Tesis de Grado
de Ingeniería en Informática

Verificación de smart contracts en Marlowe para la blockchain Cardano

Director: Dr. Ing. Mariano G. Beiró
mbeiro@fi.uba.ar

Co-director: Phd. Simon Thompson (Kent University, IOHK)
S.J.Thompson@kent.ac.uk

Alumno: Julián Ferres, *(Padrón #101.483)*
jferres@fi.uba.ar

Facultad de Ingeniería, Universidad de Buenos Aires

29 de junio de 2022

Agradecimientos

I like to acknowledge.

Índice general

Índice de figuras

Lista de cuadros de código

Capítulo 1

Introducción

En este capítulo, presentaremos al lector algunos conceptos generales que fueron utilizados a lo largo del desarrollo de esta tesis. Se presentarán temas tales como cadenas de bloques, contratos inteligentes y Cardano (la cadena de bloques en la cual se centra la escritura de dichos contratos y verificación de propiedades).

Luego de este capítulo, el lector contará con las herramientas necesarias para adentrarse en cuestiones teóricas especificas a esta tesis.

1.1. Cadena de bloques o Blockchain

Las cadenas de bloques, conocidas en inglés como *blockchains*, son estructuras de datos en las cuales la información se divide en conjuntos (bloques) que cuentan con información adicional relativa a bloques previos de la cadena.

Con esta organización relativa, y con ayuda de técnicas criptográficas, la información de un bloque solo puede ser alterada modificando todos los bloques posteriores.

Esta propiedad facilita su aplicación en un entorno distribuido, de manera tal que la cadena de bloques puede modelar una base de datos pública no relacional, que contenga un registro histórico irrefutable de información.

En la práctica esta técnica ha permitido la implementación de un registro contable o *ledger* distribuido que soporta y garantiza la seguridad de transacciones y dinero digital. El concepto de cadena de bloque fue aplicado por primera vez en 2009 como parte central de Bitcoin [Nakamoto, 2008]. En este trabajo, nos concentraremos en la cadena de bloques conocida como Cardano [Cardano, 2019, Corduan et al., 2019].

Con respecto a como se implementa en sistemas reales, una blockchain es un tipo de base de datos o libro mayor (*ledger*) que se duplica y distribuye a todos los participantes dentro de la red de esa blockchain. Está formada por un conjunto de nodos interconectados que almacenan

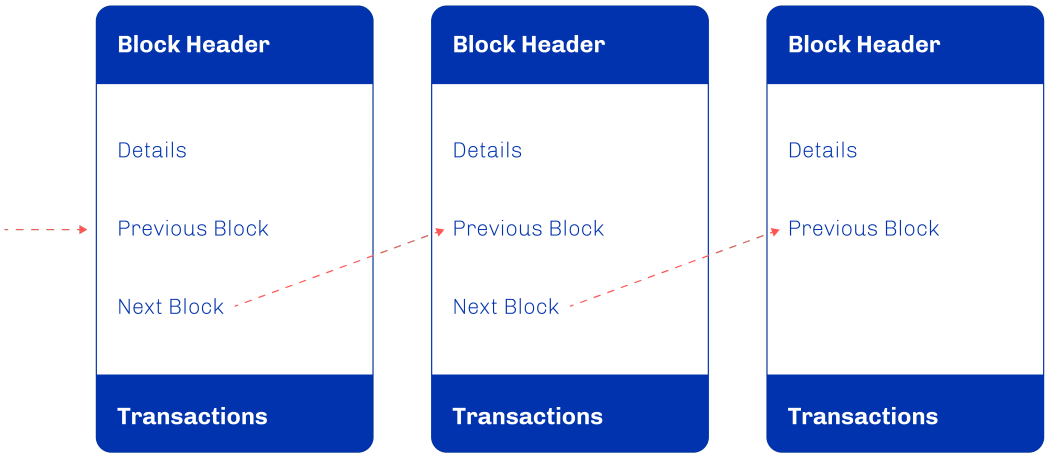


Figura 1.1: Representación simplificada de los datos en un bloque de la cadena. Extraída de [Brünjes and Vinogradova, 2019].

datos o elementos de valor en bloques. Estos bloques se verifican mediante transacciones y se vinculan entre sí mediante un orden cronológico en la cadena. Los detalles de estas transacciones están escritos de forma permanentemente en el bloque y no pueden modificarse.

Como una cadena de bloques almacena datos de manera descentralizada, es independiente de entidades de control centralizadas o intermediarios. Esto proporciona una mayor transparencia del almacenamiento de datos y su gestión. Una característica importante de blockchain es que almacena registros de forma inmutable, lo que significa que no se pueden cambiar, falsificar ni eliminar, ya que esto rompería la cadena de registros.

Las cadenas de bloques no solo proporcionan una base de datos inmutable y segura, sino que también actúan como un entorno funcional para realizar transacciones de fondos, crear monedas digitales y procesar transacciones complejas mediante acuerdos digitales, conocidos como *smart contracts*.

Los propietarios de activos digitales se identifican por sus claves públicas, y pueden ser personas o máquinas.

1.2. Smart contracts

Un contrato inteligente o *smart contract* es un acuerdo digital automatizado, escrito en código, que rastrea, verifica y ejecuta las transacciones vinculantes de un contrato entre varias par-

tes. Las transacciones del contrato se ejecutan automáticamente mediante el código del smart contract cuando se cumplen las condiciones predeterminadas. Esencialmente, un contrato inteligente es un programa corto cuyas entradas y salidas son transacciones en una cadena de bloques.

Los smart contracts son auto-ejecutables y confiables y no requieren las acciones o la presencia de terceros. El código del contrato inteligente se almacena y distribuye a través de una red blockchain descentralizada, lo que lo hace transparente e irreversible.

En resumen, los contratos inteligentes son inmutables ya que no se pueden modificar, pueden distribuirse y son a prueba de manipulaciones, rápidos y rentables, ya que no hay intermediarios. Esto es seguro gracias al cifrado del mismo.

Cardano presentará el soporte de contratos inteligentes en 2021. Como un entorno funcional, Cardano apoyará el desarrollo y la implementación de contratos inteligentes utilizando lenguajes de programación como:

- **Plutus**: Una plataforma de desarrollo y ejecución de smart contracts especialmente diseñada. Los contratos de Plutus consisten en partes que se ejecutan en la blockchain (código *on-chain*) y partes que se ejecutan en la maquina del usuario ('código *off-chain* o de cliente'). Plutus se basa en la investigación de lenguajes modernos para proporcionar un entorno de programación completo y seguro basado en Haskell, el lenguaje de programación funcional líder.
- **Marlowe**: Un lenguaje de dominio específico [Fowler, 2010] para escribir y ejecutar contratos financieros que permite construir contratos visualmente, así como en código más tradicional. Las instituciones financieras pueden usarlo para desarrollar e implementar instrumentos personalizados para sus clientes y usuarios. El propio lenguaje Marlowe está integrado tanto en JavaScript como en Haskell y ofrece una selección de editores según las preferencias y el conjunto de habilidades de los desarrolladores.

1.3. Criptomonedas

Las criptomonedas son activos digitales que se almacenan en el *ledger* y están diseñadas para servir como medio de intercambio de bienes o servicios. Suelen ser popularmente llamadas 'criptos'.

Los ledgers de blockchain son utilizados como tecnología subyacente para la creación de criptomonedas en un entorno descentralizado. Los protocolos de blockchain utilizan técnicas criptográficas rigurosas para permitir el minting (acuñación o creación) de criptomonedas, asegurar y verificar la propiedad de las mismas y los registros de movimiento de fondos. El precio de la criptomoneda no está controlado por un gobierno o una institución financiera centralizada. Se define por su valor, la correlación con las cifras del mundo real y está impulsado por la oferta y la demanda del mercado.

Las direcciones se utilizan al enviar pagos en criptomonedas. Son identificadores únicos y están representados por una cadena de números y letras que se obtienen de las claves públicas del usuario.

1.4. Cardano

Cardano [IOHK, 2015] es una plataforma blockchain de tipo *proof-of-stake* (prueba de participación) [1] descentralizada de tercera generación y el hogar de la criptomoneda *ada*. Es la primera plataforma de cadena de bloques que evoluciona a partir de una filosofía científica y un enfoque impulsado por la investigación.

La primera generación de blockchains (con Bitcoin como gran representante) ofrecía ledgers descentralizados para la transferencia segura de criptomonedas. Sin embargo, tales cadenas de bloques no proporcionaron un entorno funcional para la liquidación de acuerdos complejos y el desarrollo de aplicaciones descentralizadas (DApps). A medida que la tecnología blockchain maduró, la segunda generación (por ejemplo Ethereum) proporcionó soluciones mejoradas para redactar y ejecutar contratos inteligentes, desarrollar aplicaciones y crear diferentes tipos de tokens. Sin embargo, la segunda generación de cadenas de bloques a menudo enfrenta problemas en términos de escalabilidad.

Cardano se concibe como la cadena de bloques de tercera generación, ya que combina las propiedades de las generaciones anteriores y evoluciona para satisfacer todas las necesidades que surjan de los usuarios. Al comparar las propiedades de las blockchains, se deben considerar muchos aspectos. Por lo tanto, la mejor solución debe garantizar la máxima seguridad, escalabilidad (rendimiento de transacciones, escala de datos, ancho de banda de la red) y funcionalidad (además del procesamiento de transacciones, la cadena de bloques debe proporcionar todos los medios para la liquidación de acuerdos comerciales). Asimismo, es importante asegurarse de que la tecnología blockchain esté en constante desarrollo en términos de sostenibilidad y sea interoperable con otras blockchains e instituciones financieras.

Figura 1.2: Logo de la cadena de bloques Cardano. Extraído de la página de Wikipedia de Cardano.

La plataforma Cardano ha sido diseñada desde cero y verificada por una combinación de ingenieros y expertos académicos en los campos de blockchain y criptografía. Tiene un fuerte enfoque en la sostenibilidad, la escalabilidad y la transparencia. Es un proyecto totalmente de

[1] Definiciones más extensas sobre PoS se encuentran en los sitios web de Ethereum y Coinbase

código abierto que tiene como objetivo ofrecer una infraestructura inclusiva, justa y resistente para aplicaciones financieras y sociales a escala global.

Uno de sus principales objetivos es brindar servicios financieros confiables y seguros a aquellas personas que actualmente no tienen acceso.

Cardano ha sido diseñado con la seguridad como uno de sus principios fundamentales. Está escrito en Haskell, un lenguaje de programación funcional. Los lenguajes funcionales fomentan la construcción de software usando funciones puras, lo que conduce a un diseño en el que los componentes se pueden probar convenientemente de forma aislada. Además, las funciones avanzadas de Haskell nos permiten emplear una amplia gama de métodos potentes para garantizar la corrección del código, como basar la implementación en especificaciones formales y ejecutables, pruebas exhaustivas basadas en propiedades y ejecutar pruebas en simulación.

Cardano está desarrollando una plataforma de *smart contracts* que busca ofrecer funciones más avanzadas que cualquier protocolo desarrollado anteriormente, y servirá como una plataforma estable y segura para el desarrollo de dApps de nivel empresarial.

1.4.1. Ada como criptomoneda de Cardano

Cada ledger de blockchain tiene su criptomoneda subyacente o moneda nativa. Ada es la moneda nativa o principal en Cardano. Esto significa que ada es la principal unidad de pago en Cardano; se acepta como pago de cuotas, para realizar depósitos, y también es la única moneda en la que se distribuyen las recompensas.

Lovelace es la denominación más pequeña de ada, siendo $\boxed{1 \text{ ada} = 1{,}000{,}000 \text{ lovelaces}}$. Ada tiene seis decimales, lo que la hace fácilmente divisible en fracciones más pequeñas.

Tokens nativos

Cardano también admite la creación de tokens nativos: activos digitales que se crean para fines específicos. Esto significa que los usuarios, desarrolladores y empresas pueden usar la cadena de bloques de Cardano para crear tokens que representen una huella de valor (ya sea definida por la comunidad, el estado del mercado o la entidad autónoma). Un token puede ser fungible (intercambiable) o no fungible[2] (único) y actuar como unidad de pago, recompensa, activo comercial o contenedor de información.

1.4.2. Proof of Stake

Proof of Stake (PoS o Prueba de participación) es un tipo de protocolo de consenso que utiliza la cantidad de participación (o valor) mantenida en el sistema para determinar el consenso [Vashchuk and Shuwar, 2018, Zhang and Chan, 2020, Nair and Dorai, 2021].

[2]Ampliamente conocido por su acrónimo en inglés NFT

En esencia, un protocolo de consenso es lo que controla las leyes y los parámetros que rigen el comportamiento de las cadenas de bloques. El consenso puede resumirse como un conjunto de reglas a las que se adhiere cada participante de la red.

Dado que las blockchains no están controladas por ninguna autoridad central única, en su lugar se utiliza un protocolo de consenso para permitir que los participantes del sistema distribuido acuerden el historial de la red reflejada en la cadena de bloques, para llegar a un consenso sobre lo que ha sucedido y continuar desde una sola fuente de 'verdad'.

Cardano se basa en el protocolo de consenso PoS llamado Ouroboros [Kiayias et al., 2017], el primer protocolo de consenso de blockchain que se desarrolla a través de una investigación revisada por pares. En el corazón del protocolo se encuentran los *stake pools* (grupos de participación), nodos servidores confiables administrados por un operador de *stake pools* en los que los titulares de ada pueden delegar su participación. Los *stake pools* se utilizan para garantizar que todos puedan participar en el protocolo, independientemente de la experiencia técnica o la disponibilidad para mantener un nodo en funcionamiento.

Proof of Stake vs. Proof of Work

Por el contrario, el protocolo conocido como *Proof of Work* (PoW) o prueba de trabajo es un mecanismo síncrono que anima a los mineros a competir para ser los primeros en resolver cualquier problema dentro del bloque. Se utiliza un sistema de recompensas para incentivar esta resolución de problemas. Sin embargo, este enfoque acarrea una gran desventaja, con un mayor uso de electricidad y períodos de tiempo más largos para resolver problemas dentro de la cadena. Estos factores pueden ralentizar la red significativamente y hacerla más costosa de mantener.

Características del protocolo proof of stake

Una de las características clave de PoS es que a medida que aumenta el valor del usuario, también aumenta la oportunidad de mantener el ledger; Esto significa una mayor probabilidad de producir nuevos bloques que se pueden agregar a la cadena de bloques.

El creador de un nuevo bloque se elige en función de una combinación de selección aleatoria y una medida de su participación o riqueza. Cabe aclarar que dentro de un protocolo de prueba de participación, los participantes acumulan las tarifas de transacción, lo que aumenta su riqueza a medida que avanzan. Este enfoque fomenta el crecimiento constante y estable de la blockchain y reduce los casos de transacciones estancadas que pueden impedir el crecimiento de la misma.

Principales ventajas de Proof of Stake por sobre Proof of Work

- Se incorporan rigurosos protocolos de seguridad en un protocolo PoS.
- Centralización reducida: el riesgo de centralización se reduce al emitir sanciones por prácticas egoístas dentro de la red

- Eficiencia energética: el consumo de energía es extremadamente eficiente ya que se necesita una cantidad menor de electricidad, así como recursos de hardware, para producir y ejecutar la cadena de bloques.
- Eficiencia de costos: las monedas PoS son mucho más rentables que las que operan en los protocolos PoW.

Capítulo 2

Escritura de contratos financieros en Marlowe para Cardano

Durante este capítulo, nos adentraremos en la escritura de contratos financieros en Marlowe. Para esto será necesario describir el modelo contable de preferencia de dichas plataformas descentralizadas, el estándar involucrado en la clasificación de los contratos financieros y el lenguaje propiamente dicho.

Al final del mismo, el lector estará familiarizado con la notación correspondiente y expondremos brevemente la especificación para un tipo de contrato.

2.1. El modelo UTXO

Para entender la estructura de los contratos en Cardano, es importante tener comprensión de como se lleva a cabo la contabilidad en la misma. Tradicionalmente, pensamos en las transferencias de dinero entre dos cuentas bancarias, o quizás direcciones de Internet en el caso de la moneda digital.

La plataforma Cardano, asi como otras plataformas de criptomonedas como Bitcoin, utilizan en su lugar un enfoque contable conocido como UTXO [Corduan et al., 2019] (*Unspent transaction output*, o 'Salidas de transacción no utilizadas').

El modelo UTXO [Zahnentferner, 2018, Brünjes and Gabbay, 2020] documenta el flujo de dinero no de cuenta a cuenta, sino de **transacción a transacción**. Cada transacción tiene entradas (de dónde proviene el dinero que se gasta) y salidas (hacia donde se dirige este dinero).

Consideremos el gráfico de flujo de dinero en la figura 2.1. Las lineas negras representan outputs no gastados de las transacciones, y las lineas rojas representan dichos outputs siendo utilizados como inputs de transacciones posteriores.

Las cajas sin etiquetas representan una transacción (que contiene varios inputs y outputs). Los certificados azules denotan los outputs no gastados disponibles en nuestra ilustración.

Al comienzo del gráfico de flujo, Alice tiene 100 Ada en 'outputs sin utilizar' previos al comienzo de nuestro análisis. Este dinero proviene de una o más transacciones pasadas, que exceden el alcance del gráfico. Simplificamos el mismo con una simple caja (etiquetada con su nombre y el dinero correspondiente).

Dicha caja tiene dos lineas negras (outputs) saliendo de ella, siendo la suma del valor de las mismas 100 Ada:

- Un output de 58 Ada permanece sin ser utilizado y es parte de los outputs sin utilizar al final del análisis.
- Un output de 42 Ada se utiliza como parte de la nueva transacción.

Por su parte, Bob tiene 10 Ada de previos outputs sin utilizar. Los utiliza a todos en la nueva transacción. La transacción que ilustramos tiene dos inputs: 42 de Alice y 10 de Bob. La misma también tiene dos outputs: 2 para Bob y 50 para Charlie.

Vemos también que Charlie tiene 52 Ada provenientes de outputs previos a nuestro gráfico, totalizando 102 Ada que puede utilizar en transacciones futuras. Bob termina con solo un output de 2 Ada, y Alice con un total de 58 Ada.

El modelo anterior muestra estrictamente el flujo de dinero entre varios participantes. En esta versión simplificada, por ejemplo, las transacciones ilustradas no pagan comisiones. Sin embargo, en este modelo simplificado, vemos que los outputs deben gastarse en su totalidad. Es decir, un registro de un output no gastado no puede ser modificado (esta acción se adecúa a los modelos contables basados en cuentas), solo podría utilizarse de forma completa.

Para mantener la integridad de la contabilidad, las nuevas transacciones debe tener todas las outputs no gastados (totalizando la cantidad correcta de outputs no gastados) utilizados como entrada.

En nuestro ejemplo anterior, la nueva transacción elimina (utilizándolas como entrada) a los outputs no gastados de valor 42 de Alice y 10 de Bob, para un total de 52 Ada. Esto implica que la transacción esta obligada a totalizar 52 Ada como outputs sin gastar (que de hecho cumple, con 2 para Bob y 50 para Charlie).

Un aspecto a destacar es que Bob tiene un *unspent output* como entrada y uno como salida. Esto se podría interpretar como un 'cambio' (de 2 Ada) para esta transacción. Dicho concepto es similar al que utilizamos en el día a día al realizar pagos en efectivo: Si un producto cuesta \$98 y tenemos un billete de \$100, no podemos fraccionar dicho billete. Tenemos que pagar con todo el billete y recibir \$2 de cambio.

Dado que no existe una forma real de gastar parte de un *unspent output*, así es como el modelo UTXO trata el gasto parcial: agregando una salida de 'cambio'.

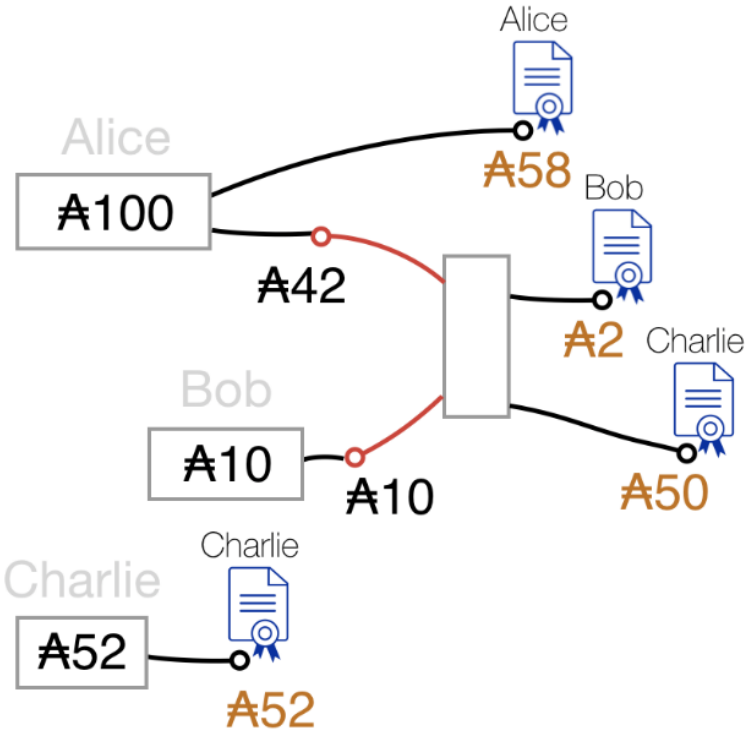


Figura 2.1: Flujo de dinero en el modelo UTXO. Extraído de [Brünjes and Vinogradova, 2019].

Cabe destacar que este modelo contable hace que sea conveniente distribuir el flujo de efectivo de varios contribuyentes a varios destinatarios haciendo que el mismo fluya hacia un fondo común, en este caso, la transacción, antes de enviarse a los beneficiarios finales. Esto representa, en un sentido muy general, el objetivo de los *smart contracts* o contratos inteligentes.

Veamos más formalmente lo que sucede durante una transacción en el modelo UTXO.Para el modelo de transacciones básico que analizaremos, podemos referirnos a las siguientes definiciones:

Datos primitivos:

$$\begin{aligned} \text{txid} \in \text{TxId} &\quad \text{id de transacción} \\ \text{ix} \in \text{Ix} &\quad \text{índice} \\ \text{addr} \in \text{Addr} &\quad \text{dirección} \\ \text{c} \in \text{Coin} &\quad \text{valor de la divisa} \end{aligned}$$

Datos derivados:

$$\begin{aligned} \text{tx} \in \text{Tx} &= (\text{inputs, outputs}) \in \mathbb{P}(\text{TxIn}) \times (\text{Ix} \mapsto \text{TxOut}) && \text{transacción} \\ \text{txin} \in \text{TxIn} &= (\text{txid, ix}) \in \text{TxId} \times \text{Ix} && \text{entrada de la transacción} \\ \text{txout} \in \text{TxOut} &= (\text{addr, c}) \in \text{Addr} \times \text{Coin} && \text{salida de la transacción} \\ \text{utxo} \in \text{UTxO} &= \text{txin} \mapsto \text{txout} \in \text{TxIn} \mapsto \text{TxOut} && \text{salidas sin gastar} \\ \text{b} \in \text{Block} &= \{\text{tx}\} \in \mathbb{P}(\text{Tx}) && \text{bloque} \\ \text{pending} \in \text{Pending} &= \{\text{tx}\} \in \mathbb{P}(\text{Tx}) && \text{transacciones pendientes} \end{aligned}$$

Funciones:

$$\begin{aligned} \text{txid} \in \text{Tx} \mapsto \text{TxId} &\quad \text{computar id de la transacción} \\ \text{ours} \in \text{Addr} \mapsto \mathbb{B} &\quad \text{direcciones que corresponden a la billetera} \end{aligned}$$

Filtros sobre Conjuntos:

$$\begin{aligned} \text{Addr}_{ours} &= \{a \mid a \in \text{Addr},\ \text{ours}\ a\} \\ \text{TxOut}_{ours} &= \text{Addr}_{ours} \times \text{Coin} \end{aligned}$$

Antes de analizar la estructura de las transacciones, haremos un pequeño repaso sobre como la contabilidad se lleva a cabo en el 'libro mayor' o *ledger*. El registro que contiene la información sobre el *ledger* es llamado `UTXO`. Este registro es un map finito, donde la key o clave is un par formado por el id de la transacción de la cual se toma el dinero y un indice, `TxIn = TxId * Ix`. El id de la transacción puede ser calculado en base a una transacción completada para procesar, y es un identificador único de la transacción.

