MEMORIA DEL PROYECTO FINAL: CREACIÓN DE SMART CONTRACTS EN HASKELL

MODELOS AVANZADOS DE COMPUTACIÓN

Universidad de Huelva

Grado en Ingeniería Informática
Especialidad en Computación
Curso 2023/24

Manuel Ramírez Ballesteros

Índice:

1- Haskell en el mundo blockchain

Haskell, nombrado así en honor al matemático estadounidense *Haskell Curry*, tiene sus raíces en el mundo académico y en el estudio de la lógica, definiéndose formalmente en 1990, aunque algunos investigadores como el profesor *Phillip Wadler* ya habían comenzado a trabajar en este proyecto desde 1987.

Los lenguajes de programación funcional hacen uso de *funciones puras*, es decir, funciones que siempre dan el mismo resultado para la misma entrada. Separando los *efectos secundarios* de la lógica principal siempre que sea posible, los programadores de *Haskell* pueden hacer mucho más fácil el razonamiento de su código. Además de facilitar la escritura de un código correcto, también es invaluable para probar, o incluso comprobar su corrección.

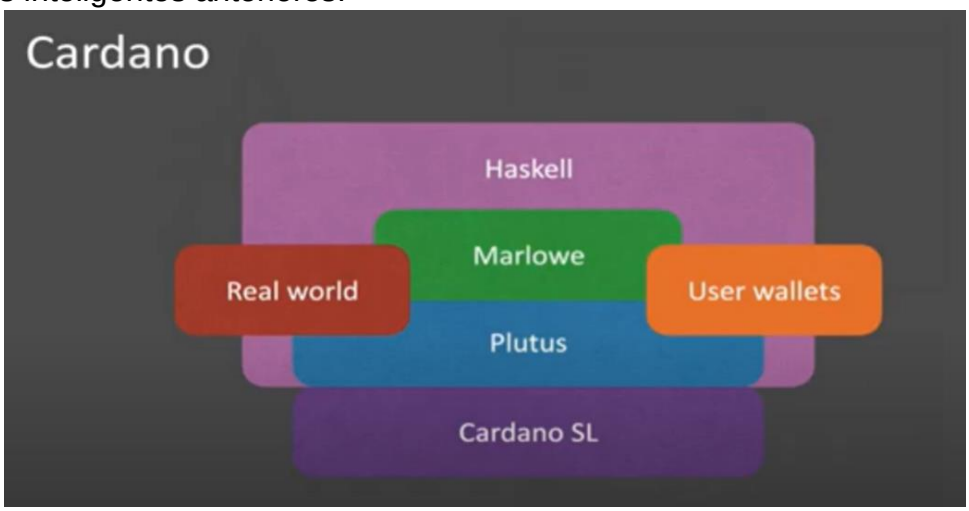
Cardano es una plataforma *Blockchain* al de código abierto diseñada tanto para transiciones financieras como para computación distribuida que cuenta con su propia moneda digital: *ADA*. La plataforma lleva el nombre de *Girolamo Cardano* y, la moneda, el de *Ada Lovelace*. Fue fundada en 2015 y lanzada en 2017 por el cofundador de Ethereum con el objetivo de crear una red que pudiera rivalizar con *Ethereum*. Utiliza como mecanismo de consenso la *prueba de participación (PoS)*, la cual se basa en validadores que son seleccionados al azar (en función de las monedas que se tenga) para crear nuevos bloques en la cadena y validar transacciones. Esta alternativa se considera una opción más ecológica y con mayor escalabilidad que la *prueba de trabajo (PoW)* de *Bitcoin*.

Cardano, al igual que *Bitcoin*, es una cadena de bloques basada en las *Salidas de Transacción no Gastadas (UTXO*, por sus siglas en inglés), que utiliza un modelo contable diferente para su libro mayor en comparación con otras cadenas de bloques basadas en cuentas, como *Ethereum*. *Cardano* implementa un innovador modelo de *Salidas de Transacción No Gastadas Extendidas* (*EUTXO*, por sus siglas en inglés), para admitir contratos inteligentes y activos múltiples.

En el modelo *UTXO*, una transacción tiene entradas y salidas, donde las entradas son salidas no gastadas de transacciones anteriores. Los activos se almacenan en el libro mayor en salidas no gastadas en lugar de en cuentas. En términos abstractos, se puede pensar en una transacción como la acción que desbloquea las salidas anteriores y crea nuevas.

Dentro del ecosistema de *Cardano*, *Marlowe* es un nuevo lenguaje específico de dominio (*DSL*) para modelar instrumentos financieros como contratos inteligentes en una cadena de bloques. Ha sido diseñado para personas que son ingenieros comerciales o expertos en el tema en lugar de desarrolladores experimentados. Es un lenguaje simple que consta de un pequeño número de bloques de construcción poderosos que se pueden ensamblar en contratos financieros expresivos. Está integrado en el lenguaje *Haskell*, que tiene su propio ecosistema y marco de prueba establecido. No necesitas experiencia en programación para usar *Marlowe* y puedes explorar tus construcciones financieras de *Marlowe* con un editor de contratos y un simulador basados en el navegador.

Plutus proporcionaba (actualmente no parece estar disponible) una plataforma de ejecución y un lenguaje de programación funcional que se ejecuta en la capa de liquidación de Cardano y ofrece considerables ventajas de seguridad. Ofrece una forma más fácil y robusta de demostrar que tus contratos inteligentes son correctos y no encontrarán los problemas que se encontraron en el diseño de lenguajes de contratos inteligentes anteriores.



2- Playground de Marlowe

El modelo *Marlowe* está diseñado para facilitar la ejecución de contratos financieros en la *Blockchain* de *Cardano*. Los contratos se construyen a partir de un conjunto limitado de bloques de construcción que describen diversos contratos financieros y transaccionales de manera más segura y directa. Estos bloques pueden describir cómo realizar un pago, cómo realizar una observación en el mundo real, esperar hasta que cierta condición se cumpla...

Los participantes (o partes) en el contrato pueden ser representados por claves públicas o roles. Los roles, representados por *tokens*, se asignan a direcciones al desplegar un contrato. Quien posea el token de un rol puede realizar las acciones asignadas a dicho rol y recibir pagos que se emitan a ese rol, permitiendo que puedan intercambiarse los roles entre participantes. Las claves públicas están representadas por el hash de la clave pública y eventualmente serán reemplazadas por direcciones. Usar claves públicas para representar a los participantes es más simple ya que no requiere manejos de *tokens*, con la desventaja de que no se podrán intercambiar.
El *Playground* de *Marlowe* permite la simulación omnisciente, donde los usuarios pueden realizar cualquier acción para cualquier rol y observar la ejecución desde múltiples perspectivas. La ejecución implica evaluar paso a paso el contrato durante la validación de la transacción. Cada transacción se agrega a la cadena de bloques y se procesa hasta que el contrato alcanza la inactividad.
Permite a los usuarios simular el comportamiento de contratos fuera de la cadena, lo que ayuda a comprender su ejecución sin implementarlos en la cadena de bloques. Se proporcionan editores específicos para cada lenguaje, como *JavaScript* y *Haskell*, que admiten funciones como autocompletado, verificación de errores y

descripciones emergentes de enlaces. Se puede compilar código escrito en estos lenguajes a *Marlowe* para su simulación.

Marlowe cuenta con seis maneras de construir contratos: Por un lado, tenemos *Close* para los contratos simples, y *Pay, Let, When, If* y *Assert* que describen 5 formas de hacer contratos complejos.

