Invalid `min_base`, `max_base`, `min_quote`, `max_quote` values
  - ☑ Negative test (indirectly, by tests for `range_check_trade` and `match`)
- mismatching custodian ID
  - ☐ Negative test

Inputs

- `user_address`:
  - **Control**: Arbitrary if coming from `place_market_order_custodian`, but must match custodian ID
  - **Authorization**: Be an address that authorized the custodian
  - **Impact**: Determines the user placing the order
- `market_id`:
  - **Control**: Arbitrary, but the user must have a corresponding market account
  - **Authorization**: None
  - **Impact**: Identifies the market where the order is placed
- `custodian_id`:
  - **Control**: None (provided by the callers by unpacking a capability)
  - **Authorization**: Must be authorized for the user, if provided
  - **Impact**: Identifies the custodian placing the order
- `integrator`:
  - **Control**: Arbitrary
  - **Authorization**: None
  - **Impact**: Identifies the integrator receiving a share of the taker fees
- `direction`:
  - **Control**: Either `BUY` or `SELL`

  - **Authorization**: None
  - **Impact**: Determines the side of the order
- `min_base`:
  - **Control**: Arbitrary, must be less than `max_base`
  - **Authorization**: None
  - **Impact**: Minimum amount of base assets to be traded
- `max_base`:
  - **Control**: Arbitrary, must be greater than zero and `min_base`, must not overflow 2^64 when summed to the user's `base_ceiling` if direction is BUY, must be less than available assets if direction is BUY
  - **Authorization**: None
  - **Impact**: Maximum amount of base assets to be traded
- `min_quote`:
  - **Control**: Arbitrary, must be less than `max_quote`
  - **Authorization**: None
  - **Impact**: Minimum amount of quote assets to be traded
- `max_quote`:
  - **Control**: Arbitrary, must be greater than zero and `min_quote`, must not overflow 2^64 when summed to the user's `base_ceiling` if direction is SELL, must be less than available assets if direction is SELL
  - **Authorization**: None
  - **Impact**: Maximum amount of quoted assets to be traded
- `limit_price`:
  - **Control**: Must be less than 2^32
  - **Authorization**: None
  - **Impact**: Determines the limit price for the order

Function: `register_market`

The public functions `register_market_base_coin` and `register_market_base_generic` first invoke, respectively, `registry::register_market_base_internal` and `registry::register_market_base_generic_internal`, registering the new market in the global registry and obtaining a market ID. Then they invoke `register_market` to create the order book and the coin store for the fees. A third public function `register_market_base_coin_from_coinstore` is also a utility function that withdraws funds from a user's coin store before invoking `register_market_base_coin`.

```
fun register_market<
    BaseType,
```

```
    QuoteType
>(
    market_id: u64,
    base_name_generic: String,
    lot_size: u64,
    tick_size: u64,
    min_size: u64,
    underwriter_id: u64
)
```

Branches and code coverage

Intended branches:

- add market to the global order book map, register a new coin store for fees
  - ☑ Test coverage

Negative behavior:

- Invalid `lot_size`, `tick_size`, `min_size`
  - ☑ Negative test (indirectly, through tests for `registry::register_market_internal`)
- Invalid quote of base type
  - ☑ Negative test (indirectly, through tests for `registry::register_market_internal`)
- Invalid `base_name_generic`
  - ☑ Negative test (indirectly, through tests for `registry::register_market_base_generic_internal`)
- Insufficient or incorrect type of utility coins (supplied to the callers)
  - ☑ Negative test (indirectly, through tests for `incentives::deposit_market_registration_utility_coins`)

Inputs

- `market_id`:
  - **Control**: None, provided by the callers
  - **Authorization**: N/A
  - **Impact**: ID of the newly registered market
- `base_name_generic`:
  - **Control**: Arbitrary if coming from `register_market_base_generic`

  - **Authorization**: None
  - **Impact**: Only cosmetic; name of the generic base asset
- `lot_size`:
  - **Control**: Arbitrary, greater than zero (checked indirectly by callers through `registry::register_market_internal`)
  - **Authorization**: N/A
  - **Impact**: Lot size of the new market
- `tick_size`:
  - **Control**: Arbitrary, greater than zero (checked indirectly by callers through `registry::register_market_internal`)
  - **Authorization**: N/A
  - **Impact**: Tick size of the new market
- `min_size`:
  - **Control**: Arbitrary, greater than zero (checked indirectly by callers through `registry::register_market_internal`)
  - **Authorization**: N/A
  - **Impact**: Min size of the orders on the new market
- `underwriter_id`:
  - **Control**: None (nonzero only if coming from `register_market_base_generic`, obtained from unpacking a capability)
  - **Authorization**: N/A
  - **Impact**: ID identifying the underwriter for the generic market