El índice `Ix` es necesario debido a que puede haber mas de un output en dicha transacción, y cada uno de los mismos tiene que tener un identificador único dentro de el conjunto de *outputs* dentro de una transacción.

Los valores o *values* en el mapa son pares formados por un *coin value* y una dirección, y el tipo de los mismos es `TxOut = Addr * Coin`. Cabe destacar que las direcciones de los usuarios

son siempre claves públicas, y los fondos en ellas pertenecen a la entidad que puede probar que posee la clave privada correspondiente. Las direcciones de *script* (smart contract o contrato inteligente) se comportan de manera ligeramente distinta, debido a que no tienen un dueño directo.

Para poder comprender la estructura de la transacción en sí, analicemos primero los *outputs*. Una transacción puede distribuir el dinero que está gastando a varias direcciones diferentes. Los *outputs*, (valores de tipo `TxOut`) se almacenan en una transacción como valores en un mapa finito. Las claves del mapa son indices únicos dentro del contexto del mapa, de manera tal que la combinación del id de la transacción y dicho indice identifica de forma global a dicho output. En el modelo UTXO, se relaciona a los valores de salida con las entradas de las cuales provienen, por medio de este identificador global compuesto.

Los inputs, cuyo orden no es relevante, son un conjunto y no una lista. Los elementos de este conjunto no contienen ni el valor de la moneda a gastar, ni la dirección de donde proviene el dinero. Esta es la principal distinción entre el modelo contable tradicional y el UTXO:el dinero que se gasta solo referencia a los outputs no gastadas de transacciones previamente procesadas en el *ledger* que reside actualmente en la blockchain. Cada elemento del mencionado set de inputs es un par formado por el id de la transacción y un indice que, como se explicó anteriormente, identifica de forma única el output no gastado en la UTXO.

Procesar una transacción implica actualizar el `UTXO` en el ledger de manera tal que los fondos gastados por la transacción que se está procesando estén disponibles para que los gasten los propietarios de las direcciones de las salidas de la transacción. Es decir, todas las entradas correspondientes a inputs de la transacción procesada se eliminan del ledger `UTXO`.

Adicionalmente, todos los valores de `TxOut` en el mapa finito de las salidas de la transacción se agregan a la `UTXO`, con la clave del mapa finito que consiste en el id de la transacción que se procesa, y el valor del índice es el mismo que en el mapa finito de salidas de esta transacción. Es decir, si `tx` contiene un par formado por el conjunto de entrada y el mapa de salidas `(ins, outs)` con id `id`, y `ix |->(a, c)` es una entrada de `outs`, la `UTXO` va a tener la entrada `(id, ix) |->(a, c)` agregada. En este párrafo, utilizamos la notación `k |->v` para referirnos a una entrada del mapa finito con clave `k` y valor `v`.

Veamos como se refleja dicha actualización del ledger en términos de notación matemática (que puede ser reflejada en código con relativa facilidad). Las siguientes son tres formas de filtrar el mapa finito de `UTXO`. El primero filtra dicho mapa mediante un subconjunto `ins` de las claves. El segundo filtro obtiene el complemento del resultado del primer filtro (en otras palabras, todas las entradas de la `UTXO` que no son indexadas por claves en la lista de inputs). El tercero filtra el mapa mediante los valores.

$$\begin{aligned}
\texttt{ins} \lhd \texttt{utxo} &= \{i \mapsto o \mid i \mapsto o \in utxo,\ i \in ins\} && \text{restricción de dominio} \\
\texttt{ins} \not\lhd \texttt{utxo} &= \{i \mapsto o \mid i \mapsto o \in utxo,\ i \notin ins\} && \text{exclusión de dominio} \\
\texttt{utxo} \rhd \texttt{outs} &= \{i \mapsto o \mid i \mapsto o \in utxo,\ o \in outs\} && \text{restricción de rango}
\end{aligned}$$

Utilizaremos la notación introducida para procesar una nueva transacción. En otras palabras,

eliminar los outputs no gastados correspondientes y construir un nuevo conjunto de outputs que serán agregados al `UTXO` (como se describió anteriormente). Los outputs a agregar serían computados de la siguiente manera:

$$
\begin{array}{rcl}
\text{txins} & \in & \mathbb{P}(\text{Tx}) \to \mathbb{P}(\text{TxIn}) \\
\text{txins } txs & = & \bigcup\{inputs \mid (inputs, \_) \in txs\} \\
\text{txouts} & \in & \mathbb{P}(\text{Tx}) \to \text{UTXO} \\
\text{txouts } txs & = & \left\{ (\text{txid } tx, ix) \mapsto txout \;\middle|\; \begin{array}{lcl} tx & \in & txs \\ (\_, outputs) & = & tx \\ ix \mapsto txout & \in & outputs \end{array} \right\}
\end{array}
$$

Usando esta notación, podemos definir la actualización del `UTXO`, debido a la transacción `tx` como:

$$(\text{txins } tx \not\lhd \ utxo) \cup \text{outs } tx$$

Hay que tener en cuenta que se debe realizar un cálculo explícito de la cantidad total de Aca en las salidas y el total de *ada* en todas las entradas de una transacción como parte de la validación de la transacción. También podría haber outputs en una transacción sin inputs correspondientes; estos se deben a la recolección de recompensas.

Ahora, para validar una transacción, se realiza una serie de cálculos que involucran el Ada en la misma y el Ada en otras cuentas del ledger, para asegurarse de que no se crea ni se destruye dinero. Esto se conoce como 'propiedad contable generalizada'. El modelo contable UTXO brinda protección integrada contra el 'doble gasto' de un output determinado.

Esta protección inherente, junto con la aplicación de la propiedad contable generalizada, asegura que no se permita que ocurra ningún gasto deshonesto. Esta es una propiedad crucial del sistema contable del ledger de Cardano, en particular porque existe una cantidad fija de Ada que nunca puede cambiar.

Para finalizar, hay que tener en cuenta que una transacción incluye una gran cantidad de datos adicionales, como testigos, certificados, y scripts juntos con sus hashes. En esta sección no hemos entrado en los detalles de los tipos y cálculos específicos utilizados en la implementación del ledger de Cardano. Sin embargo, abarcamos suficiente información como para poder entender que sucede detrás de escena cuando se genera una transacción en la blockchain.

2.2. Marlowe como DSL

Marlowe [Lamela Seijas et al., 2020a, Kondratiuk et al., 2021] es un lenguaje pequeño, con pocas sentencias soportadas que, para cada contrato, describen el comportamiento que involucra un conjunto fijo y finito de roles.

Marlowe está diseñado para crear bloques para contratos financieros: pagos o depósitos de las partes, elecciones e información del mundo real. Cuando se ejecuta un contrato, los roles que implica son satisfechos por los participantes, que son identidades en la cadena de bloques. Cada rol está representado por un token en la cadena y los roles se pueden transferir durante la ejecución del contrato, lo que significa que esencialmente se pueden intercambiar.

Los contratos se pueden construir reuniendo una pequeña cantidad de estas sentencias que, en combinación, se pueden usar para describir y modelar muchos tipos diferentes de contratos financieros. Algunos ejemplos incluyen un contrato que puede realizar un pago a un rol o a una clave pública, un contrato que puede esperar una acción por parte de uno de los roles, como un depósito de moneda, o una elección entre un conjunto de opciones.

En particular, un contrato no puede esperar indefinidamente una acción: si la misma no se ha realizado en un tiempo determinado (conocido como *timeout*), el mismo continuará con un comportamiento alternativo, por ejemplo, reembolsar los fondos en el contrato.

Los contratos de Marlowe pueden ramificarse en función de alternativas y tienen una vida finita, al final de la cual el dinero restante retenido por el mismo se devuelve a los participantes. Esta característica garantiza que el dinero no se puede bloquear para siempre en un contrato. Dependiendo del estado actual de un contrato, se puede elegir entre dos cursos de acción alternativos, que son en sí mismos contratos. Cuando no se requieran más acciones, el contrato se cerrará y se reembolsará cualquier moneda restante en el contrato.

2.2.1. El modelo de Marlowe

Marlowe está diseñado para soportar la ejecución de contratos financieros en la blockchain, específicamente en Cardano. Los contratos se construyen reuniendo una pequeña cantidad de constructores que se pueden combinar para describir muchos tipos diferentes de contratos financieros.

Antes de describir estos constructores en 2.2.2, debemos analizar el enfoque general para modelar contratos en Marlowe, y el contexto en el que se ejecutan los mismos (la blockchain de Cardano). Al hacer esto, también presentamos parte de la terminología que usaremos, indicando las definiciones en cursiva.

Contratos

Los contratos en Marlowe se ejecutan en una blockchain, pero deben interactuar con el mundo fuera de la misma. Las partes (*parties*) del contrato, a las que también llamamos participantes,

pueden realizar diversas acciones: se les puede pedir que depositen dinero o que elijan entre varias alternativas. Una notificación de un valor externo (también llamado valor de oráculo), como el precio actual de un producto en particular, es la otra forma posible de entrada.[1]

La ejecución de un contrato también producirá efectos externos, al realizar pagos a las partes en el mismo.

Participantes y roles

Los *participantes* y los *roles* se comportan de forma ligeramente distinta en un contrato en Marlowe. Los roles en un contrato son fijos e inmutables, y pueden ser llamados *party*, *counter-party*, etc. Por otro lado, los participantes que están vinculados a los roles del contrato pueden cambiar durante la ejecución de la instancia de contrato en particular.

Esto permite que los roles en la ejecución de contratos se intercambien entre los participantes, a través de un mecanismo de *tokenización*. A la fecha de escritura de este documento, la simulación en *Marlowe Playground* [IOHK, 2021] simplemente presenta roles de contrato.

Cuentas

El modelo de Marlowe permite que un contrato pueda almacenar activos. Todas las partes que participan en el contrato poseen implícitamente una cuenta con su nombre. Todos los bienes almacenados en el contrato deben estar en la cuenta de una de las partes; de esta forma, cuando se cierra el contrato, todos los bienes que quedan en el contrato pertenecen a alguien, por lo que pueden ser reembolsados a sus respectivos dueños. Estas cuentas son *locales*: solo existen mientras dura la ejecución del contrato, y durante ese tiempo son accesibles (solo desde dentro del contrato).

Pasos y estados

Los contratos de Marlowe describen una serie de *pasos*, generalmente describiendo el primer paso, junto con otro *sub-contrato* que describe qué hacer a continuación. Por ejemplo, el contrato `Pay a p t v cont` dice "realice un pago del valor `v` del token `t` a la parte `p` desde la cuenta `a`, y luego siga el contrato `cont`". Llamamos `cont` a la continuación del contrato.

Al ejecutar un contrato, debemos realizar un seguimiento del *contrato actual*: después de dar un paso en el ejemplo anterior, el contrato actual es la continuación, `cont`. También tenemos que realizar un seguimiento de información adicional, como por ejemplo: cuánto se tiene en cada cuenta. Llamamos a esta información el *estado*: este también cambia potencialmente en cada paso. Un paso también puede implicar que se lleva a cabo una acción, como el depósito de dinero o la producción de un *efecto*, por ejemplo un pago.

[1] Podemos pensar a los oráculos como otro tipo de *party* del contrato; bajo este punto de vista, las notificaciones se convierten en las elecciones (*choices*) realizadas por esa *party*.

Blockchain

Si bien Marlowe está diseñado para trabajar con cadenas de bloques en general, algunos detalles de cómo interactúa con la misma son relevantes al describir la semántica y la implementación del lenguaje.

Una cadena de bloques basada en UTXO 2.1 contiene una colección de transacciones. Cada transacción tiene un conjunto de entradas y salidas, y la cadena de bloques se construye vinculando las salidas de transacciones no gastadas (UTXO) a las entradas de una nueva transacción. Como máximo se puede generar un bloque en cada *slot*, que tienen una duración de 1 segundo.

Los mecanismos por los cuales se generan estos bloques, y por quién, no son relevantes para el alcance de este documento, pero los contratos se expresarán en términos de números de *slot*, contados desde el bloque de inicio ("génesis") de la cadena de bloques. Actualmente, utilizan números de *slot* en la simulación en el Marlowe Playground [IOHK, 2021], pero en cadena usaremos el tiempo, adoptando el estándar de tiempo *Posix*.

UTXO, billeteras y la aplicación Marlowe Run

El dinero dentro de la blockchain reside en los UTXO, que están protegidos criptográficamente por una clave privada en poder del propietario. Estas claves se pueden usar para canjear la salida y usarlas como entradas para nuevas transacciones. Los usuarios suelen realizar un seguimiento de sus claves privadas y los valores adjuntos a ellas en una billetera (o *wallet*) criptográficamente segura.

Para interactuar con un contrato que se ejecuta en blockchain, los usuarios deberán usar la aplicación cliente *Marlowe Run*. Esta interactuará con las billeteras de los usuarios para autenticar las transacciones que gastan activos, ya que los depósitos se realizan desde las billeteras de los usuarios y los pagos son recibidos por ellos. El código que realiza este tipo de acciones suele ser llamado *off-chain*, debido a que no queda inmortalizado en la misma.

Simulación omnisciente

El Playground de Marlowe [IOHK, 2021] admite la simulación de contratos. Esta es una simulación omnisciente, en la que el usuario puede realizar cualquier acción para cualquier rol y, por lo tanto, puede observar la ejecución desde la perspectiva de todos los usuarios simultáneamente. Esto contrasta con la experiencia de ejecutar un contrato en Marlowe Run, en el que cada participante ve el contrato desde su propio punto de vista. En particular, los participantes solo pueden interactuar con un contrato en ejecución que está esperando su entrada; si ese no es el caso, entonces verán que la ejecución del contrato está esperando la participación de otra persona.

Valores y *tokens*

En los ejemplos de los capítulos posteriores, veremos que siempre que se requiere un *valor*, hemos utilizado exclusivamente *Ada*. Esto tiene sentido ya que *Ada* es la moneda fundamental respaldada por Cardano.

Sin embargo, Marlowe ofrece un concepto de *valor* más general, que admite tokens *nativos* personalizados, que pueden ser fungibles, no fungibles o mixtos:

```
newtype Value = Value
    {getValue :: Map CurrencySymbol (Map TokenName Integer)}
```

Código 2.1: Definicion del tipo `Value`

Los tipos `CurrencySymbol` y `TokenName` son wrappers simples alrededor de `ByteString`.

Esta noción de valor abarca a *Ada*, tokens fungibles (como monedas), tokens no fungibles o NFT (tokens personalizados que no son intercambiables con otros tokens) y casos mixtos más exóticos:

- Ada usa la cadena de bytes vacía como `CurrencySymbol` y `TokenName`.
- Un *token fungible* está representado por un `CurrencySymbol` (o símbolo de moneda) para el que existe exactamente un `TokenName` (nombre de token) que puede tener una cantidad entera arbitraria no negativa (del cual *Ada* es un caso especial).
- Una clase de *tokens no fungibles* es un `CurrencySymbol` con varios `TokenNames`, cada uno de los cuales tiene una cantidad de uno. Cada uno de estos nombres corresponde a un *token único no fungible*.
- Los *tokens mixtos* son aquellos que tienen varios `TokenNames` y cantidades superiores a uno.

Ejecución de un contrato Marlowe

Ejecutar un contrato de Marlowe en la cadena de bloques de Cardano significa restringir las transacciones generadas por el usuario a la lógica del contrato. Si en un punto particular de ejecución, un contrato espera un depósito de `100 Ada` de Alice, solo esa transacción tendrá éxito, cualquier otra será rechazada.

Una transacción contiene una lista ordenada de *entradas* o *acciones*. El intérprete de Marlowe se ejecuta durante la validación de transacciones. Primero, evalúa el contrato *paso* a *paso* hasta que no se puede cambiar más sin procesar ninguna entrada, una condición que se llama *quiescent* (o inactiva). En esta etapa, avanza a través cualquier `When` con tiempos de espera que hayan pasado, y todos los constructores `If`, `Let`, `Pay` y `Close`, sin procesar ninguna entrada.

A continuación, se procesa la primera entrada, y luego el contrato se vuelve a procesar hasta la inactividad. Este proceso se repite hasta que se procesan todas las entradas. En cada paso, el contacto actual y el estado cambiarán, es posible que se procesen algunas entradas y se realicen pagos.

Tal *transacción*, como se muestra en el diagrama a continuación, se agrega a la cadena de bloques.

Hemos mostrado que el comportamiento de un Marlowe es independiente de cómo se recopilan las entradas en las transacciones, por lo que cuando simulamos la acción de un contrato no necesitamos agrupar las entradas en transacciones explícitamente. Para ser más concretos, podemos pensar que cada transacción tiene como máximo una entrada. Si bien la semántica de un contrato es independiente de cómo se agrupan las entradas en transacciones, los costos de ejecución pueden ser menores si se pueden agrupar varias entradas en una sola transacción.

En la simulación omnisciente disponible en el *Playground*, podemos abstraernos con seguridad de la agrupación de transacciones, ya que la agrupación no afecta el comportamiento del contrato.

Construyendo una transacción

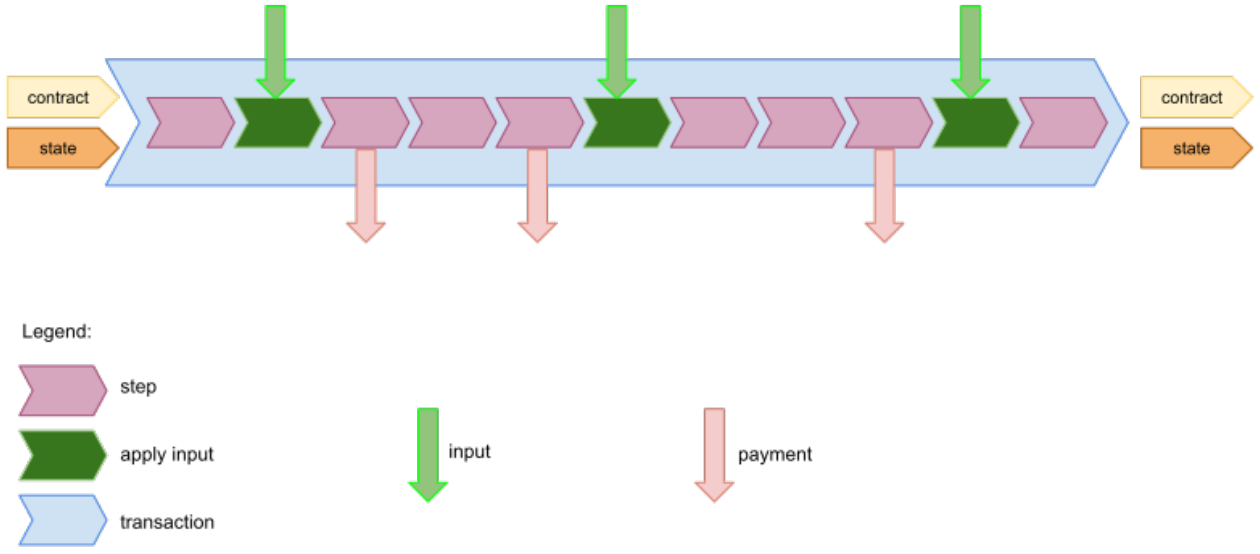


Figura 2.2: Diagrama de ejemplo de una transacción. Extraído de https://play.marlowe-finance.io/doc/marlowe/tutorials/marlowe-model.html.

2.2.2. Piezas fundamentales de un contrato en Marlowe

Un contrato en Marlowe se obtiene combinando una pequeña cantidad de sentencias o *building blocks*. Las mismas pueden llegar a describir muchos tipos de contratos financieros, como hacer

un pago, hacer una observación, esperar hasta que cierta condición se cumpla, etc. Luego, el contrato se ejecuta en una cadena de bloques, como Cardano, e interactúa con el mundo exterior.

Marlowe en sí mismo está embebido en Haskell, y se modela como una colección de tipos de datos algebraicos en Haskell [HaskellWiki, 2020]. A continuación mostramos la definición más general de un contrato en Marlowe:

```
data Contract = Close
              | Pay party payee token value contract
              | If observation contract1 contract2
              | When [Case] timeout contract
              | Let valueId value contract
              | Assert observation contract
```

Código 2.2: Tipos de contratos en Marlowe.

Marlowe tiene seis maneras de construir contratos. Cinco de esos métodos — `Pay`, `Let`, `If`, `When`, y `Assert` — construyen un contrato complejo a partir de contratos más simples, y el ultimo método, `Close` es un contrato simple. En cada paso de la ejecución, además de modificar el estado y proceder hacia un nuevo contrato, podrían generarse pagos y advertencias (*warnings*).

Antes de describir los métodos exhaustivamente, es útil conocer la definición de valores, observaciones y acciones:

1. **Valores (*values*)**: Incluyen cantidades que cambian con el tiempo, tales como: el *slot interval* o 'intervalo actual', el balance de cierto token en una cuenta o elecciones que se han realizado (conocidas como *valores volátiles*). Los valores pueden ser combinados usando operaciones como suma, resta, negación, etc. Los mismos pueden ser valores condicionales o una observación.

2. **Observaciones (*observations*)**: Valores booleanos que son obtenidos al comparar valores, y que pueden ser combinados con los operadores booleanos estándar. Además, es posible observar si alguna elección se ha realizado (para una elección en concreto). Las observaciones tendrán un valor en cada etapa de la ejecución.

3. **Acciones (*actions*)**: Suceden en momentos particulares durante la ejecución, por ejemplo: un depósito de dinero o elegir entre varias alternativas.

Pay

Un contrato de pago `(Pay acc payee tok val cont)` realizará un pago de valor `val` de un token `tok` desde una cuenta `acc` a un beneficiario `payee`, quien sera uno de los participantes del contrato, u otra cuenta en el mismo.

Se generarán *warnings* si el valor `val` no es positivo, o si no hay recursos suficientes en `acc` para realizar el pago en su totalidad (incluso si hay balances positivos de otros tokens en la misma). En este último caso, se realizará un pago parcial (conteniendo todo el dinero disponible). El contrato en el que continuará la ejecución es `cont`.