- *Close*: Tipo de contrato que permite que finalice. La única acción que realiza es proporcionar reembolsos a los propietarios de cuentas que contengan un saldo positivo.
- *Pay c p t v cont*: Contrato que se entiende como realizar un pago del token *t* con valor *v* desde la cuenta *c* a un beneficiario *p*, que será uno de los participantes, seguido del contrato *cont* que describirá como será el siguiente bloque tras realizar el pago.
- *If obs cont1 cont2*: Contrato condicional que continuará con *cont1* o *cont2*, en función del valor *booleano* de la observación *obs* al ejecutar esta construcción.
- *When cases timeout cont*: Contrato desencadenado por acciones. La lista *cases* contiene una colección de casos, cada uno de la forma *Case act con*, donde *act* es una acción y *con* el contrato de continuación. Cuando ocurre cierta acción (por ejemplo, *act*), el estado se actualiza en función de dicha acción y el contrato continuará con la continuación correspondiente *con*. Para asegurar que el contrato progrese eventualmente, tras alcanzar el tiempo de espera *timeout*, el contrato continuará con el contrato *cont*.
- *Let id val cont*: Contrato que permite nombrar y guardar un valor o expresión *val* mediante un identificador *id*, continuando con el contrato *cont*. Este mecanismo nos permite guardar valores volátiles que pueden cambiar en el tiempo.
- *Assert obs cont*: Contrato de afirmación que no tiene efectos sobre el estado del contrato, continuando inmediatamente con el contrato *cont*, pero emite una advertencia cuando la observación *obs* es falsa.

```
data Contract = Close
              | Pay Party Payee Token Value Contract
              | If Observation Contract Contract
              | When [Case] Timeout Contract
              | Let ValueId Value Contract
              | Assert Observation Contract
```

Para finalizar con los conceptos necesarios para la programación de *smart contracts*, se mencionarán los diferentes tipos más utilizados en el modelo y con los que podremos comenzar a diseñar nuestros primeros contratos. El lenguaje de *Marlowe* se modela con tipos algebraicos en *Haskell*, siendo el tipo principal ***Contract***.

Se utilizan tipos como ***Party*** para representar a los participantes. Para avanzar con un contrato de *Marlowe*, uno de los participantes debe aportar pruebas, ya sea mediante la firma válida de una transacción firmada por la clave privada de la ***PubKeyHash*** o mediante el gasto de un token de rol para un participante de tipo ***Role***.

Una cuenta de *Marlowe* puede contener múltiples cantidades de monedas y/o tokens, que pueden ser fungibles o no fungibles. Una cantidad concreta es indexada por un ***Token***, que se define por un par *SímboloDeMoneda* y *NombreDelToken*, ambos dados por un ***ByteString***. También cabe la posibilidad de crear tus propias monedas y tokens.
Value, ***Observation*** y ***Action*** son tipos que representan los componentes de los contratos. **Value** incluye representaciones de dinero disponible, constantes, operaciones aritméticas y elecciones condicionales. Observaciones como ***AndObs***, ***OrObs***, y ***ChoseSomething*** permiten comparaciones y toma de decisiones. Acciones como ***Deposit***, ***Choice*** y ***Notify*** describen depósitos, elecciones y notificaciones en contratos.
Además, el modelo extendido de *Marlowe* introduce funcionalidad de plantillas para permitir que las constantes en los contratos sean reemplazadas por parámetros. Añade ***ConstantParam*** y ***TimeParam*** como valores de parámetros para valores constantes y temporales, que también se han empleado en algunos de los ejemplos. Además, el *playground* de *Marlowe* nos permite, tras compilar un contrato, ver el estado en cada momento de este en función de las acciones que se vayan tomando. Esto facilita mucho la corrección de errores en la lógica.

```
When
    [Case
        (Deposit
            (Role "Prestamista")
            (Role "Prestamista")
            (Token "" "")
            (ConstantParam "Cantidad")
        )
        (Pay
            (Role "Prestamista")
            (Party (Role "Deudor"))
            (Token "" "")
            (ConstantParam "Cantidad")
            (When
                [Case
                    (Choice
                        (ChoiceId
                            "Continuar"
                            (Role "Prestamista")
                        )
                        [Bound 0 1]
                    )
                    (If
                        (ValueEQ
                            (ChoiceValue
                                (ChoiceId
                                    "Continuar"
                                    (Role "Prestamista")
                                ))
                            (Constant 1)
                        )
                        (When
                            [Case
                                (Deposit
                                    (Role "Deudor")
                                    (Role "Deudor")
                                    (Token "" "")
                                    (AddValue
                                        (ConstantParam "Interes")
                                        (MulValue
                                            (ConstantParam "Cantidad")
                                            (ConstantParam "Aumento de Precio")
```

3- Creación de Smart Contracts

Tras comprender mejor nuestro entorno y las características esenciales que nos permiten desarrollar *Smart Contracts,* vamos a empezar creando un *Smart Contract* desde cero que simplemente nos permita hacer un depósito de cierta cantidad de

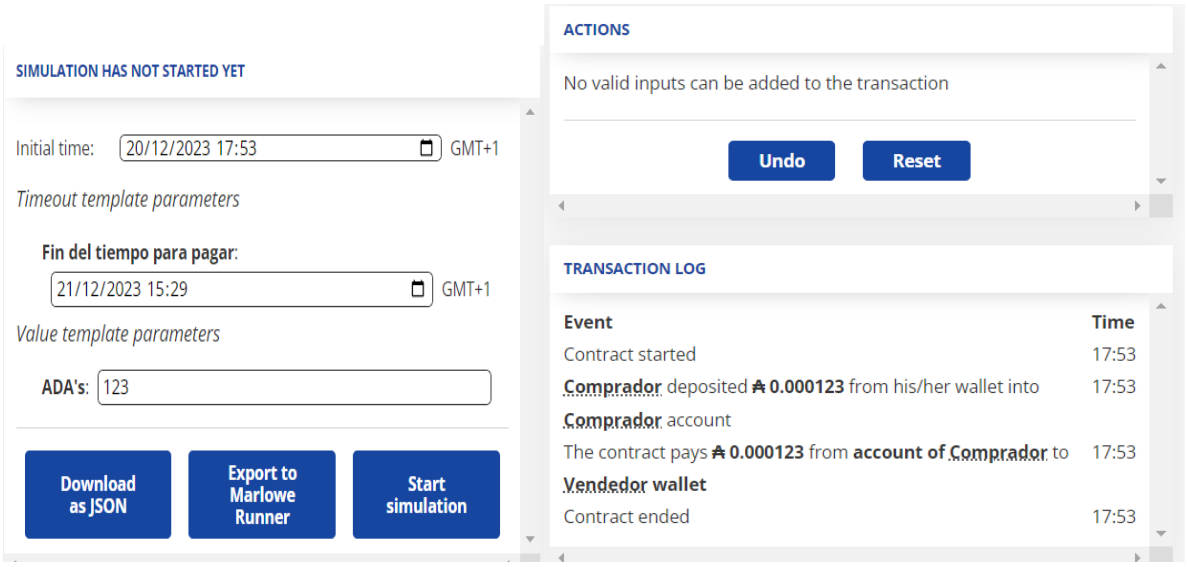
ADA's que escojamos entre dos participantes. Siempre que se comience un nuevo proyecto con el editor de *Haskell*, nos encontraremos con la siguiente plantilla:

```
{-# LANGUAGE OverloadedStrings #-}
module Example where

import Language.Marlowe.Extended.V1

main :: IO ()
main = printJSON example


{- Define a contract, Close is the simplest contract which just ends the contract straight away
-}

example :: Contract
example = Close
```

Para construir nuestro primer *Smart Contract,* sólo será necesario definir un valor de tipo *Value*, un par de participantes y un tiempo máximo para cerrar el contrato si no se ha realizado el pago:

```
{-# LANGUAGE OverloadedStrings #-}
module Prueba where

import Language.Marlowe.Extended.V1

main :: IO ()
main = printJSON example

valor :: Value
valor = ConstantParam "ADA's"

participante1, participante2 :: Party
participante1 = Role "Comprador"
participante2 = Role "Vendedor"

tiempoFin :: Timeout
tiempoFin = TimeParam "Fin del tiempo para pagar"

pago :: Timeout -> Party -> Party -> Value -> Contract -> Contract
pago tf p1 p2 v cont =
    When [Case (Deposit p1 p1 ada v)
        (Pay p1 (Party p2) ada v cont)
        ] tf
        Close

example :: Contract
example = pago tiempoFin participante1 participante2 valor Close
```

La función *main* será la encargada de invocar a la función *example*, que es el contrato en sí. En esta función se invocará a la función *pago* con todos los parámetros que requiere y que previamente se han declarado, y consiste en comprobar si se ha realizado un depósito antes del tiempo límite por parte del *participante1*. En caso de realizarse, se generará el contrato de pago *Pay* del *participante1* al *participante2* con la cantidad que ha depositado el primero. En caso de que el tiempo establecido se agote, el contrato se cerrará directamente. Por ejemplo:



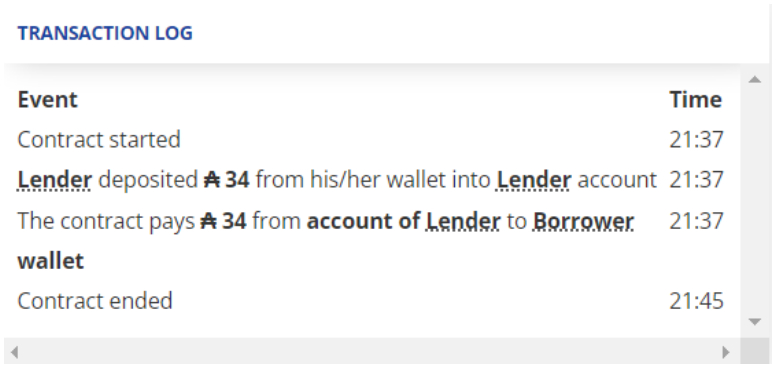
A continuación, pasaremos a estudiar algunos de los ejemplos que nos ofrece el playground y realizar algunas modificaciones sobre ellos.