Function: `swap`

The `swap` function is a thin wrapper around `match`, which validates some inputs (market ID, underwriter ID, base and quote asset types) and obtains references to the order book on which the swap takes place. It can be reached from multiple entry points:

- `swap_between_coinstores/swap_between_coinstores_entry`
- `swap_coins`
- `swap_generic`

The callers use it essentially to place the equivalent of a fill or abort limit order for the specified minimum quantities of assets (base or quote, depending on direction).

```
fun swap<
    BaseType,
    QuoteType
>(
```

```
    market_id: u64,
    underwriter_id: u64,
    taker: address,
    integrator: address,
    direction: bool,
    min_base: u64,
    max_base: u64,
    min_quote: u64,
    max_quote: u64,
    limit_price: u64,
    optional_base_coins: Option<Coin<BaseType>>,
    quote_coins: Coin<QuoteType>
)
```

Branches and code coverage

Intended branches:

- validate market ID, underwriter ID, base and quote asset types, then call `match`
  ☑ Test coverage

Negative behavior:

- Invalid market ID
  ☑ Negative test
- Invalid base or quote type
  ☑ Negative test
- Invalid underwriter ID
  ☑ Negative test

Inputs

- `market_id`:
  - **Control**: Must be consistent with underwriter, base type, quote type
  - **Authorization**: None
  - **Impact**: Identifies the market where the swap takes place
- `underwriter_id`:
  - **Control**: None; only != from `NO_UNDERWRITER` if coming from `swap_generic`
  - **Authorization**: ID from an unpacked capability; must be the underwriter for the market

  - **Impact**: Identifies the underwriter for the market
- `taker`:
  - **Control**: None; address of the user placing the order, populated by the callers. Either the signer of the transaction (if coming from `swap_between_coinstores`), otherwise `UNKNOWN_TAKER`
  - **Authorization**: None
  - **Impact**: Identifies the user placing the order, used to avoid self-matching
- `integrator`:
  - **Control**: Arbitrary
  - **Authorization**: None
  - **Impact**: Address of the integrator taking a share of the taker fees
- `direction`:
  - **Control**: Either `BUY` or `SELL`
  - **Authorization**: None
  - **Impact**: Determines the direction of the trade
- `min_base`:
  - **Control**: Arbitrary, must be less than `max_base`
  - **Authorization**: None
  - **Impact**: Determines the minimum amount of base asset to be traded
- `max_base`:
  - **Control**: Arbitrary, but bounds checked not to cause overflows and not overspend availability (if direction is BUY)
  - **Authorization**: None
  - **Impact**: Determines the maximum amount of base asset to be traded
- `min_quote`:
  - **Control**: Arbitrary, must be less than `max_quote`
  - **Authorization**: None
  - **Impact**: Determines the minimum amount of quote asset to be traded
- `max_quote`:
  - **Control**: Arbitrary, but bounds checked not to cause overflows and not overspend availability (if direction is SELL)
  - **Authorization**: None
  - **Impact**: Determines the maximum amount of quote asset to be traded
- `limit_price`:
  - **Control**: Arbitrary, up to 2^32
  - **Authorization**: None
  - **Impact**: Limit price for the swap
- `optional_base_coins`:
  - **Control**: Both type and amount are validated if user supplied

  - **Authorization**: None
  - **Impact**: Base coins to be traded (if market is not generic)
- `quote_coins`:
  - **Control**: Both type and amount are validated if user supplied
  - **Authorization**: None
  - **Impact**: Quote coins to be traded

Function: `match`

Arguably the most important and complex function of the market module, `match` performs the match between the bid and ask sides of a trade. It is invoked, directly or indirectly, by all functions placing orders (limit, market, or swaps).

```
fun match<
    BaseType,
    QuoteType
>(
    market_id: u64,
    order_book_ref_mut: &mut OrderBook,
    taker: address,
    integrator: address,
    direction: bool,
    min_base: u64,
    max_base: u64,
    min_quote: u64,
    max_quote: u64,
    limit_price: u64,
    optional_base_coins: Option<Coin<BaseType>>,
    quote_coins: Coin<QuoteType>,
)
```

Branches and code coverage

Intended branches:

This function has a particularly complex behavior, with multiple branches assigning variables affecting the final behavior in nontrivial ways, (e.g., depending on the `direction`, the `side`, the orders on the order book and the parameters to be matched against); describing branches in isolation is not possible.

