Marlowe Specification

Version 3

Pablo Lamela Seijas Alexander Nemish David Smith

Simon Thompson Hernán Rajchert Brian Bush

December 31, 1979

Contents

1 Marlowe **2**

1.1 Introduction 2

1.2 The Marlowe Model 3

1.2.1 Data types 3

1.2.2 Quiescent 3

1.2.3 Participants, accounts and state 3

1.2.4 Core and Extended 4

1.3 Specification generation and nomenclature 4

1.4 Blockchain agnostic 4

2 Marlowe Core **6**

2.1 Types 6

2.1.1 Participants, roles and addresses 6

2.1.2 Multi-Asset token 7

2.1.3 Accounts 7

2.1.4 Choices 8

2.1.5 Values and Observations 8

2.1.6 Actions and inputs 9

2.1.7 Contracts 10

2.1.8 State and Environment 12

2.2 Semantics 12

2.2.1 Compute Transaction 13

2.2.2 Fix Interval 14

2.2.3 Apply All Inputs 14

2.2.4 Reduce Contract Until Quiescent 15

2.2.5 Reduction Loop 16

2.2.6 Reduce Contract Step 16

2.2.7 Apply Input 18

2.2.8 Apply Cases 19

2.2.9 Utilitiesn 20

2.2.10 Evaluate Value 21

2.2.11 Evaluate Observation 24

3 Marlowe Guarantees 26

3.1 Money Preservation 26
3.2 Contracts Always Close 27
3.3 Positive Accounts 27
3.4 Quiescent Result 28
3.5 Reducing a Contract until Quiescence Is Idempotent 28
3.6 Split Transactions Into Single Input Does Not Affect the Result 28
3.6.1 Termination Proof 29
3.6.2 All Contracts Have a Maximum Time 29
3.6.3 Contract Does Not Hold Funds After it Closes 29
3.6.4 Transaction Bound 29

Chapter 1

Marlowe

1.1 Introduction

Marlowe is a special purpose or domain-specific language (DSL) that is designed to be usable by someone who is expert in the field of financial contracts, somewhat lessening the need for programming skills.

Marlowe is modelled on special-purpose financial contract languages popularised in the last decade or so by academics and enterprises such as LexiFi [1], which provides contract software in the financial sector. In developing Marlowe, we have adapted these languages to work on any blockchain §1.4.

Where we differ from non-blockchain approaches is in how we make sure that the contract is followed. In the smart contracts world there is a saying "Code is law", which implies that the assets deposited in a contract will follow its logic, without the ability of a human to change the rules. This applies for both the intended and not intended behaviour (in the form of bugs or exploits).

To reduce the probability of not intended behaviour, the Marlowe DSL is designed with simplicity in mind. Without loops, recursion, or other features that general purposes smart-contract languages (E.g: Plutus, Solidity) have, it is easier to make certain claims. Each Marlowe contract can be reasoned with a static analizer to avoid common pitfalls such as trying to Pay more money than the available. And the *executable semantics* that dictates the logic of **all** Marlowe contracts is formalized with the proof-assistant Isabelle.

Chapter §1 provides an overview of the Marlowe language. Chapter §2 defines the Core language and semantics in detail. Chapter §3 presents proofs that

[1] https://www.lexifi.com/

guarantee that Marlowe contracts possess properties desirable for financial agreements.

1.2 The Marlowe Model

Marlowe *Contract*s describe a series of steps, typically by describing the first step, together with another (sub-) contract that describes what to do next. For example, the contract *Pay* $a\ p\ t\ v\ c$ says “make a payment of v number of tokens t to the party p from the account a, and then follow the contract c”. We call c the continuation of the contract. All paths of the contract are made explicit this way, and each *Contract* term is executed at most once.

1.2.1 Data types

The *Value*s and *Observations* §2.1.5 only works with integers and booleans respectively. There is no custom data types, records, tuples, nor string manipulation. There is also no floating point numbers, so in order to represent currencies it is recommended to work with cents. Dates are only used in the context of Timeouts and they are absolute, but it is likely we’ll add relative times in a future version.

1.2.2 Quiescent

The blockchain can’t force a participant to make a transaction. To avoid having a participant blocking the execution of a contract, whenever an *Input* is expected, there is a Timeout with a contingency continuation. For each step, we can know in advance how long it can last, and we can extend this to know the maximum duration and the amount of transactions of a contract.

1.2.3 Participants, accounts and state

Once we define a contract, we can see how many participants it will have. The number of participants is fixed for the duration of the contract, but there are mechanisms to trade participation §2.1.1.

Each participant has an internal account that allows the contract to define default owner for assets §2.1.3. Whenever a *Party* deposits an asset in the contract, they need to decide the default owner of that asset. Payments can be made to transfer the default owner or to take the asset out of the contract.

If the contract is closed, the default owner can redeem the assets available in their internal accounts.

The accounts, choices, and variables stored in the *State* §2.1.8 are global to that contract.

1.2.4 Core and Extended

The set of types and functions that conform the semantics executed in the blockchain is called *Marlowe Core*, and it's formalized in chapter §2. To improve usability, there is another set of types and functions that compile to core, and it is called *Marlowe Extended*.

In the first version of the extended language, the only modification to the DSL is the addition of template parameters. These allows an initial form of contract reutilization, allowing to instantiate the same contract with different *Value*s and *Timeout*s. For the moment, the extended language is not formalized in this specification but it will be added in the future

1.3 Specification generation and nomenclature

The Marlowe specification is formalized using the proof assistant Isabelle[2]. The code is written in a literate programming style and this document is generated from the proofs. This improves code documentation and lowers the probability of stale information.

