Technical review of Marlowe
Final report

High Assurance Software Group
March the 24th, 2023

Contents

Chapter 1

Executive summary and scope

1.1 Disclaimer

This report is presented without warranty or guaranty of any type. It contains the concerns that have so far become apparent to Tweag after a partial inspection of the engineering work. Corrections, such as the cancellation of incorrectly reported issues, may arise. Therefore Tweag advises against making any business decision or other decision based on this report.

Tweag does not recommend for or against the use of any work or supplier referenced in this report. This report focuses on the technical implementation provided by the project's contractors and subcontractors, based on their information, and is not meant to assess the concept, mathematical validity, or business validity of IOG's product. This report does not assess the implementation regarding financial viability nor suitability for any purpose.

1.2 Executive summary

This document contains the findings discovered by Tweag while reviewing Marlowe. There are some issues affecting the safety of the Marlowe validators and the contracts expressed in Marlowe:

- Negative deposits handling (2.1.1), potentially allowing one to steal funds from contracts lacking enough validation.
- Double satisfaction vulnerability for payouts (2.1.2), allowing one to satisfy a Marlowe contract along with some other third-party contract with a single payout.
- The Isabelle proofs do not distinguish between different currencies when proving money preservation (2.1.4), potentially allowing deposits in one currency with payouts in another one.

There are also several other high and medium-severity findings described in Section 2.1 which either weaken the Isabelle proofs or their applicability to the Haskell implementation, or make the specification unclear.

1.3 Scope

The review is focused on the Marlowe language itself and its implementation for the Cardano blockchain. The team *did not* analyze the safety of any particular contract written in the Marlowe language. Any issues with the usability of the language, or any potential sources of erroneous or malicious behavior (falling outside of IOG's specification) of any current or future contracts are *outside* of the scope of the review. It is the responsibility of a user of Marlowe to avoid interacting with contracts that are vulnerable to abuse by an adversarial party.

The team has reviewed the Isabelle files, specification files, and Plutus validators provided by IOG[1]. More precisely, Tables 1.1, 1.2, and 1.3 list the files inspected alongside their sha256 sums. Additionally, the team has reviewed the testing strategy and the properties in the files listed in Table 1.4. Chapter 2 lists the findings resulting from the review, and Chapter 3 describes the tests that were implemented during the review.

Some of IOG's code had to be changed to accomodate for the differences between `plutus-apps` versions required by IOG's product and the team's tools respectively. The corresponding patches are listed in Appendix A. The sha256 sums in Table 1.3 are computed *without* these modifications, using the contents of the files as received from IOG. The line numbers in the report refer to the patched files, though.

`sha256sum`	File Name
`dcdd4f...6068c1`	`Core/TransactionBound.thy`
`708e74...a0bcd6`	`Core/Semantics.thy`
`8c52f0...5a0f53`	`Core/CloseSafe.thy`
`4abeb6...ef1164`	`Core/SemanticsTypes.thy`
`dd573b...a2279a`	`Core/TimeRange.thy`
`0f21eb...c1e587`	`Core/SingleInputTransactions.thy`
`cdc127...d2cb0c`	`Core/ListTools.thy`
`c3e8aa...74547a`	`Core/OptBoundTimeInterval.thy`
`06d820...54f0b0`	`Core/BlockchainTypes.thy`
`529fd2...b3d1da`	`Core/Timeout.thy`
`4ddc58...657b02`	`Core/PositiveAccounts.thy`
`c7e56c...9a20da`	`Core/ValidState.thy`
`ad04a3...2d60f3`	`Core/SemanticsGuarantees.thy`
`bd2a1b...594198`	`Core/QuiescentResult.thy`
`8fabfe...7427d8`	`Core/MoneyPreservation.thy`
`3239d4...d82338`	`Util/ByteString.thy`
`a18ffe...e01818`	`Util/MList.thy`

TABLE 1.1: *Isabelle files analyzed, and their* `sha256sum`

`sha256sum`	File Name
`9cee0f...b310a0`	`specification-v3-rc1.pdf`
`df8446...370dde`	`marlowe-cardano-specification.md`

TABLE 1.2: *Specification files analyzed, and their* `sha256sum`

[1] `https://github.com/input-output-hk/marlowe/commit/c8c67ad6892ec68842461d2e66b02ca87f93f70c`
`https://github.com/input-output-hk/marlowe-cardano/commit/523f3d56f22bf992ddb0b0c8a52bb7a5a188f9e9`

sha256sum	File Name
`26a96f...dbf6bf`	`Marlowe/Scripts.hs`
`3bcab9...cae538`	`Marlowe/Core/V1/Semantics.hs`
`8abfd3...9c7574`	`Marlowe/Core/V1/Semantics/Types.hs`

TABLE 1.3: *Haskell files analyzed, and their sha256sum*

sha256sum	File Name
`d9d55c...2c6ba2`	`test-report.md`
`e663a2...a565eb`	`Spec/Marlowe/Plutus/Value.hs`
`28a476...16f3da`	`Spec/Marlowe/Plutus/Arbitrary.hs`
`fdca9a...37e121`	`Spec/Marlowe/Plutus/Specification.hs`
`7ec64a...2eda23`	`Spec/Marlowe/Plutus/AssocMap.hs`
`820967...dfd3a0`	`Spec/Marlowe/Plutus/ScriptContext.hs`
`e7eaf2...451ee1`	`Spec/Marlowe/Plutus/Prelude.hs`
`252a34...e9d0ad`	`Spec/Marlowe/Semantics/Arbitrary.hs`
`87ffa5...01b5d5`	`Spec/Marlowe/Semantics/Compute.hs`
`3c775d...137df1`	`Spec/Marlowe/Semantics/Functions.hs`

TABLE 1.4: *Test-related files analyzed, and their sha256sum*

Severity	Section	Summary
High	2.1.1	Negative deposits allow stealing funds
High	2.1.2	Contracts vulnerable to double satisfaction attacks
High	2.1.3	Missing constructor in equality instance
High	2.1.4	Inaccurate formulation of Money preservation
High	2.1.6	Missing description of Merkleization
High	2.1.7	Positive balances are not checked for the output state
High	2.1.8	Non-validated Marlowe states
High	2.1.9	Total balance of ending state uncomputed
High	2.1.10	Unchecked ending balance
High	2.1.12	Different insertion functions used in Isabelle and Haskell code
Medium	2.1.5	Insufficient documentation of Money preservation
Medium	2.1.11	Partial functions used outside their domain
Medium	2.1.13	Missing specification tests
Medium	2.3.1	Unnecessarily large proofs
...	...	...

TABLE 1.5: *Table of findings*

Severity	Section	Summary
Medium	2.4.1	Variable shadowing in `applyAllLoop`
Medium	2.5.1	Lack of guidelines for creating cooperating contracts
Medium	2.6.1	Name shadowing in `applyAllInputs`
Medium	2.6.2	Non-isomorphic types in `playTraceAux`
Medium	2.7.1	More precise failure checks
Medium	2.7.2	Missing tests
Low	2.2.1	Lack of explanation regarding changing choices
Low	2.2.2	Undefined reference
Low	2.2.3	Lack of explanation for necessity of `Environment` type
Low	2.2.4	Unclear meaning of execution environment
Low	2.2.5	Unexplained interval data types
Low	2.2.6	Incomplete explanation for `TransactionOutput`
Low	2.2.7	Code snippets switch languages
Low	2.2.8	Repeated definition of `IntervalResult`
Low	2.2.9	Poorly named variable `newAccount`
Low	2.2.10	Poorly named variable `acc` in specification
Low	2.2.11	Inaccurate specification of `giveMoney`
Low	2.2.12	Redundant evaluation in `addMoneyToAccount`
Low	2.2.13	Redundant statement regarding addition
Low	2.2.14	Missing implementation for negation case of `evalValue`
Low	2.2.15	Missing parentheses in `div` specification
Low	2.2.16	Unclear division explanation
Low	2.2.17	Discrepancy with `evalValue`
Low	2.2.18	Missing `evalValue` lemmas in specification
Low	2.2.19	Typo in **Use Value** case of `evalValue`
Low	2.2.20	Unexplained parameters of `playTrace`
Low	2.2.21	Type parameter discrepancy in `playTrace`
Low	2.2.22	Money preservation on failing transactions not specified
Low	2.2.23	Complicated definition of `allAccountsPositive`
Low	2.2.24	Discrepancy with Isabelle code for `allAccountsPositive`
Low	2.2.25	Misleading or incorrect formula for contract not holding funds
Low	2.2.26	Different format for lemma statement
Low	2.2.27	Function `isClosedAndEmpty` is unexplained
Low	2.2.28	Top-down definitions
Low	2.2.29	No mention of Isabelle lemmas in specification
Low	2.3.2	Long lines in lemmas
…	…	…

TABLE 1.5: *Table of findings*

Severity	Section	Summary
■ Low	2.3.3	Confusing auxiliary lemmas
■ Low	2.3.4	Undescriptive variable names
■ Low	2.3.5	Involved proof of `insert_valid`
■ Low	2.3.6	Repeated verbose expression
■ Low	2.3.7	Inconsistent variable name `valTrans`
■ Low	2.3.8	Unused binding `interAccs`
■ Low	2.3.9	Undescriptive variable name `acc`
■ Low	2.3.10	Misleading indentation
■ Low	2.3.11	Missing theorem regarding `playTrace`
■ Low	2.3.12	Unconcise goal in `reduceContractStepPayIsQuiescent`
■ Low	2.3.13	Misleading lemma names
■ Low	2.3.14	Misleading variable name `reduced`
■ Low	2.3.15	Undescriptive name `beforeApplyAllLoopIsUseless`
■ Low	2.3.16	Unused and undocumented lemmas
■ Low	2.3.17	Redundant `reduceContractStep` lemmas
■ Low	2.3.18	Redundant `transferMoneyBetweenAccounts_preserves`
■ Low	2.3.19	Duplicated lemmas
■ Low	2.3.20	Redundant `computeTransaction` lemmas
■ Low	2.3.21	Complicated formulation of `updateMoneyInAccount_money2_aux`
■ Low	2.3.22	Complicated proofs that can be simplified
■ Low	2.3.23	Inconsistent style when applying constructor
■ Low	2.3.24	Unsimplified boolean formulas
■ Low	2.3.25	Typo with “independet” in multiple lemmas
■ Low	2.3.26	Poorly named `acc` lemmas
■ Low	2.3.27	Verbose lemma statement `playTraceAuxIterative_base_case`
■ Low	2.3.28	`playTrace_only_accepts_maxTransactionsInitialState` not written as `theorem`
■ Low	2.3.29	Inconsistent style with assumptions
■ Low	2.3.30	Function `validTimeInterval` unnecessarily unfolded in lemma
■ Low	2.3.31	Overly specific auxiliary lemma
■ Low	2.3.32	`playTrace_preserves_valid_state` not written as `theorem`
■ Low	2.3.33	Unnecessary assumptions
■ Low	2.4.2	Undescriptive name `moneyInPayment`
■ Low	2.4.3	Typo in section name
■ Low	2.4.4	Typo in comment
■ Low	2.4.5	Unclear need for multiple formulations for positive accounts
■ Low	2.4.6	Variable name discrepancy in `reductionLoop`
…	…	…