The function is extensively tested, including tests for these cases:

- complete fill as buy
- complete fill as sell
- complete fill of an order, partial fill of subsequent order in the AVLQ
- zero fill (no orders to match against)
- zero fill as buy due to limit price too low
- zero fill as sell due to limit price too low
- incomplete fill (insufficient size to fill the whole order)

Negative behavior:

- order price mismatch between the AVLQ key and order struct (defensive programming, should never happen)
  ☑ Negative test
- price > 2^32
  ☑ Negative test
- self match
  ☑ Negative test
- specified minimum base unable to be filled
  ☑ Negative test
- specified minimum quote unable to be filled
  ☑ Negative test

Inputs

- `market_id`:
  - **Control**: None
  - **Authorization**: None
  - **Impact**: ID of the market where the trade takes place
- `order_book_ref_mut`:
  - **Control**: None (callers trusted obtain the reference to the correct order book)
  - **Authorization**: None
  - **Impact**: The order book where the bid or ask queue for filling the order is taken from
- `taker`:
  - **Control**: None (callers trusted to check this parameter)
  - **Authorization**: Depending on the caller, this address is the signer, or the address of a user who has authorized a custodian, or `UNKNOWN_TAKER`
  - **Impact**: Address of the order taker, used to prevent self-matching

- `integrator`:
  - **Control**: None
  - **Authorization**: None
  - **Impact**: Address of the integrator facilitating the trade, which will receive part of the taker fees
- `direction`:
  - **Control**: Either BUY or SELL
  - **Authorization**: None
  - **Impact**: Determines the side (from the perspective of the taker, BUY means buying the base asset filling the asks)
- `min_base`:
  - **Control**: Arbitrary, must be less than `max_base`
  - **Authorization**: None
  - **Impact**: Determines the minimum amount of base asset to be traded
- `max_base`:
  - **Control**: Arbitrary, but bounds checked not to cause overflows and not overspend availability (if direction is BUY)
  - **Authorization**: None
  - **Impact**: Determines the maximum amount of base asset to be traded
- `min_quote`:
  - **Control**: Arbitrary, must be less than `max_quote`
  - **Authorization**: None
  - **Impact**: Determines the minimum amount of quote asset to be traded
- `max_quote`:
  - **Control**: Arbitrary, but bounds checked not to cause overflows and not overspend availability (if direction is SELL)
  - **Authorization**: None
  - **Impact**: Determines the maximum amount of quote asset to be traded
- `limit_price`:
  - **Control**: Arbitrary, up to 2^32
  - **Authorization**: None
  - **Impact**: Limit price for the swap
- `optional_base_coins`:
  - **Control**: Both type and amount are validated if user supplied
  - **Authorization**: None
  - **Impact**: Base coins to be traded (if market is not generic)
- `quote_coins`:
  - **Control**: Both type and amount are validated if user supplied
  - **Authorization**: None

- **Impact**: Quote coins to be traded

5.4 Module: `econia::registry`

The `registry` module contains functions used for registering capabilities.

Function: `register_custodian_capability`

The `register_custodian_capability` function increments the number of registered custodians, takes a fee in utility coins, then returns a new `CustodianCapability` resource with the current number of registered custodians as its custodian ID.

```
public fun register_custodian_capability<UtilityCoinType>(
    utility_coins: Coin<UtilityCoinType>
)
```

Branches and code coverage

Intended branches:

- Returns a `CustodianCapability` resource with a unique `custodian_id` value.
  ☑ Test coverage

Negative behavior:

- The `UtilityCoinType` must be the utility coin type registered in `IncentiveParameters` to `@econia`.
  ☐ Negative test
- The value of the `utility_coins` argument must be sufficient to pay the amount returned by `econia::incentives::get_custodian_registration_fee()`.
  ☐ Negative test

Inputs

- `utility_coins`:
  - **Control**: Type system enforces it is a `Coin<UtilityCoinType>`. Arbitrary coin value, but must be greater than the configured fee.
  - **Authorization**: None.
  - **Impact**: The coins pay for the custodian registration.