As a drawback, the code/doc organization is more rigid. Isabelle require us to define code in a bottom-up approach, having to define first the dependencies and later the most complex structures.

The notation is closer to a Mathematical formula than a functional programming language. There are some configurations in the *SpecificationLatexSugar* theory file that makes the output be closer to code.

1.4 Blockchain agnostic

Marlowe is currently implemented on the Cardano Blockchain, but is designed to be Blockchain agnostic. The same way that modern languages

[2] https://isabelle.in.tum.de/

like C++ and Java have compilers that target intel/ARM, Marlowe could be implemented in Ethereum and other blockchain platforms.

There are some assumptions we make on the underlying Blockchain that allows Marlowe Semantics to serve as a common abstraction:

In order to define the different *Token*s that are used as currency in the participants accounts §2.1.3, we need to be able to express a *TokenName* and *CurrencySymbol*.

type-synonym *TokenName* = *ByteString*
type-synonym *CurrencySymbol* = *ByteString*

To define a fixed participant in the contract §2.1.1 and to make payouts to them, we need to express an *Address*.

type-synonym *Address* = *ByteString*

In the context of this specification these string types are opaque, and we don't enforce a particular encoding or format.

The *Timeout*s that prevent us from waiting forever for external *Input*s are represented by the number of milliseconds from the Unix Epoch [3].

type-synonym *POSIXTime* = *int*

type-synonym *Timeout* = *POSIXTime*

The *TimeInterval* that define the validity of a transaction is a tuple of exclusive start and end time.

type-synonym *TimeInterval* = *POSIXTime* × *POSIXTime*

[3] January 1st, 1970 at 00:00:00 UTC

Chapter 2

Marlowe Core

2.1 Types

This section introduces the data types of *Marlowe Core*, which are composed by the Abstract Syntax Tree (AST) that define the Marlowe DSL and also the types required to compute a *Transaction*.

Because of the literate programming nature of Isabelle §1.3, the types are defined bottom-up. To follow just the DSL, a reader can start by looking at a *Contract* definition §2.1.7.

2.1.1 Participants, roles and addresses

We should separate the notions of participant, role, and addresses in a Marlowe contract. A participant (or *Party*) in the contract can be represented by either a fixed *Address* or a *Role*.

type-synonym *RoleName* = *ByteString*

datatype *Party* =
  Address Address
 | *Role RoleName*

The address party is defined by a Blockhain specific *Address* §1.4 and it cannot be traded (it's fixed in the lifetime of a contract).

A *Role* in the other hand allows the participation of the contract to be dynamic. Any user that can prove to have permissions to act as *RoleName* is able to carry out the actions assigned §2.1.6, and redeem the payments issued to that role. The roles could be implemented as tokens[1] that can be traded. By minting multiple tokens, more people could prove to have permission to that role, allowing for more complex scenarios.

2.1.2 Multi-Asset token

Inspired by Cardano's Multi-Asset tokens [2], Marlowe also supports to transact with different assets. A *Token* is constructed by a *CurrencySimbol* that represents the monetary policy of the *Token* and a *TokenName* which allows to have multiple tokens with the same policy.

datatype *Token* = *Token CurrencySymbol TokenName*

The marlowe semantics treats both types as opaque *ByteString*.

2.1.3 Accounts

The Marlowe model allows for a contract to store assets. All participants of the contract implicitly own an account identified with an *AccountId*.

type-synonym *AccountId* = *Party*

All assets stored in the contract must be in an internal account for one of the parties; this way, when the contract is closed, all remaining assets can be redeemed by their respective owners. These accounts are local: they only exist (and are accesible) within the contract.

type-synonym *Accounts* = ((*AccountId* × *Token*) × *int*) *list*

During the course of the contract, assets can be deposited into an internal account using the term "*When* [*Deposit accountId party token value*] *timeout continuation*" . We can transfer assets internally or externally by using the term "*Pay accountId payee token value continuation*", where *Payee* is:

datatype *Payee* = *Account AccountId*
| *Party Party*

A *Pay* always takes money from an internal *AccountId*, and the *Payee* defines if we transfer internally (*Account p*) or externally (*Party p*)

[1] In the Cardano implementation roles are represented by native tokens and they are distributed to addresses at the time a contract is deployed to the blockchain

[2] https://docs.cardano.org/native-tokens/learn

2.1.4 Choices

Choices – of integers – are identified by *ChoiceId* which is defined with a canonical name and the *Party* who had made the choice:

type-synonym *ChoiceName* = *ByteString*
datatype *ChoiceId* = *ChoiceId ChoiceName Party*

Choices are *Bound*ed. As an argument for the *Choice* action §2.1.6, we pass a list of *Bound*s that limit the integer that we can choose. The *Bound* data type is a tuple of integers that represents an **inclusive** lower and upper bound.

type-synonym *Bound* = *int* × *int*

2.1.5 Values and Observations

We can store a *Value* in the Marlowe State §2.1.8 using the *Let* construct §2.1.7, and we use a *ValueId* to referrence it

datatype *ValueId* = *ValueId ByteString*

*Value*s and *Observation*s are language terms that interact with most of the other constructs. *Value* evaluates to an integer and *Observation* evaluates to a boolean using *evalValue* §2.2.10 and *evalObservation* §2.2.11 respectively.