TABLE 1.5: *Table of findings*

Severity	Section	Summary
Low	2.4.7	Typo in constructor
Low	2.4.8	Unclear function name `calculateNonAmbiguousInterval`
Low	2.4.9	Non-modularized file `SingleInputTransactions.thy`
Low	2.4.10	Misleading function names
Low	2.4.11	Unused parameter in `maxTransactionCaseList`
Low	2.4.12	Duplicated `isValidInterval` function
Low	2.5.2	No reference to creating a minting policy
Low	2.5.3	Argument for Contract in `txInfoData` not specified
Low	2.5.4	Merkleization section not detailed enough
Low	2.5.5	Unnecessary constraint
Low	2.5.6	Asymmetry between role and wallet payouts
Low	2.5.7	Incorrect description of `rolePayoutValidator`
Low	2.5.8	Unspecified initial state
Low	2.5.9	Unspecified behavior when multiple cases can apply
Low	2.6.3	Variable names differ from Isabelle code
Low	2.6.4	Naming of functions and variables
Low	2.6.5	Unused functions
Low	2.6.6	Comments
Low	2.6.7	Record updates in `playTraceAux`
Low	2.6.8	Potential simplifications
Low	2.6.9	`computeTransaction` differs from the Isabelle implementation
Low	2.6.10	Constraint implementations differ from description
Low	2.6.11	Missing argument of `computeTransaction`
Low	2.6.12	Missing `smallMarloweValidator`
Low	2.6.13	Incorrect constraint reference
Low	2.6.14	`MarloweParams` differs from the specification
Low	2.6.15	Timeout boundary differs from the specification

TABLE 1.5: *Table of findings*

Chapter 2

Findings

During the inspection of the code, the team has noticed various issues, outlined in Table 1.5 and presented in detail in this chapter. Section 2.1 lists the main concerns which could endanger users.

We use the following severities to classify issues:

High Exposes users to harm or can expose them to harm if disregarded when changing the code.

Medium Can expose users to harm if disregarded when making changes, but likely need to compound with other high severity concerns to be exploitable.

Low Inconsistencies or omissions that make the review and development work harder.

2.1 Main concerns

2.1.1 Negative deposits allow stealing funds

Severity: High

File `marlowe-cardano-specification.md`, `Constraint 6` The income from deposits is computed by adding up the deposit inputs, regardless of whether they are negative, while the semantics considers them as zero deposits. Combined with the absence of a balance check on the ending Marlowe state, this allows the ending balance to differ from the value paid to the Marlowe validator.

This disagreement can be exploited to steal money from a flawed Marlowe contract that allows a negative deposit. The issue is demonstrated in Section 3.2.1.

2.1.2 Contracts vulnerable to double satisfaction attacks

Severity: High

File `marlowe-cardano-specification.md`, `Constraint 15` No datum is required for outputs fulfilling payments to addresses generated by the evaluation of a Marlowe contract. This implies that these outputs are vulnerable to double satisfaction in transactions involving other contracts that pay to the same wallets. An example is discussed in Section 3.2.2.

One way to strengthen the implementation is for the Marlowe validator to demand that outputs paid to addresses contain a datum that identifies the contract instance, like the `TxOutRef` of the validator UTxO being spent. Then cooperation with other contracts is possible without double satisfaction if the validators of the other contracts demand a different datum for their outputs.

2.1.3 Missing constructor in equality instance

Severity: High

File `Semantics.hs`, Class instance `Eq ReduceWarning`, line *845* The constructor `ReduceAssertionFailed` is not mentioned and compares `False` against itself. This might cause validators to fail checking the presence of this particular warning.

2.1.4 Inaccurate formulation of Money preservation

Severity: High

File `specification-v3-rc1.pdf`, Section `3.1 Money preservation`, page *29* As the property stands, it is permitted to make deposits in one currency and return payments in a different currency. As long as the sums of the amounts match, the equality is satisfied. Yet it is unlikely that the participants of the contract would agree that money has been preserved.

Money preservation is a property stated with an equality. The left hand side is the sum of the deposits done by a list of transactions. The right hand side of the equality is the sum of all the payments done in the same list of transactions. Each sum, in turn, is represented as a single integer which aggregates the amounts of the various payments and deposits, irrespective of what currencies correspond to these amounts.

2.1.5 Insufficient documentation of Money preservation

Severity: Medium

File `specification-v3-rc1.pdf`, Section `3.1 Money preservation`, page *29* Money preservation is formulated in terms of functions that are not discussed in the specification. It is necessary to explain the meaning of these functions in sufficient detail so readers can understand the property.

2.1.6 Missing description of Merkleization

Severity: High

File `specification-v3-rc1.pdf`, `Merkleization` There is no property about merkleization, but merkleization is implemented in the Cardano integration.

Some relevant properties could be:

a) The merkleized contract produces the same payments as the analogous regular contract.

b) If a merkleized case input is applied successfully, it implies that the contract hash in the input corresponds to the continuation of the contract.

c) Merkleizing and unmerkleizing a contract gives back the original contract.

2.1.7 Positive balances are not checked for the output state

Severity: High

File `marlowe-cardano-specification.md`, `Constraint 13` *Positive balances* are only checked for the input, not for the output Marlowe state. If the semantics are flawed, a transaction can produce an unspendable output that does not satisfy this constraint.

If such a transaction is accepted, no further evaluation will be possible since all subsequent transactions will be rejected due to the very same Constraint 13. This is an hypothetical attack vector, where a malicious actor could send a transaction to block a contract.

2.1.8 Non-validated Marlowe states

Severity: High

File `marlowe-cardano-specification.md`, `Missing constraint` The validator is not specified to check that the Marlowe states in the input and output datums are valid. This condition is necessary for the lemmas about the Marlowe semantics to be applicable. The Marlowe state could become invalid if there is a flaw in the implementation of the semantics.

It also could be possible for the Marlowe state to be invalid if someone pays an output to the Marlowe validator with an invalid Marlowe state. Though this problem could be addressed with off-chain checks that prevent sending transactions that spend outputs with invalid Marlowe states. If off-chain checks are used, a note in the specification about how this is handled would be helpful.

An example showing betrayed user expectations is discussed in Section 3.2.3.

For a valid Marlowe state, the association lists for bound values, accounts, and choices have keys sorted and without duplicates.

2.1.9 Total balance of ending state uncomputed

Severity: High

File `marlowe-cardano-specification.md`, `Constraint 6` The constraint says

> The beginning balance plus the deposits equals the ending balance plus the payments.

However, the Marlowe validator never computes the total balance of the accounts in the ending Marlowe state. Instead, the ending balance is assumed to be whatever value is paid by the transaction to the Marlowe validator. The natural language should describe precisely what is being checked.

2.1.10 Unchecked ending balance

Severity: High

File `marlowe-cardano-specification.md`, `Constraint 5` The balance of the starting Marlowe state is checked to match the value in the input. However, the validator does not check that the ending balance matches the value in the output paid to the Marlowe validator. Similarly to Issue 2.1.7, if there are flaws in the semantics that cause the ending balance to differ from the actual value paid to the validator, this constraint would prevent any transaction from spending the output.

The specification should at least discuss why the check is absent together with the other similar checks that are not implemented (checking that ending accounts have positive balances, checking that the ending Marlowe state is valid).

2.1.11 Partial functions used outside their domain

Severity: Medium

File `MoneyPreservation.thy`, various functions `moneyInRefundOneResult`, `moneyInApplyResult`, `moneyInApplyAllResult`, `moneyInTransactionOutput`, and `moneyInPlayTraceResult` have strange meanings when the result is an error. Arguably, on error there is no money to retrieve, so the return type should be `(Token × int) option` instead.

Some lemmas rely on this behavior to have equalities hold even in cases of errors, but the cost is that the meaning is so surprising that the reader may be confused by it. It would be more reliable to have explicit and weaker lemmas that assert equalities only when there are no errors.

2.1.12 Different insertion functions used in Isabelle and Haskell code

Severity: High

File `Semantics.hs`, Several functions Where `MList.insert` is used in the Isabelle semantics, AssocMap.insert is used in the Cardano implementation. However, the functions are not equivalent as demonstrated by the following examples:

```
AssocMap.insert 'a' 1 [('b', 0)] == [('b', 0), ('a', 1)]
-- whereas
MList.insert 'a' 1 [('b', 0)] == [('a', 1), ('b', 0)]

AssocMap.insert 'b' 1 [('a', 0), ('b', 0), ('b', 0)] == [('a', 0), ('b', 1), ('b', 1)]
-- whereas
MList.insert 'b' 1 [('a', 0), ('b', 0), ('b', 0)] == [('a', 0), ('b', 1), ('b', 0)]
```

This renders the Isabelle lemmas inapplicable for the Cardano integration. The lemmas need to demand some properties of an `insert` function without fully spelling it out, or the Cardano integration needs to use `MList.insert` instead of `AssocMap.insert`.