- `UtilityCoinType`:
  - **Control**: Must be the utility coin type registered in `IncentiveParameters` to `@econia`.
  - **Authorization**: None.
  - **Impact**: The type used for `Coin<UtilityCoinType>`.

Function: `register_underwriter_capability`

The `register_underwriter_capability` function increments the number of registered underwriters, takes a fee in utility coins, then returns a new `UnderwriterCapability` resource with the current number of registered underwriters as its underwriter ID.

```
public fun register_underwriter_capability<UtilityCoinType>(
    utility_coins: Coin<UtilityCoinType>
)
```

Branches and code coverage

Intended branches:

- Returns a `UnderwriterCapability` resource with a unique `underwriter_id` value.
  ☑ Test coverage

Negative behavior:

- The `UtilityCoinType` must be the utility coin type registered in `IncentiveParameters` to `@econia`.
  ☐ Negative test
- The value of the `utility_coins` argument must be sufficient to pay the amount returned by `econia::incentives::get_underwriter_registration_fee()`.
  ☐ Negative test

Inputs

- `utility_coins`:
  - **Control**: Type system enforces it is a `Coin<UtilityCoinType>`. Arbitrary coin value, but must be greater than the configured fee.
  - **Authorization**: None.
  - **Impact**: The coins pay for the underwriter registration.
- `UtilityCoinType`:

- **Control**: Must be the utility coin type registered in `IncentiveParameters` to `@econia`.
- **Authorization**: None.
- **Impact**: The type used for `Coin<UtilityCoinType>`.

Function: `register_integrator_fee_store`

The `register_integrator_fee_store` function — as the name implies — registers an integrator, charging fees to set the tier level to the specified `tier` argument. The function is publicly callable, but also has public wrapper functions `register_integrator_fee_store_base_tier` and `register_integrator_fee_store_from_coinstore`.

As the name implies, `register_integrator_fee_store_from_coinstore` function allows users to pay from an `aptos_framework::coin::CoinStore`. The `register_integrator_fee_store_base_tier` function does not charge utility coins, but only activates an integrator to the base tier.

```
public entry fun register_integrator_fee_store_base_tier<
    QuoteCoinType,
    UtilityCoinType
>(
    integrator: &signer,
    market_id: u64,
)
public entry fun register_integrator_fee_store_from_coinstore<
    QuoteCoinType,
    UtilityCoinType
>(
    integrator: &signer,
    market_id: u64,
    tier: u8
)
public fun register_integrator_fee_store<
    QuoteCoinType,
    UtilityCoinType
>(
    integrator: &signer,
    market_id: u64,
    tier: u8,
    utility_coins: Coin<UtilityCoinType>
)
```

Branches and code coverage

Intended branches:

- Registering an integrator to tier 0 with zero utility coin (`register_integrator_fee_store_base_tier`)
  - ☑ Test coverage
- Registering an integrator that has previously been registered to a higher tier.
  - ☑ Test coverage

Negative behavior:

- The `market_id` is invalid.
  - ☑ Negative test
- The quote coin type is invalid for the given `market_id`.
  - ☑ Negative test
- An insufficient amount of utility coin is provided in `utility_coins`.
  - ☐ Negative test
- An invalid type is passed in `UtilityCoinType`.
  - ☐ Negative test

Inputs

- `integrator`:
  - **Control**: Full; signer.
  - **Authorization**: None.
  - **Impact**: The integrator to register.
- `market_id`:
  - **Control**: Must be a valid market ID in the global `Registry` on `@econia`.
  - **Authorization**: None.
  - **Impact**: The market ID to register integrator on.
- `tier`:
  - **Control**: Must be a valid tier (less than the vector length of `IncentiveParameters`'s `integrator_fee_store_tiers` on `@econia`).
  - **Authorization**: None.
  - **Impact**: The coins pay for the integrator upgrade.
- `utility_coins`:
  - **Control**: Type system enforces it is a `Coin<UtilityCoinType>`. Arbitrary coin value, but must be greater than the configured fee for the specified tier.
  - **Authorization**: None.

  - **Impact**: The coins pay for the integrator upgrade.
- `UtilityCoinType`:
  - **Control**: Must be the utility coin type registered in `IncentiveParameters` to `@econia`.
  - **Authorization**: None.
  - **Impact**: The type used for `Coin<UtilityCoinType>`.