Close

Un contrato `Close` prevé que el contrato sea cerrado (o rescindido). La única acción que realiza es reembolsar a los titulares de cuentas que contienen un saldo positivo. Esto se realiza de a una cuenta a la vez, pero todas las cuentas se reembolsarán en una sola transacción.

If

El conditional `If obs cont1 cont2` continuará en `cont1` o `cont2`, dependiendo de la observación `obs` cuando el mismo es ejecutado.

When

Es el constructor de contratos mas complejo, con la forma `When cases timeout cont`. El mismo es activado por acciones, que pueden o no ocurrir en un *slot* en particular. Como continua el mismo tras una acción se declara en la sintaxis de `cases` del contrato.

En el contrato `When cases timeout cont`, la lista `cases` contiene una colección de casos. Cada caso es de la forma `Case ac co` donde `ac` es una acción y `co` un contrato de continuación. Cuando una acción en particular, por ejemplo `ac`, ocurre, el estado del contrato es actualizado correspondientemente y y mismo continuara su ejecución en `co`.

Para garantizar que el contrato eventualmente progresará, la ejecución de `When cases timeout cont` continuará como `cont` una vez que el slot `timeout` es alcanzado.

Let

Un contrato `Let id val cont` permite registrar un valor, en un punto particular en el tiempo, y darle nombre usando un identificador. En este caso, la expresión `val` se evalúa y se almacena con el nombre `id`. El contrato entonces continúa como `cont`.

Además de permitirnos usar abreviaturas, este mecanismo nos brinda la capacidad de capturar y guardar valores volátiles que pueden cambiar con el tiempo, por ejemplo: '*el precio actual del petróleo*', '*el slot actual, en un punto particular de la ejecución del contrato*', para ser utilizado más adelante en la ejecución del mismo.

Assert

Un contrato `Assert obs cont` no tiene ningún efecto en el estado de un contrato, que continua inmediatamente en `cont`, pero genera una advertencia cuando la observación `obs` es falsa. Puede

ser utilizado para asegurar que alguna propiedad se cumple en un momento particular de la ejecución del contrato. Esta sentencia es útil porque permite que un *análisis estático* detecte que algún `assert` es falso, para alguna ejecución específica del contrato.

2.3. Un contrato de ejemplo en Marlowe

Esta sección introduce un contrato financiero simple en pseudocódigo y luego explica como puede ser modificado para funcionar en Marlowe, dando el primer ejemplo de un contrato en el lenguaje. La misma se basa principalmente en el contrato de ejemplo brindado por la documentación oficial[2].

2.3.1. El contrato *Escrow*

Supongamos que Alice quiere comprarle un libro a Bob, pero ninguno de los dos confía en el otro. Afortunadamente, tienen una amiga en común, Carol, en quien ambos confían para que sea neutral (pero no lo suficiente como para darle el dinero y actuar como intermediaria). Entonces, acuerdan el siguiente contrato, escrito en pseudocódigo funcional. Este tipo de contrato es un ejemplo simple de depósito en garantía o *escrow*:

```
When aliceChoice
     (When bobChoice
           (If (aliceChosen `ValueEQ` bobChosen)
               agreement
               arbitrate))
```

Código 2.3: Primer pseudocódigo del contrato Escrow.

El contrato se describe utilizando los constructores vistos en la sección previa. El constructor más externo `When` tiene dos argumentos: el primero es una observación y el segundo es otro contrato. El significado esperado es que cuando ocurre la acción, se activa el segundo contrato.

El segundo contrato es en sí mismo otro `When` — esperando una decisión de Bob — pero dentro de este, hay una opción: Si (`If`) Alice y Bob están de acuerdo en qué hacer, se procede; en caso contrario, se le pide a Carol que arbitre y tome una decisión.

En general, `When` ofrece una lista de casos[3], cada uno con una acción y un contrato correspondiente que se activa cuando ocurre esa acción. Sabiendo esto, podemos permitir la opción de que Bob realice la primera elección, en lugar de Alice, de la siguiente forma:

[2]Disponible en https://play.marlowe-finance.io/doc/marlowe/tutorials/escrow-ex.html
[3]Las listas en Marlowe se representan entre corchetes, como en `[2,3,4]`.

```
When [ Case aliceChoice
            When [ Case bobChoice
                        (If (aliceChosen `ValueEQ` bobChosen)
                           agreement
                           arbitrate) ],
       Case bobChoice
            When [ Case aliceChoice
                      (If (aliceChosen `ValueEQ` bobChosen)
                            agreement
                            arbitrate) ]
     ]
```

Código 2.4: Pseudocódigo agnóstico al orden de las elecciones.

En este contrato, Alice o Bob pueden hacer la primera elección y el otro luego hará la siguiente. Si están de acuerdo, entonces se procede con el mismo; en caso contrario, Carol arbitra. En esta sección, y sin perdida de generalidad, supondremos que Alice elige primero (esto nos permite omitir el código simétrico y repetido que cubriría el caso en el que Bob comienza).

A continuación, se incluye un diagrama de secuencia que ilustra el comportamiento esperado del contrato, descrito anteriormente en pseudocódigo:

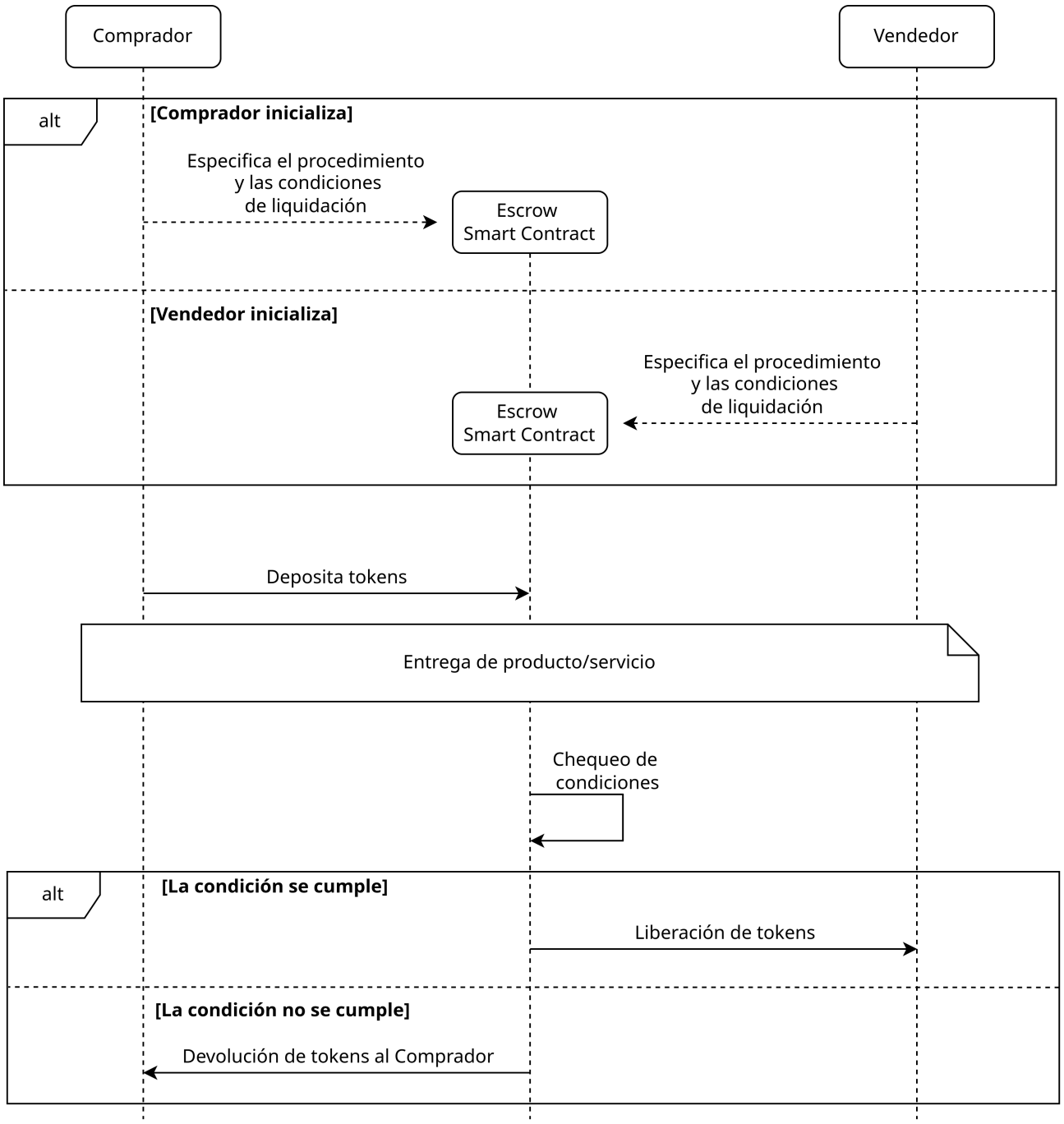


Figura 2.3: Diagrama de secuencia del contrato *Escrow*.

En [Peyton Jones et al., 2000, Bahr et al., 2015] se encuentran los trabajos iniciales sobre la utilización de programación funcional para la descripción de contratos financieros.

2.3.2. Escrow en Marlowe

Los contratos de Marlowe incorporan parámetros adicionales para garantizar que progresen correctamente. Cada vez que vemos un `When`, debemos proporcionar dos cosas adicionales:

- El tiempo de espera (o *timeout*) luego del cual el contrato progresará.
- El contrato de continuación al que progresa.

Agregando *timeouts*

Primero, examinemos cómo modificar el pseudocódigo escrito para solucionar el caso de que la condición del `When` no se cumpla. Para eso, agregamos valores de *timeout* y 'contrato de continuación' para cada `When` en el contrato:

```
When [ Case aliceChoice
            (When [ Case bobChoice
                         (If (aliceChosen `ValueEQ` bobChosen)
                            agreement
                            arbitrate) ]
                   60              -- AGREGADO
                   arbitrate)      -- AGREGADO
     ]
     40                            -- AGREGADO
     Close                         -- AGREGADO
```

Código 2.5: Pseudocódigo con timeouts.

El `When` más externo requiere que Alice haga la primera elección. Si Alice no ha hecho una elección luego del `slot`[4] 40, el contrato se cierra y se reembolsan todos los fondos del mismo.

El contrato `Close` suele ser el último paso en cada 'camino' a través de un contrato en Marlowe, y su efecto es el de reembolsar el dinero del contrato a los participantes. Describiremos esto con más detalle en 2.2.2. En este caso particular, el reembolso se realizará en el *slot* número 40.

Viendo los constructores internos, si se ha hecho la elección de Alice, entonces esperamos una de Bob. Si eso no está disponible para el *slot* 60, entonces se llama a Carol para que arbitre.

Agregando compromisos

A continuación, debemos ver cómo se compromete el efectivo como primer paso del contrato.

[4]Los *slots* representan la unidad básica de tiempo en la cadena de bloques.

```
When [Case (Deposit "alice" "alice" ada price)   -- AGREGADO
 (When [ Case aliceChoice
             (When [ Case bobChoice
                          (If (aliceChosen `ValueEQ` bobChosen)
                              agreement
                              arbitrate) ]
                   60
                   arbitrate)
       ]
       40
       Close)
   ]
   10        -- AGREGADO
   Close     -- AGREGADO
```

Código 2.6: Pseudocódigo con validación de fondos.

Se solicita un depósito de valor `price` de `"alice"`: si se da, entonces se mantiene en una cuenta, también llamada `"alice"`. Cuentas como esta existen solo durante la vigencia del contrato; cada cuenta pertenece a un solo contrato.

Hay un *timeout* en el *slot* número 10 para realizar el depósito; si se alcanza sin que se haya realizado el mismo, el contrato se cierra y se reembolsa todo el dinero que ya estaba en el contrato. En este caso, no hay ningún reembolso pendiente y simplemente finaliza del contrato.

Completando las funciones

En la descripción del contrato, hemos utilizado variables como `agreement`, `arbitrate` y `price`. Las mismas se aprovechan de la capacidad de Marlowe de ser embebido en Haskell 2.2.2, y por lo tanto es posible dar definiciones abreviadas. También se utilizó sobrecarga de cadenas (en el caso de `ValueEQ`) para hacer algunas descripciones y cuentas mas concisas.

Las funciones restantes pueden ser escritas en Haskell de la siguiente manera:

```
agreement :: Contract
agreement =
  If
    (aliceChosen `ValueEQ` (Constant 0))
    (Pay "alice" (Party "bob") ada price Close)
    Close

arbitrate :: Contract
arbitrate =
  When  [ Case carolClose Close,
          Case carolPay (Pay "alice" (Party "bob") ada price Close) ]
```

```
        100
        Close
```

Código 2.7: Funciones `agreement` y `arbitrate`.

Dentro de los contratos podemos usar abreviaciones como:

```
price :: Value
price = Constant 450
```

Código 2.8: Funcion `price`.

También podemos describir las elecciones realizadas por Alice y Bob:

```
aliceChosen, bobChosen :: Value

aliceChosen = ChoiceValue (ChoiceId choiceName "alice")
bobChosen   = ChoiceValue (ChoiceId choiceName "bob")

choiceName :: ChoiceName
choiceName = "choice"
```

Código 2.9: Definicion de las elecciones.

Sin abreviaciones ni la utilización de Haskell, se obtiene el siguiente contrato escrito en "Marlowe puro":

```
When [
  (Case
     (Deposit
        "alice" "alice" ada
        (Constant 450))
     (When [
           (Case
              (Choice
                 (ChoiceId "choice" "alice") [
                 (Bound 0 1)])
              (When [
                 (Case
                    (Choice
```

```
                        (ChoiceId "choice" "bob") [
                        (Bound 0 1)])
                    (If
                        (ValueEQ
                            (ChoiceValue
                                (ChoiceId "choice" "alice"))
                            (ChoiceValue
                                (ChoiceId "choice" "bob")))
                        (If
                            (ValueEQ
                                (ChoiceValue
                                    (ChoiceId "choice" "alice"))
                                (Constant 0))
                            (Pay
                                "alice"
                                (Party "bob") ada
                                (Constant 450) Close) Close)
                        (When [
                                (Case
                                    (Choice
                                        (ChoiceId "choice" "carol") [
                                        (Bound 1 1)]) Close)
                                ,
                                (Case
                                    (Choice
                                        (ChoiceId "choice" "carol") [
                                        (Bound 0 0)])
                                    (Pay
                                        "alice"
                                        (Party "bob") ada
                                        (Constant 450) Close))] 100 Close)))]
    60
                (When [
                        (Case
                            (Choice
                                (ChoiceId "choice" "carol") [
                                (Bound 1 1)]) Close)
                        ,
                        (Case
                            (Choice
                                (ChoiceId "choice" "carol") [
                                (Bound 0 0)])
                            (Pay
                                "alice"
                                (Party "bob") ada
                                (Constant 450) Close))] 100 Close)))
      ]
```

Código 2.10: Contrato *Escrow* completamente expandido.

2.4. El estándar ACTUS

Los contratos financieros son acuerdos legales entre dos (o más) partes sobre el futuro intercambio de dinero. Dichos acuerdos legales se definen sin ambigüedades por medio de un conjunto de términos y lógica contractual. Como resultado, los mismos pueden describirse matemáticamente y representarse digitalmente como algoritmos. Los beneficios de representar contratos financieros de esta forma son múltiples; Tradicionalmente, el procesamiento de transacciones ha sido un campo en el que se pueden lograr mejoras de eficiencia mediante la automatización de contratos.

Adicionalmente, el análisis financiero (por naturaleza del dominio) se basa en la disponibilidad de representaciones computables de estos acuerdos, donde a menudo se utilizan aproximaciones analíticas. Recientemente, el auge de las blockchain, de contabilidad distribuida y los diversos casos de uso de los contratos inteligentes han abierto nuevas posibilidades para los contratos financieros digitales.

En general, el intercambio de flujos de efectivo entre partes sigue ciertos patrones. Un patrón típico es un contrato de préstamo de tipo *bullet*, donde un monto de dinero inicial se entrega, a cambio de pagos de intereses cíclicos y la devolución del dinero inicial en el vencimiento del contrato. Si bien los pagos son fijos, existen muchas variantes que determinan cómo se programan y/o pagan los pagos de intereses cíclicos. Por ejemplo, los pagos de intereses pueden ser mensuales, anuales, mediante períodos arbitrarios. Pueden además ser de tasa fija o variable, pueden usarse diferentes métodos de cálculo de fracciones anuales o puede que no haya ningún interés.

Otro patrón popular es el de amortización de préstamos, en el que, a diferencia de los préstamos *bullet*, el dinero inicial prestado puede devolverse en porciones de monto fijo o variable, y de acuerdo con cronogramas cíclicos o personalizados. Otros tipos de contratos financieros a mencionar incluyen, acciones, contratos a plazo, opciones, swaps, mejoras crediticias, acuerdos de compra, titularización, etc.

Al centrarse en las principales características distintivas, ACTUS describe la gran mayoría de todos los contratos financieros con un conjunto de alrededor de 32 patrones generales de flujo de efectivo, también conocidos como 'tipos de contrato'.

La taxonomía ACTUS [ACTUS, 2019e] proporciona un sistema de clasificación que organiza los contratos financieros según sus patrones distintivos de flujo de dinero. Aparte de este sistema de clasificación, la taxonomía también incluye una descripción de los instrumentos del mundo real cubiertos por cada contrato.

Por otro lado, los acuerdos legales en los contratos financieros representan una lógica puramente determinista. Es decir, un contrato financiero define un conjunto fijo de reglas y condiciones bajo las cuales, dado cualquier conjunto de variables externas, las obligaciones de flujo de efectivo pueden determinarse sin ambigüedades. Por ejemplo, en un préstamo de tasa fija, las obligaciones de flujo de efectivo se definen explícitamente.

Las propiedades de los contratos financieros descritos anteriormente sientan las bases para una descripción algorítmica estandarizada y determinista de las obligaciones de flujo de dinero que surgen de tales acuerdos. Por lo tanto, esta descripción es agnóstica de la tecnología y es compatible con todos los casos de uso necesarios para que este mismo estándar se utilice en todas las funciones financieras. Entre estas se podrían mencionar: fijación de precios, creación de acuerdos, procesamiento de transacciones, así como el análisis en general, proyecciones de liquidez, valoración, cálculos y proyecciones de pérdidas y ganancias, y medición y agregación de riesgos, etc.

Adicionalmente, este estándar crea una base formidable para las máquinas de estados financieras y los *smart contracts*. En la documentación técnica [ACTUS, 2018] es posible encontrar la descripción matemática de los contratos financieros.

2.4.1. Notación ACTUS

Antes de adentrarnos en la especificación de un contrato, es necesario poder entender algunos aspectos de la notación del mismo:

- **Atributos de contrato**: Representan los términos contractuales que definen el flujo de dinero en un contrato financiero. Estos atributos están definidos en [ACTUS, 2019c].
- **Starting date**: t_0 representa la fecha de comienzo del contrato, y marca el instante en el cual las condiciones y estado del contrato esta siendo representado. En general, partiendo desde la lógica contractual, se podrán determinar los eventos del contrato y el estado para todo $t > t_0$, pero no para $s < t_0$
- **Variables de estado**: Las variables de estado describen el estado de un contrato, para un tiempo determinado de su ciclo de vida. Algunos ejemplos de las mismas son: *Notional Principal*, *Nominal Interest Rate*, o *Contract Performance*.

  El diccionario de ACTUS [ACTUS, 2019b] define todas las variables de estado y provee información adicional sobre el tipo de dato esperado por cada una, el formato, etc.

  En general, el 'estado' representa ciertos términos de un contrato que pueden cambiar a lo largo de su ciclo de ejecución, de acuerdo a eventos programados o no programados. Las variables están escritas en su forma abreviada con la primera letra en mayúscula, en negrita e indexadas mediante el tiempo.
- **Eventos**: Un evento de contrato (o simplemente evento) e_t^k se refiere a cualquier evento programado o no programado en un momento determinado t y de un tipo determinado k.

  Los eventos del contrato marcan puntos específicos en el tiempo (durante la ejecución del mismo) en el que se intercambian flujos flujo de efectivo o se actualizan los estados del contrato. El diccionario de eventos [ACTUS, 2019a] enumera y describe todos los tipos de eventos k definidos por el estándar ACTUS.

- **Funciones de transición de estado**: Dichas funciones, conocidas en Inglés como *'State Transition Functions'* (STF) definen la transición de las variables de estado desde el *pre-evento* hacia el *post-evento*, cuando un cierto evento e_t^k ocurre. Esto provoca que el *pre-evento* y *post-evento* reciban la notación de t^- y t^+ respectivamente.

  Estas funciones son específicas para un tipo de evento y contrato. Las mismas son escritas de acuerdo al siguiente formato **STF_[event type]_[contract type] ()**, donde [event type] y [contract type] hacen alusión al tipo de evento y contrato al cual la STF pertenece.

  Por ejemplo: La STF para un evento de tipo IP en el contrato PAM se escribe como STF_IP_PAM () y modifica (entre otras) a la variable **Ipac** desde el pre-evento $\mathbf{Ipac}_{t^-}$ al post-evento $\mathbf{Ipac}_{t^+}$.

- **Funciones de pago**: Las funciones de pago, o Payoff Functions (POF) definen como el flujo de dinero $c \in \mathbb{R}$ ocurre para un determinado evento e_t^k. El mismo es obtenido del estado actual y los términos del contrato. Si fuera necesario, el flujo de dinero puede ser indexado con el tiempo del evento: c_t.

  Las funciones de pago (de forma análoga a las STF), son específicas para un tipo de evento y contrato, y su notación es la siguiente: **POF_[event type]_[contract type] ()**, donde [event type] y [contract type] hacen alusión al tipo de evento y contrato al cual la POF pertenece.

  Por ejemplo: La POF para un evento de tipo IP e_t^{IP} en el contrato PAM se escribe como POF_IP_PAM ().

- **Fechas/Tiempo**: Sin adentrarnos demasiado en particularidades, cabe aclarar que ACTUS utiliza el formato de fechas ISO 8601. Por lo tanto, las fechas son usualmente expresadas en el siguiente formato: [YYYY]-[MM]-[DD]T[hh]:[mm]:[ss]. El formato no soporta husos horarios.

- **Secuencia de eventos**: Los eventos (de diferentes tipos) de un contrato pueden ocurrir en el mismo instante de tiempo t. En este caso, la secuencia de evaluación de su STF y POF es crucial para los flujos de efectivo resultantes y las transiciones de estado. Por lo tanto, se utiliza un indicador de secuencia de eventos que se puede encontrar para cada evento en el diccionario de eventos. Este implica el orden de ejecución de diferentes eventos en el mismo tiempo t.

- **Lifetime de contrato**: La vida útil de un contrato ACTUS es el período de tiempo de su existencia, desde la perspectiva del usuario que analiza. Para cada punto en el tiempo durante su vida, se puede analizar un contrato ACTUS en términos de estado actual y flujos de efectivo futuros.

2.4.2. Un contrato de ejemplo

En esta sección, recorreremos brevemente la especificación técnica ofrecida por [ACTUS, 2018].

En particular, nos centraremos en un tipo de contrato llamado *Principal at Maturity* (PAM). El propósito del contrato PAM puede ser resumido en el siguiente párrafo:

'Se efectuará un pago del valor total en la fecha de intercambio inicial (simbolizada con la variable de contrato IED) y es reembolsado en la fecha de vencimiento (MD). Dependiendo de las variables de contrato, podrían aplicarse tarifas fijas o variables.'