Similarly, functions `AssocMap.delete` and `MList.delete` differ in behavior when the input map is not sorted:

```
AssocMap.delete 'a' [('b', 0), ('a', 0)] == [('b', 0)]
-- whereas
MList.delete 'a' [('b', 0), ('a', 0)] == [('b', 0), ('a', 0)]
```

Functions `AssocMap.lookup` and `MList.lookup` also differ in behavior when the input map is not sorted:

```
AssocMap.lookup 'a' [('b', 0), ('a', 0)] == Just 0
-- whereas
MList.lookup 'a' [('b', 0), ('a', 0)] == Nothing
```

The following usage places were found:

- Line 395, `evalValue` depends on `moneyInAccount` which depends on `AssocMap.lookup`.
- Line 413, `evalValue` depends on `AssocMap.lookup`.
- Line 428, `evalObservation` depends on `AssocMap.member`.
- Line 456, function `updateMoneyInAccount` relies on `AssocMap.delete` and `AssocMap.insert`.
- Line 482, function `reduceContractStep` relies on `AssocMap.insert`.
- Line 567, function `applyAction` relies on `AssocMap.insert`.

2.1.13 Missing specification tests

Severity: Medium

File `Spec/Marlowe/Semantics/Compute.hs`, There are no tests for the properties in Section 3 of `specification-v3-rc1.pdf`. Besides checking that there are no translation mistakes, these properties would also help contrasting the assumptions in the Isabelle and the Haskell sides, like the meaning of validity of an association list, which is focused in the previous issue.

2.2 Marlowe specification

2.2.1 ■ Lack of explanation regarding changing choices

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.1.4 Choices`, page *10* Choices can only be changed when evaluating `When` statements. This is something only evident after looking at the implementation of `computeTransaction`. It needs to be discussed when first introducing choices and the `When` contract.

2.2.2 ■ Undefined reference

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.1.7 Contracts`, page *13* There is an undefined reference.

2.2.3 ■ Lack of explanation for necessity of `Environment` type

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.1.8 State and Environment`, page *14* An `Environment` type is introduced, but it is unclear why it is needed as it is defined as a synonym for time intervals.

2.2.4 ■ Unclear meaning of execution environment

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.1.8 State and Environment`, page *14* The meaning of the execution environment of the transaction is unclear. This is due to the concept of *transaction* being assumed by the specification and never formally introduced.

The specification reads

> The execution environment of a Marlowe contract simply consists of the (inclusive) time interval within which the transaction is occurring.

One has to infer that evaluating a Marlowe contract is undefined if it does not happen within a transaction, as otherwise the description of the execution environment would not make sense. It would be necessary to establish more explicitly the relationship between the contract evaluation and the notion of transaction.

2.2.5 ■ Unexplained interval data types

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.1.8 State and Environment`, page *14* The meaning of the data types `IntervalError` and `IntervalResult` needs to be explained.

2.2.6 ■ Incomplete explanation for `TransactionOutput`

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.2.1 Compute Transaction`, page *15* The meaning of the data type `TransactionOutput` needs to be explained. More generally, the meaning of the return types of most functions has to be explained. Currently, the meaning can only be inferred from looking at how the types are used, which makes it harder to identify if they are used as intended.

The purpose of these types needs to be made explicit so it can be checked if the code is doing what is intended.

2.2.7 Code snippets switch languages

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.2.1 Compute Transaction`, page *15* The specification changes from using Isabelle to using Haskell henceforth. Making the reader aware of the criteria for the language change would help maintaining the document.

2.2.8 Repeated definition of `IntervalResult`

Severity: Low

File `specification-v3-rc1.pdf`, Sections `2.1.8 State and Environment, 2.2.2 Fix Interval`, pages *14, 16* The `IntervalResult` type is defined twice in the specification. One should be removed.

2.2.9 Poorly named variable `newAccount`

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.2.6 Reduce Contract Step`, page *19* In the implementation of the function `reduceContractStep`, the variable `newAccount` should be named `newAccounts`.

2.2.10 Poorly named variable `acc` in specification

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.2.8 Apply Cases`, page *22* On the last equation of `applyCases`, `acc` should be named `input`.

2.2.11 Inaccurate specification of `giveMoney`

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.2.9 Utilities`, page *22* It says

> The *giveMoney* function transfers funds internally between accounts.

which is not accurate. It should say instead

> The *giveMoney* function deposits funds to an internal account.

This function is confusing in that it takes the account identifier of the paying account which is not used for anything other than filling a field in the returned value.

2.2.12 Redundant evaluation in `addMoneyToAccount`

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.2.9 Utilities`, page *22* `addMoneyToAccount` is redundantly evaluating `money <= 0` when invoking `updateMoneyInAccount`. The else branch could be replaced instead with `insert (accId, token) money accountsV`.

2.2.13 Redundant statement regarding addition

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.2.10 Evaluate Value`, page *24* It says that addition is associative and commutative. This is true but it is already implied by the equation preceding the statement. Maybe change to

> Note that addition is associative and commutative.

or remove the redundant statement.

2.2.14 Missing implementation for negation case of `evalValue`

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.2.10 Evaluate Value`, page *24* Negation for `evalValue` does not show the implementation, just one lemma about `NegValue`, which is inconsistent with how other operations are presented.

2.2.15 Missing parentheses in `div` specification

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.2.10 Evaluate Value`, page *25* On page 25 formula

$$c \neq 0 \Rightarrow c * a \text{ div } (c * b) = a \text{ div } b$$

needs additional parentheses around the term $c * a$, otherwise it can be parsed as

$$c \neq 0 \Rightarrow c * (a \text{ div } (c * b)) = a \text{ div } b$$

which does not hold (Counter-example: $c = 2, a = 3, b = 2$). The lemma `divMultiply` in the file `Semantics.thy` does use extra parentheses around $c * a$.

2.2.16 Unclear division explanation

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.2.10 Evaluate Value`, page *25* It says

> Division is a special case because we only evaluate to natural numbers.

The meaning of this statement needs to be further explained, since the arguments of `DivValue` could evaluate to negative numbers.

2.2.17 Discrepancy with `evalValue`

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.2.10 Evaluate Value`, pages *23–26* The order of some cases for `evalValue` is different in the specification text and in the actual Isabelle code, and several cases (for example, `NegValue`) are missing from the specification entirely.

Moreover, the definition of `evalValue` is juxtaposed with some lemmas about its behavior (for example, `AddValue` being associative and commutative), making it harder to match the specification text with the Isabelle code.

2.2.18 Missing `evalValue` lemmas in specification

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.2.10 Evaluate Value`, pages *23–26* Not all lemmas about `evalValue` are listed in the specification. The absent lemmas include `evalDoubleNegValue`, `evalMulValue`, `evalSubValue`, and all division lemmas.

2.2.19 Typo in **Use Value** case of `evalValue`

Severity: Low

File `specification-v3-rc1.pdf`, Section `2.2.10 Evaluate Value`, page *26* The **Use Value** case mentions `TimeIntervalEnd` instead of `UseValue`.

2.2.20 Unexplained parameters of `playTrace`

Severity: Low

File `specification-v3-rc1.pdf`, Section `3 Marlowe Guarantees`, page *28* The parameters of the function `playTrace` need to be explained.

2.2.21 Type parameter discrepancy in `playTrace`

Severity: Low

File `specification-v3-rc1.pdf`, Section `3 Marlowe Guarantees`, page *28* The first parameter of `playTrace` in the specification is `int`, while it is `POSIXTime` in the code. Even though the latter is an alias for the former, it is beneficial to use the `POSIXTime` name both for consistency and readability.

2.2.22 Money preservation on failing transactions not specified

Severity: Low

File `specification-v3-rc1.pdf`, Section `3.1 Money preservation`, page *29* Money preservation is expressed with an equality. This equality, however, only ensures money preservation for those lists of transactions that produce no error. In other words, there is no guarantee that money will be preserved for those lists of transactions that fail.

This is not a concern in practice because the lists of transactions that fail to evaluate are not accepted in the blockchain. However, this should be made explicit in the explanation of the property.

2.2.23 Complicated definition of `allAccountsPositive`

Severity: Low

File `specification-v3-rc1.pdf`, Section `3.3 Possitive Accounts`, page *30* The definition of `allAccountsPositive` is complicated and can be refactored as `all ((_, money) -> money > 0)`.

2.2.24 Discrepancy with Isabelle code for `allAccountsPositive`

Severity: Low

File `specification-v3-rc1.pdf`, Sections `3.3 Positive Accounts`, page *30* The `allAccountsPositive` function is defined differently in the specification and in the Isabelle code, although both definitions show the same behavior. These definitions need to be consolidated.

2.2.25 Misleading or incorrect formula for contract not holding funds

Severity: Low

File `specification-v3-rc1.pdf`, Section `3.6.3 Contract Does Not Hold Funds After it Closes`, page *32* The statement in natural language looks unconnected from the proposed formula. Otherwise, it is unclear how not holding funds forever is a consequence of producing no warnings.

2.2.26 Different format for lemma statement

Severity: Low

File `specification-v3-rc1.pdf`, Sections `3.6.2 All Contracts Have a Maximum Time`, page *32* The lemma is stated using the proof derivation tree format as opposed to the rest of the specification and the Isabelle code.

2.2.27 Function `isClosedAndEmpty` is unexplained

Severity: Low

File `specification-v3-rc1.pdf`, Section `3.6.2 All Contracts Have a Maximum Time`, page *32* The function `isClosedAndEmpty` needs to be explained.

2.2.28 Top-down definitions

Severity: Low

File `specification-v3-rc1.pdf`, Section `2` In Section 2, the order of definitions is reversed, and the reader is thus faced with functions which call other functions that have not been introduced yet, despite the claim in Section 1.3 that the definitions will be presented bottom-up.

2.2.29 No mention of Isabelle lemmas in specification

Severity: Low

File `specification-v3-rc1.pdf`, Multiple sections Generally, readability can be improved by mentioning the Isabelle lemma names alongside their statements. This way, it would be much easier to search for the actual Isabelle code and proofs matching the informal specification text, and compare the two.

2.3 Lemmas and proofs

2.3.1 Unnecessarily large proofs

Severity: Medium

Several Isabelle files, several lemmas Some Isabelle proofs are written with long apply-scripts, where Isar would document the proof better. Proofs could also be split using more auxiliary lemmas.

As the proofs stand, it is hard to figure out why a proof step fails, after changes elsewhere required a proof to be updated. Since the newly-failing proof step was designed with specific goals in mind, and changes in the code may lead to it facing a different set of goals, the maintainer might need to reconstruct the whole structure of the proof from an older version to infer state that Isabelle produces at each step.