Function: `set_recognized_market`

The `set_recognized_market` function stores info about the market passed through the `market_id` parameter in the `map` attribute of `RecognizedMarkets` resource on `@econia` to mark a market as "recognized".

```
public entry fun set_recognized_markets(
    account: &signer,
    market_ids: vector<u64>
)
public entry fun set_recognized_market(
    account: &signer,
    market_id: u64
)
```

Branches and code coverage

Intended branches:

- Configures a `market_id` value's `trading_pair` that does not exist yet in the `map`.
  ☑ Test coverage
- Updates a `market_id` value's `trading_pair` if it already exists in the `map`.
  ☑ Test coverage

Negative behavior:

- Expects the caller to be `@econia`.
  ☑ Negative test
- Expects the `market_id` to exist in the `market_id_to_info` map in `Registry` resource on `@econia`.
  ☐ Negative test

Inputs

- `account`:
  - **Control**: Full; signer.
  - **Authorization**: Must be `@econia`.
  - **Impact**: Used to validate that the caller is econia.
- `market_id`:
  - **Control**: Full; any u64 value. Must exist in the `market_id_to_info` map in `Registry` resource on `@econia` (i.e. must be valid).
  - **Authorization**: None.
  - **Impact**: The `market_id` to be registered.

Function: `remove_recognized_market`

The `remove_recognized_market` function removes the `market_id` parameter from the `map` attribute of `RecognizedMarkets` resource on `@econia` to unmark a market as "recognized".

```
public entry fun remove_recognized_markets(
    account: &signer,
    market_ids: vector<u64>
)
public entry fun remove_recognized_market(
    account: &signer,
    market_id: u64
)
```

Branches and code coverage

Intended branches:

- Removes a `market_id` value's `trading_pair` from the `map`.
  ☑ Test coverage

Negative behavior:

- Expects the caller to be `@econia`.
  ☑ Negative test
- Expects the `market_id` to exist in the `market_id_to_info` map in `Registry` resource on `@econia`.
  ☐ Negative test

- Expects the `trading_pair` to exist in the `map` in `RecognizedMarkets` resource on `@econia`.
  ☑ Negative test
- Expects the recognized market ID from the `map` in `RecognizedMarkets` resource on `@econia` to be equal to the `market_id` argument.
  ☑ Negative test

Inputs

- `account`:
  - **Control**: Full; signer.
  - **Authorization**: Must be `@econia`.
  - **Impact**: Used to validate that the caller is econia.
- `market_id`:
  - **Control**: Full; any u64 value. Must exist in the `market_id_to_info` map in `Registry` resource on `@econia` (i.e. must be valid). Must exist in the `map` in `RecognizedMarkets` resource on `@econia`.
  - **Authorization**: None.
  - **Impact**: The `market_id` to be unregistered.

5.5 Module: `econia::user`

The `user` module contains functionality for tracking a user's assets and open orders. It tends to have separate functions to support use of specific coins versus generic assets, and separate functions for use by users versus custodians.

Function: `deposit_asset`

The `deposit_coins` function deposits coins of type `CoinType` into the specified market and custodian accounts. It calls the `deposit_asset` function with the provided `coins`, a `None` option for the asset, and `NO_UNDERWRITER` as the underwriter.

The `deposit_generic_asset` function deposits a generic asset into the specified market and custodian accounts. It calls the `deposit_asset` function with the provided `amount`, a `None` option for the asset, and the underwriter ID specified in the `underwriter_capability_ref` parameter.

The `deposit_from_coinstore` function deposits coins of type `CoinType` into the specified market and custodian accounts from an `aptos_framework::coin::CoinStore`. It calls the `deposit_coins` function with the provided `amount`, withdrawing the coins from the specified `user`'s coin store. The `deposit_coins` function is a wrapper around `depo`

`sit_asset` for depositing coins.