They are defined in a mutually recursive way as follows:

```
datatype Value = AvailableMoney AccountId Token
               | Constant int
               | NegValue Value
               | AddValue Value Value
               | SubValue Value Value
               | MulValue Value Value
               | DivValue Value Value
               | ChoiceValue ChoiceId
               | TimeIntervalStart
               | TimeIntervalEnd
               | UseValue ValueId
               | Cond Observation Value Value
and Observation = AndObs Observation Observation
                | OrObs Observation Observation
                | NotObs Observation
                | ChoseSomething ChoiceId
                | ValueGE Value Value
                | ValueGT Value Value
```

$\mid$ *ValueLT Value Value*
$\mid$ *ValueLE Value Value*
$\mid$ *ValueEQ Value Value*
$\mid$ *TrueObs*
$\mid$ *FalseObs*

Three of the *Value* terms look up information in the Marlowe state: *AvailableMoney p t* reports the amount of token *t* in the internal account of party *p*; *ChoiceValue i* reports the most recent value chosen for choice *i*, or zero if no such choice has been made; and *UseValue i* reports the most recent value of the variable *i*, or zero if that variable has not yet been set to a value.

Constant v evaluates to the integer *x*, while *NegValue x*, *AddValue x y*, *SubValue x y*, *MulValue x y*, and *DivValue x y* provide the common arithmetic operations $- x$, $x + y$, $x - y$, $x * y$, and x / y, where division always rounds (truncates) its result towards zero.

Cond b x y represents a condition expression that evaluates to *x* if *b* is true and to *y* otherwise.

The last *Value*s *TimeIntervalStart* and *TimeIntervalEnd* evaluate respectively to the start or end of the validity interval for the Marlowe transaction.

For the observations, the *ChoseSomething i* term reports whether a choice *i* has been made thus far in the contract.

The terms *TrueObs* and *FalseObs* provide the logical constants *true* and *false*. The logical operators $\neg x$, $x \wedge y$, and $x \vee y$ are represented by the terms *NotObs x*, *AndObs x y*, and *OrObs x y*, respectively.

Value comparisons $x < y$, $x \leq y$, $x > y$, $x \geq y$, and $x = y$ are represented by *ValueLT x y*, *ValueLE x y*, *ValueGT x y*, *ValueGE x y*, and *ValueEQ x y*.

2.1.6 Actions and inputs

*Action*s and *Input*s are closely related. An *Action* can be added in a list of *Case*s §2.1.7 as a way to declare the possible external *Input*s a *Party* can include in a *Transaction* at a certain time.

The different types of actions are:

datatype *Action* = *Deposit AccountId Party Token Value*
$\mid$ *Choice ChoiceId Bound list*
$\mid$ *Notify Observation*

A *Deposit a p t v* makes a deposit of $\#v$ *Token*s *t* from *Party p* into account *a*.

A choice *Choice i bs* is made for a particular choice identified by the ChoiceId §2.1.4 *i* with a list of inclusive bounds *bs* on the values that are acceptable. For example, [*Bound 0 0*, *Bound 3 5*] offers the choice of one of 0, 3, 4 and 5.

A notification can be triggered by any participant as long as the *Observation* evaluates to *true*. If multiple notify are present in the *Case* list, the first one with a *true* observation is matched.

For each *Action*, there is a corresponding *Input* that can be included inside a *Transaction*

```
type-synonym ChosenNum = int

datatype Input = IDeposit AccountId Party Token int
               | IChoice ChoiceId ChosenNum
               | INotify
```

The differences between them are:

- *Deposit* uses a *Value* while *IDeposit* has the *int* it was evaluated to with *evalValue* §2.2.10.
- *Choice* defines a list of valid *Bound*s while *IChoice* has the actual *ChosenNum*.
- *Notify* has an *Observation* while *INotify* doesn't have arguments, the *Observation* must evaluate to true inside the *Transaction*

2.1.7 Contracts

Marlowe is a continuation based language, this means that a *Contract* can either be a *Close* or another construct that recursively has a *Contract*. Eventually, **all** contracts end up with a *Close* construct.

Case and *Contract* are defined in a mutually recursive way as follows:

```
datatype Case = Case Action Contract
and Contract = Close
             | Pay AccountId Payee Token Value Contract
             | If Observation Contract Contract
```

| *When Case list Timeout Contract*
| *Let ValueId Value Contract*
| *Assert Observation Contract*

Close is the simplest contract, when we evaluate it, the execution is completed and we generate *Payment*s §**??** for the assets in the internal accounts to their default owners [3].

The contract *Pay a p t v c*, generates a *Payment* from the internal account *a* to a payee §2.1.3 *p* of $\#v$ *Token*s and then continues to contract *c*. Warnings will be generated if the value *v* is negative, or if there is not enough in the account to make the payment in full. In the latter case, a partial payment (of the available amount) is made

The contract *If obs x y* allows branching. We continue to branch *x* if the *Observation obs* evaluates to *true*, or branch*y* otherwise.

When is the most complex constructor for contracts, with the form *When cs t c*. The list *cs* contains zero or more pairs of *Action*s and *Contract* continuations. When we do a *computeTransaction* §2.2.1, we follow the continuation associated to the first *Action* that matches the *Input*. If no action is matched it returns a *ApplyAllNoMatchError*. If a valid *Transaction* is computed with a *TimeInterval* with a start time bigger than the *Timeout t*, the contingency continuation *c* is evaluated. The explicit timeout mechanism is what allow Marlowe to avoid waiting forever for external inputs.

A *Let* contract *Let i v c* allows a contract to name a value using an identifier *i*. In this case, the expression *v* is evaluated, and stored with the name *i*. The contract then continues as *c*. As well as allowing us to use abbreviations, this mechanism also means that we can capture and save volatile values that might be changing with time, e.g. the current price of oil, or the current time, at a particular point in the execution of the contract, to be used later on in contract execution.

An assertion contract *Assert b c* does not have any effect on the state of the contract, it immediately continues as *c*, but it issues a warning when the observation *b* is false. It can be used to ensure that a property holds in any given point of the contract, since static analysis will fail if any execution causes a warning. the *Assert* term might be removed from future on-chain versions of Marlowe.