What Isar brings is making the intention of the author explicit at every step of the proof. This helps the maintainer of the proofs and fixes the concerns mentioned above.

IOG will likely have to update the proofs. We conjecture that it will happen at least every time they target a new platform. In the case of Cardano, they need to extend the semantics to explain Merkleization. Another action that would make long proofs easier to understand is to split them using more auxiliary lemmas, thus feeding the information to the reader in smaller chunks.

Some examples of large proofs:

- in `MoneyPreservation.thy`, lemmas `reduceContractStep_preserves_money` and `reductionLoop_preserves_money`
- in `SingleInputTransactions.thy`, lemmas `applyAllInputsPrefix_aux`, `computeTransactionIterative`, and `computeTransactionStepEquivalence_error`

2.3.2 Long lines in lemmas

Severity: Low

Several Isabelle files, several lemmas Lines are sometimes long which makes it difficult to understand the lemmas. Lemmas need to be formulated expressing one hypothesis per line and the conclusion on a separate line. Complex hypotheses need to be indented using several lines to expose their structure.

Besides the effort of scrolling the text horizontally, the hypotheses are hard to separate visually, and so is the conclusion. Furthermore, when a hypothesis is a nested implication it is difficult to see where it ends without further indentation.

Some examples of lemmas with long lines or non-trivial hypothesis follow.

- in `CloseSafe.thy`, lemmas `closeIsSafe_reduceContractUntilQuiescent`, and `closeIsSafe_reductionLoop`
- in `MoneyPreservation.thy`, lemmas `reductionLoop_preserves_money_Payment_not_ReduceNoWarning`, `reductionLoop_preserves_money_Payment` and `reduceContractStep_preserves_money_acc_to_party`
- in `SingleInputTransactions.thy`, lemma `applyAllLoop_longer_doesnt_grow`
- in `TimeRange.thy`, lemmas `reduceStep_ifCaseLtCt` and `reduceLoop_ifCaseLtCt`
- in `ValidState.thy`, lemma `reductionLoop_preserves_valid_state_aux`

2.3.3 Confusing auxiliary lemmas

Severity: Low

Several Isabelle files, several lemmas Some Isabelle proofs resort to declaring auxiliary lemmas with names suffixed with *_aux*. Sometimes these lemmas are not expressed succinctly, and look more like a punctual copy of the state of some particular proof that is later developed. For the sake of maintaining the proofs, it would be necessary to structure them in a way that presents the information piecewise to the reader. More generally, even auxiliary lemmas should have a well-defined meaning.

We found this problem at least in the following:

- in `QuiescentResult.thy`, lemmas `reduceContractStepPayIsQuiescent`, `reductionLoopIsQuiescent_aux`, and `applyAllInputsLoopIsQuiescent_loop`
- in `PositiveAccounts.thy`, lemma `positiveMoneyInAccountOrNoAccountImpliesAllPositive_aux2`
- in `SingleInputTransactions.thy`, lemma `applyAllInputsPrefix_aux`

2.3.4 Undescriptive variable names

Severity: Low

Several Isabelle files, several lemmas Many Isabelle proof statements and proofs use uninformative variable names. The most common example occurs with variables named *x11*, *x12*, etc. These inhibit the reader from easily understanding the lemma statements, and often require looking back at constructors to understand what these variables represent.

Some examples of lemmas with these uninformative variable names follow:

- in `QuiescentResult.thy`, lemma `reductionLoopIsQuiescent_aux`
- in `SingleInputTransactions.thy`, lemmas `beforeApplyAllLoopIsUseless` and `applyAllInputsPrefix_aux`
- in `ValidState.thy`, lemma `reductionLoop_preserves_valid_state_aux`
- in `TimeRange.thy`, lemmas `resultOfReduceIsCompatibleToo`, `resultOfReductionLoopIsCompatibleToo`, `resultOfReduceUntilQuiescentIsCompatibleToo`, `reduceLoop_ifCaseLtCt`, and `reduceContractUntilQuiescent_ifCaseLtCt`

2.3.5 Involved proof of `insert_valid`

Severity: Low

File `MList.thy`, theorem `insert_valid`, line *66* The proof of `insert_valid` sprouts three other lemmas of difficult characterization: `insert_valid_aux`, `insert_valid_aux2`, and `insert_valid_aux3`. These lemmas make assumptions with implications that get in the way of understanding them in isolation.

An alternative to make the proof pieces more reusable is to use instead the following set of lemmas, which also offers insight on how function `insert` interacts with predicates `sorted` and `distinct`:

```
lemma insert_set:
  "set (map fst (insert a b xs)) = set (map fst xs) ∪ { a }"

lemma insert_sorted:
  "List.sorted (map fst c) ⟹ List.sorted (map fst (insert a b c))"
```

```
lemma insert_distinct :
  "List.distinct (map fst c)
   ⟹
   List.sorted (map fst c)
   ⟹
   List.distinct (map fst (MList.insert a b c))"
```

which then can be combined in the proof of `insert_valid` as follows:

```
theorem insert_valid2 : "valid_map c ⟹ valid_map (MList.insert a b c)"
  using insert_sorted[of c a b] insert_distinct[of c a b] by fastforce
```

The proofs of the lemmas can be found in Appendix B.

2.3.6 Repeated verbose expression

Severity: Low

File `MoneyPreservation.thy`, lemma `removeMoneyFromAccount_preservation`, line *202* The expression

```
giveMoney
  accId
  (Party p)
  tok
  paidMoney
  (updateMoneyInAccount accId tok (balance - paidMoney) accs)
```

is large and used in other lemmas as well. It would need to be moved to a separate function to save the effort of reading it repeteadly.

2.3.7 Inconsistent variable name `valTrans`

Severity: Low

File `MoneyPreservation.thy`, lemma `transferMoneyBetweenAccounts_preserves_aux`, line *257* The lemma uses a variable `valTrans` where other proofs use the name `paidMoney`. To convey the meaning of the variable faster, the same name should be used consistently in all places.

2.3.8 Unused binding `interAccs`

Severity: Low

File `MoneyPreservation.thy`, lemma `transferMoneyBetweenAccounts_preserves_aux`, line *263* The binding `interAccs` was probably intended to be used on this line. It should either be used or removed from the premise.

2.3.9 Undescriptive variable name `acc`

Severity: Low

File `MoneyPreservation.thy`, lemma `transferMoneyBetweenAccounts_preserves`, line *295* This lemma has a variable `acc` that is used together with `tok2`. It would be more descriptive to call it `accId2`.

2.3.10 Misleading indentation

Severity: Low

File `MoneyPreservation.thy`, lemmas `reductionLoop_preserves_money_NoPayment_not_ReduceNoWarning`, `reductionLoop_preserves_money_NoPayment`, lines *430, 439* The indentation is misleading: the premises on these lines are indented as if they are a part of the previous functional premise.

2.3.11 Missing theorem regarding `playTrace`

Severity: Low

File `PositiveAccounts.thy`, `playTrace preserves valid and positive state` There is no theorem that `playTrace` keeps the state valid and positive when given a state which is valid and positive. This trivially follows from `playTraceAux_preserves_validAndPositive_state` but no such theorem is present.

2.3.12 Unconcise goal in `reduceContractStepPayIsQuiescent`

Severity: Low

File `QuiescentResult.thy`, lemma `reduceContractStepPayIsQuiescent`, line *8* This lemma does not express its goal concisely, as it makes no mention of `reduceContractStep` in the formulation. Changing the first assumption to `reduceContractStep` *env sta* (`Pay` *x21 x22 tok x23 x24*) makes more explicit in which contexts this lemma can be useful. Modifying this assumption requires an additional `apply simp` to be added to the proof (before line 30) for the lemma to go through. Further, an additional `apply simp` will need to be added in lemmas `reduceContractStepIsQuiescent` (before line 44) and `timedOutReduce_only_quiescent_in_close` (`Timeout.thy`, before line 128) as well.

2.3.13 Misleading lemma names

Severity: Low

File `PositiveAccounts.thy`, lemma `reduceOne_gtZero`, line *80* This lemma should be renamed as `refundOne_gtZero`.

File `QuiescentResult.thy`, lemma `reduceOneIsSomeIfNotEmptyAndPositive`, line *32* This lemma should be renamed as `refundOneIsSomeIfNotEmptyAndPositive`.

File `TransactionBound.thy`, lemma `computeTransaction_decreases_maxTransaction_aux`, line *240* This lemma should be renamed as `applyAllInputs_decreases_maxTransactions` or `applyAllInputs_reduced_decreases_maxTransactions`.

2.3.14 Misleading variable name `reduced`

Severity: Low

File `QuiescentResult.thy`, lemmas `reductionLoop_reduce_monotonic`, `reduceContractUntilQuiescent_ifDifferentReduced`, lines *138, 153* The boolean variable name `reduce` would be better named `reduced` as it is signifying that the contract has been reduced.

2.3.15 Undescriptive name `beforeApplyAllLoopIsUseless`

Severity: Low

File `SingleInputTransactions.thy`, lemma `beforeApplyAllLoopIsUseless`, line *270* This lemma seems to say that `reduceContractUntilQuiescent` has no effect when composed with `applyAllLoop`, because `applyAllLoop` evaluates `reduceContractUntilQuiescent`, and `reduceContractUntilQuiescent` is idempotent.

A more descriptive name for this lemma could be `reduceContractUntilQuiescent_hasNoEffect-_before_applyAllLoop`

2.3.16 Unused and undocumented lemmas

Severity: Low

Several Isabelle files, several lemmas Some lemmas are never used, and they would need comments motivating their presence:

a. In file `MoneyPreservation.thy`, line 257, lemma `transferMoneyBetweenAccounts_preserves_aux`.

b. In file `QuiescentResult.thy`

   b.1. Line 5, lemma `reduceOne_onlyIfNonEmptyState`

   b.2. Line 153, lemma `reduceContractUntilQuiescent_ifDifferentReduced`

c. In file `PositiveAccounts.thy`, line 66, lemma `positiveMoneyInAccountOrNoAccount_sublist_gtZero`. Furthermore, it is identical to `positiveMoneyInAccountOrNoAccount_gtZero_preservation`, but with an additional assumption `money > 0`.

d. In file `ValidState.thy`

   d.1. Line 9, lemma `valid_state_valid_choices`

   d.2. Line 13, lemma `valid_state_valid_valueBounds`

e. In file `SingleInputTransactions.thy`, line 1214, lemma `traceListToSingleInput_isSingleInput`. It is mentioned in a commented out line in `StaticAnalysis.thy`. Furthermore, the lemma can be expressed more concisely as

$$(\!|interval = inte, inputs = inp\_h \,\#\, inp\_t|\!)\ \#\ t = traceListToSingleInput\ t2 \Longrightarrow inp\_t = []$$

2.3.17 Redundant `reduceContractStep` lemmas

Severity: Low

File `MoneyPreservation.thy`, lemma `reduceContractStep_preserves_money_acc_to_acc_aux`, line *310* This lemma is weaker than `transferMoneyBetweenAccounts_preserves`. If we replace its usage at line 351 with `transferMoneyBetweenAccounts_preserves`, the proof goes through.