The internal `deposit_asset` function deposits an asset to a user's market account. It updates the asset counts, and also deposits any optional coins as collateral.

```
public fun deposit_coins<
    CoinType
>(
    user_address: address,
    market_id: u64,
    custodian_id: u64,
    coins: Coin<CoinType>
)

public fun deposit_generic_asset(
    user_address: address,
    market_id: u64,
    custodian_id: u64,
    amount: u64,
    underwriter_capability_ref: &UnderwriterCapability
)

fun deposit_asset<
    AssetType
>(
    user_address: address,
    market_id: u64,
    custodian_id: u64,
    amount: u64,
    optional_coins: Option<Coin<AssetType>>,
    underwriter_id: u64
)
```

Branches and code coverage

Intended branches:

- Deposits the asset to the user's account.
  ☑ Test coverage

Negative behavior:

- Expects that a `MarketAccounts` resource exists for the given `user_address`.
  ☑ Test coverage
- Expects that a `MarketAccount` resource exists for the given `market_id` and `custodian_id`.
  ☑ Test coverage
- Expects that the `AssetType` is either the `base_type` or `quote_type` of the `MarketAccount` resource.
  ☑ Test coverage
- Expects that the deposit does not exceed the `base_ceiling`/`quote_ceiling` of the `MarketAccount` resource.
  ☑ Test coverage
- Expects that the underwriter ID is valid for the given `MarketAccount` resource if the `AssetType` is `GenericAsset`.
  ☑ Test coverage
- Expects that the `amount` passed to the function is equal to the `value` of the `Coin` if the `AssetType` is not `GenericAsset`.
  ☑ Test coverage

Inputs

- `user_address`:
  - **Control**: None.
  - **Authorization**: N/A.
  - **Impact**: The address of the user whose market account is being deposited into.
- `market_id`:
  - **Control**: The parameter must be an unsigned 64-bit integer.
  - **Authorization**: N/A.
  - **Impact**: Identifies the specific market account being deposited into.
- `custodian_id`:
  - **Control**: The parameter must be an unsigned 64-bit integer.
  - **Authorization**: N/A.
  - **Impact**: Identifies the specific market account being deposited into.
- `amount`:
  - **Control**: The parameter must be an unsigned 64-bit integer.
  - **Authorization**: N/A.
  - **Impact**: The amount of asset being deposited.
- `optional_coins`:
  - **Control**: The parameter must be an `Option` type containing either `None` or a

`Coin` type matching the `AssetType` type parameter.
  - **Authorization**: N/A.
  - **Impact**: The optional coins being deposited.
- `underwriter_id`:
  - **Control**: The parameter must be an unsigned 64-bit integer.
  - **Authorization**: N/A.
  - **Impact**: Identifies the underwriter for the market, only used when depositing a generic asset.

Function: `withdraw_asset`

The `withdraw_coins_custodian()` function withdraws coins from a user's market account to a custodian's address under the authority of a delegated custodian. This is different from the `withdraw_coins_user()` function, which withdraws coins from a user's market account to the user's address under the authority of the signing user.

The `withdraw_generic_asset_custodian()` function withdraws generic assets from a user's market account to a custodian's address under the authority of a delegated custodian. This is distinct from the `withdraw_generic_asset_user()` function, which withdraws generic assets from a user's market account to the user's address under the authority of the signing user.

The `withdraw_coins_user()` function withdraws coins from a user's market account to the user's address under the authority of the signing user. This is distinct from the `withdraw_coins_custodian()` function, which withdraws coins from a user's market account to a custodian's address under the authority of a delegated custodian.

The `withdraw_generic_asset_user()` function withdraws generic assets from a user's market account to the user's address under the authority of the signing user. This is different from the `withdraw_generic_asset_custodian()` function, which withdraws generic assets from a user's market account to a custodian's address under the authority of a delegated custodian.

The `withdraw_to_coinstore()` function is a wrapper for the `withdraw_coins_user()` function. It withdraws coins from a user's market account and deposits them into their `aptos_framework::coin::CoinStore`.

The internal `withdraw_asset` function withdraws `amount` of `AssetType` from the `MarketAccount` of a user specified by `user_address`, `market_id` and `custodian_id`. If the `AssetType` is a `GenericAsset`, it updates the relevant counts, and if it is a Coin, it also withdraws the corresponding collateral coins.

```
public fun withdraw_coins_custodian<
    CoinType
>(
    user_address: address,
    market_id: u64,
    amount: u64,
    custodian_capability_ref: &CustodianCapability
)
public fun withdraw_coins_user<
    CoinType
>(
    user: &signer,
    market_id: u64,
    amount: u64,
)
public entry fun withdraw_to_coinstore<
    CoinType
>(
    user: &signer,
    market_id: u64,
    amount: u64,
)
public fun withdraw_generic_asset_custodian(
    user_address: address,
    market_id: u64,
    amount: u64,
    custodian_capability_ref: &CustodianCapability,
    underwriter_capability_ref: &UnderwriterCapability
)
public fun withdraw_generic_asset_user(
    user: &signer,
    market_id: u64,
    amount: u64,
    underwriter_capability_ref: &UnderwriterCapability
)
fun withdraw_asset<
    AssetType
>(
    user_address: address,
    market_id: u64,
    custodian_id: u64,
```

```
    amount: u64,
    underwriter_id: u64
)
```