[3] Even if the payments are generated one at a time (per account and per Token), the cardano implementation generates a single transaction

2.1.8 State and Environment

The internal state of a Marlowe contract consists of the current balances in each party's account, a record of the most recent value of each type of choice, a record of the most recent value of each variable, and the minimum time for which input can be applied to the contract. The data for accounts, choices, and bound values are stored as association lists.

record *State* = *accounts* :: *Accounts*
choices :: (*ChoiceId* × *ChosenNum*) *list*
boundValues :: (*ValueId* × *int*) *list*
minTime :: *POSIXTime*

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

record *Environment* = *timeInterval* :: *TimeInterval*

— TODO: see if we want to add data types of Semantic here (Transaction, etc) or if we want to move this types to Semantic

datatype *IntervalError* = *InvalidInterval TimeInterval*
| *IntervalInPastError POSIXTime TimeInterval*

datatype *IntervalResult* = *IntervalTrimmed Environment State*
| *IntervalError IntervalError*

2.2 Semantics

Marlowe's behavior is defined via the *operational semantics* (or *executable semantics*) of the Isabelle implementation of its *computeTransaction* function. That function calls several auxiliary functions to apply inputs and find a quiescent state of the contract. These, in turn, call evaluators for *Value* and *Observation*.

2.2.1 Compute Transaction

The entry point into Marlowe semantics is the function *computeTransaction* that applies input to a prior state to transition to a posterior state, perhaps reporting warnings or throwing an error, all in the context of an environment for the transaction.

computeTransaction :: *Transaction* ⇒ *State* ⇒ *Contract* ⇒ *TransactionOutput*

FIXME: Print record: *Transaction*

datatype *TransactionOutput* =
  TransactionOutput
    TransactionOutputRecord
  | *TransactionError TransactionError*

FIXME: Print record: *TransactionOutputRecord*

This function adjusts the time interval for the transaction using *fixInterval* and then applies all of the transaction inputs to the contract using *applyAllInputs*. It reports relevant warnings and throws relevant errors.

```
computeTransaction ::
  Transaction_ext () -> State_ext () -> Contract -> TransactionOutput;
computeTransaction tx state contract =
  let {
    inps = inputs tx;
  } in (case fixInterval (interval tx) state of {
         IntervalTrimmed env fixSta ->
           (case applyAllInputs env fixSta contract inps of {
             ApplyAllSuccess reduced warnings payments newState cont
->
               (if not reduced &&
                     (not (equal_Contract contract Close) ||
                       null (accounts state))
                 then TransactionError TEUselessTransaction
                 else TransactionOutput
                        (TransactionOutputRecord_ext warnings payments
newState
                          cont ()));
             ApplyAllNoMatchError -> TransactionError TEApplyNoMatchError;
             ApplyAllAmbiguousTimeIntervalError ->
```

```
              TransactionError TEAmbiguousTimeIntervalError;
          });
        IntervalError errora -> TransactionError (TEIntervalError errora);
      });
```

2.2.2 Fix Interval

The *fixInterval* functions combines the minimum-time constraint of *State* with the time interval of *Environment* to yield a “trimmed” validity interval and a minimum time for the new state that will result from applying the transaction. It throws an error if the interval is nonsensical or in the past.

FIXME: print type synonym: *IntervalResult*

```
fixInterval :: (Int, Int) -> State_ext () -> IntervalResult;
fixInterval (low, high) state =
  let {
    curMinTime = minTime state;
    newLow = max low curMinTime;
    curInterval = (newLow, high);
    env = Environment_ext curInterval ();
    newState = minTime_update (\ _ -> newLow) state;
  } in (if less_int high low then IntervalError (InvalidInterval (low,
high))
         else (if less_int high curMinTime
                then IntervalError (IntervalInPastError curMinTime (low,
high))
                else IntervalTrimmed env newState));
```

2.2.3 Apply All Inputs

The *applyAllInputs* function iteratively progresses the contract and applies the transaction inputs to the state, checking for errors along the way and continuing until all the inputs are consumed and the contract reaches a quiescent state.

```
applyAllInputs ::
  Environment_ext () -> State_ext () -> Contract -> [Input] -> ApplyAllResult;
applyAllInputs env state contract inputs =
  applyAllLoop False env state contract inputs [] [];
applyAllLoop ::
  Bool ->
    Environment_ext () ->
      State_ext () ->
        Contract ->
          [Input] -> [TransactionWarning] -> [Payment] -> ApplyAllResult;
applyAllLoop contractChanged env state contract inputs warnings payments
=
  (case reduceContractUntilQuiescent env state contract of {
    ContractQuiescent reduced reduceWarns pays curState cont ->
      (case inputs of {
        [] -> ApplyAllSuccess (contractChanged || reduced)
                (warnings ++ convertReduceWarnings reduceWarns)
                (payments ++ pays) curState cont;
        input : rest ->
          (case applyInput env curState input cont of {
            Applied applyWarn newState conta ->
              applyAllLoop True env newState conta rest
                (warnings ++
                  convertReduceWarnings reduceWarns ++
                    convertApplyWarning applyWarn)
                (payments ++ pays);
            ApplyNoMatchError -> ApplyAllNoMatchError;
          });
      });
    RRAmbiguousTimeIntervalError -> ApplyAllAmbiguousTimeIntervalError;
  });
```

2.2.4 Reduce Contract Until Quiescent

The *reduceContractUntilQuiescent* executes as many non-input steps of the contract as is possible. Marlowe semantics do not allow partial execution of a series of non-input steps.

```
reduceContractUntilQuiescent ::
  Environment_ext () -> State_ext () -> Contract -> ReduceResult;
reduceContractUntilQuiescent env state contract =
  reductionLoop False env state contract [] [];
```