Empezaremos con uno de los ejemplos más fáciles de entender que hay, que se trata de un contrato *Zero Coupon Bond*: Un tipo de préstamo simple donde el inversor paga al emisor un precio inicial en *ADA* y recibe al final el precio completo o nominal. El contrato de ejemplo que aparece al abrirlo por primera vez es:

```
{-# LANGUAGE OverloadedStrings #-}
module ZeroCouponBond where

import Language.Marlowe.Extended.V1

main :: IO ()
main = printJSON zcb

discountedPrice, notionalPrice :: Value
discountedPrice = ConstantParam "Amount"
notionalPrice = AddValue (ConstantParam "Interest") discountedPrice

investor, issuer :: Party
investor = Role "Lender"
issuer = Role "Borrower"

initialExchange, maturityExchangeTimeout :: Timeout
initialExchange = TimeParam "Loan deadline"
maturityExchangeTimeout = TimeParam "Payback deadline"

transfer :: Timeout -> Party -> Party -> Value -> Contract -> Contract
transfer timeout from to amount continuation =
    When [ Case (Deposit from from ada amount)
                (Pay from (Party to) ada amount continuation) ]
         timeout
         Close

zcb :: Contract
zcb = transfer initialExchange investor issuer discountedPrice
          $ transfer maturityExchangeTimeout issuer investor notionalPrice
            Close

```

En este contrato tenemos un par de valores: El precio descontado y el nominal, que se obtiene a partir del descontado más el interés. Tenemos los participantes ya mencionados: Inversor y emisor. Además, contamos con dos parámetros que nos indicarán las fechas límites tanto de préstamo como de devolución. Con todo esto, el contrato, mediante la función *zcb*, se encargará de hacer una transferencia desde el inversor hasta el emisor del precio descontado con el límite de tiempo indicado para el préstamo. Para que se realice la transferencia, primero el inversor debe

realizar un depósito desde su wallet hasta su cuenta para que automáticamente el contrato realice un pago hasta la wallet del emisor. Si no se realiza la transferencia en el tiempo máximo para el préstamo, el contrato finalizará. Si se realiza, se procederá a esperar otra transferencia del emisor al inversor con la cantidad nominal estipulada de la misma manera que la transferencia anterior. Si no se hace en el plazo indicado, podemos observar que el dinero no regresa al inversor:



En caso contrario, sí se realiza correctamente la devolución del préstamo. Debido al tipo de contrato, no se ha podido realizar ninguna modificación, pese a intentarlo de varias formas, para recuperar la cantidad invertida, por lo que las modificaciones que se han planteado son las siguientes:

- Se han modificado los Value para multiplicar el precio descontado por el parámetro que se indique antes de sumarle el interés. También se podrían haber aplicado funciones como SubValue o DivValue:

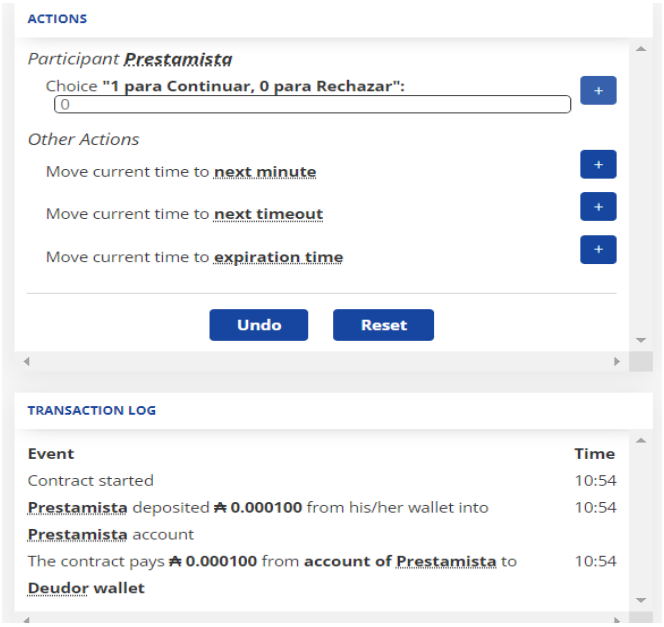
```
precioDescontado, precioNominal, aumentoDePrecio :: Value
precioDescontado = ConstantParam "Cantidad"
aumentoDePrecio = ConstantParam "Aumento de Precio"
precioNominal = AddValue (ConstantParam "Interes") $ MulValue precioDescontado aumentoDePrecio  -- Aumento aplicado
```

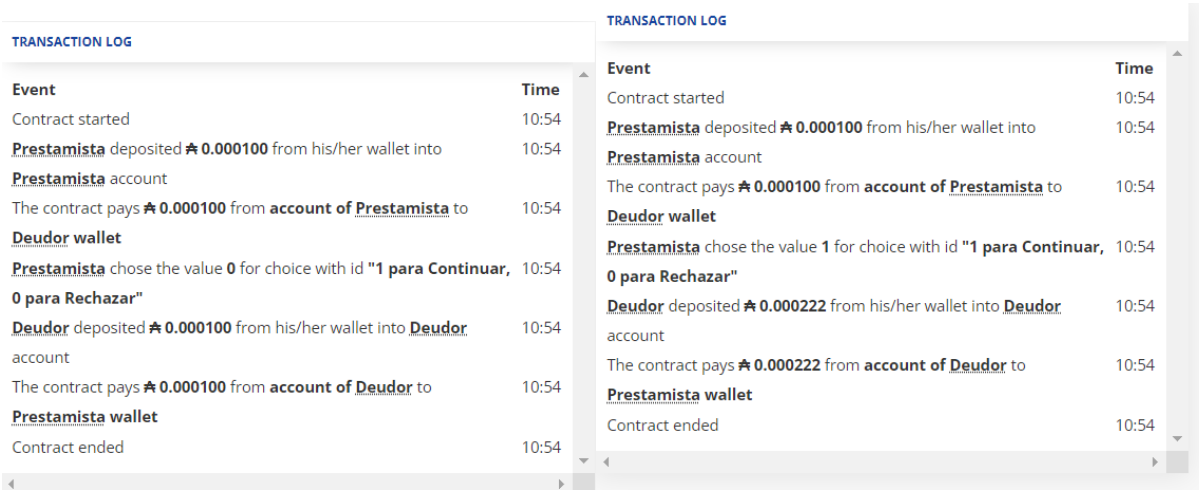
- Se ha añadido una decisión para el inversor para continuar o no con el contrato. Esto se ha implementado de dos maneras distintas donde la lógica no varía, sólo la forma de tomar dicha decisión:

```
transferir :: Timeout -> Party -> Party -> Value -> Contract -> Contract
transferir tiempoFin de a cantidad continuacion =
    When [ Case (Deposit de de ada cantidad)
                (Pay de (Party a) ada cantidad continuacion) ]
         tiempoFin
         Close

zcb :: Contract
zcb = transferir intercambioInicial inversor deudor precioDescontado
         (When [ Case (Choice (ChoiceId "1 para Continuar, 0 para Rechazar" inversor) [Bound 0 1])
                      (If (ValueEQ (ChoiceValue (ChoiceId "1 para Continuar, 0 para Rechazar" inversor)) (Constant 1))
                          (transferir tiempoFinIntercambio deudor inversor precioNominal Close)
                          (transferir tiempoFinIntercambio deudor inversor precioDescontado Close)) ]
                 tiempoFinIntercambio Close)
```

```
elecciones :: Timeout -> Party -> [(ChoiceId, [Bound], Contract)] -> Contract
elecciones tiempoFin participante elec =
    When (map (\(cId, b, cont) -> Case (Choice cId b) cont) elec) tiempoFin Close

zcb :: Contract
zcb = transferir intercambioInicial inversor deudor precioDescontado
        $ elecciones tiempoFinIntercambio inversor
          [ (ChoiceId "Continuar" inversor, [Bound 0 1],
             transferir tiempoFinIntercambio deudor inversor precioNominal Close)
          , (ChoiceId "Rechazar" inversor, [Bound 0 1],
             transferir tiempoFinIntercambio deudor inversor precioDescontado  Close)
          ]
```