2.3.18 Redundant `transferMoneyBetweenAccounts_preserves`

Severity: Low

File `MoneyPreservation.thy`, lemma `reduceContractStep_preserves_money_acc_to_acc`, line *332* This lemma is weaker than `transferMoneyBetweenAccounts_preserves`. We can replace its usage site in line 376

```
using
  reduceContractStep_preserves_money_acc_to_acc
  validAndPositive_state.simps
 by blast
```

with

```
using transferMoneyBetweenAccounts_preserves validAndPositive_state.simps by auto
```

2.3.19 Duplicated lemmas

Severity: Low

File `PositiveAccounts.thy`, theorems `computeTransaction_gtZero`, `accountsArePositive`, lines *257, 369* These theorems are identical (modulo variable names), and one of them should be removed.
File `PositiveAccounts.thy`, `ValidState.thy`, lemma `valid_state_valid_accounts`, lines *381, 5* This lemma is defined twice, once in each of these files. One of them should be removed.

2.3.20 Redundant `computeTransaction` lemmas

Severity: Low

File `ValidState.thy`, lemmas `computeTransaction_preserves_valid_state_aux`, `computeTransaction_preserve_valid_state`, lines *160, 176* If `computeTransaction_preserves_valid_state_aux` is rewritten to have the same formulation as `computeTransaction_preserves_valid_state`, then the lemma (with the exact same proof) is still accepted, and these lemmas become duplicates of each other. Thus, no auxiliary lemma is needed.

2.3.21 Complicated formulation of `updateMoneyInAccount_money2_aux`

Severity: Low

File `MoneyPreservation.thy`, lemma `updateMoneyInAccount_money2_aux`, line *159* `updateMoneyInAccount_money2_aux` could be expressed simpler by removing the hypothesis `moneyToPay >= 0`, leaving

```
valid_map (((thisAccId, tok), money) # tail) ⟹
allAccountsPositive (((thisAccId, tok), money) # tail) ⟹
moneyInAccount thisAccId tok (((thisAccId, tok), money) # tail) > 0"
```

The proof of `updateMoneyInAccount_money2` can then in turn be trivially adjusted so it still works, by changing the step cases

```
cases "moneyInAccount accId tok ((thisAccIdTok, money) # tail) + moneyToPay ≤ 0"
```

to

```
cases "moneyInAccount accId tok ((thisAccIdTok, money) # tail) ≤ 0"
```

2.3.22 Complicated proofs that can be simplified

Severity: Low

File `MoneyPreservation.thy`, lemma `moneyInInput_is_positive`, line *53* The proof could be more general with `apply (cases x; simp)` instead of using `metis`.

File `MoneyPreservation.thy`, lemma `reductionLoop_preserves_money_NoPayment_not_ReduceNoWarning`, line *434* This lemma can be proved directly with `metis reductionLoop_preserves_money_NoPayment`, and reversing the order in which the lemmas are defined.
File `TimeRange.thy`, lemma `inIntervalIdempotentToIntersectInterval`, line *5* The lemma can use a shorter proof: `apply (cases min2;cases max2;auto) done`.
File `TimeRange.thy`, lemma `inIntervalIdempotency1, inIntervalIdempotency2`, lines *20, 36* These lemmas use the `smt` tactic and `metis` where a simpler Isar proof would work, for example:

```
lemma inIntervalIdempotency1 :
  assumes "inInterval (x, y) (intersectInterval b c)"
  shows "inInterval (x, y) b"
proof (cases b)
  case [simp]:(Pair b1 b2)
  thus ?thesis proof (cases c)
    case (Pair c1 c2)
    thus ?thesis using assms by (cases c1; cases c2; cases b1;cases b2; simp)
  qed
qed
```

File `SemanticsGuarantees.thy`, `Various lemmas/instantiations` Multiple lemmas and `linorder` instantiations in this file repeat auxiliary facts within the proof that are not necessary. For example, in the `linorder` instantiation for `Party`, lines 51–53 state

```
have "(x < y) = (x ≤ y ∧ ¬ y ≤ (x :: Token))"
  by (meson less_Tok.simps less_Token_def less_eq_Token_def linearToken)
thus "(x < y) = (x ≤ y ∧ ¬ y ≤ x)" by simp
```

This can be rewritten to avoid repeating the fact as

```
show "(x < y) = (x ≤ y ∧ ¬ y ≤ (x :: Token))"
  by (meson less_Tok.simps less_Token_def less_eq_Token_def linearToken)
```

This pattern appears many times in this file. For example, in the `Party` instantation alone, it is present on lines 51 − 53, 56 − 57, 77 − 80, and 83 − 84.

2.3.23 ■ Inconsistent style when applying constructor

Severity: Low

File `SingleInputTransactions.thy`, lemmas `beforeApplyAllLoopIsUseless, fixIntervalOnlySummary`, lines *275, 398* The lines mentioned in these lemmas display the resulting constructor before the function application, which differs from the general style in the rest of the codebase.

2.3.24 ■ Unsimplified boolean formulas

Severity: Low

File `SingleInputTransactions.thy`, lemma `computeTransactionIterative_aux2`, lines *708, 710* In multiple places, this lemma formulation includes top-level negation in front of nontrivial conjunctions and disjunctions. These negations should be distributed. Otherwise, the reader is taxed with the chore to mentally distribute the negation to understand the lemma.

2.3.25 Typo with "independet" in multiple lemmas

Severity: Low

File `SingleInputTransactions.thy`, lemmas `applyAllLoop_independet_of_acc_error1`, `applyAllLoop_independet_of_acc_error2`, lines *977, 987* In both of these lemmas, there is a typo with the word "independet".

2.3.26 Poorly named `acc` lemmas

Severity: Low

File `SingleInputTransactions.thy`, lemmas `applyAllLoop_independet_of_acc_error1`, `applyAllLoop_independet_of_acc_error2`, lines *977, 987* It is unclear what acc refers to in these lemma names, as the lemmas are about the independence of warnings and payments, not accounts.

2.3.27 Verbose lemma statement `playTraceAuxIterative_base_case`

Severity: Low

File `SingleInputTransactions.thy`, lemma `playTraceAuxIterative_base_case`, line *1063* The statement of this lemma is very verbose. A more natural (and slightly stronger) formulation could be

$$\begin{aligned} &\mathit{playTraceAux}\ \mathit{txOut}\ [\ (\!|\mathit{interval} = \mathit{inte}, \mathit{inputs} = [h]|\!), (\!|\mathit{interval} = \mathit{inte}, \mathit{inputs} = t|\!)\ ] \\ = &\ \mathit{playTraceAux}\ \mathit{txOut}\ [\ (\!|\mathit{interval} = \mathit{inte}, \mathit{inputs} = h\ \#\ t|\!)\ ] \end{aligned}$$

2.3.28 `playTrace_only_accepts_maxTransactionsInitialState` not written as `theorem`

Severity: Low

File `TransactionBound.thy`, lemma `playTrace_only_accepts_maxTransactionsInitialState`, line *316* This lemma seems like the main result of this file. Assuming it is an important result, we recommend writing it as a `theorem` rather than a `lemma`.

2.3.29 Inconsistent style with assumptions

Severity: Low

File `Timeout.thy`, lemmas `timedOutReduceContractUntilQuiescent_closes_contract`, `timedOutReduceContractStep_empties_accounts`, lines *201/204, 211/214* These lemmas use the hypothesis $\mathit{minTime}\ \mathit{sta} \leq \mathit{iniTime}$ and build a state $\mathit{sta}\ (\!|\mathit{minTime} := \mathit{iniTime}|\!)$ while other lemmas simply say $\mathit{minTime}\ \mathit{sta} = \mathit{iniTime}$. Readability would be improved by presenting these lemmas in the same style as the others, or documenting the need for these distinct presentations via code comments.

2.3.30 Function `validTimeInterval` unnecessarily unfolded in lemma

Severity: Low

File `TimeRange.thy`, lemma `reduceStep_ifCaseLtCt_aux`, line *234* For consistency, $a \leq b$ should be replaced by `validTimeInterval`.

2.3.31 Overly specific auxiliary lemma

Severity: Low

File `ValidState.thy`, lemma `reductionLoop_preserves_valid_state_aux`, line *73* This lemma on its own is very specific, and is only used in `reductionLoop_preserves_valid_state`. If possible, we recommend this lemma to be generalized or broken down into smaller lemmas, in order to present the arguments to the reader in smaller pieces.

2.3.32 `playTrace_preserves_valid_state` not written as `theorem`

Severity: Low

File `ValidState.thy`, lemma `playTrace_preserves_valid_state`, line *194* This lemma seems like the main result of this file. Assuming it is an important result, we recommend writing it as a `theorem` instead.

2.3.33 Unnecessary assumptions

Severity: Low

File `PositiveAccounts.thy`, lemmas `addMoneyToAccountPositive_match`, `addMoneyToAccountPositive_noMatch`, lines *12, 23* The assumptions

$$\forall x\ tok.\ positiveMoneyInAccountOrNoAccount\ x\ tok\ accs$$

in `addMoneyToAccountPositive_match` and

$$money > 0$$

in `addMoneyToAccountPositive_noMatch` are unnecessary.

File `PositiveAccounts.thy`, lemma `reduceContractStep_gtZero_Refund`, line *93* The lemma has an assumption that is mostly redundant.

```
Reduced
  ReduceNoWarning
  (ReduceWithPayment (Payment party (Party party) tok2 money))
  (state⦇accounts := newAccount⦈) Close
  =
Reduced wa eff newState newCont
```

A stronger lemma would be valid, which results from eliminating the assumption and changing the conclusion to `positiveMoneyInAccountOrNoAccount y tok3 newAccount`.