2.2.5 Reduction Loop

The *reductionLoop* function attempts to apply the next, non-input step to the contract. It emits warnings along the way and it will through an error if it encounters an ambiguous time interval.

```
reductionLoop ::
  Bool ->
    Environment_ext () ->
      State_ext () -> Contract -> [ReduceWarning] -> [Payment] -> ReduceResult;
reductionLoop reduced env state contract warnings payments =
  (case reduceContractStep env state contract of {
    Reduced warning effect newState ncontract ->
      let {
        newWarnings =
          (if equal_ReduceWarning warning ReduceNoWarning then warnings
            else warning : warnings);
        a = (case effect of {
              ReduceNoPayment -> payments;
              ReduceWithPayment payment -> payment : payments;
            });
      } in reductionLoop True env newState ncontract newWarnings a;
    NotReduced ->
      ContractQuiescent reduced (reverse warnings) (reverse payments)
state
        contract;
    AmbiguousTimeIntervalReductionError -> RRAmbiguousTimeIntervalError;
  });
```

2.2.6 Reduce Contract Step

The *reduceContractStep* function handles the progression of the *Contract* in the absence of inputs: it performs the relevant action (payments, state-change, etc.), reports warnings, and throws errors if needed. It stops reducing the contract at the point when the contract requires external input.

Note that this function should report an implicit payment of zero (due to lack of funds) as a partial payment of zero, not as a non-positive payment. An explicit payment of zero (due to the contract actually specifying a zero payment) should be reported as a non-positive payment.

```
reduceContractStep ::
  Environment_ext () -> State_ext () -> Contract -> ReduceStepResult;
reduceContractStep uu state Close =
  (case refundOne (accounts state) of {
    Nothing -> NotReduced;
    Just ((party, (token, money)), newAccount) ->
      let {
        newState = accounts_update (\ _ -> newAccount) state;
      } in Reduced ReduceNoWarning
             (ReduceWithPayment (Payment party (Party party) token money))
             newState Close;
  });
reduceContractStep env state (Pay accId payee token val cont) =
  let {
    moneyToPay = evalValue env state val;
  } in (if less_eq_int moneyToPay Zero_int
         then let {
                warning = ReduceNonPositivePay accId payee token moneyToPay;
              } in Reduced warning ReduceNoPayment state cont
         else let {
                balance = moneyInAccount accId token (accounts state);
                paidMoney = min balance moneyToPay;
                newBalance = minus_int balance paidMoney;
                newAccs =
                  updateMoneyInAccount accId token newBalance (accounts
state);
                warning =
                  (if less_int paidMoney moneyToPay
                    then ReducePartialPay accId payee token paidMoney
moneyToPay
                    else ReduceNoWarning);
              } in (case giveMoney accId payee token paidMoney newAccs
of {
                     (payment, finalAccs) ->
                       Reduced warning payment
                         (accounts_update (\ _ -> finalAccs) state) cont;
                   }));
reduceContractStep env state (If obs cont1 cont2) =
  let {
    a = (if evalObservation env state obs then cont1 else cont2);
  } in Reduced ReduceNoWarning ReduceNoPayment state a;
reduceContractStep env state (When uv timeout cont) =
  (case timeInterval env of {
    (startTime, endTime) ->
      (if less_int endTime timeout then NotReduced
        else (if less_eq_int timeout startTime
               then Reduced ReduceNoWarning ReduceNoPayment state cont
               else AmbiguousTimeIntervalReductionError));
  });
reduceContractStep env state (Let valId val cont) =
  let {
    evaluatedValue = evalValue env state val;
    boundVals = boundValues state;
    newState =
      boundValues_update (\ _ -> insert valId evaluatedValue boundVals)
state;
    warn = (case lookup valId boundVals of {
              Nothing -> ReduceNoWarning;
              Just oldVal -> ReduceShadowing valId oldVal evaluatedValue;
            });
  } in Reduced warn ReduceNoPayment newState cont;
reduceContractStep env state (Assert obs cont) =
  let {
    warning =
      (if evalObservation env state obs then ReduceNoWarning
        else ReduceAssertionFailed);
  } in Reduced warning ReduceNoPayment state cont;
```

2.2.7 Apply Input

The *applyInput* function attempts to apply the next input to each *Case* in the *When*, in sequence.

```
applyInput ::
  Environment_ext () -> State_ext () -> Input -> Contract -> ApplyResult;
applyInput env state input (When cases t cont) =
  applyCases env state input cases;
applyInput env state input Close = ApplyNoMatchError;
applyInput env state input (Pay v va vb vc vd) = ApplyNoMatchError;
applyInput env state input (If v va vb) = ApplyNoMatchError;
applyInput env state input (Let v va vb) = ApplyNoMatchError;
applyInput env state input (Assert v va) = ApplyNoMatchError;
```