Al describir el contrato, la especificación técnica separa al mismo en tres tablas. Las mismas expresan de forma declarativa, que acción debe ocurrir ante determinado evento.

Las filas de las tablas representan los diferentes tipos de eventos que el contrato tolera, y en las columnas se encuentra la acción correspondiente, junto con comentarios apropiados:

- **Contract Schedule (Cronograma del contrato)**: Contiene información acerca de los eventos programados para dicho contrato. En general, se realiza la asignación a las variables de estado correspondiente. Dichas variables reciben fechas o vectores de fechas (en caso de que el tipo de evento pueda ocurrir en múltiples instantes de la vida del contrato).

  Por ejemplo, para el evento de *monitoring* (AD), un contrato podría definir $\vec{t}^{AD} = (t_0, t_1, \ldots, t_n)$, siendo $t_1, \ldots, t_n$ tiempos definidos por el usuario.

- **State Variables Initialization (Inicialización de variables de estado)**: Esta tabla contiene información acerca del estado inicial de las variables del contrato. Muchas variables son simplemente extraídas de los términos del contrato, mientas que otras tienen estructuras condicionales en su definición.

  Dichas variables serán luego utilizadas para definir pagos y funciones de transición de estado.

- **State Transition Functions and Payoff Functions (Funciones de transición de estado y de pago)**: Esta tabla reúne las funciones de transición y de pago correspondientes a un contrato, para cada tipo de evento.

A continuación, se muestran fragmentos de las 3 tablas para el contrato, extraídos de la especificación [ACTUS, 2018]:

PAM: Contract Schedule

Event	Schedule	Comments
AD	$\vec{t}^{AD} = (t_0, t_1, \ldots, t_n)$	With $t_i, i = 1, 2, \ldots$ a custom input
IED	$t^{IED} =$ `IED`	
MD	$t^{MD} = \mathbf{Md}_{t_0}$	

Continua en la siguiente página

Continua desde la página anterior

Event	Schedule	Comments
PP	$\vec{t}^{PP} = \begin{cases} \varnothing & \text{if} \quad \texttt{PPEF} = \text{'N'} \\ (\vec{u}, \vec{v}) & \text{else} \end{cases}$ where $\vec{u} = S(s, \texttt{OPCL}, T^{MD})$ $\vec{v} = O^{ev}(\texttt{CID}, \texttt{PP}, t)$	with $s = \begin{cases} \varnothing & \text{if} \quad \texttt{OPANX} = \varnothing \wedge \texttt{OPCL} = \varnothing \\ \texttt{IED} + \texttt{OPCL} & \text{else if} \quad \texttt{OPANX} = \varnothing \\ \texttt{OPANX} & \text{else} \end{cases}$
PY	$\vec{t}^{PY} = \begin{cases} \varnothing & \text{if} \quad \texttt{PYTP} = \text{'O'} \\ \vec{t}^{PP} & \text{else} \end{cases}$	
FP	$\vec{t}^{FP} = \begin{cases} \varnothing & \text{if} \quad \texttt{FER} = \varnothing \vee \texttt{FER} = 0 \\ S(s, \texttt{FECL}, T^{MD}) & \text{else} \end{cases}$	with $s = \begin{cases} \varnothing & \text{if} \quad \texttt{FEANX} = \varnothing \wedge \texttt{FECL} = \varnothing \\ \texttt{IED} + \texttt{FECL} & \text{else if} \quad \texttt{FEANX} = \varnothing \\ \texttt{FEANX} & \text{else} \end{cases}$
PRD	$t^{PRD} = \texttt{PRD}$	
TD	$t^{TD} = \texttt{TD}$	
IP	$\vec{t}^{IP} = \begin{cases} \varnothing & \text{if} \quad \texttt{IPNR} = \varnothing \\ S(s, \texttt{IPCL}, T^{MD}) & \text{else} \end{cases}$	with $s = \begin{cases} \varnothing & \text{if} \quad \texttt{IPANX} = \varnothing \wedge \texttt{IPCL} = \varnothing \\ \texttt{IPCED} & \text{else if} \quad \texttt{IPCED} \neq \varnothing \\ \texttt{IED} + \texttt{IPCL} & \text{else if} \quad \texttt{IPANX} = \varnothing \\ \texttt{IPANX} & \text{else} \end{cases}$
IPCI	$\vec{t}^{IPCI} = \begin{cases} \varnothing & \text{if} \quad \texttt{IPCED} = \varnothing \\ S(s, \texttt{IPCL}, \texttt{IPCED}) & \text{else} \end{cases}$	with $s = \begin{cases} \varnothing & \text{if} \quad \texttt{IPANX} = \varnothing \wedge \texttt{IPCL} = \varnothing \\ \texttt{IED} + \texttt{IPCL} & \text{else if} \quad \texttt{IPANX} = \varnothing \\ \texttt{IPANX} & \text{else} \end{cases}$
RR	$\vec{t}^{RR} = \begin{cases} \varnothing & \text{if} \quad \texttt{RRANX} = \varnothing \wedge \texttt{RRCL} = \varnothing \\ \vec{t} \setminus t^{RRY} & \text{else if} \texttt{RRNXT} \neq \varnothing \\ \vec{t} & \text{else} \end{cases}$ where $\vec{t} = S(s, \texttt{RRCL}, T^{MD})$	with $s = \begin{cases} \texttt{IED} + \texttt{RRCL} & \text{if} \quad \texttt{RRANX} = \varnothing \\ \texttt{RRANX} & \text{else} \end{cases}$ $t^{RRY} = \text{ínf}\, t \in \vec{t} \mid t > \texttt{SD}$
RRF	$t^{RRF} = \begin{cases} \varnothing & \text{if} \quad \texttt{RRANX} = \varnothing \wedge \texttt{RRCL} = \varnothing \\ \text{ínf}\, t \in \vec{t} \mid t > \texttt{SD} & \text{else} \end{cases}$ where $\vec{t} = S(s, \texttt{RRCL}, T^{MD})$	with $s = \begin{cases} \texttt{IED} + \texttt{RRCL} & \text{if} \quad \texttt{RRANX} = \varnothing \\ \texttt{RRANX} & \text{else} \end{cases}$
SC	$\vec{t}^{SC} = \begin{cases} \varnothing & \text{if} \quad \texttt{SCEF} = \text{'000'} \\ S(s, \texttt{SCCL}, T^{MD}) & \text{else} \end{cases}$	with $s = \begin{cases} \varnothing & \text{if} \quad \texttt{SCANX} = \varnothing \wedge \texttt{SCCL} = \varnothing \\ \texttt{IED} + \texttt{SCCL} & \text{else if} \quad \texttt{SCANX} = \varnothing \\ \texttt{SCANX} & \text{else} \end{cases}$
CE	$\vec{t}^{CE} = t(e^k) \vert \mathbf{Prf}_{t^-} \neq \mathbf{Prf}_{t^+}, \forall k$	

PAM: State Variables Initialization

State	Initialization per t_0	Comments
Md	$\mathbf{Md}_{t_0} = \texttt{MD}$	
Nt	$\mathbf{Nt}_{t_0} = \begin{cases} 0{,}0 & \text{if} \quad \texttt{IED} > t_0 \\ R(\texttt{CNTRL}) \times \texttt{NT} & \text{else} \end{cases}$	
Ipnr	$\mathbf{Ipnr}_{t_0} = \begin{cases} 0{,}0 & \text{if} \quad \texttt{IED} > t_0 \\ \texttt{IPNR} & \text{else} \end{cases}$	
Ipac	$\mathbf{Ipac}_{t_0} = \begin{cases} 0{,}0 & \text{if} \quad \texttt{IPNR} = \varnothing \\ \texttt{IPAC} & \text{else if} \quad \texttt{IPAC} \neq \varnothing \\ Y(t^-, t_0) \times \mathbf{Nt}_{t_0} \times \mathbf{Ipnr}_{t_0} & \text{else} \end{cases}$	with $t^- = \sup t \in \vec{t}^{IP} \mid t < t_0$

Continua en la siguiente página

Continua desde la página anterior

State	Initialization per t_0	Comments
Feac	$\mathbf{Feac}_{t_0} = \begin{cases} 0{,}0 & \text{if } \texttt{FER} = \varnothing \\ \texttt{FEAC} & \text{else if } \texttt{FEAC} \neq \varnothing \\ Y(t^{FP-}, t_0) \times \mathbf{Nt}_{t_0} \times \texttt{FER} & \text{else if } \texttt{FEB} = \text{'N'} \\ \frac{Y(t^{FP-}, t_0)}{Y(t^{FP-}, t^{FP+})} \times \texttt{FER} & \text{else} \end{cases}$	with $t^{FP-} = \sup t \in \bar{t}^{FP} \mid t < t_0$, $t^{FP+} = \text{ínf}\, t \in \bar{t}^{FP} \mid t > t_0$
Nsc	$\mathbf{Nsc}_{t_0} = \begin{cases} \texttt{SCIXSD} & \text{if } \texttt{SCEF} = \text{'[x]N[x]'} \\ 1{,}0 & \text{else} \end{cases}$	
Isc	$\mathbf{Isc}_{t_0} = \begin{cases} \texttt{SCIXSD} & \text{if } \texttt{SCEF} = \text{'I[x][x]'} \\ 1{,}0 & \text{else} \end{cases}$	
Prf	$\mathbf{Prf}_{t_0} = \texttt{PRF}$	
Sd	$\mathbf{Sd}_{t_0} = t_0$	

PAM: State Transition Functions and Payoff Functions

Event	Payoff Function	State Transition Function
AD	0.0	$\mathbf{Ipac}_{t^+} = \mathbf{Ipac}_{t^-} + Y(\mathbf{Sd}_{t^-1}, t)\mathbf{Ipnr}_{t^-}\mathbf{Nt}_{t^-}$ $\mathbf{Sd}_{t^+} = t$
IED	$X^{\texttt{CURS}}_{\texttt{CUR}}(t)R(\texttt{CNTRL})(-1)(\texttt{NT} + \texttt{PDIED})$	$\mathbf{Nt}_{t^+} = R(\texttt{CNTRL})\texttt{NT}$ $\mathbf{Ipnr}_{t^+} = \begin{cases} 0{,}0 & \text{if } \texttt{IPNR} = \varnothing \\ \texttt{IPNR} & \text{else} \end{cases}$ $\mathbf{Ipac}_{t^+} = \begin{cases} \texttt{IPAC} & \text{if } \texttt{IPAC} \neq \varnothing \\ y\mathbf{Nt}_{t^+}\mathbf{Ipnr}_{t^+} & \text{if } \texttt{IPANX} \neq \varnothing \wedge \texttt{IPANX} < t \\ 0{,}0 & \text{else} \end{cases}$ $\mathbf{Sd}_{t^+} = t$ with $y = Y(\texttt{IPANX}, t)$
MD	$X^{\texttt{CURS}}_{\texttt{CUR}}(t)(\mathbf{Nsc}_{t^-}\mathbf{Nt}_{t^-} + \mathbf{Isc}_{t^-}\mathbf{Ipac}_{t^-} + \mathbf{Feac}_{t^-})$	$\mathbf{Nt}_{t^+} = 0{,}0$ $\mathbf{Ipac}_{t^+} = 0{,}0$ $\mathbf{Feac}_{t^+} = 0{,}0$ $\mathbf{Sd}_{t^+} = t$
PP	$X^{\texttt{CURS}}_{\texttt{CUR}}(t)f(O^{ev}(\texttt{CID}, \texttt{PP}, t))$	$\mathbf{Ipac}_{t^+} = \mathbf{Ipac}_{t^-} + Y(\mathbf{Sd}_{t^-}, t)\mathbf{Ipnr}_{t^-}\mathbf{Nt}_{t^-}$ $\mathbf{Feac}_{t^+} = \begin{cases} \mathbf{Feac}_{t^-} + Y(\mathbf{Sd}_{t^-}, t)\mathbf{Nt}_{t^-}\texttt{FER} & \text{if } \texttt{FEB} = \text{'N'} \\ \frac{Y(t^{FP-}, t)}{Y(t^{FP-}, t^{FP+})}R(\texttt{CNTRL})\texttt{FER} & \text{else} \end{cases}$ $\mathbf{Nt}_{t^+} = \mathbf{Nt}_{t^-} - f(O^{ev}(\texttt{CID}, \texttt{PP}, t))$ $\mathbf{Sd}_{t^+} = t$ with $t^{FP-} = \sup t \in \bar{t}^{FP} \mid t < t_0$ $t^{FP+} = \text{ínf}\, t \in \bar{t}^{FP} \mid t > t_0$
PY	$X^{\texttt{CURS}}_{\texttt{CUR}}(t)R(\texttt{CNTRL})\texttt{PYRT}$ if $\texttt{PYTP} = \text{'A'}$ $c\texttt{PYRT}$ if $\texttt{PYTP} = \text{'N'}$ $c\,\text{máx}(0, \mathbf{Ipnr}_{t^-} - O^{rf}(\texttt{RRMO}, t))$ if $\texttt{PYTP} = \text{'I'}$ with $c = X^{\texttt{CURS}}_{\texttt{CUR}}(t)R(\texttt{CNTRL})Y(\mathbf{Sd}_{t^-}, t)\mathbf{Nt}_{t^-}$	$\mathbf{Ipac}_{t^+} = \mathbf{Ipac}_{t^-} + Y(\mathbf{Sd}_{t^-}, t)\mathbf{Ipnr}_{t^-}\mathbf{Nt}_{t^-}$ $\mathbf{Feac}_{t^+} = \begin{cases} \mathbf{Feac}_{t^-} + Y(\mathbf{Sd}_{t^-}, t)\mathbf{Nt}_{t^-}\texttt{FER} & \text{if } \texttt{FEB} = \text{'N'} \\ \frac{Y(t^{FP-}, t)}{Y(t^{FP-}, t^{FP+})}R(\texttt{CNTRL})\texttt{FER} & \text{else} \end{cases}$ $\mathbf{Sd}_{t^+} = t$ with $t^{FP-} = \sup t \in \bar{t}^{FP} \mid t < t_0$ $t^{FP+} = \text{ínf}\, t \in \bar{t}^{FP} \mid t > t_0$
FP	$R(\texttt{CNTRL})c$ if $\texttt{FEB} = \text{'A'}$ $cY(\mathbf{Sd}_{t^-}, t)\mathbf{Nt}_{t^-} + \mathbf{Feac}_{t^-}$ if $\texttt{FEB} = \text{'N'}$ with $c = X^{\texttt{CURS}}_{\texttt{CUR}}(t)\texttt{FER}$	$\mathbf{Ipac}_{t^+} = \mathbf{Ipac}_{t^-} + Y(\mathbf{Sd}_{t^-}, t)\mathbf{Ipnr}_{t^-}\mathbf{Nt}_{t^-}$ $\mathbf{Feac}_{t^+} = 0{,}0$ $\mathbf{Sd}_{t^+} = t$

Continua en la siguiente página

Continua desde la página anterior

Event	Payoff Function	State Transition Function
PRD	$X^{\texttt{CURS}}_{\texttt{CUR}}(t)R(\texttt{CNTRL})(-1)(\texttt{PPRD}+\mathbf{Ipac}_{t^-}+Y(\mathbf{Sd}_{t^-},t)\mathbf{Ipnr}_{t^-}\mathbf{Nt}_{t^-})$	$\mathbf{Ipac}_{t^+}=\mathbf{Ipac}_{t^-}+Y(\mathbf{Sd}_{t^-},t)\mathbf{Ipnr}_{t^-}\mathbf{Nt}_{t^-}$ $\mathbf{Feac}_{t^+}=\begin{cases}\mathbf{Feac}_{t^-}+Y(\mathbf{Sd}_{t^-},t)\mathbf{Nt}_{t^-}\texttt{FER} & \text{if } \texttt{FEB}=\text{'N'}\\ \frac{Y(t^{FP-},t)}{Y(t^{FP-},t^{FP+})}R(\texttt{CNTRL})\texttt{FER} & \text{else}\end{cases}$ $\mathbf{Sd}_{t^+}=t$ with $t^{FP-}=\sup t\in\vec{t}^{FP}\mid t<t_0$ $t^{FP+}=\text{ínf}\, t\in\vec{t}^{FP}\mid t>t_0$
TD	$X^{\texttt{CURS}}_{\texttt{CUR}}(t)R(\texttt{CNTRL})(\texttt{PTD}+\mathbf{Ipac}_{t^-}+Y(\mathbf{Sd}_{t^-},t)\mathbf{Ipnr}_{t^-}\mathbf{Nt}_{t^-})$	$\mathbf{Nt}_{t^+}=0{,}0$ $\mathbf{Ipac}_{t^+}=0{,}0$ $\mathbf{Feac}_{t^+}=0{,}0$ $\mathbf{Ipnr}_{t^+}=0{,}0$ $\mathbf{Sd}_{t^+}=t$
IP	$X^{\texttt{CURS}}_{\texttt{CUR}}(t)\mathbf{Isc}_{t^-}(\mathbf{Ipac}_{t^-}+Y(\mathbf{Sd}_{t^-},t)\mathbf{Ipnr}_{t^-}\mathbf{Nt}_{t^-})$	$\mathbf{Ipac}_{t^+}=0{,}0$ $\mathbf{Feac}_{t^+}=\begin{cases}\mathbf{Feac}_{t^-}+Y(\mathbf{Sd}_{t^-},t)\mathbf{Nt}_{t^-}\texttt{FER} & \text{if } \texttt{FEB}=\text{'N'}\\ \frac{Y(t^{FP-},t)}{Y(t^{FP-},t^{FP+})}R(\texttt{CNTRL})\texttt{FER} & \text{else}\end{cases}$ $\mathbf{Sd}_{t^+}=t$ with $t^{FP-}=\sup t\in\vec{t}^{FP}\mid t<t_0$ $t^{FP+}=\text{ínf}\, t\in\vec{t}^{FP}\mid t>t_0$
IPCI	0.0	$\mathbf{Nt}_{t^+}=\mathbf{Nt}_{t^-}+\mathbf{Ipac}_{t^-}+Y(\mathbf{Sd}_{t^-},t)\mathbf{Nt}_{t^-}\mathbf{Ipnr}_{t^-}$ $\mathbf{Ipac}_{t^+}=0{,}0$ $\mathbf{Feac}_{t^+}=\begin{cases}\mathbf{Feac}_{t^-}+Y(\mathbf{Sd}_{t^-},t)\mathbf{Nt}_{t^-}\texttt{FER} & \text{if } \texttt{FEB}=\text{'N'}\\ \frac{Y(t^{FP-},t)}{Y(t^{FP-},t^{FP+})}R(\texttt{CNTRL})\texttt{FER} & \text{else}\end{cases}$ $\mathbf{Sd}_{t^+}=t$ with $t^{FP-}=\sup t\in\vec{t}^{FP}\mid t<t_0$ $t^{FP+}=\text{ínf}\, t\in\vec{t}^{FP}\mid t>t_0$
RR	0.0	$\mathbf{Ipac}_{t^+}=\mathbf{Ipac}_{t^-}+Y(\mathbf{Sd}_{t^-},t)\mathbf{Ipnr}_{t^-}\mathbf{Nt}_{t^-}$ $\mathbf{Feac}_{t^+}=\begin{cases}\mathbf{Feac}_{t^-}+Y(\mathbf{Sd}_{t^-},t)\mathbf{Nt}_{t^-}\texttt{FER} & \text{if } \texttt{FEB}=\text{'N'}\\ \frac{Y(t^{FP-},t)}{Y(t^{FP-},t^{FP+})}R(\texttt{CNTRL})\texttt{FER} & \text{else}\end{cases}$ $\mathbf{Ipnr}_{t^+}=\text{mín}(\text{máx}(\mathbf{Ipnr}_{t^-}+\Delta r,\texttt{RRLF}),\texttt{RRLC})$ $\mathbf{Sd}_{t^+}=t$ with $\Delta r=\text{mín}(\text{máx}(O^{rf}(\texttt{RRMO},t)\texttt{RRMLT}+\texttt{RRSP}-\mathbf{Ipnr}_{t^-},\texttt{RRPF}),\texttt{RRPC})$ $t^{FP-}=\sup t\in\vec{t}^{FP}\mid t<t_0$ $t^{FP+}=\text{ínf}\, t\in\vec{t}^{FP}\mid t>t_0$
RRF	0.0	$\mathbf{Ipac}_{t^+}=\mathbf{Ipac}_{t^-}+Y(\mathbf{Sd}_{t^-},t)\mathbf{Ipnr}_{t^-}\mathbf{Nt}_{t^-}$ $\mathbf{Feac}_{t^+}=\begin{cases}\mathbf{Feac}_{t^-}+Y(\mathbf{Sd}_{t^-},t)\mathbf{Nt}_{t^-}\texttt{FER} & \text{if } \texttt{FEB}=\text{'N'}\\ \frac{Y(t^{FP-},t)}{Y(t^{FP-},t^{FP+})}R(\texttt{CNTRL})\texttt{FER} & \text{else}\end{cases}$ $\mathbf{Ipnr}_{t^+}=\texttt{RRNXT}$ $\mathbf{Sd}_{t^+}=t$ with $t^{FP-}=\sup t\in\vec{t}^{FP}\mid t<t_0$ $t^{FP+}=\text{ínf}\, t\in\vec{t}^{FP}\mid t>t_0$

Continua en la siguiente página

Continua desde la página anterior

Event	**Payoff Function**	**State Transition Function**
SC	0.0	$\mathbf{Ipac}_{t^+} = \mathbf{Ipac}_{t^-} + Y(\mathbf{Sd}_{t^-}, t)\mathbf{Ipnr}_{t^-}\mathbf{Nt}_{t^-}$ $\mathbf{Feac}_{t^+} = \begin{cases} \mathbf{Feac}_{t^-} + Y(\mathbf{Sd}_{t^-}, t)\mathbf{Nt}_{t^-}\texttt{FER} & \text{if} \quad \texttt{FEB} = \text{'N'} \\ \frac{Y(t^{FP-}, t)}{Y(t^{FP-}, t^{FP+})} R(\texttt{CNTRL})\texttt{FER} & \text{else} \end{cases}$ $\mathbf{Nsc}_{t^+} = \begin{cases} \mathbf{Nsc}_{t^-} & \text{if} \quad \texttt{SCEF} = [x]0[x] \\ \frac{O^{rf}(\texttt{SCMO}, t) - \texttt{SCIED}}{\texttt{SCIED}} & \text{else} \end{cases}$ $\mathbf{Isc}_{t^+} = \begin{cases} \mathbf{Isc}_{t^-} & \text{if} \quad \texttt{SCEF} = 0[x][x] \\ \frac{O^{rf}(\texttt{SCMO}, t) - \texttt{SCIED}}{\texttt{SCIED}} & \text{else} \end{cases}$ $\mathbf{Sd}_{t^+} = t$ with $t^{FP-} = \sup t \in \vec{t}^{FP} \mid t < t_0$ $t^{FP+} = \text{ínf}\, t \in \vec{t}^{FP} \mid t > t_0$
CE	0.0	STF_AD_PAM()

Capítulo 3

Verificación de programas

En este capítulo abordaremos los conceptos teóricos en los cuales se basan los distintos sistemas automáticos de demostración de teoremas.