File `QuiescentResult.thy`, lemma `reduceContractStepPayIsQuiescent`, line *17* The assumption $cont =$ `Pay` $x21\ x22\ tok\ x23\ x24$ is unnecessary.

File `Timeout.thy`, lemma `timedOutReduce_only_quiescent_in_close_When`, line *43* The assumption $minTime\ sta \leq iniTime$ is unnecessary.

File `Timeout.thy`, lemma `timedOutReduce_only_quiescent_in_close`, line *122* The assumption $minTime\ sta \leq iniTime$ is unnecessary. However, removing it will require the later proof `timedOutReduceContractLoop_closes_contract` to be adjusted.

File `Timeout.thy`, lemma `timedOutReduceContractLoop_closes_contract`, lines *170, 173* The assumptions $minTime\ sta \leq iniTime$ and $minTimesta = iniTime$ are both present. The former is redundant.

File `TimeRange.thy`, lemma `reduceStep_ifCaseLtCt_aux`, line *234* The assumption $env = (|timeInterval = (a, b)|)$ is unnecessary.

File `ValidState.thy`, lemma `reductionStep_preserves_valid_state_Refund`, line *29* The assumption

$$newState = (|accounts = newAccounts, choices = newChoices, boundValues = newBoundValues, minTime = newMinTime|)$$

is unnecessary.

2.4 Isabelle implementation

2.4.1 Variable shadowing in `applyAllLoop`

Severity: Medium

File `Semantics.thy`, function `applyAllLoop`, line *575* The `cont` variable introduced by the pattern match shadows another `cont` variable, coming from the pattern match of an outer case expression, making the function harder to follow while also making it more error-prone to future changes.

2.4.2 Undescriptive name `moneyInPayment`

Severity: Low

File `MoneyPreservation.thy`, function `moneyInPayment`, line *5* The name of the function can be more precise. Perhaps `moneyInPaymentToParty` or `moneyInExternalPayment` would work.

2.4.3 Typo in section name

Severity: Low

File `OptBoundTimeInterval.thy`, line 37 Typo in section name: “Interval intesection”.

2.4.4 Typo in comment

Severity: Low

File `OptBoundTimeInterval.thy`, line 42 Typo in comment: “endpoits”.

2.4.5 Unclear need for multiple formulations for positive accounts

Severity: Low

File `PositiveAccounts.thy`, `Throughout` It is unclear what the use is for multiple formulations (and lemmas about) positive accounts. The first formulation (with the theorems `playTraceAux_gtZero` and `playTrace_gtZero`) is not used in any other modules but the alternative formulation is used instead. If both formulations are relevant, then it should be explained why.

2.4.6 Variable name discrepancy in `reductionLoop`

Severity: Low

File `Semantics.thy`, function `reductionLoop` When comparing this function against `specification-v3-rc1.pdf`, different names are used for a let-bound variable. It is `a` in the pdf and `newPayments` in the file `Semantics.thy`. There are similar issues in the function `reduceContractStep` in the equation for the `If` case, and in the function `giveMoney`.

2.4.7 Typo in constructor

Severity: Low

File `Semantics.thy`, function `applyCases`, line *505* Apparent typo in the error message constructor: the party mentioned should be `party2`.

2.4.8 Unclear function name `calculateNonAmbiguousInterval`

Severity: Low

File `Semantics.thy`, function `calculateNonAmbiguousInterval`, line *725* The meaning of the function is not obvious. It needs a comment to explain it.

2.4.9 Non-modularized file `SingleInputTransactions.thy`

Severity: Low

File `SingleInputTransactions.thy`, `Splitting File` This file is very long, and it covers more than just single-input transactions. For instance, about 530 lines at the beginning are rather dedicated to idempotence of certain operations. Then, the lemmas around lines 530 – 700 focus on “well-foundedness” of the recursion on contract steps. Then there is also a clear block of lemmas about “distributivity” of semantics over transaction lists.

Splitting the module, grouping the related lemmas, would help understanding the relationships between the groups.

2.4.10 Misleading function names

Severity: Low

File `SingleInputTransactions.thy`, function `inputsToTransactions`, line *9* This function name is not very descriptive of its meaning. It takes a transaction (both a time interval and a list of inputs) and returns a list of transactions at the same interval containing a single input each. A name like `splitTransactionIntoSingleInputTransactions` would convey better what the input and the output are.

Moreover, the code would be cleaner if the function takes a single argument of type `Transaction`, instead of asking the caller to rip apart its fields.

File `SingleInputTransactions.thy`, function `traceListToSingleInput`, line *18* This function name is not descriptive of what it does. Perhaps a more telling name could be `splitTransactionsIntoSingleInputTransactions`.

File `SingleInputTransactions.thy`, function `isSingleInput`, line *1222* This function should be renamed or repurposed. If renamed, `allAreSingleInput` more accurately reflects the meaning of the function. If repurposed, it should check that a single transaction has a single input, and `all isSingleInput` can be used to express the current behavior.

2.4.11 Unused parameter in `maxTransactionCaseList`

Severity: Low

File `TransactionBound.thy`, function `maxTransactionCaseList`, line *16* This function has a parameter of type `State` that is completely unused and can be removed.

2.4.12 Duplicated `isValidInterval` function

Severity: Low

File `TimeRange.thy`, function `isValidInterval`, line *231* This function duplicates `validTimeInterval` from `OptBoundTimeInterval.thy`, and the latter has certain additional properties proven about it specifically, so it makes sense to use the latter in both cases.

2.5 marlowe-cardano specification

2.5.1 Lack of guidelines for creating cooperating contracts

Severity: Medium

File `marlowe-cardano-specification.md`, Section `Life Cycle of a Marlowe Contract` Given that transactions are expected to work with Marlowe and non-Marlowe contracts simultaneously, it would be helpful to offer some guidelines for other contracts to avoid double satisfaction. Some degree of cooperation between the contracts that can appear in the same transaction is unavoidable.

One measure could be to ask every cooperating contract to refrain from paying to the payout validator. In this way, double satisfaction can not affect the payments of the Marlowe contract, if the Marlowe contract only pays to roles rather than addresses.

Another alternative would be to demand other contracts' outputs to use datums that are different from the roles used by the Marlowe contract for payments.

2.5.2 No reference to creating a minting policy

Severity: Low

File `marlowe-cardano-specification.md`, Section `Monetary Policy for Role Tokens` The minting policy is not specified, but a reference needs to be offered to explain how to create one.

2.5.3 Argument for Contract in `txInfoData` not specified

Severity: Low

File `marlowe-cardano-specification.md`, Section `Types` The argument by which the `Contract` in the `txInfoData` list corresponds to the given hash needs to be made explicit.

2.5.4 Merkleization section not detailed enough

Severity: Low

File `marlowe-cardano-specification.md`, Section `Merkleization` This section is too terse. It needs to explain what Merkleization is, and to motivate why it is needed.

When explaining how it works, it needs to make explicit that only the `Case` type is modified, and that in the semantics, only the `Input` type is modified. It needs to explain why the `Input` type needs to carry a hash and a contract, and why the evaluation of the contract is changed as described.

2.5.5 Unnecessary constraint

Severity: Low

File `marlowe-cardano-specification.md`, `Constraint 12. Merkleized continuations` This constraint is unnecessary to have in the Marlowe validator, since the construction of the arguments for evaluation of the Marlowe contract would fail. However, it would be useful to have it appear in the specification for users to be aware of it when crafting transactions. A note to motivate the presence of the constraint could be helpful.

2.5.6 Asymmetry between role and wallet payouts

Severity: Low

File `marlowe-cardano-specification.md`, `Constraint 15`, The marlowe validator allows multiple outputs to be paid to a wallet, but it demands that a single output exists when paying to a role instead. The motivation to use different approaches needs to be documented. This is implemented in `Scripts.hs` at line 371, in function `payoutConstraints`.

2.5.7 Incorrect description of `rolePayoutValidator`

Severity: Low

File `marlowe-cardano-specification.md`, Section `Plutus Validator for Marlowe Payouts` The description of the Marlowe payout validator in the specification states that it is parameterized by the currency symbol. However, this is not correct as the validator is unparameterized; rather, the datum type of the validator includes the currency symbol (as well as token name). The description should be modified to reflect this.

2.5.8 Unspecified initial state

Severity: Low

File `marlowe-cardano-specification.md`, Section `Life Cycle of a Marlowe Contract` The specification should say what the initial state of a Marlowe contract should be. In particular, creating a contract requires giving the minimum Ada to some account in the Marlowe state. Otherwise, Constraint 5 will reject the transactions that try to spend the output.

2.5.9 Unspecified behavior when multiple cases can apply

Severity: Low

File `Semantics.hs`, Function `applyCases`, line *597* If multiple cases in a case list can apply, the first one is taken. This behavior should be better communicated in the specification.

2.6 Haskell implementation

2.6.1 Name shadowing in `applyAllInputs`

Severity: Medium

File `Semantics.hs`, Function `applyAllInputs`, line *658* The binding `cont` from the `Applied` constructor shadows the `cont` variable coming from the pattern match in an enclosing case expression. This makes the code error-prone to subsequent changes and refactorings.

2.6.2 Non-isomorphic types in `playTraceAux`

Severity: Medium

File `Semantics.hs`, Function `playTraceAux`, line *710* The function in the Isabelle code takes a `TransactionOutputRecord` while the Haskell version takes a `TransactionOutput`. This means `TransactionError` cannot be an input to `playTraceAux` in Isabelle, possibly invalidating proofs about its properties.

2.6.3 Variable names differ from Isabelle code

Severity: Low

File `Semantics.hs`, Several functions Different variable names in Isabelle and Haskell make comparison harder. It is less of an issue when only one variable has been renamed in a function, but multiple variable renames require carefully mapping between the names to avoid confusion. For example, the code of `reduceContractStep` in line 482 (`Pay` case) is hard to compare.