Branches and code coverage

Intended branches:

- Withdraws the asset from the user's account.
  ☑ Test coverage

Negative behavior:

- Expects that a `MarketAccounts` resource exists for the given `user_address`.
  ☑ Test coverage
- Expects that a `MarketAccount` resource exists for the given `market_id` and `custodian_id`.
  ☑ Test coverage
- Expects that the `AssetType` is either the `base_type` or `quote_type` of the `MarketAccount` resource.
  ☑ Test coverage
- Expects that the withdrawal does not exceed the `base_available` or `quote_available` of the `MarketAccount` resource.
  ☑ Test coverage
- Expects that the underwriter ID is valid for the given `MarketAccount` resource if the `AssetType` is `GenericAsset`.
  ☑ Test coverage

Inputs

- `user_address`:
  - **Control**: For custodian functions, the CustodianCapability and UnderwriterCapability references must be passed. For the user functions, the `user` parameter must be a signer.
  - **Authorization**: Input comes from wrapper functions.
  - **Impact**: Used to access the `MarketAccounts` resource for the user.
- `market_id`:
  - **Control**: Must be a valid `u64` ID value for a market.
  - **Authorization**: N/A.

  - **Impact**: Used to find the `market_account_id` used to access the `MarketAccount` resource.
- `custodian_id`:
  - **Control**: Must be a valid `u64` ID value for a registered custodian.
  - **Authorization**: N/A.
  - **Impact**: Used to find the `market_account_id` used to access the `MarketAccount` resource.
- `amount`:
  - **Control**: Must be a valid `u64` value that is less than or equal to the available base/quote (depending on `AssetType`).
  - **Authorization**: N/A.
  - **Impact**: Used to specify the amount of the asset to withdraw.
- `underwriter_id`:
  - **Control**: Must be a valid `u64` ID value for an underwriter that is registered to the market.
  - **Authorization**: Input comes from wrapper functions.
  - **Impact**: Used to check if the `underwriter_id` is valid for a generic asset withdrawal.

Function: `register_market_account`

Registers a market account for the given market ID and custodian ID.

```
public entry fun register_market_account<
    BaseType,
    QuoteType
>(
    user: &signer,
    market_id: u64,
    custodian_id: u64
)
public entry fun register_market_account_generic_base<
    QuoteType
>(
    user: &signer,
    market_id: u64,
    custodian_id: u64
)
```

Branches and code coverage

Intended branches:

- Withdraws the asset from the user's account.
  ☑ Test coverage

Negative behavior:

- Expects that a custodian ID has not been registered yet.
  ☑ Test coverage
- Expects that the market account does not exist yet.
  ☑ Test coverage

Inputs

- `user`:
  - **Control**: Full; signer.
  - **Authorization**: A signer reference.
  - **Impact**: The address of the user is used to register new market account entries.
- `market_id`:
  - **Control**: Must be a valid u64.
  - **Authorization**: None.
  - **Impact**: The `market_id` is used to register a new market account.
- `custodian_id`:
  - **Control**: Must be a valid u64, or `NO_CUSTODIAN`.
  - **Authorization**: Must have a registered custodian ID, or `NO_CUSTODIAN`.
  - **Impact**: The `custodian_id` is used to register a new market account with the given custodian, or without any custodian.
- `BaseType`:
  - **Control**: Must be a valid asset type.
  - **Authorization**: None.
  - **Impact**: The BaseType is used to set the base asset for the given market.
- `QuoteType`:
  - **Control**: Must be a valid asset type.
  - **Authorization**: None.
  - **Impact**: The QuoteType is used to set the quote asset for the given market.

6 Audit Results

At the time of our audit, the code was not deployed to the Aptos mainnet.

During our audit, we discovered three findings, one of which was of critical severity, one of low impact, and one was a suggestion (informational). Econia Labs acknowledged and implemented fixes for all the findings.

6.1 Disclaimers

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any additional code added to the assessed project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.