En particular, nos centraremos en Isabelle, uno de los más reconocidos y utilizados en la industria.

3.1. Concepto general, herramientas y metodologías

Los asistentes de pruebas formales son herramientas de software diseñadas para ayudar a sus usuarios a realizar pruebas, especialmente en cálculo lógico. Por lo general, los llamamos asistentes de demostración o demostradores interactivos de teoremas.

Pruebas rigurosas y formales

La prueba interactiva de teoremas tiene su propia terminología, comenzando con la noción de 'prueba'. Una prueba formal es un argumento lógico expresado dentro un formalismo lógico. En este contexto, 'formal' significa 'lógico' o 'basado en la lógica'. Los matemáticos realizaron pruebas formales en papel décadas antes de la llegada de las computadoras, pero hoy en día las pruebas formales se llevan a cabo utilizando un asistente de prueba.

Por el contrario, una prueba informal es lo que un matemático normalmente llamaría una prueba. A menudo se llevan a cabo en LaTeX o en una pizarra. El nivel de detalle puede variar mucho, y frases como 'es obvio que', 'claramente' y 'sin pérdida de generalidad' delegan parte de la carga de la prueba al lector. Una prueba rigurosa es una prueba informal muy detallada.

La principal fortaleza de los asistentes de prueba es que ayudan a desarrollar pruebas altamente confiables e inequívocas de enunciados matemáticos, usando lógica precisa. Se pueden usar para probar resultados arbitrariamente avanzados, y no solo ejemplos simples.

Las pruebas formales también ayudan a los estudiantes a comprender lo que constituye una definición válida o una prueba válida.

Cuando desarrollamos una nueva teoría, las pruebas formales pueden ayudarnos a explorarla. Son útiles cuando generalizamos o modificamos una prueba existente, de la misma manera que un compilador nos ayuda a desarrollar programas correctos. Brindan un alto nivel de confiabilidad que facilita que otros revisen la prueba. Además, las pruebas formales pueden formar la base de herramientas computacionales verificadas (por ejemplo, sistemas de álgebra computacional verificados).

La mayoría de los usuarios de asistentes de pruebas en la actualidad provienen de las ciencias de la Computación. Algunas empresas, incluidas AMD [Russinoff, 1999] e Intel [Harrison, 2003], han utilizado asistentes de prueba para verificar sus diseños.

3.1.1. Algunos asistentes de pruebas

Hay una gran cantidad de asistentes de prueba en desarrollo o uso alrededor del mundo. A continuación presentamos una lista de los principales, clasificados por sus fundamentos lógicos:

- **Teoría de conjuntos**: Isabelle/ZF, Metamath, Mizar
- **Teoría simple de tipos**: HOL4, HOL Light, Isabelle/HOL
- **Teoría dependiente de tipos**: Agda, Coq, Lean, Matita, PVS
- **Lógica de primer orden, de tipo Lisp**: ACL2

Para una historia de los asistentes de demostración y la demostración interactiva de teoremas, es altamente recomendable referirse a [Harrison et al., 2014].

3.2. Isabelle

Isabelle es un sistema genérico para implementar formalismos lógicos, e Isabelle/HOL es la especialización de Isabelle para HOL, abreviatura de *Higher-Order Logic*.

En resumidas palabras, HOL puede ser definido como:

$$\boxed{\text{HOL} = \text{Programación Funcional} + \text{Lógica}}$$

Isabelle permite expresar fórmulas matemáticas en un lenguaje formal y proporciona herramientas para probar dichas fórmulas. Entre otras, las aplicaciones principales son la formalización de pruebas matemáticas y, en particular, la verificación formal.

Isabelle/HOL acuñó su éxito debido a su facilidad de uso y potente automatización. Gran parte de la misma la realizan herramientas externas: el meta-probador *Sledgehammer* se basa en probadores de resolución y en el solver SMT para su búsqueda de pruebas, el generador

de contra-ejemplos *Quickcheck* usa el compilador ML como un rápido evaluador para fórmulas básicas. Junto con el formato de prueba estructurada de Isar y una nueva interfaz de usuario asincrónica, estas herramientas han transformado radicalmente la experiencia del usuario de Isabelle.

3.2.1. Programando y probando en Isabelle/HOL

Esta sección introduce a HOL como un lenguaje de programación funcional y muestra como probar propiedades de programas funcionales mediante inducción [Nipkow, 2013].

Definiciones básicas

Tipos, términos y fórmulas: HOL es considerado una lógica tipada, cuyo sistema de tipos se asemeja al de los lenguajes de programación funcional. Esto implica la existencia de:

- **Tipos base**: Los cuales están conformados por `bool`, el tipo de valor de verdad, `nat`, el tipo de los números naturales ($\mathbb{N}$), e `int`, el tipo de los números enteros ($\mathbb{Z}$).
- **Constructores de tipo**: En particular `list`, el tipo de listas y `set`, el tipo de conjuntos. Los constructores de tipos se escriben como posfijos, es decir, después de sus argumentos. Por ejemplo, `nat list` es el tipo de listas cuyos elementos son números naturales.
- **Funciones**: Denotadas con $\Rightarrow$.
- **Variables de tipo**: Denotadas por $'a$, $'b$, etc., como en ML [Ullman, 1994].

Los **términos** se forman como en la programación funcional, aplicando funciones a los argumentos. Si f es una función de tipo $\tau_1 \Rightarrow \tau_2$ y t es un término de tipo τ_1 entonces $f\ t$ es un término de tipo τ_2. Escribimos $t :: \tau$ para indicar que el término t tiene tipo τ. Los términos abarcan también abstracciones lambda. Por ejemplo $\lambda x.\ x$ es la función identidad.

Las **fórmulas** son términos de tipo *bool*. Entre ellas se encuentran las constantes básicas `True` y `False` y los conectores lógicos usuales (en orden decreciente de precedencia): $\neg$, $\vee$, $\wedge$, $\rightarrow$.

La **igualdad** está disponible en la forma de la función infijo $=$, del tipo $'a \Rightarrow' b \Rightarrow bool$. También funciona para formulas, donde significa 'si o solo si'.

Los **cuantificadores** se escriben como $\forall\ x.\ P$ y $\exists\ x.\ P$.

Isabelle automáticamente computa el tipo de cada variable en un término. Esto se conoce popularmente como inferencia de tipos. A pesar de la inferencia de tipos, en ocasiones es necesario agregar el tipo explícito a una variable o término. La sintaxis es $t :: \tau$ como en $m+(n :: nat)$. Las anotaciones explicitas de tipos son especialmente necesarias para desambiguar términos que incluyen funciones sobrecargadas como $+$.

Finalmente nos encontramos con el cuantificador universal $\bigwedge$ y la implicación $\Longrightarrow$. Estos son parte del framework Isabelle, no de HOL.

Teorías

Una teoría es una colección de tipos, funciones y teoremas, similar a un módulo en un lenguaje de programación. Todo el contenido que escribamos en Isabelle debe pertenecer a una teoría. El formato general de una teoría T es:

```
theory T
imports T_1, ··· , T_n
begin
definitions, theorems and proofs
end
```

Código 3.1: Estructura general de un archivo de prueba

donde $T_1 \dots T_n$ son los nombres de teorías existentes en las que T está basado. Se dice que T_i son las teorías padre de T. Todo lo definido en los padres de T (y los padres de sus padres, recursivamente) es visible automáticamente. Cada teoría T debe estar guardada en un archivo llamado `T.thy`.

Ademas de las teorías que incluye por defecto la distribución de Isabelle/HOL[1], se encuentra disponible un archivo de pruebas formales (https://isa-afp.org), el cual está en constante crecimiento y en el que cualquier persona puede contribuir.

Comillas: La definición textual de una teoría sigue una sintaxis especifica incluyendo keywords como **begin** y **datatype**. En esta sintaxis están embebidos los tipos y fórmulas de HOL. Para distinguir los dos niveles, todo lo específico de HOL (términos y tipos) debe estar entre comillas: `"..."`. Las comillas alrededor de un único identificador se pueden eliminar. Cuando Isabelle imprime un mensaje de error de sintaxis, se refiere a la sintaxis HOL como la sintaxis interna y al lenguaje teórico adjunto como la sintaxis externa.

Estado de la prueba: En una instalación habitual de Isabelle[2], con el cursor en un *lemma* o *theorem*, es posible ver en la pestaña *state*, los *subgoals* y estado parcial de nuestra prueba en proceso.

[1] https://isabelle.in.tum.de/library/HOL
[2] https://isabelle.in.tum.de/

```
proof (prove)
goal (2 subgoals):
 1. (case inputList of
     [] ⇒ ApplyAllSuccess (False ∨ False) ([] @ convertReduceWarnings [])
           ([] @ []) sta COM4
     | input # rest ⇒
         case applyInput env sta input COM4 of
         Applied applyWarn newState cont ⇒
           applyAllLoop True env newState cont rest
            ([] @ convertReduceWarnings [] @ convertApplyWarning applyWarn)
            ([] @ [])
         | ApplyNoMatchError ⇒ ApplyAllNoMatchError) =
    ApplyAllSuccess reduced warnings payments newstate cont ⟹
    reduceContractStep env sta COM4 = NotReduced ⟹
    inputList = [] ⟹ minSlot sta ≤ minSlot newstate
 2. ⋀a list.
       (case inputList of
        [] ⇒ ApplyAllSuccess (False ∨ False) ([] @ convertReduceWarnings [])
              ([] @ []) sta COM4
        | input # rest ⇒
            case applyInput env sta input COM4 of
            Applied applyWarn newState cont ⇒
              applyAllLoop True env newState cont rest
               ([] @ convertReduceWarnings [] @ convertApplyWarning applyWarn)
               ([] @ [])
            | ApplyNoMatchError ⇒ ApplyAllNoMatchError) =
       ApplyAllSuccess reduced warnings payments newstate cont ⟹
       reduceContractStep env sta COM4 = NotReduced ⟹
       inputList = a # list ⟹ minSlot sta ≤ minSlot newstate
```

Figura 3.1: Pantalla de *status* para una prueba en proceso de la semántica de Marlowe.

Algunos tipos y pruebas básicas

En esta sección detallaremos los tipos *bool*, *nat* y *list*, los más importantes y utilizados en Isabelle/HOL. Mostremos algunos ejemplos de pruebas sencillas y las probaremos con inducción y simplificación.

bool Este tipo esta predefinido mediante:

```
datatype bool = True | False
```

Código 3.2: Tipo de dato *bool*

El mismo cuenta con los valores `True` y `False` y algunas funciones predefinidas, como: $\neg$, $\vee$, $\wedge$, $\rightarrow$, etc. El siguiente ejemplo muestra como puede definirse la conjunción lógica (conocida también como intersección en teoría de conjuntos) mediante *pattern matching*:

```
fun conj :: "bool ⇒ bool ⇒ bool" where
"conj True True = True" |
"conj _ _ = False"
```

Código 3.3: Definición de la función *conjunción lógica* (también conocida como *and*)

Las definiciones de datatype y funciones siguen la sintaxis de lenguajes de programación funcional.

nat Los números naturales:

```
datatype nat = 0 | Suc nat
```

Código 3.4: Tipo de dato que representa los números naturales.

Todos los valores de tipo `nat` son generados por los constructores `0` y `Suc`. Por lo tanto, los valores de tipo nat son `0`, `Suc 0`, `Suc (Suc 0)`, etc. Algunas de las funciones predefinidas son: $+$, $*$, $\leq$, etc. A continuación, vemos como podríamos definir nuestra propia operación de suma:

```
fun add :: "nat ⇒ nat ⇒ nat" where
"add 0 n = n" |
"add (Suc m) n = Suc(add m n)"
```

Código 3.5: Definición de la suma para dos números naturales.

Y a continuación vemos una prueba de `add m 0 = m`:

```
lemma add_02: "add m 0 = m"
apply(induction m)
apply(auto)
done
```

Código 3.6: Prueba de la neutralidad del cero frente a la suma.

La palabra reservada `lemma` comienza la prueba y le da al lema un nombre, por ejemplo '`add_02`'. Las propiedades definidas recursivamente suelen ser probadas por inducción en la mayoría de

las ocasiones. El commando `apply (induction m)` le indica a Isabelle que debe comenzar una prueba por inducción, en la variable `m`. Como respuesta, va a mostrar el siguiente 'estado de prueba':

```
1. add 0 0 = 0
2. ⋀ m. add m 0 = m ⟹ add (Suc m) 0 = Suc m
```

Código 3.7: Subgoals para la prueba de neutralidad del cero luego de aplicar inducción estructural.

Las lineas numeradas son conocidas como subgoals. El primer subgoal es el caso base, y el segundo es el paso inductivo. El prefijo ⋀ `m.` es la forma en la que Isabelle nos comunica "para un m arbitrario, pero fijo". El símbolo ⟹ separa las suposiciones de la conclusión. El comando `apply (auto)` (mencionado por primera vez en 3.3.2) le indica a Isabelle que intente probar todos los subgoals automáticamente, en este caso simplemente mediante simplificación. Debido a la sencillez de ambos *subgoals*, Isabelle puede manejarlos con las teorías incluidas por defecto.

El caso base `add 0 0 = 0` se prueba por la definición de `add` y el paso inductivo es bastante simple: `add (Suc m) 0 = Suc (add m 0) = Suc m` usando la definición de `add` y la hipótesis inductiva respectivamente. En resumen, ambas 'sub-pruebas' se basan en la simplificación mediante la definición y la hipótesis inductiva. Como resultado de la keyword '`done`', Isabelle asocia el lema a su identificador.

De ahora en más dicho lema podría ser utilizado como paso de prueba por lemas subsiguientes.

Listas Las listas ya están predefinidas por Isabelle[3], pero su definición puede ser simplificada como:

```
datatype 'a list = Nil | Cons 'a "'a list"
```

Código 3.8: Tipo de datos *lista*.

- La lista `'a list` es el conjunto de listas con elementos de tipo `'a`. Debido a que `'a` es una variable de tipo, las listas son de hecho polimórficas: Los elementos de una lista pueden ser de tipo arbitrario (pero todos los elementos deben compartir el tipo).
- Las listas tienen dos constructores: `Nil`, la lista vacía y `Cons`, que agrega un elemento de tipo `'a` en el comienzo de la misma (de tipo `'a list`). Esto provoca que todas las listas sean de la forma `Nil`, `(Cons x Nil)`, `(Cons x (Cons y Nil))`, etc.

[3]Dicha definición puede ser encontrada en https://isabelle.in.tum.de/library/HOL/HOL/List.html

3.3. Métodos de prueba en Isabelle

Isabelle proporciona una serie de métodos de prueba (de propósito general) que realizan la búsqueda de las mismas [Blanchette et al., 2011]. En esta sección, discutiremos los más importantes.

3.3.1. Simplificación

La simplificación es el principal caballo de batalla en Isabelle. El sistema puede reescribir y simplificar fácil y eficientemente expresiones. También tolera *hooks* para personalizar las mismas:

- *Procedimientos de simplificación basados en patrones* que derivan y aplican reglas de reescritura de forma dinámica. Muchos de estos procedimientos están pre-instalados, en particular los de simplificación aritmética para números y términos simbólicos.
- *Solvers especiales para reglas de reescritura condicional.* Los ejemplos típicos son: fragmentos de aritmética lineal y un *prover* de clausuras para relaciones transitivas arbitrarias.
- *Loopers especiales* que advierten el objetivo luego de cada ronda de simplificación. Los separadores de casos se proveen de esta manera.

El poder del simplificador se debe principalmente a la reescritura, junto a la inmensa (y en constante crecimiento) librería de reglas preexistentes en la plataforma.

3.3.2. Auto

Desde el lado del usuario, invocado con la keyword *auto*, podemos mencionar a un método de prueba que intercala la simplificación con una pequeña cantidad de búsqueda dentro del espacio de pruebas.

Es imposible describir sucintamente al método *auto* debido a su naturaleza heurística, de tipo ad-hoc.

Su gran fortaleza es la capacidad para resolver las partes simples de un objetivo y dejar al usuario las más difíciles. Esto ayuda a concentrarse rápidamente en el núcleo de un problema.

Versiones mejoradas de *auto* realizan búsquedas de pruebas más sofisticadas, al mismo tiempo que intercalan simplificación. Estos métodos suelen ser útiles, pero dado que implican cierto tipo de búsqueda, no solo son más lentos que el simplificador y *auto*, sino que además no brindan ninguna pista cuando no logran demostrar el objetivo.

3.3.3. Blast y Metis

Las versiones más sofisticadas de *auto* mencionadas anteriormente pueden ser lentas debido a que cada paso de inferencia es ejecutado directamente en el estado de la prueba, mediante el

kernel de Isabelle. Para mayor performance, los usuarios pueden usar *blast*, un prover de tipo tableau [Paulson, 1997] escrito directamente en ML [Ullman, 1994] que evita el kernel.

Una vez que la prueba fue encontrada, es ejecutada nuevamente en el kernel para validarla. El método blast supera a la implementación de tableau basada en el kernel por un amplio margen, pero no es rival de los mejores probadores automáticos. Tampoco explota el concepto de simplificación, lo cual es una gran pérdida.

Yendo un paso más allá, Metis es un probador de teoremas de resolución escrito en ML, con muy buenos resultados en competiciones de prueba de teoremas [Sutcliffe, 2008].

Ha sido portado a Isabelle y sigue la misma filosofía que blast: la búsqueda de la prueba se realiza directamente en ML, y en caso de ser encontrada, se verifica en el kernel de Isabelle. El método blast se basa en una base de datos extensible de lemas que impulsa la búsqueda y que está pre-configurada para 'razonar' sobre conjuntos, funciones y relaciones, lo que lo hace bastante fácil de usar. Por otro lado, la versión de Metis de Isabelle solo conoce la lógica pura y deriva su conocimiento sobre otros operadores de lemas proporcionados explícitamente.

Aunque Metis se puede invocar directamente, en la práctica las llamadas a Metis casi siempre son generadas por *Sledgehammer*.

3.3.4. Sledgehammer: Descubriendo pruebas con la ayuda de otros provers

Sledgehammer es el subsistema de Isabelle que concentra el poder de los probadores automáticos de teoremas de primer orden.

Dada una conjetura, selecciona de forma heurística unos cientos de hitos relevantes (lemas, definiciones o axiomas) de las bibliotecas de Isabelle, los traduce a lógica de primer orden junto con la conjetura y delega la búsqueda de pruebas a probadores de resolución externos (E, SPASS, y Vampire) y solvers SMT (CVC3, Yices y Z3 [de Moura and Bjørner, 2008]).

Sledgehammer es muy eficaz y ha alcanzado una gran popularidad entre usuarios, novatos y expertos por igual.

Filtros por relevancia

La mayoría de los probadores automáticos funcionan mal en presencia de miles de axiomas. Sledgehammer emplea un filtro de relevancia simple para extraer unos pocos cientos de datos de las bibliotecas de Isabelle que podrían ser relevantes para el problema en cuestión.

A pesar de su sencillez, este filtro mejora en gran medida la tasa de éxito de Sledgehammer. El filtro funciona de forma iterativa. La primera iteración selecciona hechos que comparten todas, o casi todas, sus constantes (símbolos) con la conjetura. Las iteraciones posteriores incluyen hechos que comparten constantes con hechos previos, hasta alcanzar el número deseado de

hechos. Al observar que algunos provers manejan mejor las bases de axiomas grandes que otros, ese número se optimizó de forma independiente para cada prover.

Traducción a lógica de primer orden

El formalismo de Isabelle, con lógica polimórfica de orden superior (polymorphic HOL) y clases tipadas [Wenzel, 2012], es mucho más complejo que la lógica de primer orden soportada por los provers automáticos.

Sledgehammer se basa en diferentes traducciones dependiendo de la clase de probador.

Para los probadores de resolución, se emplean técnicas estándar para traducir fórmulas HOL a lógica clásica de primer orden: las abstracciones λ se reescriben a combinadores y las funciones currificadas son traducidas variando el número de argumentos, mediante operador de `apply` explícito.

Hasta hace poco, la traducción de tipos no era sólida: proporcionaba suficiente información de tipos para hacer cumplir el razonamiento de clase de tipos correcto, pero no para especificar el tipo de cada término. La implementación actual soluciona de forma segura la mayoría de la problemas de información de tipos infiriendo la monotonicidad de los mismos [Blanchette and Krauss, 2011], la cual brinda una codificación sólida y eficiente.

Para los solvers de SMT, la traducción asigna operadores aritméticos y de igualdad a conceptos correspondientes en SMT-LIB [Ranise and Tinelli, 2006]. La lógica de SMT-LIB está ordenada de forma múltiple, lo que la hace más apropiada para codificar información de escritura HOL que la lógica clásica de primer orden, pero no admite polimorfismo. La solución para este inconveniente es monomorfizar las fórmulas: las fórmulas polimórficas se instancian de forma iterativa con 'instancias fundamentales' de sus constantes polimórficas. Este proceso se itera para obtener el problema monomorfizado. Las aplicaciones parciales se traducen usando un operador de *apply*, pero en contraste con el enfoque de combinación que se usa cuando se comunica con probadores de resolución, las abstracciones λ se elevan a nuevas reglas, introduciendo así nuevas constantes.

Invocación de probadores externos

Sledgehammer permite que los probadores externos se ejecuten en paralelo, tanto de forma local como remota. En una instalación habitual de Isabelle[4], E, SPASS, y Z3 son ejecutados localmente, mientras que Vampire y el metaprover SInE son utilizados mediante SystemOnTPTP remoto. Los usuarios pueden también habilitar CVC3 y Yices.

La siguiente figura representa la arquitectura, omitiendo la reconstrucción y minimización de la prueba. En la misma, se puede apreciar que se ejecutan dos instancias del filtro de relevancia para tener en cuenta diferentes conjuntos de constantes integradas. Los hechos relevantes y la conjetura se traducen a la versión TPTP o SMT (lógica de primer orden), y los problemas

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

resultantes se delegan a los probadores. La traducción de Z3 se realiza de forma ligeramente diferente a la de CVC3 e Yices, para beneficiarse del soporte de Z3 en aritmética no lineal.