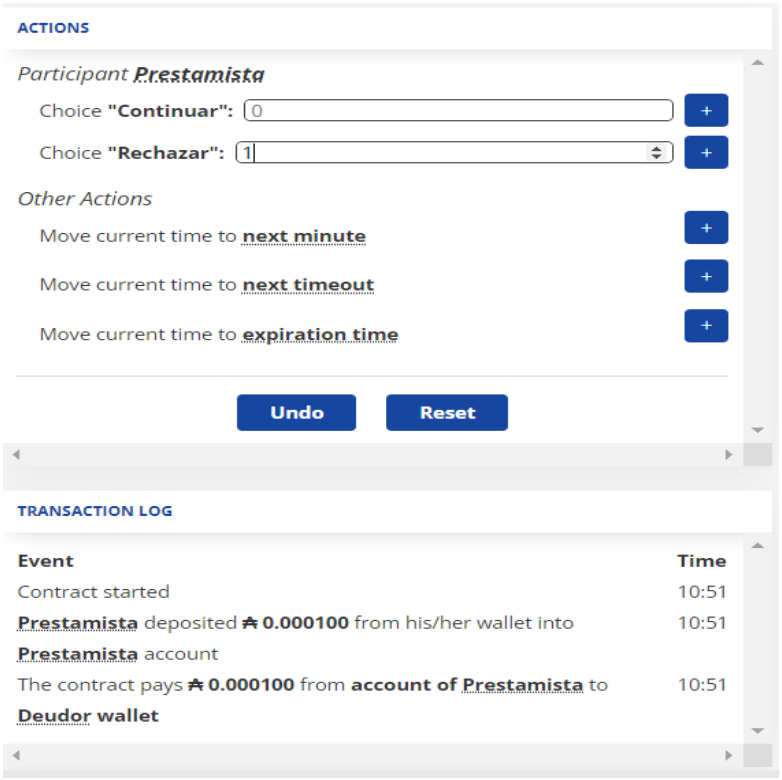
En la primera imagen se ha modificado la función *zcb* para que, tras realizar la primera transferencia, el inversor pueda decidir si desea continuar con la operación o no, indicando un 1 o un 0 respectivamente. Si escoge un 1, el otro participante deberá transferir la cantidad nominal acordada inicialmente mientras que, si escoge un 0, se tratará de cancelar la operación y sólo tendrá que devolver la cantidad inicial con el descuento.

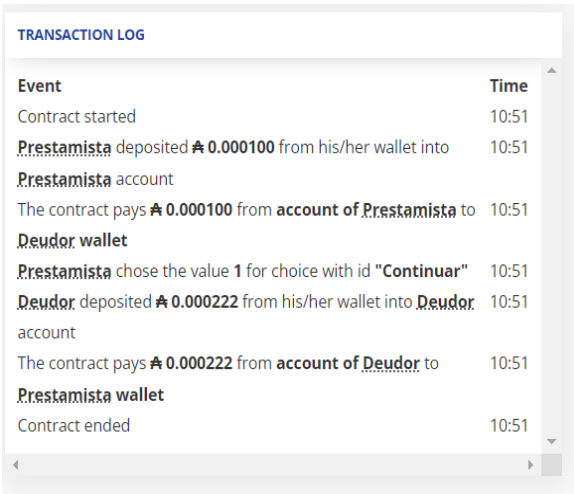


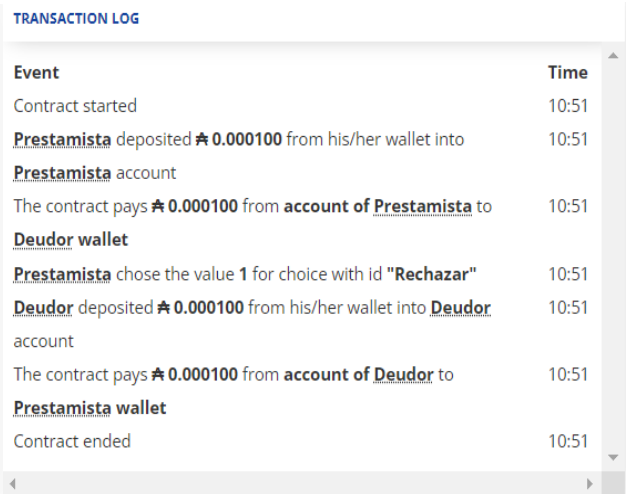


En la segunda imagen también se ha modificado la función *zcb* de manera similar a la anterior, solo que ahora se mostrarán las dos opciones que puede escoger, y no

una única opción con múltiples respuestas. Además, se ha añadido la función elecciones que recibe como parámetros el tiempo máximo para el intercambio, el participante implicado en la elección y una lista con el *String* asociado a la decisión, una lista de valores válidos para la elección (realmente no hay diferencias entre escoger un valor u otro en un mismo caso, ha sido por probar funciones como *map* en *Smart Contracts* y practicar con los tipos *Choice*) y el contrato que siga, y devuelve un contrato. La idea de esta función ha sido mapear todas las listas de decisiones que tengamos en un *Case,* que se utilizará para continuar o rechazar el contrato tal y como hacíamos en la primera imagen.







También se ha trabajado con el contrato de ejemplo *swap*, que no es más que un intercambio de manera atómica de cierta cantidad de *ADA* de un participante por

cierta cantidad de tokens de dólares del otro participante. Al abrir por primera vez el contrato de ejemplo:

```
{-# LANGUAGE OverloadedStrings #-}
module Swap where

import Language.Marlowe.Extended.V1

main :: IO ()
main = printJSON swap

-- We can set explicitRefunds True to run Close refund analysis
-- but we get a shorter contract if we set it to False
explicitRefunds :: Bool
explicitRefunds = False

lovelacePerAda, amountOfAda, amountOfLovelace, amountOfDollars :: Value
lovelacePerAda = Constant 1000000
amountOfAda = ConstantParam "Amount of Ada"
amountOfLovelace = MulValue lovelacePerAda amountOfAda
amountOfDollars = ConstantParam "Amount of dollars"

adaDepositTimeout, dollarDepositTimeout :: Timeout
adaDepositTimeout = TimeParam "Timeout for Ada deposit"
dollarDepositTimeout = TimeParam "Timeout for dollar deposit"

dollars :: Token
dollars = Token "85bb65" "dollar"

data SwapParty = SwapParty { party    :: Party
                           , currency :: Token
                           , amount   :: Value
                           }

adaProvider, dollarProvider :: SwapParty
adaProvider = SwapParty { party = Role "Ada provider"
                        , currency = ada
                        , amount = amountOfLovelace
                        }
dollarProvider = SwapParty { party = Role "Dollar provider"
                           , currency = dollars
                           , amount = amountOfDollars
                           }

makeDeposit :: SwapParty -> Timeout -> Contract -> Contract -> Contract
makeDeposit src timeout timeoutContinuation continuation =
  When [ Case (Deposit (party src) (party src) (currency src) (amount src))
               continuation
       ] timeout
         timeoutContinuation

refundSwapParty :: SwapParty -> Contract
refundSwapParty swapParty
  | explicitRefunds = Pay (party swapParty) (Party (party swapParty)) (currency swapParty) (amount swapParty) Close
  | otherwise = Close

makePayment :: SwapParty -> SwapParty -> Contract -> Contract
makePayment src dest =
  Pay (party src) (Party $ party dest) (currency src) (amount src)

swap :: Contract
swap = makeDeposit adaProvider adaDepositTimeout Close
         $ makeDeposit dollarProvider dollarDepositTimeout (refundSwapParty adaProvider)
         $ makePayment adaProvider dollarProvider
         $ makePayment dollarProvider adaProvider
           Close