2.2.8 Apply Cases

The *applyCases* function attempts to match an *Input* to an *Action*, compute the new contract state, emit warnings, throw errors if needed, and determine the appropriate continuation of the contract.

```
applyCases ::
  Environment_ext () -> State_ext () -> Input -> [Case] -> ApplyResult;
applyCases env state (IDeposit accId1 party1 tok1 amount)
  (Case (Deposit accId2 party2 tok2 val) cont : rest) =
  (if equal_Party accId1 accId2 &&
        equal_Party party1 party2 &&
          equal_Token tok1 tok2 && equal_int amount (evalValue env state
val)
    then let {
           warning =
             (if less_int Zero_int amount then ApplyNoWarning
               else ApplyNonPositiveDeposit party1 accId2 tok2 amount);
           newState =
             accounts_update
               (\ _ -> addMoneyToAccount accId1 tok1 amount (accounts
state))
               state;
         } in Applied warning newState cont
    else applyCases env state (IDeposit accId1 party1 tok1 amount) rest);
applyCases env state (IChoice choId1 choice)
  (Case (Choice choId2 bounds) cont : rest) =
  (if equal_ChoiceId choId1 choId2 && inBounds choice bounds
    then let {
           newState =
             choices_update (\ _ -> insert choId1 choice (choices state))
state;
         } in Applied ApplyNoWarning newState cont
    else applyCases env state (IChoice choId1 choice) rest);
applyCases env state INotify (Case (Notify obs) cont : rest) =
  (if evalObservation env state obs then Applied ApplyNoWarning state
cont
    else applyCases env state INotify rest);
applyCases env state (IDeposit accId1 party1 tok1 amount)
  (Case (Choice vb vc) va : rest) =
  applyCases env state (IDeposit accId1 party1 tok1 amount) rest;
applyCases env state (IDeposit accId1 party1 tok1 amount)
  (Case (Notify vb) va : rest) =
  applyCases env state (IDeposit accId1 party1 tok1 amount) rest;
applyCases env state (IChoice choId1 choice)
  (Case (Deposit vb vc vd ve) va : rest) =
  applyCases env state (IChoice choId1 choice) rest;
applyCases env state (IChoice choId1 choice) (Case (Notify vb) va : rest)
=
  applyCases env state (IChoice choId1 choice) rest;
applyCases env state INotify (Case (Deposit vb vc vd ve) va : rest) =
  applyCases env state INotify rest;
applyCases env state INotify (Case (Choice vb vc) va : rest) =
  applyCases env state INotify rest;
applyCases env state acc [] = ApplyNoMatchError;
```

2.2.9 Utilitiesn

The *moneyInAccount*, *updateMoneyInAccount*, and *addMoneyToAccount* functions read, write, and increment the funds in a particular account of the *State*, respectively. The *giveMoney* function transfer funds internally between accounts. The *refundOne* function finds the first account with funds in it.

```
moneyInAccount :: Party -> Token -> [((Party, Token), Int)] -> Int;
moneyInAccount accId token accountsV =
  findWithDefault Zero_int (accId, token) accountsV;


updateMoneyInAccount ::
  Party -> Token -> Int -> [((Party, Token), Int)] -> [((Party, Token),
Int)];
updateMoneyInAccount accId token money accountsV =
  (if less_eq_int money Zero_int then delete (accId, token) accountsV
    else insert (accId, token) money accountsV);


addMoneyToAccount ::
  Party -> Token -> Int -> [((Party, Token), Int)] -> [((Party, Token),
Int)];
addMoneyToAccount accId token money accountsV =
  let {
    balance = moneyInAccount accId token accountsV;
    newBalance = plus_int balance money;
  } in (if less_eq_int money Zero_int then accountsV
        else updateMoneyInAccount accId token newBalance accountsV);
```

```
giveMoney ::
  Party ->
    Payee ->
      Token ->
        Int ->
          [((Party, Token), Int)] -> (ReduceEffect, [((Party, Token),
Int)]);
giveMoney accountId payee token money accountsV =
  let {
    a = (case payee of {
          Account accId -> addMoneyToAccount accId token money accountsV;
          Party _ -> accountsV;
        });
  } in (ReduceWithPayment (Payment accountId payee token money), a);
```

```
refundOne ::
  [((Party, Token), Int)] ->
    Maybe ((Party, (Token, Int)), [((Party, Token), Int)]);
refundOne (((accId, tok), money) : rest) =
  (if less_int Zero_int money then Just ((accId, (tok, money)), rest)
    else refundOne rest);
refundOne [] = Nothing;
```

2.2.10 Evaluate Value

Given the *Environment* and the current *State*, the *evalValue* function evaluates a *Value* into a number

$evalValue :: Environment \Rightarrow State \Rightarrow Value \Rightarrow int$

Available Money

For the *AvailableMoney* case, *evalValue* will give us the amount of *Token*s that a *Party* has in their internal account.

$evalValue\ env\ state\ (AvailableMoney\ accId\ token) = findWithDefault\ 0\ (accId, token)\ (accounts\ state)$

Constant

For the *Constant* case, *evalValue* will always evaluate to the same value

$evalValue\ env\ state\ (Constant\ integer) = integer$

Addition

For the *AddValue* case, *evalValue* will evaluate both sides and add them together.

$evalValue\ env\ state\ (AddValue\ lhs\ rhs) = evalValue\ env\ state\ lhs + evalValue\ env\ state\ rhs$

Addition is associative and commutative:

$evalValue\ env\ sta\ (AddValue\ x\ (AddValue\ y\ z)) = evalValue\ env\ sta\ (AddValue\ (AddValue\ x\ y)\ z)$

$evalValue\ env\ sta\ (AddValue\ x\ y) = evalValue\ env\ sta\ (AddValue\ y\ x)$

Subtraction

For the *SubValue* case, *evalValue* will evaluate both sides and subtract the second value from the first.

$evalValue\ env\ state\ (SubValue\ lhs\ rhs) = evalValue\ env\ state\ lhs - evalValue\ env\ state\ rhs$

Negation

For every value x there is the complement *NegValue* x so that

$evalValue\ env\ sta\ (AddValue\ x\ (NegValue\ x)) = 0$

Multiplication

For the *MulValue* case, *evalValue* will evaluate both sides and multiply them.