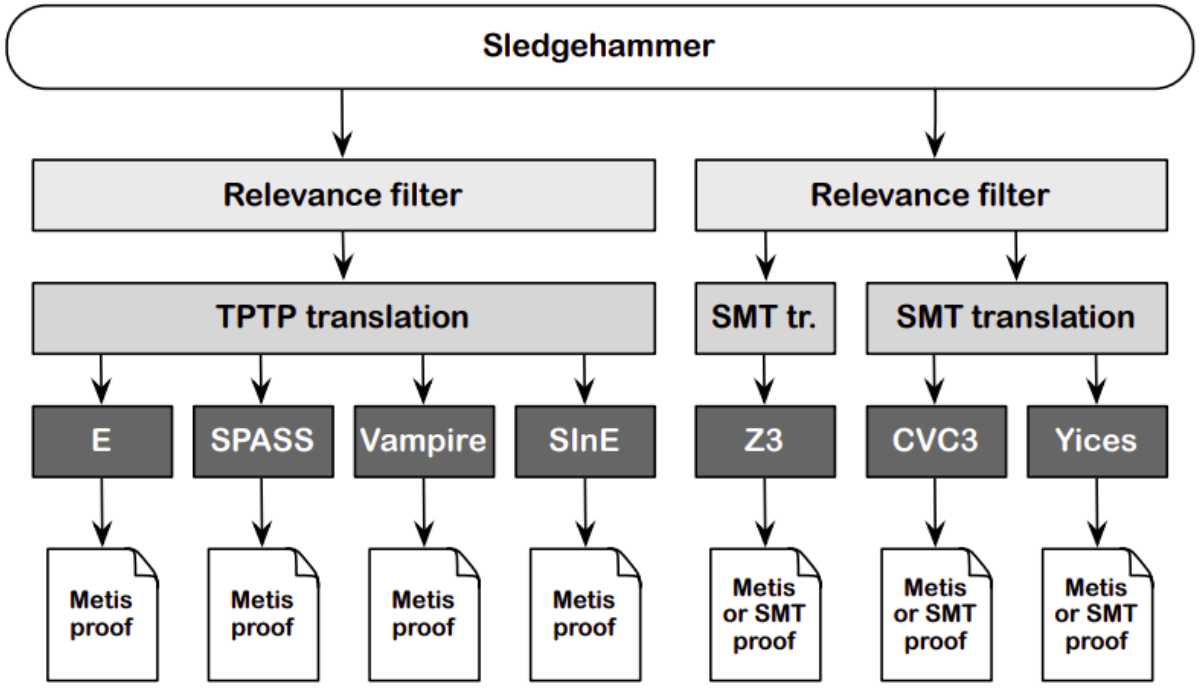


Figura 3.2: Arquitectura de Sledgehammer. Extraída de [Blanchette et al., 2011].

Idealmente, los probadores de terceros se incluyen en un paquete con Isabelle y están listos para usarse sin necesidad de configuración. Isabelle incluye ejecutables CVC3, E, SPASS y Z3 para las principales plataformas de hardware; los usuarios pueden descargar Yices y Vampire, cuyas licencias prohíben la redistribución, pero la mayoría simplemente ejecuta Vampire de forma remota en SystemOnTPTP.

Los servidores remotos son ideales para la búsqueda de pruebas, al menos cuando están funcionando y el usuario tiene acceso a Internet. También ayudan a distribuir la carga: a menos que la máquina del usuario posea un procesador de ocho núcleos, sería imprudente lanzar cuatro probadores de resolución y tres solvers SMT y esperar que la interfaz de usuario de Isabelle siga respondiendo. La invocación paralela de probadores es invaluable: ejecutar E, SPASS y Vampire juntos durante cinco segundos resuelve tantos problemas como ejecutar un solo probador durante dos minutos [Böhme and Nipkow, 2010].

Reconstruyendo la prueba

Para probadores de resolución, Sledgehammer realiza una reconstrucción de prueba ejecutando el probador de resolución integrado de Isabelle, Metis, brindando la breve lista de hechos utilizados en la prueba encontrada por el prover. Dado solo unos pocos hechos, Metis generalmente tiene éxito en milisegundos. Dado que Metis tiene que volver a encontrar la prueba, los provers externos se utilizan esencialmente como filtros de relevancia de gran precisión.

Las pruebas generadas usualmente requieren una edición a posteriori para volverlas sintácticamente correctas. Las nuevas actualizaciones siempre realizan esfuerzos para hacer que el resultado generado sea más robusto y conciso.

Con respecto a SMT, las pruebas que no involucran razonamiento aritmético generalmente pueden ser reproducidas por Metis; de lo contrario, se admite la reproducción de la prueba, paso a paso, para Z3. Mientras que CVC3 e Yices se pueden invocar como oráculos. La repetición de la prueba Z3 se basa en gran medida en los procedimientos de decisión aritmética, simplificador y probador de tipo tableau de Isabelle.

Minizando la prueba

Los probadores externos usualmente usan muchos más hechos de los necesarios. La herramienta de minimización de Sledgehammer toma el conjunto de hechos usados retornados por un prover e invoca repetidamente al mismo con subconjuntos de los hechos para encontrar un conjunto mínimo.

Dependiendo del número de hechos iniciales, se basa en un algoritmo lineal *naive* que intenta eliminar un hecho a la vez o en un algoritmo binario que biseca recursivamente los hechos [Böhme and Nipkow, 2010].

La minimización frecuentemente mejora el rendimiento y la tasa de éxito de Metis, y al mismo tiempo elimina el desorden de las formalizaciones de Isabelle. Para algunos provers, es difícil o imposible extraer la lista de hechos usados de la prueba; la minimización es entonces la única opción. Por ejemplo, las pruebas detalladas devueltas por CVC3 siempre se refieren a todos los hechos, sean realmente necesarios o no, y no existe un criterio fácil para aislar los hechos necesarios.

3.3.5. Quickcheck: Generación de contraejemplos

Los métodos de prueba de Isabelle y Sledgehammer son efectivos frente a conjeturas válidas, pero dada una conjetura inválida, normalmente no detectan la invalidez de la misma, y mucho menos producen un contraejemplo relevante.

Aquí es donde entra en escena Quickcheck. Quickcheck se modeló originalmente a partir de la herramienta QuickCheck para Haskell [Claessen and Hughes, 2000], que prueba las propiedades proporcionadas por el usuario de un programa Haskell para valores generados aleatoriamente.

Recientemente Quickcheck se ha ampliado con pruebas exhaustivas y basadas en restricciones como complemento a las pruebas aleatorias.

Las pruebas exhaustivas verifican la fórmula para cada posible conjunto de valores hasta un límite dado, y por lo tanto encuentran contraejemplos que las pruebas aleatorias podrían pasar por alto. La reducción puede ser más precisa y eficiente que los otros dos enfoques porque considera la fórmula simbólicamente, en lugar de probar un conjunto finito de valores fundamentales. Gracias a un análisis de flujo de datos estáticos, inspirado en la programación lógica, Quickcheck deriva generadores de datos de prueba que tienen en cuenta las premisas para ayudar a evitar los casos de prueba vacíos que afectan a la mayoría de las herramientas de prueba de especificaciones.

3.4. Semántica de Marlowe en Isabelle

Antes de comenzar a realizar pruebas sobre cualquier contrato escrito en Marlowe, es necesario definir formalmente en Isabelle los pilares del DSL.

Al momento de comenzar a realizar pruebas, la semántica de Marlowe estaba traducida a Isabelle [Lamela Seijas et al., 2020a, Lamela Seijas et al., 2020b] (junto con algunos *lemmas* y teoremas correspondientes). El mayor responsable de dicha traducción es Pablo Lamela[5], quien colaboró con la supervisión de este trabajo, en particular durante la escritura de mis pruebas.

Debido a que no existen actualmente traductores de Haskell [Torrini et al., 2007], dicho trabajo debe realizarse manualmente.

Esto implica la traducción (por momentos debiendo reestructurar la organización de las funciones en Haskell, debido a que no hay una biyección trivial entre los lenguajes) y la prueba de terminación y exclusión mutua de las funciones. Debido a que HOL es una lógica de funciones totales[6], la terminación es un requisito fundamental para prevenir inconsistencias. Isabelle intenta probar la terminación automáticamente cuando se escribe una definición, pero en casos complejos es posible que el usuario tenga que proveerla utilizando técnicas de prueba.

La totalidad de las pruebas de Marlowe en Isabelle se encuentran en [IOHK, 2019b]. Algunas de las teorías ya probadas se detallan a continuación. En [Lamela Seijas et al., 2020a] se menciona que se intentará mantener las versiones (escritas Haskell e Isabelle) tan similares como sea posible. Sin embargo, tomaron la decisión de implementar de forma diferente dos aspectos:

- *Enteros como identificadores*, ya que son mas sencillos para manejar que las strings.
- *Implementaciones personalizadas de map y set*, basadas en listas, debido a que Isabelle ya provee un extenso arsenal de teoremas probados en listas.

Algunas de las propiedades probadas para la semántica de Marlowe pueden verse a continuación.

[5] https://iohk.io/en/team/pablo-lamela-seijas

[6] Una función se conoce como total cuando está definida para todos los valores posibles de entrada.

3.4.1. Prueba de Terminación (*Termination proof*)

Isabelle prueba la terminación de forma automática para la mayoría de las funciones.Sin embargo, este no es el caso de `reductionLoop`.

Esta función llama repetidamente a `reduceContractStep`, hasta que la misma retorna *NotReduced*, por lo tanto la terminación global requiere una prueba de que `reduceContractStep` va a retornarlo eventualmente. Para probar esto, se define una medida para un par `(Contract` × `State)`:

```
fun evalBound :: "State ⇒ Contract ⇒ nat" where
"evalBound sta cont = length (accounts sta) + 2 * (size cont)"
```

Código 3.9: Función que será utilizada por *measure* para probar la terminación de *reductionLoop*

donde `size` es una medida generada automáticamente por Isabelle.

Se requirió indicar la cantidad de cuentas en `State` porque el tamaño del contrato `Close` no disminuirá al llamar a `reduceContractStep`, pero la cantidad de cuentas sí lo hará, a menos que estén todas vacías.

Adicionalmente, se debió multiplicar el tamaño del *Contrato* por dos porque el `Deposit` puede aumentar el número de cuentas en uno, por lo que hubo que multiplicar el efecto de la reducción del tamaño del contrato para compensarlo.

3.4.2. Estado válido y preservación de cuentas en positivo

Hay algunos valores para `State` que están permitidos por su tipo pero que no tienen sentido, especialmente en el caso de la semántica de Isabelle donde usamos listas en lugar de mapas:

1. Las listas representan mapas, por lo tanto no deberían tener claves repetidas.
2. Es deseable que dos mapas iguales sean representados de la misma forma, entonces se forzó a las claves a estar en orden ascendiente.
3. Solo se requiere almacenar las cuentas con una valor positivo.

Se define a un valor de `State` como *valido* si las dos primeras propiedades son verdaderas. Y decimos que tiene cuentas positivas si la tercer propiedad es verdadera.

Se demostró que las funciones en la semántica conservan las tres propiedades.

Resultado inactivo (Quiescent result)

Un contrato es inactivo o *quiescent* si y solo si el constructor raíz es `When` (esperando *inputs* de usuario), o si el contrato es `Close` y todas las cuentas están vacías. Fue probado que, si un

`State` de entrada es válido y las cuentas son positivas, entonces la salida será inactiva.

3.4.3. Preservación del dinero y *timeout* del contrato

Uno de los peligros de usar *smart contracts* es que uno mal escrito puede potencialmente bloquear sus fondos para siempre. Al final del contrato, todo el dinero retenido en el contrato debe distribuirse, de alguna manera, a un subconjunto de los participantes del mismo. Para asegurarse de que este sea el caso, se probaron dos propiedades:

Preservación del dinero

El dinero no se crea ni se destruye por la semántica. Concretamente, el dinero que ingresa más el dinero en el contrato antes de la transacción debe ser igual al dinero que sale más el dinero en el contrato después de la transacción, salvo en casos en que ocurra un error.

Un *timeout* cierra un contrato

Por cada contrato en Marlowe hay un *slot* después del cual se puede emitir una transacción vacía que cerrará el mismo y reembolsará todo el dinero en sus cuentas.

Un límite superior conservador para el *slot* de vencimiento se puede calcular de manera eficiente utilizando la función `maxTime` (o `maxTimeContract` en la semántica de Isabelle), que simplemente toma el máximo de todos los *timeouts* en el contrato.

Se ha probado que este límite superior conservador es lo suficientemente general para cada contrato, mostrando que, si el contrato no está cerrado y vacío, entonces una transacción vacía enviada después de `maxTime` cerrará el contrato y vaciará las cuentas.

3.4.4. Límite en el número máximo de transacciones

Otra propiedad de Marlowe es que cualquier contrato tiene un límite finito implícito en el número máximo de transacciones aceptadas. Esta es una propiedad conveniente por dos razones:

En primer lugar, reduce el peligro de ataques de denegación de servicio (DoS): ya que el número de entradas válidas es limitado, un participante atacante no puede bloquear arbitrariamente el contrato mediante la emisión de una cantidad ilimitada de transacciones inútiles.

En segundo lugar, el número de transacciones limita la longitud de los rastros que necesita explorar la ejecución simbólica (consultar la Sección 4 de [Lamela Seijas et al., 2020a]). Establecemos la propiedad de la siguiente manera:

```
lemma playTrace_only_accepts_maxTransactionsInitialState :
"playTrace sl c l = TransactionOutput txOut
    ⟹ length l ≤ maxTransactionsInitialState c"
```

Código 3.10: Lema que verifica que *playTrace* no supere el número máximo de transacciones.

donde `maxTransactionsInitialState` es el máximo número de constructores `When` anidados en el contrato más uno.

Esta propiedad implica que cualquier seguimiento (*trace*) que sea más largo que esto produce al menos un error. Debido a que las transacciones que producen un error no alteran el estado del contrato, tal lista de transacciones (un *trace*) será equivalente a una lista de transacciones que no tiene la transacción errónea.

Por lo tanto, no perdemos generalidad al explorar solo *traces* más cortos.

Capítulo 4

Desarrollo: Verificación de contratos financieros usando Isabelle

En este capítulo se describirán los aportes realizados en la presente tesis.

En primer lugar, la escritura de contratos financieros ACTUS para la blockchain Cardano, junto con la creación de casos de prueba y validación con los tests oficiales del estándar [ACTUS, 2019f].

En segundo lugar, la traducción de algunos contratos financieros sencillos a Isabelle, junto con la ejecución y prueba de propiedades en los mimos.

Finalmente, probaremos la ausencia de *warnings* en el contrato *Auction* [IOHK, 2019a]. Para esto, traduciremos el contrato a Isabelle, demostraremos la terminación del mismo y algunos lemas auxiliares que nos ayudan en la prueba del teorema.

4.1. Escritura de contratos ACTUS para Cardano

En esta sección, veremos como se desarrolló la escritura de tres contratos ACTUS para la blockchain Cardano, bajo la supervisión de IOHK.

Cabe destacar que durante esta tarea, tuve la colaboración de Yves Hauser[1], con quien conversamos sobre decisiones de diseño e implementación de los contratos correspondientes. Yves fue el responsable de integrar mis cambios a la rama *master* del repositorio de `marlowe-cardano` [IOHK, 2016].

[1] https://iohk.io/en/team/yves-hauser

4.1.1. Descripción de la estructura del proyecto

En la siguiente sub-sección describiremos brevemente algunos de las entidades relevantes para la implementación de un contrato. Para eso, es importante conocer la estructura de archivos del mismo, junto con la responsabilidad de algunos de ellos.

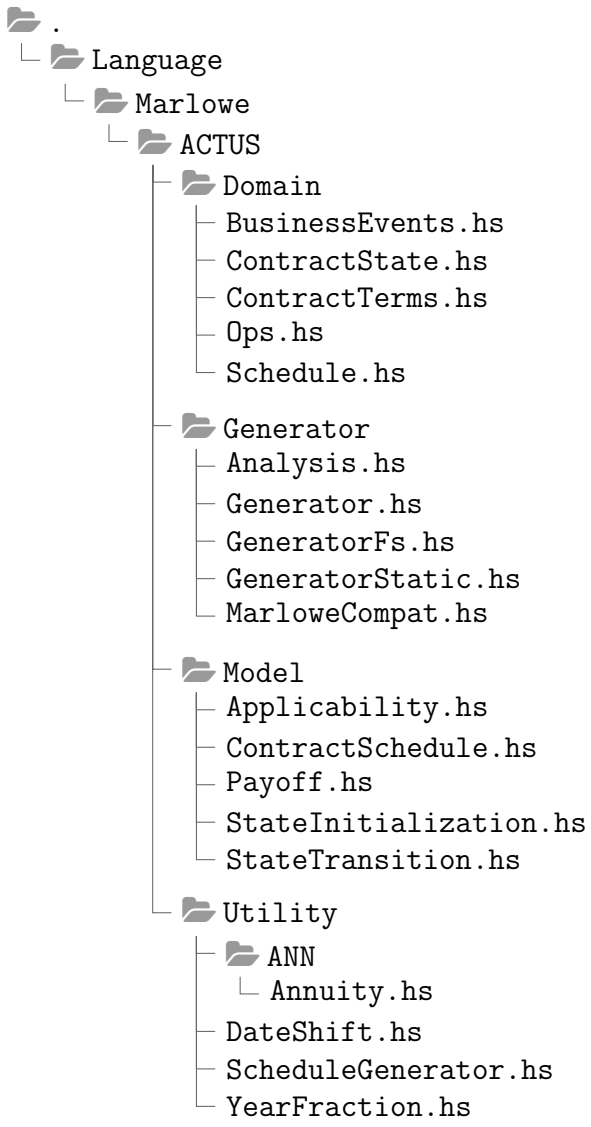
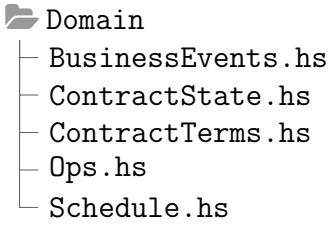


Figura 4.1: Árbol de archivos en el repositorio del lenguaje Marlowe. Extraído de [IOHK, 2020].

Para comenzar, podemos observar los cuatro directorios principales: `Domain`, `Generator`, `Model` y `Utility`. En los mismos es posible encontrar el código necesario para modelar los contratos ACTUS, y generar el código de los mismos en Marlowe o en Haskell para luego ser ejecutados.

Domain

El mismo está conformado por los archivos que modelan el dominio de los contratos ACTUS.



A lo largo de los mismos, se ven definiciones de tipos necesarios en ACTUS. Como por ejemplo: *EventType* o *RiskFactorsPoly* en `BusinessEvents.hs` o *ContractStatePoly* en `ContractState.hs`.

A continuación vemos algunos fragmentos de algunas definiciones de tipos:

```
{-| ACTUS event types
    https://github.com/actusfrf/actus-dictionary/blob/master/actus-
   dictionary-event.json
-}
data EventType =
       IED  -- ^ Initial Exchange
     | FP   -- ^ Fee Payment
     | PR   -- ^ Principal Redemption
     | PD   -- ^ Principal Drawing
     | PY   -- ^ Penalty Payment
     | PP   -- ^ Principal Prepayment (unscheduled event)
     | IP   -- ^ Interest Payment
     | IPFX -- ^ Interest Payment Fixed Leg
     | IPFL -- ^ Interest Payment Floating Leg
     | IPCI -- ^ Interest Capitalization
     | CE   -- ^ Credit Event

     [...]

     | MD   -- ^ Maturity
     | XD   -- ^ Exercise
     | STD  -- ^ Settlement
     | PI   -- ^ Principal Increase
     | AD   -- ^ Monitoring
```

Código 4.1: Algunos tipos de eventos.

```
{-| ACTUS contract states are defined in
    https://github.com/actusfrf/actus-dictionary/blob/master/actus-
   dictionary-states.json
-}
data ContractStatePoly a = ContractStatePoly
  {
    tmd   :: Maybe LocalTime -- ^ Maturity Date (MD): The timestamp as
   per which the contract matures according to the initial terms or as
   per unscheduled events
  , nt    :: a               -- ^ Notional Principal (NT): The
   outstanding nominal value
  , ipnr  :: a               -- ^ Nominal Interest Rate (IPNR) : The
   applicable nominal rate
  , ipac  :: a               -- ^ Accrued Interest (IPAC): The current
   value of accrued interest
  , ipac1 :: Maybe a         -- ^ Accrued Interest (IPAC1): The current
   value of accrued interest of the first leg
  , ipac2 :: Maybe a         -- ^ Accrued Interest (IPAC2): The current
   value of accrued interest of the second leg
  , ipla  :: Maybe a         -- ^ Last Interest Period
  , feac  :: a               -- ^ Fee Accrued (FEAC): The current value
   of accrued fees
  , nsc   :: a               -- ^ Notional Scaling Multiplier (SCNT):
   The multiplier being applied to principal cash flows
  , isc   :: a               -- ^ InterestScalingMultiplier (SCIP): The
   multiplier being applied to interest cash flows
  , prf   :: PRF             -- ^ Contract Performance (PRF)
  , sd    :: LocalTime       -- ^ Status Date (MD): The timestamp as per
    which the state is captured at any point in time
  , prnxt :: a               -- ^ Next Principal Redemption Payment (
   PRNXT): The value at which principal is being repaid
  , ipcb  :: a               -- ^ Interest Calculation Base (IPCB)
  , xd    :: Maybe LocalTime -- ^ Exercise Date (XD)
  , xa    :: Maybe a         -- ^ Exercise Amount (XA)
  }
```

Código 4.2: Contract state poly.

También es posible ver tipos relacionados a características de un contrato, como *ContractType*, *ContractRole*, *DayCountConvention*, *BusinessDayConvention*, etc en `ContractTerms.hs`.

A continuación vemos algunos fragmentos de algunas definiciones de los mismos:

```
-- |ContractType
data CT = PAM   -- ^ Principal at maturity
        | LAM   -- ^ Linear amortizer
        | NAM   -- ^ Negative amortizer
        | ANN   -- ^ Annuity
        | STK   -- ^ Stock
        | OPTNS -- ^ Option
        | FUTUR -- ^ Future
        | COM   -- ^ Commodity
        | CSH   -- ^ Cash
        | CLM   -- ^ Call Money
        | SWPPV -- ^ Plain Vanilla Swap
        | CEG   -- ^ Guarantee
        | CEC   -- ^ Collateral
```

Código 4.3: Tipos de contrato.

```
-- |ContractRole
data CR = CR_RPA -- ^ Real position asset
        | CR_RPL -- ^ Real position liability
        | CR_CLO -- ^ Role of a collateral
        | CR_CNO -- ^ Role of a close-out-netting
        | CR_COL -- ^ Role of an underlying to a collateral
        | CR_LG  -- ^ Long position
        | CR_ST  -- ^ Short position
        | CR_BUY -- ^ Protection buyer
        | CR_SEL -- ^ Protection seller

        [...]
```

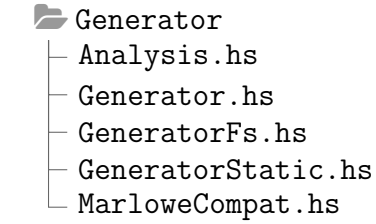
Código 4.4: Algunos tipos de roles de contrato.

```
-- |BusinessDayConvention
data BDC = BDC_NULL -- ^ No shift
         | BDC_SCF  -- ^ Shift/calculate following
         | BDC_SCMF -- ^ Shift/calculate modified following
         | BDC_CSF  -- ^ Calculate/shift following
         | BDC_CSMF -- ^ Calculate/shift modified following
         | BDC_SCP  -- ^ Shift/calculate preceding
         | BDC_SCMP -- ^ Shift/calculate modified preceding
         | BDC_CSP  -- ^ Calculate/shift preceding
         | BDC_CSMP -- ^ Calculate/shift modified preceding
```

Código 4.5: Tipos de convención sobre días laborables.

Generator

En este directorio se implementan los diferentes generadores y compatibilidad hacia el lenguaje Marlowe.



En el archivo `Analysis.hs`, se definen las funciones:

- `genProjectedCashflows`
- `genCashflow`
- `genSchedules`
- `genStates`
- `genPayoffs`

Las mismas, como sus nombres lo indican, generan los flujos de dinero proyectados, flujos de dinero, cronogramas, estados y pagos respectivamente. Los flujos de dinero pueden ser utilizados para generar los pagos en un contrato en Marlowe.