```

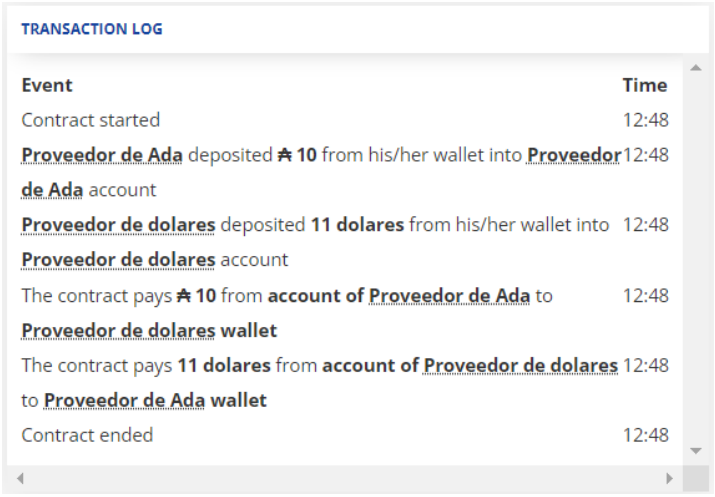
En este contrato se utilizan 4 *Value's:* La cantidad de *ADA* y de *dólares* que se pueden escoger como parámetros, un valor constante para cambiar *Lovelace* por

ADA y la cantidad de *Lovelace* que es el producto de la contante por la cantidad de *ADA*. Dos parámetros temporales para definir los tiempos máximos para depositar *ADA's* y *dólares*. El token para representar los *dólares*. También se crea un tipo de dato llamado SwapParty, que se compone de un participante, un token y una cantidad. Este nuevo tipo se utiliza para representar a los participantes del swap: El proveedor de *ADA's* y el de dólares. En cuanto a las funciones que implementan el contrato tenemos una función encargada de realizar un depósito desde la *wallet* a la cuenta del participante del swap, otra función encargada en realizar el pago entre participantes, otra para el reembolso de fondos a un participante en función de un parámetro definido al principio del programa y la función *swap* que es la encargada del intercambio. En esta, primero se realizará el depósito de *ADA*, después el de *dólares* y, cuando ambas partes hayan realizado sus depósitos, se procederá automáticamente con el intercambio. En caso de que el proveedor de *dólares* no haga su depósito en el tiempo establecido, se procederá a reembolsar al proveedor de *ADA* la cantidad que ha depositado.

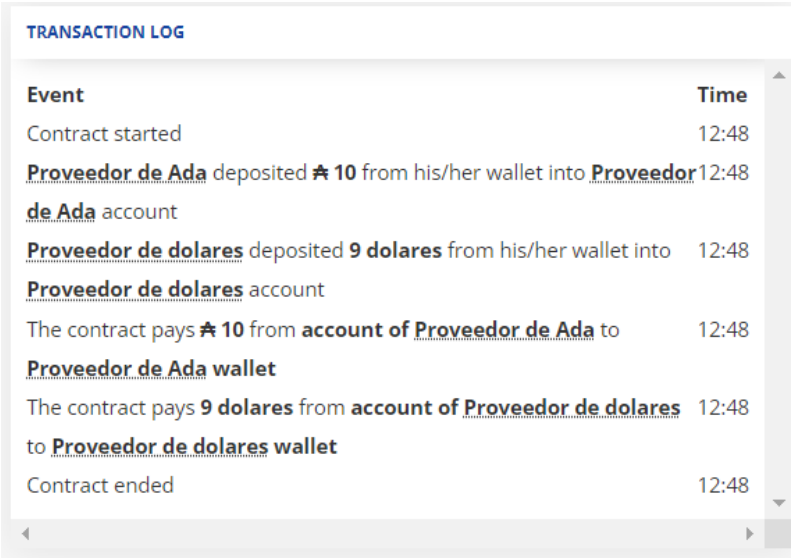
Como modificación a este contrato, se ha propuesto establecer un valor mínimo de dólares para que el intercambio pueda efectuarse. Para ello se ha distinguido entre realizar un depósito de *ADA* o de *dólares.* El depósito de ADA será igual al del ejemplo, mientras que el de *dólares,* en caso de que se realice un depósito, distinguiremos entre si la cantidad depositada es menor o mayor/igual que 10 dólares. Si es mayor continuaremos con el contrato inicial y, si es menor, se reembolsará la cantidad ingresada a ambas partes y finalizará el contrato.

```
realizarDepositoADA :: SwapParty -> Timeout -> Contract -> Contract -> Contract
realizarDepositoADA origen timeout continuation timeoutContinuation =
  When [ Case (Deposit (participante origen) (participante origen) (moneda origen) (cantidad origen))
               continuation
       ] timeout
         timeoutContinuation

realizarDepositoDollars :: SwapParty -> Timeout -> Contract -> Contract -> Contract
realizarDepositoDollars origen timeout continuation timeoutContinuation =
  When [ Case (Deposit (participante origen) (participante origen) (moneda origen) (cantidad origen))
               (If (ValueGE (cantidad origen) (Constant 10))
                   continuation
                   (reembolsarParticipante origen))
       ] timeout
         timeoutContinuation
```

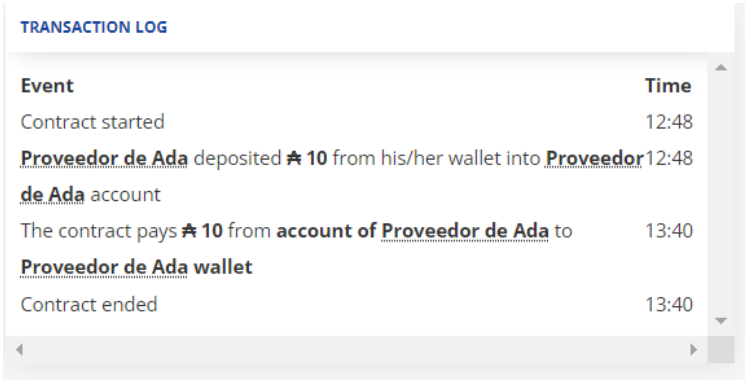
Los resultados al finalizar el contrato correctamente son:



Y cuando la cantidad de dólares es menor que 10:



Este tipo de contrato, a diferencia del anterior, si reembolsa correctamente si el tiempo límite se alcanza. Esto es debido al tipo de contrato, donde no se hacen los ingresos hasta que ambas partes han depositado y todo es correcto. En el contrato anterior de Zero Coupon Bond esto no era posible ya el inversor primero tenía que realizar un pago antes de que el otro haya podido conseguir la cantidad con intereses estipulada en el contrato.



Por último, veremos otro ejemplo interesante que es el contrato *Escrow With Collateral*. Este tipo de contrato trata de regular un intercambio entre un comprador y un vendedor utilizando un colateral para incentivar la participación de ambos. En caso de que haya un desacuerdo, el colateral se quema. El código de ejemplo que nos ofrece el entorno es el siguiente:

```
{-# LANGUAGE OverloadedStrings #-}
module EscrowWithCollateral where

import Language.Marlowe.Core.V1.Semantics.Types.Address (testnet)
import Language.Marlowe.Extended.V1
import qualified Plutus.V1.Ledger.Address as P
import qualified Plutus.V1.Ledger.Credential as P

main :: IO ()
main = printJSON escrowC

-- We can set explicitRefunds True to run Close refund analysis
-- but we get a shorter contract if we set it to False
explicitRefunds :: Bool
explicitRefunds = False

seller, buyer, burnAddress :: Party
buyer = Role "Buyer"
seller = Role "Seller"
burnAddress = Address testnet (P.Address (P.PubKeyCredential "0000000000000000000000000000000000000000000000000000000000000000") Nothing)

price, collateral :: Value
price = ConstantParam "Price"
collateral = ConstantParam "Collateral amount"

sellerCollateralTimeout, buyerCollateralTimeout, depositTimeout, disputeTimeout, answerTimeout :: Timeout
sellerCollateralTimeout = TimeParam "Collateral deposit by seller timeout"
buyerCollateralTimeout = TimeParam "Deposit of collateral by buyer timeout"
depositTimeout = TimeParam "Deposit of price by buyer timeout"
disputeTimeout = TimeParam "Dispute by buyer timeout"
answerTimeout = TimeParam "Complaint deadline"

depositCollateral :: Party -> Timeout -> Contract -> Contract -> Contract
depositCollateral party timeout timeoutContinuation continuation =
    When [Case (Deposit party party ada collateral) continuation]
         timeout
         timeoutContinuation

burnCollaterals :: Contract -> Contract
burnCollaterals =
    Pay seller (Party burnAddress) ada collateral
    . Pay buyer (Party burnAddress) ada collateral

deposit :: Timeout -> Contract -> Contract -> Contract
deposit timeout timeoutContinuation continuation =
    When [Case (Deposit seller buyer ada price) continuation]
         timeout
         timeoutContinuation

choice :: ChoiceName -> Party -> Integer -> Contract -> Case
choice choiceName chooser choiceValue = Case (Choice (ChoiceId choiceName chooser)
                                                     [Bound choiceValue choiceValue])

choices :: Timeout -> Party -> Contract -> [(Integer, ChoiceName, Contract)] -> Contract
choices timeout chooser timeoutContinuation list =
    When [choice choiceName chooser choiceValue continuation
          | (choiceValue, choiceName, continuation) <- list]
         timeout
         timeoutContinuation

sellerToBuyer :: Contract -> Contract
sellerToBuyer = Pay seller (Account buyer) ada price

refundSellerCollateral :: Contract -> Contract
refundSellerCollateral
  | explicitRefunds = Pay seller (Party seller) ada collateral
  | otherwise = id

refundBuyerCollateral :: Contract -> Contract
refundBuyerCollateral
  | explicitRefunds = Pay buyer (Party buyer) ada collateral
  | otherwise = id

refundCollaterals :: Contract -> Contract
refundCollaterals = refundSellerCollateral . refundBuyerCollateral
```