$evalValue\ env\ state\ (MulValue\ lhs\ rhs) = evalValue\ env\ state\ lhs * evalValue\ env\ state\ rhs$

Division

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

- If the denominator is 0, the result is also 0. Other languages uses NaN or Infinity to represent this case
- The result will be rounded towards zero.

```
evalValue env state (DivValue lhs rhs) =
(let n = evalValue env state lhs;
     d = evalValue env state rhs
 in if d = 0 then 0 else n quot d)
```

TODO: lemmas around division? maybe extend the following to proof evalValue and not just div

$c \neq 0 \Longrightarrow c * a \ div \ (c * b) = a \ div \ b$

$c \neq 0 \Longrightarrow |c * a| \ div \ |c * b| = |a| \ div \ |b|$

COMMENT(BWB): I suggest that the lemmas be (i) exact multiples divide with no remainder, (ii) the remainder equals the excess above an exact multiple, and (iii) negation commutues with division.

Choice Value

For the *ChoiceValue* case, *evalValue* will look in its state if a *Party* has made a choice for the *ChoiceName*. It will default to zero if it doesn't find it.

```
evalValue env state (ChoiceValue choId) = findWithDefault 0 choId (choices state)
```

Time Interval Start

All transactions are executed in the context of a valid time interval. For the *TimeIntervalStart* case, *evalValue* will return the beginning of that interval.

```
evalValue env state TimeIntervalStart = fst (timeInterval env)
```

Time Interval End

All transactions are executed in the context of a valid time interval. For the *TimeIntervalEnd* case, *evalValue* will return the end of that interval.

evalValue env state TimeIntervalEnd = *snd* (*timeInterval env*)

Use Value

For the *TimeIntervalEnd* case, *evalValue* will look in its state for a bound *ValueId*. It will default to zero if it doesn't find it.

evalValue env state (*UseValue valId*) = *findWithDefault 0 valId* (*boundValues state*)

Conditional Value

For the *Cond* case, *evalValue* will first call *evalObservation* on the condition, and it will evaluate the the true or false value depending on the result.

evalValue env state (*Cond cond thn els*) = (if *evalObservation env state cond* then *evalValue env state thn* else *evalValue env state els*)

2.2.11 Evaluate Observation

Given the *Environment* and the current *State*, the *evalObservation* function evaluates an *Observation* into a number

evalObservation :: *Environment* ⇒ *State* ⇒ *Observation* ⇒ *bool*

True and False

The logical constants *true* and *false* are trivially evaluated.

evalObservation env state TrueObs = *True*

evalObservation env state FalseObs = *False*

Not, And, Or

The standard logical operators ¬, ∧, and ∨ are evaluated in a straightforward manner.

evalObservation env state (*NotObs subObs*) = (¬ *evalObservation env state subObs*)

evalObservation env state (*AndObs lhs rhs*) = (*evalObservation env state lhs* ∧ *evalObservation env state rhs*)

evalObservation env state (*OrObs lhs rhs*) = (*evalObservation env state lhs* ∨ *evalObservation env state rhs*)

Comparison of Values

Five functions are provided for the comparison (equality and ordering of integer values) have traditional evaluations: $=$,$<$, $\leq$, $>$, and $\geq$.

evalObservation env state (*ValueEQ lhs rhs*) = (*evalValue env state lhs* = *evalValue env state rhs*)

evalObservation env state (*ValueLT lhs rhs*) = (*evalValue env state lhs* < *evalValue env state rhs*)

evalObservation env state (*ValueLE lhs rhs*) = (*evalValue env state lhs* ≤ *evalValue env state rhs*)

evalObservation env state (*ValueGT lhs rhs*) = (*evalValue env state rhs* < *evalValue env state lhs*)

evalObservation env state (*ValueGE lhs rhs*) = (*evalValue env state rhs* ≤ *evalValue env state lhs*)

Chose Something

The *ChoseSometing* i term evaluates to true if the a choice i was previously made in the history of the contract.

evalObservation env state (*ChoseSomething choId*) = *member choId* (*choices state*)

Chapter 3

Marlowe Guarantees

We can also use proof assistants to demonstrate that the Marlowe semantics presents certain desirable properties, such as that money is preserved and anything unspent is returned to users by the end of the execution of any contract.

Auxillary Functions

Many of the proofs in this chapter rely on function *playTrace* and *playTraceAux* that execute a sequence of transactions using the Marlowe semantics defined in *computeTransaction*. They also rely on starting from a valid and positive contract state, *validAndPositive-state* and a function *maxTimeContract* that extracts the latest timeout from the contract.

playTrace :: *int* ⇒ *Contract* ⇒ *Transaction list* ⇒ *TransactionOutput*

playTraceAux :: *TransactionOutputRecord* ⇒ *Transaction list* ⇒ *TransactionOutput*

validAndPositive-state :: *State* ⇒ *bool*

maxTimeContract :: *Contract* ⇒ *int*

3.1 Money Preservation

One of the dangers of using smart contracts is that a badly written one can potentially lock its funds forever. By the end of the contract, all the money paid to the contract must be distributed back, in some way, to a subset of the participants of the contract. To ensure this is the case we proved two properties: “Money Preservation” and “Contracts Always Close”.

Regarding money preservation, money is not created or destroyed by the semantics. More specifically, the money that comes in plus the money in the contract before the transaction must be equal to the money that comes out plus the contract after the transaction, except in the case of an error.

$\mathit{moneyInTransactions}\ \mathit{tra} = \mathit{moneyInPlayTraceResult}\ \mathit{tra}\ (\mathit{playTrace}\ \mathit{sl}\ \mathit{contract}\ \mathit{tra})$

where *moneyInTransactions* and *moneyInPlayTraceResult* measure the funds in the transactions applied to a contract versus the funds in the contract state and the payments that it has made while executing.