Other name changes include:

- Line 456, function `updateMoneyInAccount` uses variable `money` where the Isabelle code uses `amount` and omits naming the last parameter.
- Line 473, function `giveMoneyToPay` uses variables `amount` and `accounts` instead of `money` and `accountsV` as in the Isabelle code.
- Line 541, function `reductionLoop` uses variable `con` instead of `ncontract`.
- Line 300, the data type `TransactionInput` corresponds to the type `Transaction` in the Isabelle code.
- Line 313, the data type `TransactionOutput` is isomorphic but not identical to the homonymous data type in the Isabelle code.
- Line 439, function `refundOne` uses a variable `balance` where the Isabelle code uses `money`.
- Line 463, function `addMoneyToAccount` uses variable `accounts` where the Isabelle code uses `accountsV`.

2.6.4 Naming of functions and variables

Severity: Low

File `Several files, several functions`
Several functions or variables could have more descriptive or precise names, for example:

- `Scripts.hs:193`: `validateBalances` could be called `allBalancesArePositive`.
- `Scripts.hs:206`: `validateInputs` could be called `allInputsAreAuthorized`.
- `Scripts.hs:324`: `checkScriptOutputAny` could be called `noOutputPaysToOwnAddress`, as it checks that *no* outputs pay to the script address.
- `Semantics.hs:439`: `refundOne` is named somewhat confusingly, and understanding the name requires the context of `reduceContractStep` where the function is called. Perhaps a better name would be `dropWhileNonPositiveAndUncons`.
- `Semantics.hs:597`: the binding `tailCase` should rather be named `tailCases`.

2.6.5 Unused functions

Severity: Low

File `Several files, several functions`
Several functions are unused and perhaps should be removed:

- `Semantics.hs:741`: `contractLifespanUpperBound` does not seem to be used anywhere, including tests.
- `Semantics.hs:680`: `isClose` is not used in the rest of the codebase (besides checking its behavior via testing). It should either be removed, or comments justifying its existence should be included.

In addition to that, the functions `validateBalances` and `totalBalance` (defined at `Semantics.hs:755` and `:762`) are only used in `Scripts.hs` and never reused, so they should probably be moved to `Scripts.hs`.

2.6.6 Comments

Severity: Low

File `Semantics.hs`, Function `refundOne`, line *439* The comment describing the function is overly concise, as it does not mention the function removing all non-positive accounts before the first positive one, and effectively `uncons`-ing the list.
File `Semantics.hs`, Function `addMoneyToAccount`, line *461* There is a typo in the comment: `accoun` is written instead of `account`.

2.6.7 Record updates in `playTraceAux`

Severity: Low

File `Semantics.hs`, Function `playTraceAux`, line *710* The function could have followed the Isabelle code more closely if it used a record update instead of creating a new `TransactionOutput` record from scratch.

2.6.8 Potential simplifications

Severity: Low

File `Semantics.hs`, Function `totalBalance`, line *755* The function uses `foldMap f . AssocMap.toList`. Here, `AssocMap.toList` is redundant.
File `Types.hs`, Class instance `Eq Contract`, line *873* The equality of cases for the `When` constructor would be simplified by using `cases1 == cases2`. If there is a reason for the more verbose equality condition, it should be outlined in a comment.

2.6.9 `computeTransaction` differs from the Isabelle implementation

Severity: Low

Helper function `evalValue, evalObservation`, lines *391 (Semantics.hs), 34 (Semantics.thy)* `evalValue` and `evalObservation` differ from the Isabelle implementation in the introduction of auxiliary variables to abbreviate the recursive calls. The comparison would be simpler if both definitions were consolidated.
Helper function `evalValue`, lines *395 (Semantics.hs), 35 (Semantics.thy)* The Isabelle implementation should use the helper function `moneyInAccount` instead of inlining its definition, so as to maintain consistency with the Haskell implementation.
Helper function `applyCases`, lines *596 (Semantics.hs), 498 (Semantics.thy)* The structure of function `applyCases` differs between the Haskell and Isabelle files. Specifically, the Haskell implementation has an additional function `applyAction` where the Isabelle implementation does not. A comment motivating the discrepancy would be needed. This is likely due to the lack of Merkleization in the Isabelle implementation.
Helper function `convertReduceWarnings`, lines *617 (Semantics.hs), 537 (Semantics.thy)* The Haskell function is implemented using `foldr`, while the Isabelle function uses explicit recursion, making a one-to-one comparison less obvious. If there is a reason for this discrepancy, such as `foldr` yielding some optimizations, this should be outlined in a comment.

2.6.10 Constraint implementations differ from description

Severity: Low

File `marlowe-cardano-specification.md`, Section `Plutus Validator for Marlowe Semantics` Some constraints mentioned in the specification are written in a different structure than the corresponding constraint in `Scripts.hs`. While such a discrepancy may be useful to minimize verbosity, a unified structure when possible would alleviate a side-by-side comparison. Examples of these differing structures include Constraint 6 and Constraint 14.

2.6.11 Missing argument of `computeTransaction`

Severity: Low

File `marlowe-cardano-specification.md`, Section `Relationship between Marlowe Validator and Semantics` The specification mentions the output datum as the (fifth) argument for the `computeTransaction` function, while it is not an argument to it.

2.6.12 Missing `smallMarloweValidator`

Severity: Low

File `marlowe-cardano-specification.md`, Various sections The specification mentions `smallMarlowe-Validator` in a few places, but it is never mentioned in the source code.

2.6.13 Incorrect constraint reference

Severity: Low

File `Scripts.hs`, Function `mkRolePayoutValidator`, line *150* This line should refer to Constraint 17 rather than Constraint 16.

2.6.14 `MarloweParams` differs from the specification

Severity: Low

File `Semantics.hs`, type `MarloweParams`, line *355* The specification defines `MarloweParams` to contain just the payout validator hash, while the definition in the Haskell code contains just the roles currency symbol.

2.6.15 Timeout boundary differs from the specification

Severity: Low

File `Semantics.hs`, type `reduceContractStep`, line *518* The specification mentions

> If a valid Transaction is computed with a TimeInterval with a start time bigger than the Timeout t, the contingency continuation c is evaluated.

where "bigger" implies strict inequality, while the code makes non-strict comparison. This difference needs to be acknowledged and further explained in the specification.

2.7 Haskell tests

2.7.1 More precise failure checks

Severity: Medium

File `Spec/Marlowe/Plutus/Specification.hs`, `Various tests` The tests use the functions `checkSemanticsTransaction` and `checkPayoutTransaction` to verify that various error conditions cause transactions to be rejected. These functions test that a transaction passes or fails, but when it fails, the functions do not consider the error cause. Checking the exact cause is necessary to ensure the transaction is rejected because of the intended reason and not because of some other error condition arising in a particular test case by coincidence.

The absence of this information makes it easier to accidentally produce a test that is not testing what is intended.

2.7.2 Missing tests

Severity: Medium

File `Spec/Marlowe/Semantics/Compute.hs`, The following properties could additionally be tested for `computeTransaction`:

- payment subtracts from an account,
- deposit adds to an account,
- `INotify` input produces the expected continuation,
- `IChoice` input produces the expected continuation,
- the hash of a successfully applied merkleized input matches the hash of the merkleized case.

Some of these are tested in `Spec/Marlowe/Semantics/Functions.hs` already for auxiliary functions.

File `Spec/Marlowe/Semantics/Functions.hs`, `Missing merkleization tests` The properties in this module do not seem to be tested with merkleized contracts or inputs except for `checkGetContinuation`. More merkleization tests should be added.

File `Spec/Marlowe/Semantics/Compute.hs`, function `checkFixInterval`, lines *100-102* The test `checkFixInterval` is never instantiated with an invalid interval that is in the past, meaning the function `fixInterval` is never tested for that case.

Chapter 3

Testing report

In this chapter we describe the tests implemented during the review. These tests are implemented with `cooked-validators`, a test framework developed by Tweag to find vulnerabilities in smart contracts, and can be found in the source code repository located at `https://github.com/tweag/audit-2023-marlowe`.

`cooked-validators` allows constructing tests which simulate the execution of sequences of transactions, also known as traces. There are three categories of tests in our test suite:

Happy traces Each test produces a single trace that evaluates as expected.

Uncovered vulnerabilities Each test produces a single trace showing some unexpected and often exploitable behavior noticed during the review.

Additional attack attempts Each test produces multiple traces that abuse the UTxOs paid to or from the Marlowe validator to discover a successful transaction that should not have been accepted.

3.1 Happy traces

For these tests, we copied example contracts from the `marlowe-cardano` repository, namely the trivial contract, the swap contract, and the escrow contract. We tested all possible paths through these contracts, including confirming the expected behavior upon reaching timeouts. Additionally, for the swap contract, we tested traces where roles are used instead of addresses, and we tested that transactions are rejected if the role tokens are missing.

3.2 Uncovered vulnerabilities

3.2.1 Negative deposits

This trace exhibits Issue 2.1.1. The flawed contract reads as follows:

```
When
  [ Case
      (Deposit
        (party bob)
        (party alice)
        (token silverCoinAsset)
        (Constant (-1))
      )
      continuationContract
  ]
  timeout
  Close
```

This allows a deposit of -1 silver coins from Bob to Alice. Then, the contract is started by paying a silver coin to the Marlowe validator, in addition to the minimum Ada required. Finally, Alice is able to leverage the negative deposit in the contract to steal the silver coin that is in the Marlowe script when the first input is executed. See the test `traceNegative` for more information.

3.2.2 Double satisfaction

This test exhibits Issue 2.1.2. To exhibit this vulnerability, the swap contract is instantiated to swap 5 silver coins from Alice with 2 golden coins from Bob.

However, in the same transaction, Carol pays 2 golden coins to a separate script, which only accepts transactions that send 2 golden coins to Alice. The intended behavior would pay 4 golden coins to Alice: 2 from Bob using the Marlowe script, and 2 from Carol with the separate script.

But when we pay from both scripts in a single transaction, both validators succeed even if only 2 golden coins are sent to Alice. In this fashion, Bob can steal the remaining two golden coins that should have gone to Alice. See the test `traceDoubleSat` for more information.

3.2.3 Duplicate account keys

This test exhibits Issue 2.1.8. This test starts with a Marlowe state with duplicate account keys. The Marlowe state is invalid, yet the Marlowe validator allows spending the output.

The contract makes a payment with the result of evaluating `AvailableMoney`. Users of Marlowe would expect the account to be left with 0 value after the payment. However, the final Marlowe state still reports a positive value.