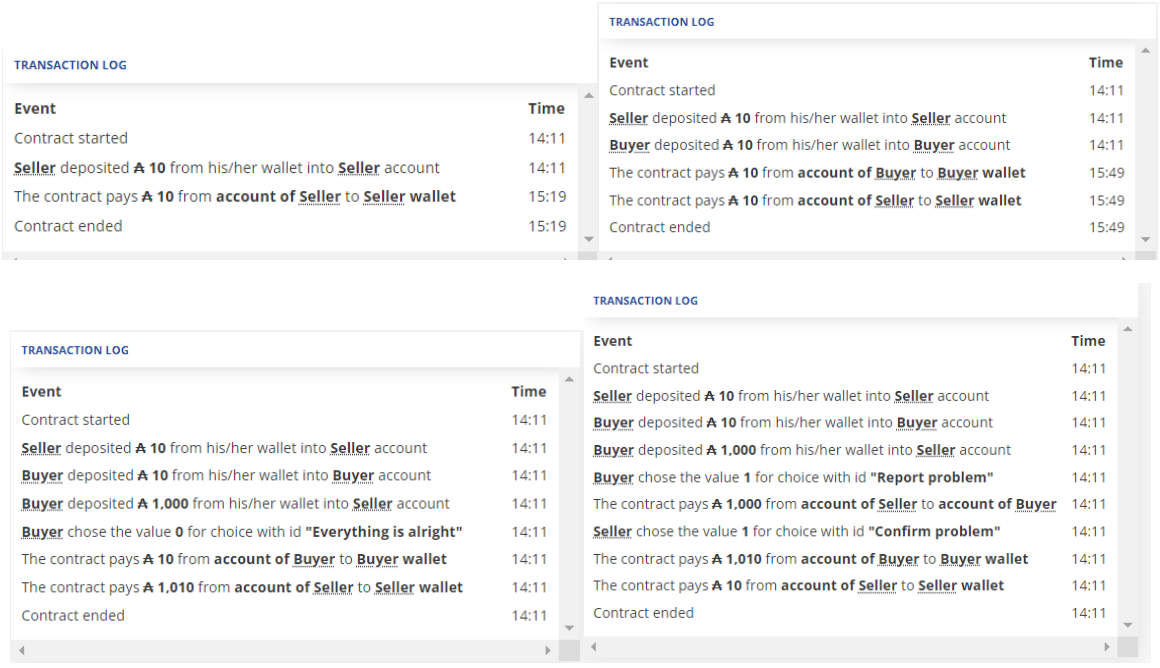
```
refundBuyer :: Contract
refundBuyer
  | explicitRefunds = Pay buyer (Party buyer) ada price Close
  | otherwise = Close

refundSeller :: Contract
refundSeller
  | explicitRefunds = Pay seller (Party seller) ada price Close
  | otherwise = Close

escrowC :: Contract
escrowC = depositCollateral seller sellerCollateralTimeout Close $
          depositCollateral buyer buyerCollateralTimeout (refundSellerCollateral Close) $
          deposit depositTimeout (refundCollaterals Close) $
          choices disputeTimeout buyer (refundCollaterals refundSeller)
            [ (0, "Everything is alright"
              , refundCollaterals refundSeller
              )
            , (1, "Report problem"
              , sellerToBuyer $
                choices answerTimeout seller (refundCollaterals refundBuyer)
                  [ (1, "Confirm problem"
                    , refundCollaterals refundBuyer
                    )
                  , (0, "Dispute problem"
                    , burnCollaterals refundBuyer
                    )
                  ]
              )
            ]
```

En este ejemplo se definen tres participantes: Comprador y vendedor, definidos mediante roles, y la dirección de quemado del colateral a partir de su clave pública. Además, se definen como valores en ADA el precio y el colateral. También existen varios parámetros para definir los tiempos máximos para que tanto comprador como vendedor ingresen el colateral, para que el comprador ingrese el precio del artículo y presente una queja y para que el vendedor la responda.

Las funciones que se definen para la lógica del contrato son:

- *depositCollateral party timeout timeoutContinuation continuation*: Encargada de que cada *party* realice el depósito desde su *wallet* hasta su cuenta antes del tiempo establecido *timeout*. Si la acción de depositar se realiza en el tiempo previsto, se ejecutará el contrato *continuation*, sino el contrato *timeoutContinuation*.
- *burnCollaterals:* Se encarga de quemar el colateral de ambas partes en caso de desacuerdo. Para ello se emplea la composición de funciones para ejecutar las dos acciones consecutivas para cada participante.
- *deposit timeout timeoutContinuation continuation:* Función encargada de que el comprador deposite el precio acordado por el artículo, similar al depósito de colateral, pero entre ambos participantes.
- *choice y choices:* Nos permiten manejar elecciones, ya sean individuales o múltiples.
- *sellerToBuyer*: Nos permite realizar el pago por el artículo.
- *refundSellerCollateral, refundBuyerCollateral y refundCollaterals*: Funciones que deciden si devolver la cantidad de colateral a cada participante o a ambos a la vez.
- *refundSeller y refundBuyer:* Funciones similares a las anteriores, pero con el precio del artículo en lugar del colateral.
- *escrowC:* Define la lógica principal del contrato y sus condiciones. Primero se esperará a que ambas partes realicen el depósito de la cantidad de colateral indicada antes de sus respectivos tiempos límites. Si el vendedor, que es quien realiza primero el depósito, se excede del tiempo, el contrato se

cancela. Si se excede el comprador, se reembolsa al vendedor la cantidad de colateral depositada y se cancela el contrato. Si ambos participantes proceden correctamente con el contrato, se le solicitará al comprador que deposite la cantidad acordada en la compra en la cuenta del vendedor y, si se excede el tiempo límite acordado, se reembolsará el colateral a ambos terminando el contrato. Si el depósito es correcto, el comprador decidirá si todo el proceso de compra es correcto o existe algún problema. Si no hay problemas, el contrato pagará a cada participante la cantidad correspondiente. Si hay problemas, se le consultará al vendedor, operando de manera similar en caso de que confirme el problema o quemando el colateral de ambas partes y devolviendo la cantidad depositada para la compra al comprador en caso de que lo rechace.