3.2 Contracts Always Close

For every Marlowe Contract there is a time after which an empty transaction can be issued that will close the contract and refund all the money in its accounts.
FIXME: This theorem doesn't actually prove the narrative. Are we missing a theorem?

$[\![\mathit{validAndPositive\text{-}state}\ \mathit{sta};\ \mathit{accounts}\ \mathit{sta} \neq [] \lor \mathit{cont} \neq \mathit{Close}]\!] \Longrightarrow \exists\, \mathit{inp}.\ \mathit{isClosedAndEmpty}\ (\mathit{computeTransaction}\ \mathit{inp}\ \mathit{sta}\ \mathit{cont})$

3.3 Positive Accounts

There are some values for State that are allowed by its type but make no sense, especially in the case of Isabelle semantics where we use lists instead of maps:

1. The lists represent maps, so they should have no repeated keys.

2. We want two maps that are equal to be represented the same, so we force keys to be in ascending order.

3. We only want to record those accounts that contain a positive amount.

We call a value for State valid if the first two properties are true. And we say it has positive accounts if the third property is true.

FIXME: Address the review comment "Is this a note for us or the explanation to the user of what *playTraceAux-preserves-validAndPositive-state* proves?".

$\llbracket \mathit{validAndPositive\text{-}state}\ (\mathit{txOutState}\ \mathit{txIn});\ \mathit{playTraceAux}\ \mathit{txIn}\ \mathit{transList} = \mathit{TransactionOutput}\ \mathit{txOut} \rrbracket \Longrightarrow \mathit{validAndPositive\text{-}state}\ (\mathit{txOutState}\ \mathit{txOut})$

3.4 Quiescent Result

A contract is quiescent if and only if the root construct is *When*, or if the contract is *Close* and all accounts are empty. If an input *State* is valid and accounts are positive, then the output will be quiescent, *isQuiescent*.

The following always produce quiescent contracts:

- reductionLoop §2.2.5
- reduceContractUntilQuiescent §2.2.4
- applyAllInputs §2.2.3
- computeTransaction §2.2.1
- playTrace §3

$\mathit{playTrace}\ \mathit{sl}\ \mathit{cont}\ (h : t) = \mathit{TransactionOutput}\ \mathit{traOut} \Longrightarrow \mathit{isQuiescent}\ (\mathit{txOutContract}\ \mathit{traOut})\ (\mathit{txOutState}\ \mathit{traOut})$

3.5 Reducing a Contract until Quiescence Is Idempotent

Once a contract is quiescent, further reduction will not change the contract or state, and it will not produce any payments or warnings.

$\mathit{reduceContractUntilQuiescent}\ \mathit{env}\ \mathit{state}\ \mathit{contract} = \mathit{ContractQuiescent}\ \mathit{reducedAfter}\ \mathit{wa}\ \mathit{pa}\ \mathit{nsta}\ \mathit{ncont} \Longrightarrow \mathit{reduceContractUntilQuiescent}\ \mathit{env}\ \mathit{nsta}\ \mathit{ncont} = \mathit{ContractQuiescent}\ \mathit{False}\ []\ []\ \mathit{nsta}\ \mathit{ncont}$

3.6 Split Transactions Into Single Input Does Not Affect the Result

Applying a list of inputs to a contract produces the same result as applying each input singly.

$\mathit{playTraceAux}\ \mathit{acc}\ \mathit{tral} = \mathit{playTraceAux}\ \mathit{acc}\ (\mathit{traceListToSingleInput}\ \mathit{tral})$

3.6.1 Termination Proof

Isabelle automatically proves termination for most function. However, this is not the case for *reductionLoop*, but it is manually proved that the reduction loop monotonically reduces the size of the contract (except for *Close*, which reduces the number of accounts), this is sufficient to prove termination.

$\mathit{reduceContractStep}\ \mathit{env}\ \mathit{sta}\ c = \mathit{Reduced}\ \mathit{twa}\ \mathit{tef}\ \mathit{nsta}\ \mathit{nc} \Longrightarrow \mathit{evalBound}\ \mathit{nsta}\ \mathit{nc} < \mathit{evalBound}\ \mathit{sta}\ c$

3.6.2 All Contracts Have a Maximum Time

If one sends an empty transaction with time equal to *maxTimeContract*, then the contract will close.

$$\frac{\begin{array}{c}\mathit{validAndPositive\text{-}state}\ \mathit{sta} \\ \mathit{minTime}\ \mathit{sta} \le \mathit{iniTime} \qquad \mathit{maxTimeContract}\ \mathit{cont} \le \mathit{iniTime} \\ \mathit{iniTime} \le \mathit{endTime} \qquad \mathit{accounts}\ \mathit{sta} \neq [] \vee \mathit{cont} \neq \mathit{Close}\end{array}}{\mathit{isClosedAndEmpty}\ (\mathit{computeTransaction}\ \llparenthesis \mathit{interval} = (\mathit{iniTime},\ \mathit{endTime}),\ \mathit{inputs} = [] \rrparenthesis\ \mathit{sta}\ \mathit{cont}}$$

3.6.3 Contract Does Not Hold Funds After it Closes

Funds are not held in a contract after it closes.

$\mathit{computeTransaction}\ \mathit{tra}\ \mathit{sta}\ \mathit{Close} = \mathit{TransactionOutput}\ \mathit{trec} \Longrightarrow \mathit{txOutWarnings}\ \mathit{trec} = []$

3.6.4 Transaction Bound

There is a maximum number of transaction that can be accepted by a contract.

$\mathit{playTrace}\ \mathit{sl}\ c\ l = \mathit{TransactionOutput}\ \mathit{txOut} \Longrightarrow |l| \le \mathit{maxTransactionsInitialState}\ c$