En `GeneratorStatic.hs`, mediante la función `genFsContract` genera un contrato en Marlowe a partir de los términos de contrato ACTUS.En `genFsContract`, los *risk factors* son agregados al contrato en Marlowe, en otras palabras, será 'observado' durante la vida del mismo.

La función `genStaticContract`, en `GeneratorStatic.hs`, valida que los términos del contrato sean correctos, para poder genera un contrato en Marlowe donde los *risk factors* sean conocidos de antemano. Esta restricción desencadena en contratos que solo consisten de transacciones, i.e. `Deposit` y `Pay`.

Con respecto a `MarloweCompat.hs`, contiene algunas funciones auxiliares y la función `toMarlowe`, que se ocupa de recibir los términos del contrato en haskell y realizar la traducción a términos del contrato en Marlowe.

A continuación se muestra el fragmento inicial de la misma:

```
toMarlowe :: ContractTerms -> ContractTermsMarlowe
toMarlowe ct =
  ContractTermsPoly
    { contractId = contractId ct,
```

```
        contractType = contractType ct,
        contractStructure = map trans (contractStructure ct),
        contractRole = contractRole ct,
        settlementCurrency = settlementCurrency ct,
        initialExchangeDate = initialExchangeDate ct,
        dayCountConvention = dayCountConvention ct,
        scheduleConfig = scheduleConfig ct,
        statusDate = statusDate ct,
        contractPerformance = contractPerformance ct,
        creditEventTypeCovered = creditEventTypeCovered ct,

        [...]
```

Código 4.6: Comienzo de la función `toMarlowe`.

Model

En este directorio se encuentran los archivos que modelan la lógica expuesta por el estándar ACTUS, tales como Schedule, Inicialización de variables de estado y funciones de transición de estado y de pago.

```
Model
├── Applicability.hs
├── ContractSchedule.hs
├── Payoff.hs
├── StateInitialization.hs
└── StateTransition.hs
```

Las reglas de aplicabilidad[2] son definidas, para cada tipo de contrato. Las reglas revisan la consistencia de los atributos de contrato, para los términos de un contrato ACTUS dado. Si los términos son validados satisfactoriamente, se espera que el contrato sea significativo y flujos de dinero puedan ser proyectados. Dicha condición es chequeada en `Applicability.hs`

Con respecto a `ContractSchedule.hs`, `Payoff.hs`, `StateInitialization.hs` y `StateTransition.hs`, se ocupan de ajustar el cronograma, los pagos, inicializar las variables de estado, y las transiciones sobre las mismas.

Como ejemplo, puede verse reflejado el cambio de estado para el contrato PAM y evento IPCI (Interest Capitalization) en parte de la implementación de la función `stateTransition`:

[2]Especificadas en https://www.actusfrf.org/aplicability

```
[...]
-------------------------------------
-- Interest Capitalization (IPCI) --
-------------------------------------
-- STF_IPCI_PAM
-- STF_IPCI_CLM
stf
  IPCI
  rf
  ct@ContractTermsPoly
    { contractType,
      dayCountConvention = Just dcc,
      maturityDate
    }
  st@ContractStatePoly
    { ..
    } | contractType `elem` [PAM, CLM] =
    let y_sd_t = _y dcc sd t maturityDate
        st' = stf IP rf ct st
     in st'
          { nt = nt + ipac + y_sd_t * nt * ipnr
          }
[...]
```

Código 4.7: Parte de la función que modela la transicción de estados.

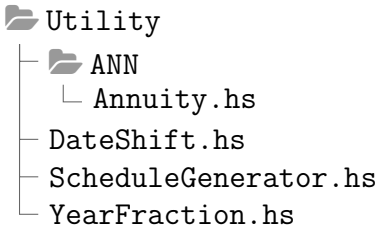
La misma refleja las variables siendo ajustadas de acuerdo al estándar:

PAM: State Transition Functions and Payoff Functions

Event	Payoff Function	State Transition Function
IPCI	0.0	$\mathbf{Nt}_{t^+} = \mathbf{Nt}_{t^-} + \mathbf{Ipac}_{t^-} + Y(\mathbf{Sd}_{t^-}, t)\mathbf{Nt}_{t^-}\mathbf{Ipnr}_{t^-}$ $\mathbf{Ipac}_{t^+} = 0{,}0$ $\mathbf{Feac}_{t^+} = \begin{cases} \mathbf{Feac}_{t^-} + Y(\mathbf{Sd}_{t^-}, t)\mathbf{Nt}_{t^-}\mathtt{FER} & \text{if } \mathtt{FEB} = \text{'N'} \\ \frac{Y(t^{FP-}, t)}{Y(t^{FP-}, t^{FP+})} R(\mathtt{CNTRL})\mathtt{FER} & \text{else} \end{cases}$ $\mathbf{Sd}_{t^+} = t$ with $t^{FP-} = \sup t \in \bar{t}^{FP} \mid t < t_0$ $t^{FP+} = \inf t \in \bar{t}^{FP} \mid t > t_0$

Utility

En este directorio se encuentran algunos archivos con funciones que se utilizan para aislar la lógica del calculo de fechas, que suele tornarse complejo y repetitivo durante los contratos.



Funciones para calcular determinadas fechas, para cada tipo de convención de día hábil, son extraídas en el módulo `DateShift.hs`, como por ejemplo: `applyBDC`, `getPreceedingBusinessDay`, `moveToEndOfMonth`, etc.

4.1.2. Metodología de escritura de un Contrato ACTUS en Cardano

Para poder escribir un contrato, es necesario revisar los eventos que disparan acciones del contrato, las mismas están especificadas en [ACTUS, 2018], también esta disponible una implementación oficial en Java de los mismos, en la cual es posible basarse. Es conveniente utilizar ambos, ya que se reduce el esfuerzo de implementación en Haskell cuando se cuenta con ellos.

Para cada contrato, se agregan casos a los archivos vistos previamente en 4.1.1, tanto para el modelo como para el dominio. En algunos casos, quizás sea necesario agregar alguna función en Utility.

Con respecto a la implementación en Java, la misma no es pública, pero con fines de la implementación de los siguientes contratos, me fue otorgado por parte de IOHK un token de acceso para poder clonarla y usarla durante la implementación.

Todas las implementaciones son probadas usando algunos ejemplos[3], así como también la batería de tests propuestos por ACTUS [ACTUS, 2019f].

En los tests propuestos por ACTUS, un conjunto de términos de contrato y eventos disparadores son provistos, y se espera que ciertos eventos sean generados por la implementación. Debido a que ACTUS es agnóstica a la misma, los test solo reflejan los eventos esperados, pero no como serán ejecutados en la plataforma (en este caso, en la blockchain Cardano).

A continuación se muestra un test de ejemplo para el contrato COM, provisto por ACTUS:

```
"com04": {
        "identifier": "com04",
        "terms": {
            "contractType": "COM",
            "contractID": "com04",
            "statusDate": "2014-12-30T00:00:00",
```

[3]Disponibles en https://github.com/input-output-hk/marlowe-cardano/blob/main/marlowe-actus/test/Spec/Marlowe/ACTUS/Examples.hs

```
            "contractDealDate": "2014-12-15T00:00:00",
            "purchaseDate": "2015-03-30T00:00:00",
            "priceAtPurchaseDate": "700",
            "terminationDate": "2016-02-20T00:00:00",
            "priceAtTerminationDate": "900",
            "contractRole": "RPL",
            "currency": "GBP",
            "marketValueObserved": "100",
            "quantity": "3",
            "unit": "BRL"
        },
        "to": "2016-03-30T00:00:00",
        "dataObserved": {

        },
        "eventsObserved": [

        ],
        "results": [
            {
                "eventDate": "2015-03-30T00:00:00",
                "eventType": "PRD",
                "payoff": 2100,
                "notionalPrincipal": 0,
                "nominalInterestRate": 0,
                "accruedInterest": 0,
                "currency": "GBP"
            },
            {
                "eventDate": "2016-02-20T00:00:00",
                "eventType": "TD",
                "payoff": -2700,
                "notionalPrincipal": 0,
                "nominalInterestRate": 0,
                "accruedInterest": 0,
                "currency": "GBP"
            }
        ]
    }
```


TUS para el contrato COM. Se pueden ovservar tanto los términos como eventos observados y los eventos que se esp

Dicho `json` es procesado en TestFramework.hs, donde se analiza gramaticalmente la entrada, se ejecuta el generador, y se comparan los campos del json inicial con los del objeto generado por la implementación.

Por otro lado, podemos ver el test de ejemplo agregado para el contrato COM

```
-- |ex_com1 defines a contract of type COM
ex_com1 :: IO ()
ex_com1 =
  contractFromFile "test/Spec/Marlowe/ACTUS/ex_com1.json"
    >>= either msg run
  where
    msg err = putStr err
    run ct = case genFsContract defaultRiskFactors (toMarlowe ct) of
      Failure _ -> assertFailure "Terms validation should not fail"
      Success contract ->
          let principal = NormalInput . IDeposit (Role "party") "party"
    ada
              out =
                computeTransaction
                  ( TransactionInput
                      (0, 0)
                      [ principal 1400
                      ]
                  )
                  (emptyState 0)
                  contract
           in case out of
                Error _ -> assertFailure "Transactions are not expected
    to fail"
                TransactionOutput txWarn txPay _ con -> do
                  assertBool "Contract is in Close" $ con == Close
                  assertBool "No warnings" $ null txWarn

                  assertBool "total payments to party" (totalPayments (
    Party "party") txPay == 0)
                  let tc = totalPayments (Party "counterparty") txPay
                  assertBool ("total payments to counterparty: " ++ show
     tc) (tc == 1400)

contractFromFile :: FilePath -> IO (Either String ContractTerms)
contractFromFile f = eitherDecode <$> B.readFile f
```

Código 4.9: Test de ejemplo escrito durante el desarrollo de la tesis para el contrato COM.

El mismo se ocupa de tomar los términos del contrato de un archivo `ex_com1.json` (de sintaxis similar a los de ACTUS) y ejecutar el generador. Luego valida que las transacciones computadas sean las esperadas, así como también que los balances en las wallets corresponientes sean consistentes.

4.1.3. Contrato COM (*Commodity*)

Según su descripción en [ACTUS, 2019d]:

> "Este contrato no es de tipo financiero en el sentido estricto. Sin embargo, realiza el seguimiento de los movimientos de commodities como el petroleo, gas o incluso viviendas. Dichas commodities pueden utilizarse servir como garantía subyacente de futuros en materias primas, garantías o simplemente posiciones de activos."

Especificación técnica

A continuación adjunto la especificación técnica del contrato COM [ACTUS, 2018]:

COM: Contract Schedule

Event	Schedule	Comments
AD		Same as PAM
PRD		Same as STK
TD		Same as STK

COM: State Variables Initialization

State	Initialization per t_0	Comments
Sd		Same as STK

COM: State Transition Functions and Payoff Functions

Event	Payoff Function	State Transition Function
AD	POF_AD_PAM()	STF_AD_STK()
PRD	POF_PRD_STK()	STF_PRD_STK()
TD	POF_PRD_STK()	STF_PRD_STK()

El contrato COM tiene una implementación similar al contrato STK (Stock), debido a que comparte alguna de las funciones de cambio de estado. Los detalles de la implementación, junto con la de los demas contratos, se encuentran disponibles en la rama `fiuba-collaboration-writing-actus-contracts`[4].

4.1.4. CSH (*Cash*)

Según su descripción en [ACTUS, 2019d]:

> "Es un contrato simple, que representa tenencias de dinero en efectivo."

Especificación técnica

A continuación adjunto la especificación técnica del contrato CSH [ACTUS, 2018]:

[4]Disponible en el repositorio marlowe-cardano de input-output-hk

CSH: Contract Schedule

Event	Schedule	Comments
AD		Same as PAM

CSH: State Variables Initialization

State	Initialization per t_0	Comments
Nt	$\mathbf{Nt}_{t_0} = R(\texttt{CNTRL})\texttt{NT}$	
Sd		Same as PAM

CSH: State Transition Functions and Payoff Functions

Event	Payoff Function	State Transition Function
AD	POF_AD_PAM()	$\mathbf{Sd}_{t^+} = t$

4.1.5. CLM (*Call Money*)

Según su descripción en [ACTUS, 2019d]:

> "Este tipo de contrato modela préstamos que se renuevan, mientras no sea notificado. Una vez realizada, debe devolverse después del período de notificación estipulado."

Especificación técnica

A continuación adjunto la especificación técnica del contrato CLM [ACTUS, 2018]:

CLM: Contract Schedule

Event	Schedule	Comments
AD		Same as PAM
IED		Same as PAM
PR		Same as PAM
FP		Same as PAM
IP	$t^{IP} = \mathbf{Md}_{t_0}$	
IPCI	$\vec{t}^{IPCI} = \begin{cases} \varnothing & \text{if} \quad \texttt{IPNR} = \varnothing \\ S(s, \texttt{IPCL}, \mathbf{Md}_{t_0}) & \text{else} \end{cases}$	where $s = \begin{cases} \texttt{IPANX} & \text{if} \quad \texttt{IPANX} \neq \varnothing \\ \texttt{IED} + \texttt{IPCL} & \text{else} \end{cases}$
RR		Same as PAM
RRF		Same as PAM
CE		Same as PAM

CLM: State Variables Initialization

State	Initialization per t_0	Comments
Md	$\mathbf{Md}_{t_0} = \begin{cases} \texttt{MD} & \text{if} \quad \texttt{MD} \neq \varnothing \\ s & \text{else if} \quad O^{ev}(\texttt{CID}, t_0) \neq \{\} \\ t^{max} & \text{else} \end{cases}$	where $s = \sup t \in \tau(O^{ev}(\texttt{CID}, t_0))$
Nt		Same as PAM
Ipnr		Same as PAM

Continua en la siguiente página

Continua desde la página anterior

State	Initialization per t_0	Comments
Ipac		Same as PAM
Feac		Same as PAM
Prf		Same as PAM
Sd		Same as PAM

CLM: State Transition Functions and Payoff Functions

Event	Payoff Function	State Transition Function
AD	POF_AD_PAM()	STF_AD_PAM()
IED	$X^{\texttt{CURS}}_{\texttt{CUR}}(t)R(\texttt{CNTRL})(-1)\texttt{NT}$	STF_IED_PAM()
PR	POF_PR_PAM()	STF_PR_PAM()
FP	POF_FP_PAM()	STF_FP_PAM()
IP	$X^{\texttt{CURS}}_{\texttt{CUR}}(t)(\mathbf{Ipac}_{t^-} + Y(\mathbf{Sd}_{t^-}, t)\mathbf{Ipnr}_{t^-}\mathbf{Nt}_{t^-})$	$\mathbf{Ipac}_{t^+} = 0{,}0$ $\mathbf{Sd}_{t^+} = t$
IPCI	POF_IPCI_PAM()	STF_IPCI_PAM()
RR	POF_RR_PAM()	STF_RR_PAM()
RRF	POF_RRF_PAM()	STF_RRF_PAM()
CE	POF_CE_PAM()	STF_AD_PAM()

4.2. Traduciendo algunos contratos financieros a Isabelle (*Pay* y *Swap*)

Con el fin de poder comprender mejor la Semantica de Marlowe escrita en Isabelle [IOHK, 2019b], procedí a traducir algunos contratos financieros relativamente sencillos de ejemplo.

4.2.1. Contrato *Pay*

Pese a ser un contrato sencillo, este contrato permite comprender la lógica de tipos que sigue la semántica de Marlowe en Isabelle, como mencionamos en 3.4.

En el siguiente contrato se pueden ver tipos como `Party`, `Payee` o `Token`, así como también el uso de `Constant` para definir un `Value`.

```
theory PayContract
  imports Semantics
begin

definition role_borrower :: Party where
"role_borrower = Role [1]"

definition role_lender :: Party where
"role_lender = Role [2]"
```

```
definition payee :: Payee where
"payee = Party role_lender"

definition token_ada :: Token where
"token_ada = Token [] []"

definition payContract :: Contract where
"payContract = (Pay role_borrower payee token_ada (Constant 37) Close)"

end
```

Código 4.10: Contrato *Pay*

4.2.2. Contrato *Swap*

En el contrato *Swap*, traducido de uno de los ejemplos del *Playground*[5], hace uso de funciones, definiciones, datatypes, etc.

Se puede destacar la definición del tipo de datos *SwapParty*, junto con sus *getters*, que hacen uso de parte del poderoso *Pattern matching* que provee Isabelle:

```
datatype SwapParty = SwapParty Party Token Value
```

Código 4.11: Definición del tipo de dato *SwapParty*.

Este es utilizado por el contrato *Swap* para encapsular parte de la lógica del mismo:

```
theory SwapExample
  imports Semantics
begin

(* Traduccion de Swap.hs *)

fun ConstantParam :: "int ⇒ Value" where
"ConstantParam x = Constant x"
```

[5] https://github.com/input-output-hk/marlowe-cardano/blob/main/marlowe-playground-server/contracts/Swap.hs

```
definition explicitRefunds :: Observation where
"explicitRefunds = TrueObs"

definition lovelacePerAda :: Value where
"lovelacePerAda = Constant 1000000"

definition amountOfAda :: Value where
"amountOfAda = ConstantParam 10"

definition amountOfLovelace :: Value where
"amountOfLovelace = MulValue lovelacePerAda amountOfAda"

definition amountOfDollars :: Value where
"amountOfDollars = ConstantParam 1000"

fun adaDepositTimeout :: "int ⇒ Slot" where
"adaDepositTimeout x = x"

fun dollarDepositTimeout :: "int ⇒ Slot" where
"dollarDepositTimeout x = x"

definition dollar :: TokenName where
"dollar = [0b1]"

definition token_dollars :: Token where
"token_dollars = Token [0x85bb65] dollar"

definition token_ada :: Token where
"token_ada = Token [] []"

datatype SwapParty = SwapParty Party Token Value

fun getParty :: "SwapParty ⇒ Party" where
"getParty (SwapParty p _ _) = p"

fun getCurrency :: "SwapParty ⇒ Token" where
"getCurrency (SwapParty _ t _) = t"

fun getAmount :: "SwapParty ⇒ Value" where
"getAmount (SwapParty _ _ v) = v"
```

```
definition role_ada_provider :: Party where
"role_ada_provider = Role [1]"

definition adaProvider :: SwapParty where
"adaProvider = SwapParty role_ada_provider token_ada amountOfLovelace"

definition role_dollar_provider :: Party where
"role_dollar_provider = Role [2]"

definition dollarProvider :: SwapParty where
"dollarProvider = SwapParty role_dollar_provider
                            token_dollars
                            amountOfDollars"

fun makeDeposit :: "SwapParty ⇒ Timeout ⇒ Contract ⇒ Contract ⇒
                    Contract" where
"makeDeposit src timeout timeoutContinuation continuation =
  When [ Case (Deposit
                (getParty src)
                (getParty src)
                (getCurrency src)
                (getAmount src))
                continuation
       ] timeout
         timeoutContinuation"

fun refundSwapParty :: "SwapParty ⇒ Contract" where
"refundSwapParty swapParty = (If explicitRefunds
                                (Pay
                                  (getParty swapParty)
                                  (Party (getParty swapParty))
                                  (getCurrency swapParty)
                                  (getAmount swapParty)
                                  Close)
                                Close
                              )"

fun makePayment :: "SwapParty ⇒ SwapParty ⇒ Contract ⇒ Contract" where
"makePayment src dest =
  Pay (getParty src)
      (Party (getParty dest))
```

```
       (getCurrency src)
       (getAmount src)"


definition contract :: Contract where
"contract = makeDeposit
              adaProvider
              (adaDepositTimeout 100)
              Close
              (makeDeposit
                  dollarProvider
                  (dollarDepositTimeout 150)
                  (refundSwapParty adaProvider)
                  (makePayment
                      adaProvider
                      dollarProvider
                      (makePayment
                          dollarProvider
                          adaProvider
                          Close
                      )
                  )
              )"
```

Código 4.12: Definición del contrato *Swap*

4.3. Probando una propiedad en un contrato COM y PAM

4.3.1. Generación de codigo Marlowe de algunos contratos COM y PAM

Para poder obtener contratos una instancia de contrato Marlowe, es necesario modificar levemente el código del mismo. En este caso, generamos algunos contratos COM y un contrato PAM a partir de los test cases propuestos por ACTUS [ACTUS, 2019f].

Por ejemplo, agregando la siguiente función al archivo `marlowe-actus/test/Spec/Marlowe/ACTUS/Examples.hs`, dentro del repositorio de `marlowe`:

```
printCOM :: IO ()
printCOM =
  contractFromFile "test/Spec/Marlowe/ACTUS/ex_com1.json"
    >>= either msg run
  where
    msg err = putStr err
    run ct = case genFsContract defaultRiskFactors (toMarlowe ct) of
      Failure _ -> assertFailure "Terms validation should not fail"
      Success contract -> do
          assertBool (pTrace (".\n\nMostrando el contrato:\n\n" ++ show
   contract) "") False
```

Código 4.13: Función para imprimir contratos en Marlowe.

Con esto, podemos obtener el contrato que necesitemos, según los términos del mismo y las condiciones de riesgo (para el siguiente ejemplo utilizamos los riesgos por defecto).

Se puede ver a continuación el contrato generado en Marlowe:

```
When
    [Case
        (Deposit "party" "party"
            (Token "" "")
            (Constant 2100)
        )
        (Pay "party"
            (Party "counterparty")
            (Token "" "")
            (Constant 2100)
            (When
                [Case
                    (Deposit "counterparty" "counterparty"
                        (Token "" "")
                        (NegValue (Constant (-2700)))
                    )
                    (Pay "counterparty"
                        (Party "party")
                        (Token "" "")
                        (NegValue (Constant (-2700)))
                        Close
                    )
                ]
                (Slot { getSlot = 1455926300 })
                Close
            )
        )
```

```
    ]
    (Slot { getSlot = 1427673500 })
    Close
```

Código 4.14: Contrato COM4 en Marlowe.

De la misma forma, se generó un contrato PAM con los términos del primer contrato[6] en los tests propuestos por ACTUS [ACTUS, 2019f]. Se omite en este caso el contrato PAM generado en Marlowe debido a la extensión del mismo.

Traducción del contrato a Isabelle

Como se menciono en 3.4, no se ha desarrollado aún ningun tipo de traductor de Haskell a Isabelle/HOL.Esto tuvo como consecuencia la traducción manual de los contratos generados:

```
theory COM
imports Semantics StaticAnalysis
begin

definition party :: Party where
"party = Role [0]"

definition counterparty :: Party where
"counterparty = Role [1]"

definition token_ada :: Token where
"token_ada = Token [] []"

definition COM1 :: Contract where
"COM1 = When
            [ Case
                (Deposit
                    party
                    party
                    token_ada
                    (Constant 1400)
                )
            ( Pay party
```

[6] Disponible en https://github.com/actusfrf/actus-tests/blob/master/tests/actus-tests-pam.json

```
                ( Party counterparty )
                token_ada
                ( Constant 1400 ) Close
            )
            ]
            1427673500
            Close"

[...]

definition COM4 :: Contract where
"COM4 = When
    [Case
        ( Deposit
            party
            party
            token_ada
            ( Constant 2100 )
        )
        (Pay
            party
            (Party counterparty)
            token_ada
            (Constant 2100)
            ( When
                [ Case
                    ( Deposit
                        counterparty
                        counterparty
                        token_ada
                        (NegValue (Constant (-2700)))
                    )
                    (Pay
                        counterparty
                        (Party party)
                        token_ada
                        (NegValue (Constant (-2700)))
                        Close
                    )
                ]
                1455926300
```

```
                Close
            )
        )
    ]
    1427673500
    Close"
```

Código 4.15: Teoría COM.