TRANSACTION LOG

Event	Time
Contract started	14:11
Seller deposited **Ⱥ 10** from his/her wallet into **Seller** account	14:11
Buyer deposited **Ⱥ 10** from his/her wallet into **Buyer** account	14:11
Buyer deposited **Ⱥ 1,000** from his/her wallet into **Seller** account	14:11
Buyer chose the value **0** for choice with id **"Everything is alright"**	14:11
The contract pays **Ⱥ 10** from **account of Buyer** to **Buyer wallet**	14:11
The contract pays **Ⱥ 1,010** from **account of Seller** to **Seller wallet**	14:11
Contract ended	14:11

TRANSACTION LOG

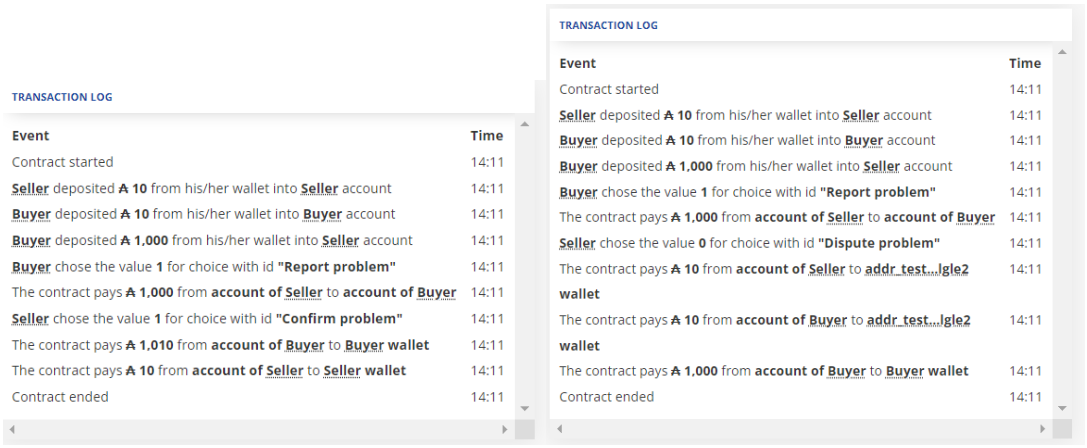
Event	Time
Contract started	14:11
Seller deposited **Ⱥ 10** from his/her wallet into **Seller** account	14:11
Buyer deposited **Ⱥ 10** from his/her wallet into **Buyer** account	14:11
Buyer deposited **Ⱥ 1,000** from his/her wallet into **Seller** account	14:11
Buyer chose the value **1** for choice with id **"Report problem"**	14:11
The contract pays **Ⱥ 1,000** from **account of Seller** to **account of Buyer**	14:11
Seller chose the value **1** for choice with id **"Confirm problem"**	14:11
The contract pays **Ⱥ 1,010** from **account of Buyer** to **Buyer wallet**	14:11
The contract pays **Ⱥ 10** from **account of Seller** to **Seller wallet**	14:11
Contract ended	14:11

TRANSACTION LOG

Event	Time
Contract started	14:11
Seller deposited **Ⱥ 10** from his/her wallet into **Seller** account	14:11
Buyer deposited **Ⱥ 10** from his/her wallet into **Buyer** account	14:11
The contract pays **Ⱥ 10** from **account of Buyer** to **Buyer wallet**	15:49
The contract pays **Ⱥ 10** from **account of Seller** to **Seller wallet**	15:49
Contract ended	15:49

TRANSACTION LOG

Event	Time
Contract started	14:11
Seller deposited **Ⱥ 10** from his/her wallet into **Seller** account	14:11
Buyer deposited **Ⱥ 10** from his/her wallet into **Buyer** account	14:11
Buyer deposited **Ⱥ 1,000** from his/her wallet into **Seller** account	14:11
Buyer chose the value **0** for choice with id **"Everything is alright"**	14:11
The contract pays **Ⱥ 10** from **account of Buyer** to **Buyer wallet**	14:11
The contract pays **Ⱥ 1,010** from **account of Seller** to **Seller wallet**	14:11
Contract ended	14:11



TRANSACTION LOG

Event	Time
Contract started	14:11
Seller deposited ₳ 10 from his/her wallet into Seller account	14:11
Buyer deposited ₳ 10 from his/her wallet into Buyer account	14:11
Buyer deposited ₳ 1,000 from his/her wallet into Seller account	14:11
Buyer chose the value 1 for choice with id "Report problem"	14:11
The contract pays ₳ 1,000 from account of Seller to account of Buyer	14:11
Seller chose the value 1 for choice with id "Confirm problem"	14:11
The contract pays ₳ 1,010 from account of Buyer to Buyer wallet	14:11
The contract pays ₳ 10 from account of Seller to Seller wallet	14:11
Contract ended	14:11

TRANSACTION LOG

Event	Time
Contract started	14:11
Seller deposited ₳ 10 from his/her wallet into Seller account	14:11
Buyer deposited ₳ 10 from his/her wallet into Buyer account	14:11
Buyer deposited ₳ 1,000 from his/her wallet into Seller account	14:11
Buyer chose the value 1 for choice with id "Report problem"	14:11
The contract pays ₳ 1,000 from account of Seller to account of Buyer	14:11
Seller chose the value 0 for choice with id "Dispute problem"	14:11
The contract pays ₳ 10 from account of Seller to addr_test...lgle2 wallet	14:11
The contract pays ₳ 10 from account of Buyer to addr_test...lgle2 wallet	14:11
The contract pays ₳ 1,000 from account of Buyer to Buyer wallet	14:11
Contract ended	14:11

Como modificación de este contrato, se ha propuesto añadir un árbitro para que, en caso de que no se pongan ambas partes de acuerdo con si existe o no un problema, este decida quién tiene razón y se le devuelva los fondos depositados del colateral únicamente a ese participante, quemando el del otro.

Para ello se ha añadido otro participante con su respectiva fecha límite:

```
    seller, buyer, burnAddress, arbitro :: Party
    arbitro = Role "Arbitro"
    buyer = Role "Buyer"
    seller = Role "Seller"
sellerCollateralTimeout, buyerCollateralTimeout, depositTimeout, disputeTimeout, answerTimeout, answerArbitroTimeout :: Timeout
answerArbitroTimeout = TimeParam "Tiempo maximo para el arbitraje"
```

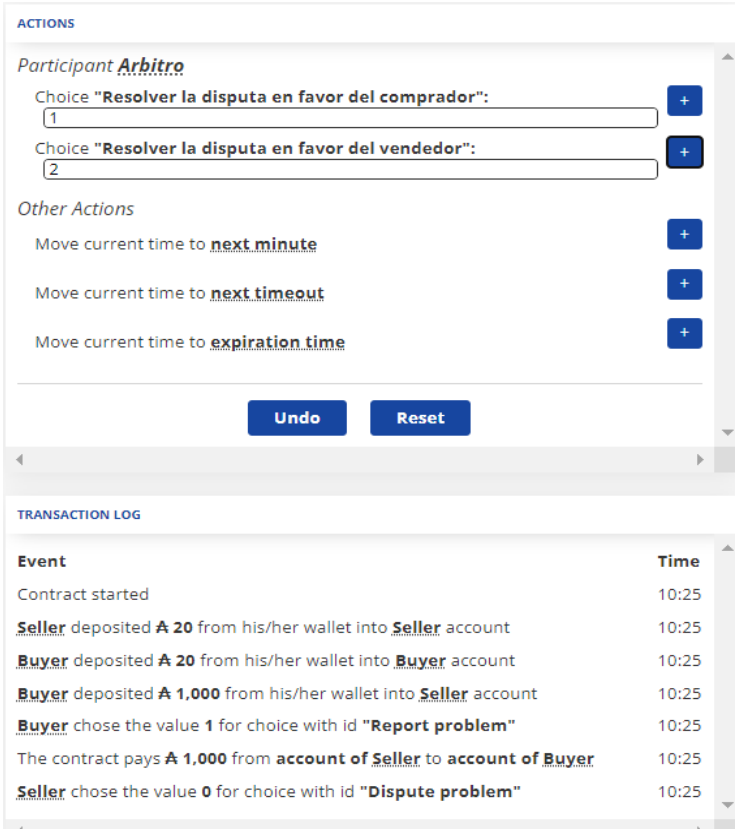
Además, se ha añadido la función *burnCollateral*, que se encargue de quemar los fondos del participante que no lleve razón:

```
41  burnCollateral :: Party -> Contract
42  burnCollateral party = Pay party (Party burnAddress) ada collateral Close
43
```