There is no exploit identified with this unexpected behavior, yet it is plausible that an adversary could use it to confuse other users in a more complex scenario. See the test `traceExploitDupEntries` for more information.

3.3 Additional attack attempts

These attacks did not uncover additional vulnerabilities.

The attack mechanism in `cooked-validators` is to define a modification of a single transaction, known as a `Tweak`. Then, multiple traces can be generated from a single trace, each time applying the tweak to a different transaction in the source trace.

In general, tweaks are supposed to perform modifications that should be rejected by the validators being tested. If any trace is produced where a modified transaction does not cause the trace to be rejected, then `cooked-validators` can flag the trace as exposing a vulnerability.

Each kind of attack is characterized by the particular tweak being used.

3.3.1 Datum hijacking

The datum hijacking attack involves redirecting the datum when spending an output from a script to an unintended address. For Marlowe, if this attack succeeds, the contract could not be executed as intended, as the continuation would no longer be the Marlowe script.

We used `cooked-validators` to automatically run the datum hijacking attack on all the happy traces, attempting to hijack the datum in all relevant transactions. However, none of these tests succeeded, meaning we did not find any instance of a Marlowe contract vulnerable to datum hijacking. These tests have names prefixed with `traceDatumHijack`.

3.3.2 Datum tampering

Rather than redirecting the datum as above, another potential attack is altering a field or fields of the datum when it is being paid to the Marlowe script. This is known as a datum tampering attack. We attempted to tamper the datum in the trivial contract in two different ways:

- By modifying the continuation contract to `Close`.
- By modifying the accounts in the state of the datum to be empty.

In both cases, the modified transaction failed as expected, so there is no evidence that datum tampering is possible for Marlowe contracts. These tests have names prefixed with `traceTamperDatum`.

3.3.3 Adding extraneous tokens

Another attack involves adding an extra token to either the value paid to the Marlowe validator or as an additional input during a transaction. If accepted, this would give an attacker a method to bring transactions close to protocol limits, to the point that we may be unable to spend produced UTxOs. These tests, whose names are prefixed with `traceExtraneousToken` and `traceExtraRedeemerToken`, all fail as expected.

Appendix A

Tweag's Modifications

One of the modifications is to support Aeson 1.5.x, which is a hard dependency of the team's tools:

```
diff --git a/marlowe/src/Language/Marlowe/Core/V1/Semantics/Types.hs b/marlowe/src/Language/Marlowe/Core/V1/Semantics/Types.hs
index a333b8f..a78f7df 100644
--- a/marlowe/src/Language/Marlowe/Core/V1/Semantics/Types.hs
+++ b/marlowe/src/Language/Marlowe/Core/V1/Semantics/Types.hs
@@ -84,12 +84,12 @@ import Control.Applicative ((<*>), (<|>))
 import Control.Newtype.Generics (Newtype)
 import qualified Data.Aeson as A
 import qualified Data.Aeson as JSON
-import qualified Data.Aeson.KeyMap as KeyMap
 import Data.Aeson.Types hiding (Error, Value)
 import qualified Data.Aeson.Types as JSON
 import Data.ByteString.Base16.Aeson (EncodeBase16(EncodeBase16))
 import qualified Data.ByteString.Base16.Aeson as Base16.Aeson
 import qualified Data.Foldable as F
+import qualified Data.HashMap.Strict as HashMap
 import Data.Scientific (floatingOrInteger, scientific)
 import Data.String (IsString(..))
 import Data.Text (pack)
@@ -367,8 +367,8 @@ instance ToJSON Input where
     let
       obj = case toJSON content of
         Object obj -> obj
-        _          -> KeyMap.empty
-    in Object $ obj `KeyMap.union` KeyMap.fromList
+        _          -> HashMap.empty
+    in Object $ obj Haskell.<> HashMap.fromList
         [ ("merkleized_continuation", toJSON continuation)
         , ("continuation_hash", toJSON $ EncodeBase16 $ fromBuiltin hash)
         ]
```

Listing 1: Patch for **`Language.Marlowe.Core.V1.Semantics.Types`**

The other modification is to use the Plutus V1 analogs of the features from Plutus V2, since the team's tools for Plutus V2 are not stable enough yet:

```
diff --git a/marlowe/src/Language/Marlowe/Scripts.hs b/marlowe/src/Language/Marlowe/Scripts.hs
index 6027202..faec1fd 100644
--- a/marlowe/src/Language/Marlowe/Scripts.hs
+++ b/marlowe/src/Language/Marlowe/Scripts.hs
@@ -61,12 +61,11 @@ import GHC.Generics (Generic)
 import Language.Marlowe.Core.V1.Semantics as Semantics
 import Language.Marlowe.Core.V1.Semantics.Types as Semantics
 import Language.Marlowe.Pretty (Pretty(..))
-import qualified Plutus.Script.Utils.Typed as Scripts
-import Plutus.Script.Utils.V2.Typed.Scripts (mkTypedValidator, mkUntypedValidator)
-import qualified Plutus.Script.Utils.V2.Typed.Scripts as Scripts
+import qualified Ledger.Typed.Scripts as Scripts
+import Plutus.Script.Utils.V1.Typed.Scripts (mkTypedValidator, mkUntypedValidator)
 import qualified Plutus.V1.Ledger.Address as Address (scriptHashAddress)
 import qualified Plutus.V1.Ledger.Value as Val
-import Plutus.V2.Ledger.Api
+import Plutus.V1.Ledger.Api
    ( Credential(..)
    , CurrencySymbol
    , Datum(Datum)
@@ -85,9 +84,9 @@ import Plutus.V2.Ledger.Api
    , ValidatorHash
    , mkValidatorScript
    )
-import qualified Plutus.V2.Ledger.Api as Ledger (Address(Address))
-import Plutus.V2.Ledger.Contexts (findDatum, findDatumHash, txSignedBy, valueSpent)
-import Plutus.V2.Ledger.Tx (OutputDatum(OutputDatumHash), TxOut(TxOut, txOutAddress, txOutDatum, txOutValue))
+import qualified Plutus.V1.Ledger.Api as Ledger (Address(Address))
+import Plutus.V1.Ledger.Contexts (findDatum, findDatumHash, txSignedBy, valueSpent)
+import Plutus.V1.Ledger.Tx (TxOut(TxOut, txOutAddress, txOutDatumHash, txOutValue))
 import PlutusTx (makeIsDataIndexed, makeLift)
 import qualified PlutusTx
 import qualified PlutusTx.AssocMap as AssocMap
@@ -310,13 +309,13 @@ mkMarloweValidator

     -- Check that address, value, and datum match the specified.
     checkScriptOutput :: Ledger.Address -> Maybe DatumHash -> Val.Value -> TxOut -> Bool
-    checkScriptOutput addr hsh value TxOut{txOutAddress, txOutValue, txOutDatum=OutputDatumHash svh} =
+    checkScriptOutput addr hsh value TxOut{txOutAddress, txOutValue, txOutDatumHash=Just svh} =
                   txOutValue == value && hsh == Just svh && txOutAddress == addr
     checkScriptOutput _ _ _ _ = False

     -- Check that address and datum match the specified, and that value is at least that required.
     checkScriptOutputRelaxed :: Ledger.Address -> Maybe DatumHash -> Val.Value -> TxOut -> Bool
-    checkScriptOutputRelaxed addr hsh value TxOut{txOutAddress, txOutValue, txOutDatum=OutputDatumHash svh} =
+    checkScriptOutputRelaxed addr hsh value TxOut{txOutAddress, txOutValue, txOutDatumHash=Just svh} =
                   txOutValue `Val.geq` value && hsh == Just svh && txOutAddress == addr
     checkScriptOutputRelaxed _ _ _ _ = False
```

Listing 2: Patch for **`Language.Marlowe.Scripts`**

Appendix B

A proof of lemma `insert_valid`

This proof was checked in `MList.thy`.

```
lemma insert_set:
  "set (map fst (insert a b xs)) = set (map fst xs) ∪ { a }"
proof (induct xs)
  case Nil
  thus ?case by simp
next
  case (Cons y ys)
  thus ?case proof (cases y)
    case [simp]:(Pair y1 y2)
    thus ?thesis using Cons by auto
  qed
qed

lemma insert_sorted:
  "List.sorted (map fst c) ⟹ List.sorted (map fst (insert a b c))"
proof (induct c)
  case Nil
  show ?case by simp
next
  case (Cons c1 rest)
  show ?case proof (cases c1)
    case [simp]:(Pair x y)
    thus ?thesis proof (cases "a<x")
      assume "a<x"
      thus ?thesis using Cons.prems by auto
    next
      assume "¬ a<x"
      thus ?thesis proof (cases "a>x")
        assume ax:"a>x"
        have h1: "sorted (map fst (insert a b rest))" using Cons by auto
        from Cons.prems have "∀y ∈ set (map fst rest). x≤y" by simp
        then have "sorted (map fst (c1 # insert a b rest))" using h1 ax insert_set[of a b rest] by auto
        thus ?thesis using ax by auto
      next
        assume "¬ a>x"
        thus ?thesis using Cons.prems by auto
      qed
    qed
  qed
qed

lemma insert_distinct :
  "List.distinct (map fst c)
   ⟹
   List.sorted (map fst c)
   ⟹
   List.distinct (map fst (MList.insert a b c))"
proof (induct c)
  case Nil
```

```
  show ?case by simp
next
  case (Cons c1 rest)
  show ?case proof (cases c1)
    case [simp]:(Pair x y)
    thus ?thesis proof (cases "a<x")
      assume "a<x"
      thus ?thesis using Cons.prems by auto
    next
      assume nax:"¬ a<x"
      thus ?thesis proof (cases "a>x")
        assume ax:"a>x"
        have "distinct (map fst (insert a b rest))" using Cons by auto
        then have "distinct (map fst (c1 # insert a b rest))" using ax Cons.prems insert_set[of a b rest] by auto
        thus ?thesis using ax by auto
      next
        assume "¬ a>x"
        thus ?thesis using Cons.prems by auto
      qed
    qed
  qed
qed

theorem insert_valid2 : "valid_map c ⟹ valid_map (MList.insert a b c)"
  using insert_sorted[of c a b] insert_distinct[of c a b] by fastforce
```