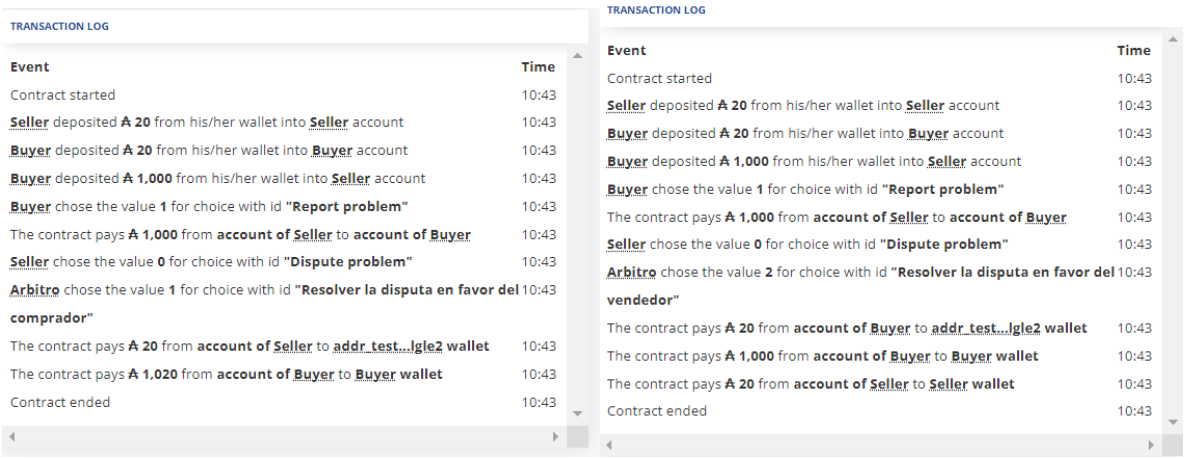
Por último, se ha modificado la función principal del contrato para que el árbitro decida después de que ambos participantes estén disputando el problema. Si el árbitro le da la razón al comprador, se quemará el colateral del vendedor mientras que el comprador recibirá todos los fondos depositados y viceversa en caso de que le dé la razón al vendedor. El resto del contrato funcionará tal y como se ha explicado en la plantilla de ejemplo.

```
escrowC :: Contract
escrowC = depositCollateral seller sellerCollateralTimeout Close $
          depositCollateral buyer buyerCollateralTimeout (refundSellerCollateral Close) $
          deposit depositTimeout (refundCollaterals Close) $
          choices disputeTimeout buyer (refundCollaterals refundSeller)
            [ (0, "Everything is alright"
              , refundCollaterals refundSeller
              )
            , (1, "Report problem"
              , sellerToBuyer $
                choices answerTimeout seller (refundCollaterals refundBuyer)
                  [ (1, "Confirm problem"
                    , refundCollaterals refundBuyer
                    )
                  , (0, "Dispute problem"
                    , choices answerArbitroTimeout arbitro (burnCollaterals refundBuyer)
                      [ (1, "Resolver la disputa en favor del comprador"
                        , burnCollateral seller
                        )
                      , (2, "Resolver la disputa en favor del vendedor"
                        , burnCollateral buyer
                        )
                      ]
                    )
                  ]
              )
            ]
  where
    arbitratorChoice = ChoiceValue (ChoiceId "Eleccion del arbitro" arbitro)

```

A continuación, veremos cómo se ejecuta esta modificación según las distintas acciones que se escojan:



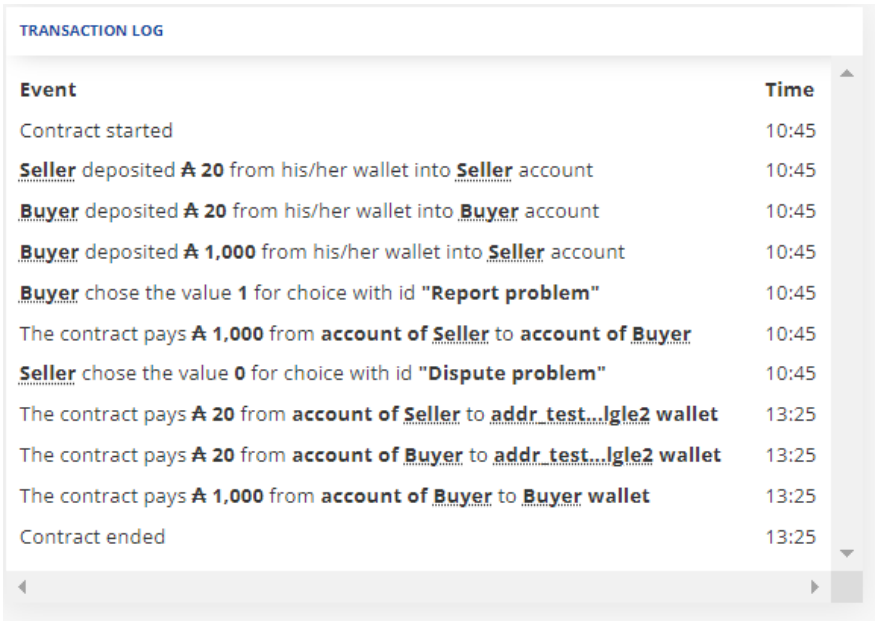
Según la decisión del árbitro, podrá ocurrir:



TRANSACTION LOG

Event	Time
Contract started	10:43
Seller deposited **₳ 20** from his/her wallet into **Seller** account	10:43
Buyer deposited **₳ 20** from his/her wallet into **Buyer** account	10:43
Buyer deposited **₳ 1,000** from his/her wallet into **Seller** account	10:43
Buyer chose the value **1** for choice with id **"Report problem"**	10:43
The contract pays **₳ 1,000** from **account of Seller** to **account of Buyer**	10:43
Seller chose the value **0** for choice with id **"Dispute problem"**	10:43
Arbitro chose the value **2** for choice with id **"Resolver la disputa en favor del vendedor"**	10:43
The contract pays **₳ 20** from **account of Buyer** to **addr_test...lgle2 wallet**	10:43
The contract pays **₳ 1,000** from **account of Buyer** to **Buyer wallet**	10:43
The contract pays **₳ 20** from **account of Seller** to **Seller wallet**	10:43
Contract ended	10:43

En caso de que no responda el árbitro, se quemarán los fondos tal y como ocurría en la plantilla de ejemplo cuando existía una disputa entre los participantes:



TRANSACTION LOG

Event	Time
Contract started	10:45
Seller deposited **₳ 20** from his/her wallet into **Seller** account	10:45
Buyer deposited **₳ 20** from his/her wallet into **Buyer** account	10:45
Buyer deposited **₳ 1,000** from his/her wallet into **Seller** account	10:45
Buyer chose the value **1** for choice with id **"Report problem"**	10:45
The contract pays **₳ 1,000** from **account of Seller** to **account of Buyer**	10:45
Seller chose the value **0** for choice with id **"Dispute problem"**	10:45
The contract pays **₳ 20** from **account of Seller** to **addr_test...lgle2 wallet**	13:25
The contract pays **₳ 20** from **account of Buyer** to **addr_test...lgle2 wallet**	13:25
The contract pays **₳ 1,000** from **account of Buyer** to **Buyer wallet**	13:25
Contract ended	13:25

4- Conclusiones

En resumen, *Marlowe* proporciona un entorno para la creación y comprobación de contratos financieros en *Cardano*, con un enfoque claro en la interacción segura y eficiente dentro de la cadena de bloques. El uso de *Haskell* para describir contratos en *Marlowe* puede conducir a códigos más expresivos, legibles y reutilizables. El enfoque modular y las abstracciones facilitan la construcción y comprensión de contratos complejos.

En mi opinión, este trabajo no es tanto generar código como pueden ser otros proyectos, debido al entorno con el que hemos trabajado donde te proporcionan muchas plantillas con ejemplos funcionales, sino el trabajo previo de familiarizarte con el ecosistema *blockchain* y con la manera de definir contratos en este entorno. Para demostrar la comprensión de todo esto se han propuesto algunas modificaciones que implementen funcionalidades extra a los contratos, pero como la

mayoría ya contaba con su idea general bien implementada, para realizar las modificaciones debes adaptarte a esa idea general, no siendo válida cualquier modificación que se nos ocurra.
Se ha tratado de explicar los ejemplos más relevantes donde se muestren la mayor parte de los contenidos implicados en la creación de contratos inteligentes, así como previamente se ha construido la base teórica necesaria para poder comprenderlos y modificarlos.

5- ***Bibliografía***

- https://play.marlowe.iohk.io/#/haskell
- https://docs.marlowe.iohk.io/tutorials/concepts/marlowe-model
- https://docs.cardano.org/learn/eutxo-explainer/
- https://clubcryptoinvest.com/historia-y-caracteristicas-de-cardano/
- https://haskell.dev/article/10_The_role_of_Haskell_in_blockchain_technology.html
- https://forum.cardano.org/t/por-que-cardano-eligio-haskell-y-por-que-deberia-importarte/43321
- https://clubcryptoinvest.com/historia-y-caracteristicas-de-cardano/
- https://bitcoin.es/criptomonedas/que-es-cardano/
- https://www.ledger.com/es/academy/que-es-la-prueba-de-participacion
- https://www.profesionalreview.com/2021/06/27/que-es-cardano/
- https://cardanofortheworld.com/es-es/marlowe-tutorial_playground-overview-es-ES