4.3.2. Algunas ejecuciones de prueba

```
definition COMTransactionList :: "Transaction list" where
"COMTransactionList  = [ (| interval = (0, 1),
                            inputs = [IDeposit party party token_ada
                               1400]
                         |)]"

definition COMDepositTransaction :: "Transaction" where
"COMDepositTransaction = (| interval = (0, 1),
                            inputs = [IDeposit party party token_ada
                               1400]
                          |)"

value "computeTransaction COMDepositTransaction (emptyState 0) COM1"

(* Se genero el siguiente TransactionOutput:

"TransactionOutput
  (| txOutWarnings = [],
     txOutPayments = [Payment
                        (Role [0]) (Party (Role [1])) (Token [] [])
                           1400],
     txOutState = (|accounts = [],
                    choices = [],
                    boundValues = [],
                    minSlot = 0|),
     txOutContract = Close |)"
  :: "TransactionOutput"
```

```
*)

(* Concateno dos deposits para poder ver si COM4 genera los dos pagos *)

definition base_env :: "Environment" where
"base_env = (| slotInterval = (0, 0) |)"

definition inputListCOM4 :: "Input list" where
"inputListCOM4 = [(IDeposit party party token_ada 2100),
                  (IDeposit counterparty counterparty token_ada (2700))
                 ]"

value "applyAllInputs base_env (emptyState 0) COM4 inputListCOM4"

(* Se generaron los dos pagos esperados:

"ApplyAllSuccess True []
  [Payment (Role [0]) (Party (Role [1])) (Token [] []) 2100,
   Payment (Role [1]) (Party (Role [0])) (Token [] []) 2700]
  (|accounts = [], choices = [], boundValues = [], minSlot = 0|) Close"
  :: "ApplyAllResult"
*)

value "reduceContractStep base_env (emptyState 0) COM4"

(*
"NotReduced"
  :: "ReduceStepResult"
*)
```

Código 4.16: Algunas ejecuciones al contrato COM.

Con estas ejecuciones se puede ver el funcionamiento del modelo de Marlowe (detallado en 2.2.1). Esto nos podrá ayudar a probar algunas propiedades sobre los contratos.

4.3.3. Prueba de *minSlot* no decreciente en COM

En esta sección detallaremos el proceso de prueba de la siguiente propiedad para el contrato COM4. Cabe destacar que se eligió este contrato en particular debido a que, entre todos los contratos generados a partir de los tests de ACTUS, es el que mayor anidación de constructores posee:

> "Evaluar `applyAllInputs` en el contrato COM4, para cualquiero *environment* y *state* dado, produce un estado con *minSlot* mayor al actual."

Este tipo de prueba evita la 'vuelta al pasado' por parte del contrato, que podrían llevar a caminos de ejecución no deseados.

```
lemma applyAllInputsDoesNotDecreaseMinSlot :
"applyAllInputs env sta COM4 inputList =
    ApplyAllSuccess reduced warnings payments newstate cont ⟹
    reduceContractStep env sta COM4 = NotReduced ⟹
  (minSlot sta) ≤ (minSlot newstate)"
  apply auto
  apply (cases inputList)
  apply auto
  by (smt (z3) ApplyAllResult.distinct(1) (* propuesto por Sledgehammer
     *)
                 ApplyResult.case(1)
                 ApplyResult.case(2)
                 ApplyResult.exhaust
                 COM4_def
                 applyAllLoop_preserves_minSlot
                 applyCases_preserves_minSlot applyInput.simps(1))
  done
```

Código 4.17: Prueba del lema de *minSlot* no decreciente para un contrato COM.

En el mismo se pueden ver distintas técnicas de prueba, algunas ya mencionadas, como *auto*; otras, como *cases*, que permiten separar la prueba según casos en alguna variable (en este caso `inputList`).

También se hizo uso de *Sledgehammer* (detallado en 3.3.4) para completar la prueba, quien sugirió utilizar la ultima linea de la misma.

El estado de prueba en el que se invocó a *Sledgehammer* (el cual se puede visualizar colocando el cursor sobre el segundo *auto*) es el siguiente:

```
proof (prove)
goal (1 subgoal):
 1. ⋀ a list.
       (case applyInput env sta a COM4 of
```

```
         Applied applyWarn newState cont ⇒
           applyAllLoop True env newState cont list
            ([] @
             convertReduceWarnings [] @
             convertApplyWarning applyWarn)
            ([] @ [])
         | ApplyNoMatchError ⇒
             ApplyAllNoMatchError) =
        ApplyAllSuccess reduced warnings payments
         newstate cont ⟹
        reduceContractStep env sta COM4 =
        NotReduced ⟹
        inputList = a # list ⟹
        minSlot sta ≤ minSlot newstate
```

Código 4.18: Estado parcial del lemma de *minSlot* no decreciente para el contrato COM.

4.3.4. Prueba de *minSlot* no decreciente en PAM

En el caso del contrato PAM, una vez que se incluye el módulo *ValidState* y se pone a un estado valido como precondición de la prueba, se logra la siguiente prueba:

```
lemma applyAllLoopDoesNotDecreaseMinSlot :
"validAndPositive_state sta ⟹
  applyAllLoop reduced env sta PAM inp wa pa =
  ApplyAllSuccess reduced warnings payments newstate cont ⟹
  (minSlot sta) ≤ (minSlot newstate)"
    by (simp add: applyAllLoop_preserves_minSlot)
```

Código 4.19: Prueba del lema de *minSlot* para el contrato PAM

La misma demuestra que aplicar el *loop* sobre el contrato PAM no hace decrecer el *minSlot*.

4.4. Probando la ausencia de Warnings en el contrato Auction

En esta sección detallaremos el trabajo realizado para probar la ausencia de *warnings* en el contrato *Auction*(subasta) [IOHK, 2019a].

4.4.1. El contrato *Auction*

Una subasta o remate (*Auction*) es una venta organizada basada en la competencia directa, es decir, a aquel comprador (postor) que pague la mayor cantidad de dinero o de bienes a cambio del producto. El bien subastado se adjudica al postor que más dinero haya ofrecido por él.

Tradicionalmente en la teoría se reconocen dos grandes tipos: la subasta en sobre cerrado (que pueden ser de primer precio o de segundo precio) y la subasta dinámica, que puede ser subasta ascendente (inglesa), descendente (holandesa), o de 'todos pagan' (subasta americana). En este caso se modela una subasta dinámica de tipo ascendente.

En una subasta basada en una *blockchain*, cada postor debe declarar el dinero primero para convertirse en el mejor postor. Los mismos recuperan su dinero cuando se los supera.

Se puede ver en el siguiente diagrama de secuencia, el comportamiento esperado por la subasta:

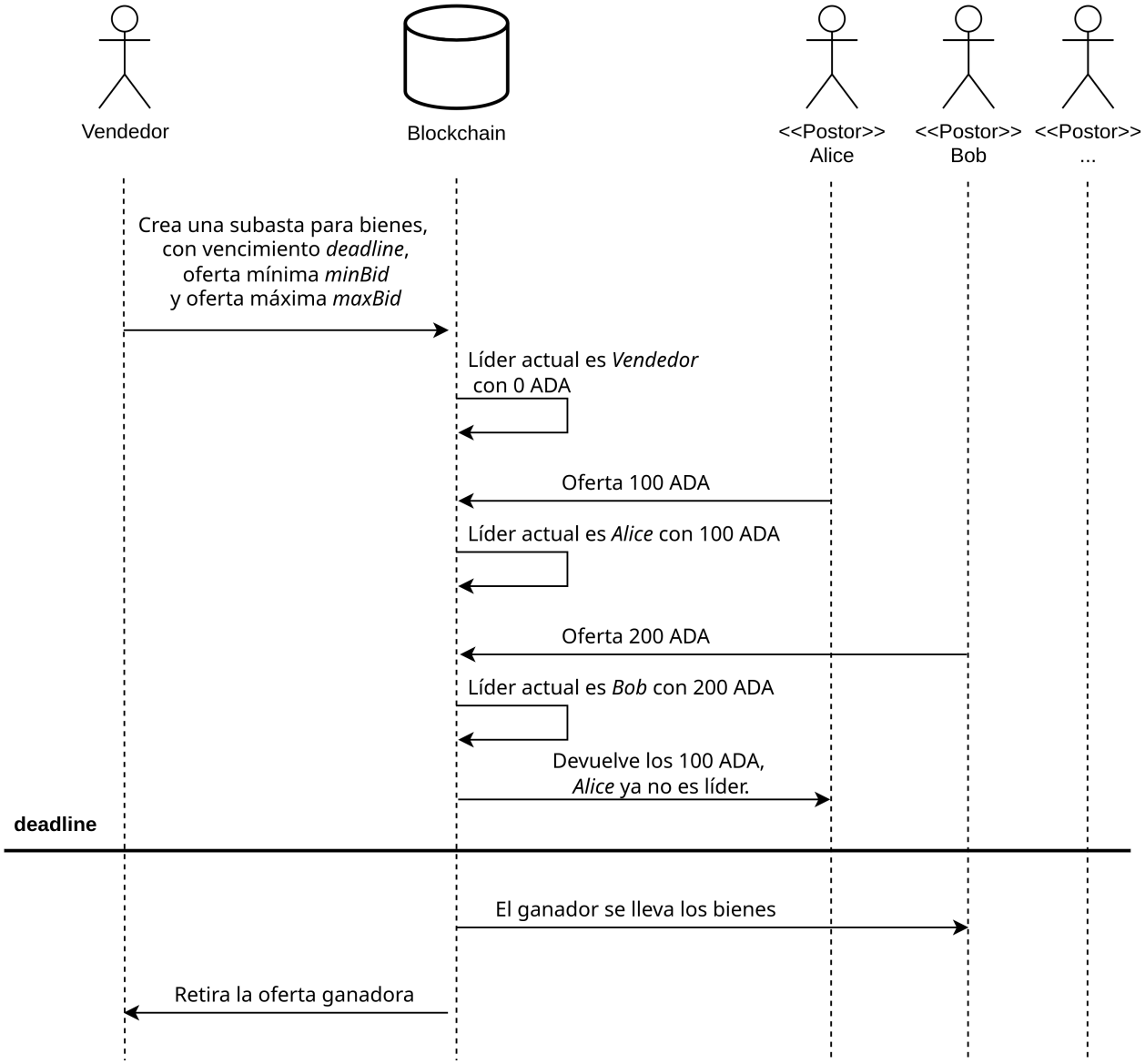


Figura 4.2: Diagrama de secuencia del contrato *Auction*.

Con el fin de mantener la claridad del diagrama. No se muestran casos donde las ofertas son menores a *minBid* o mayores a *maxBid*, las cuales deben ser rechazadas. Tampoco casos en los que se oferta menos del valor actual del líder, en los cuales se devuelve el deposito automáticamente y se mantiene el líder y valor actual.

Con respecto a la implementación de IOHK, el mismo se encuentra escrito en Haskell [IOHK, 2019a], y forma parte de los ejemplos del *Playground*:

```
{-# LANGUAGE OverloadedStrings #-}
module Language.Marlowe.Examples.Auction where

import           Data.Map.Strict  (Map)
import qualified Data.Map.Strict  as M
import           Data.String      (IsString (..) )
import           Language.Marlowe
main :: IO ()
main = print . pretty $ contract "Alice" 100 1000 ["Bob", "Charlie", "
   Dora"] 10
contract :: Party -> Integer -> Integer -> [String] -> Slot -> Contract
contract owner minBid maxBid bidders deadline = go Nothing [] $ map
   fromString bidders
  where
    go :: Maybe (Value, Party) -> [Party] -> [Party] -> Contract
    go m [] [] = settle m
    go m ps qs = When (map choose qs ++ map deposit ps) deadline $
   settle m
      where
        --choose :: Party -> Case
        choose p = Case (Choice (choice p) [Bound minBid maxBid]) $
            Let (value p) (ChoiceValue (choice p)) $
                go m (p : ps) $ filter (/= p) qs
        --deposit :: Party -> Case
        deposit p =
            let v   = UseValue $ value p
                ps' = filter (/= p) ps
            in  Case (Deposit p p ada v) $ case m of
                    Nothing       -> go (Just (v, p)) ps' qs
                    Just (v', p') -> If (ValueGT v v')
                        (go (Just (v, p)) ps' qs)
                        (go m ps' qs)
    settle :: Maybe (Value, Party) -> Contract
    settle Nothing       = Close
    settle (Just (v, p)) = Pay p (Party owner) ada v Close
    choice :: Party -> ChoiceId
    choice = ChoiceId "Bid"
    value :: Party -> ValueId
    value p = head [fromString q | q <- bidders, fromString q == p]
```

Código 4.20: Contrato *Auction* escrito en Marlowe

El mismo posee varias funciones auxiliares, y tres funciones que se llaman mutuamente:

- `go`: Maneja el contrato en general, y se ocupa de llenar los *cases* de un constructor `When` de Marlowe con *deposits* y *choices*. Popula el resto de los parametros del constructor con la información provista por los terminos del contrato.

- `choose`: Recibe una *Party* (que `go` toma de la lista `qs`) y retorna un *Case* populado adecuadamente. Dicha *Party* se remueve de la lista de *choices* (`qs`) y se agrega a la de *deposits* (`ps`). Para popular los campos del `Case`, se llama nuevamente a `go`. Para popular los campos del ttCase, se llama nuevamente a ttgo.
- `deposit`: Si la oferta más alta hasta el momento (`m`) es menor al valor del depósito que se está procesando, se actualiza `m` con el nuevo valor y dueño. También se realiza la actualiza si nadie ha ofertado aún (simbolizado con el valor `Nothing` para `m`). Se remueve a la *Party* de la lista `ps` y se retorna el `Case` correspondiente (que termina de ser populado llamando recursivamente a `go`).

El contrato también tiene algunas funciones auxiliares para poder resolver la finalización. Por ejemplo, `settle` se ocupa de construir el `Pay` desde el ganador de la subasta (en el caso de exista) hacia el dueño (`owner`) de la misma.

Cabe mencionar que el contrato acumuló algunos *deposits* (de las personas que participaron). Todos los mismos (expecto el del ganador), serán devueltos por un `Close` eventualmente.

4.4.2. Traducción del contrato *Auction* a Isabelle

La traducción del contrato en Haskell se realizó manualmente, teniendo en cuenta algunas características y consideraciones teóricas.

fun vs. *function*

El comando `fun` provee una notación abreviada, conveniente para definiciones de funciones simples. En este modo, Isabelle trata de resolver automáticamente todas las obligaciones necesarias en la definición. Si alguna prueba falla, la definición es rechazada. Esto puede significar que la definición en sí es defectuosa, o que los procedimientos utilizados no son suficientes para manejar la definición. Una definición que utiliza *fun* sigue el siguiente formato:

```
fun f :: τ
where
    equations
    .
    .
    .
```

Código 4.21: Definición de una función mediante *fun*

Al expandir la *fun* al commando más detallado *function*, estas obligaciones de prueba se vuelven visibles y se pueden analizar o resolver manualmente. La notación de *function* es la siguiente:

```
function (sequential) f :: τ
where
    equations
    .
    .
    .
by pat_completeness auto
termination by lexicographic_order
```

Código 4.22: Definición de una función mediante *function*

Algunos detalles ahora se han hecho explícitos:

1. La opción *secuential* permite el preprocesamiento con superposicion de patrones[7]. Sin esta opción, las ecuaciones ya deben estar disjuntas y completas.
2. La definición de una función produce una obligación de prueba, que expresa la completitud y la compatibilidad de los patrones. La combinación de los métodos *pat_completeness* y *auto* se usa para resolver esta obligación de prueba.
3. Una prueba de terminación finaliza la definición, iniciada por el comando *termination*. Cada vez que falla un comando *fun*, generalmente es una buena idea expandir la sintaxis a la forma de *function*, para ver qué está sucediendo realmente.

Los detalles de la prueba de terminación del contrato *Auction* podrán verse al final de la sección.

Recursión mutua

Como mencionamos anteriormente, las funciones *contractLoop*, *handleChoose* y *handleDeposit* se llaman mutuamente, esto provoca un error al intentar definirlas de forma independiente en Isabelle, debido a que las definiciones se procesan y conocen en orden.

Veamos un ejemplo más sencillo para entender la solución que propone Isabelle en estos casos:

```
function even :: "nat ⇒ bool"
and odd :: "nat ⇒ bool"
where
"even 0 = True"
| "odd 0 = False"
| "even (Suc n) = odd n"
```

[7]Los mismos definen el comportamiento de la función ante determinadas entradas, y obedecen reglas de *pattern matching*

```
| "odd (Suc n) = even n"
by pat_completeness auto
```

Código 4.23: Definición de funciones *odd* y *even* mediante recursión mutua.

En este caso, las funciones *even* y *odd* (que reciben un número natural y retornan si es par o impar respectivamente) hacen uso de la recursión mutua.

La defición consta del constructor *function*, seguido por las firmas de ambas funciones (separadas por *and*) y luego por la definición de todas las funciones, para todos los parametros de entrada posibles.

Al utilizar *function*, las obligaciones de la definición se deben probar manualmente.

Para eliminar las dependencias mutuas, Isabelle crea internamente una sola función, operando en el tipo *suma*: *nat* + *nat*[8]. Luego *Functions.even* y *Functions.odd* son definidas como proyecciones. Consecuentemente, la terminación tiene que se probada simultáneamente para ambas funciones, especificando una medida en el tipo especifico:

```
termination
by (relation "measure (λ x. case x of Inl n ⇒ n | Inr n ⇒ n)") auto
```

Código 4.24: Prueba de terminación de las funciones *odd* y *even*.

Podríamos haber usado *lexicographic_order* para la prueba de terminación, que soporta terminación en recursión mutua hasta cierto punto.

Contrato *Auction* en Isabelle

Teniendo en cuenta las siguientes técnicas de traducción:

- Renombramos las funciones *go*, *choose* y *deposit* a *contractLoop*, *handleChoose* y *handleDeposit* respectivamente.
- Los identificadores son números enteros en lugar de *String*, como se mencionó en 3.4.
- Debido a que las funciones *go*, *handleChoose* y *handleDeposit* son mutuamente recursivas 4.4.2, no pueden definirse secuencialmente (debido a que la primer función que se defina no podrá acceder a las siguientes).

[8]La documentación de funciones en Isabelle se pueden encontrar en https://isabelle.in.tum.de/library/Doc/Functions/Functions.html

- A diferencia de Haskell, Isabelle no soporta definiciones anidadas de funciones, por lo que las funciones auxiliares tuvieron que ser extraidas (las variables que obtenia del *scope* se convierten en parametros de las mismas).
- Extrajimos funciones *lambda* para poder facilitar pruebas posteriores (por ejemplo *remove*).
- Reemplazamos el tipo *Maybe* de Haskell con *option* de Isabelle (para la variable `m`).
- Para comprimir la notación, todos los términos del contrato (*owner*, *minBid*, *maxBid* y *deadline*) encapsulamos en una variable de tipo *AuctionTerms* (que es un *record*, y puede contener varios tipos de datos).
- El pattern matching para definir las funciones según los datos de entrada es similar al de Haskell.

podemos ver a continuación la traducción del contrato *Auction* a Isabelle:

```
[...]

definition bid :: ByteString where
"bid = [1]"

definition token_ada :: Token where
"token_ada = Token [] []"

type_synonym AuctionWinner = "(Value × Party) option"

record AuctionTerms = owner :: Party
                      minBid :: int
                      maxBid :: int
                      deadline :: Slot

fun settle :: "AuctionWinner ⇒ AuctionTerms ⇒ Contract" where
  "settle None _ = Close" |
  "settle (Some (v, p)) terms = (Pay p (Party (owner terms))
                                      token_ada v Close)"

fun choice :: "Party ⇒ ChoiceId" where
  "choice p = (ChoiceId bid p)"

(* Los bidders directamente como parties en lugar de String *)
fun partyToValueId :: "Party ⇒ ValueId" where
  "partyToValueId (PubKey pk) = ValueId pk" |
  "partyToValueId (Role r) = ValueId r"

fun remove :: "Party ⇒ Party list ⇒ Party list" where
"remove p ls = filter ((≠) p) ls"

function (sequential) contractLoop :: "AuctionWinner ⇒ Party list ⇒
   Party list ⇒ AuctionTerms ⇒ Contract"
and handleChoose :: "AuctionWinner ⇒ Party list ⇒ Party list ⇒
   AuctionTerms ⇒ Party ⇒ Case"
and handleDeposit :: "AuctionWinner ⇒ Party list ⇒ Party list ⇒
   AuctionTerms ⇒ Party ⇒ Case"
where
"handleChoose m ps qs terms p =
                Case (Choice (choice p) [(minBid terms, maxBid terms)])
                     (Let (partyToValueId p)
```

```
                    (ChoiceValue (choice p))
                    (contractLoop m (p # ps) (remove p qs) terms))" |
"handleDeposit m ps qs terms p =
   (let v = (UseValue (partyToValueId p)) in
    let ps' = (remove p ps) in
    (Case (Deposit p p token_ada v)
          (case m of
              None          ⇒ contractLoop (Some (v, p)) ps' qs terms
            | Some (v', p') ⇒ If (ValueGT v v')
                                (contractLoop (Some (v, p)) ps' qs
                                   terms)
                                (contractLoop m ps' qs terms))))" |

"contractLoop m [] [] terms =
                settle None terms" |
"contractLoop m ps qs terms =
                (When ((map (handleChoose m ps qs terms) qs) @
                       (map (handleDeposit m ps qs terms) ps))
                      (deadline terms)
                      (settle m terms))"
  [...]
```

Código 4.25: Contracto *Auction* traducido a Isabelle.

Se omitieron las definiciones necesarias para las distintas pruebas, así como tambien las pruebas de exclusión mútua, completitud y terminación en *function*.

4.4.3. Probando la terminación del contrato

Dado que HOL es una lógica de funciones totales[9], la terminación es un requerimiento fundamental. Isabelle intenta probar la terminación automáticamente cuando se realiza la definición. Pero en algunas circunstancias es posible que falle, y la misma tiene que ser probada manualmente.

La terminación no solo es una propiedad deseable en los contratos, sino que también le permite a Isabelle utilizar teoremas adicionales, como por ejemplo el de inducción. Esto es de gran importancia a la de probar teoremas que puedan utilizar las funciones.

[9]Una función se conoce como total cuando está definida para todos los valores posibles de entrada.

Probando terminación mediante el uso de *relation*

Como se menciona en la sección 4 de [Bulwahn et al., 2007], el metodo *lexicographic_order* es el más popular en las pruebas de terminación. Puede probar la terminación de una cierta clase de funciones buscando una combinación lexicográfica adecuada de *medidas de tamaño*. Por supuesto, no todas las funciones tienen un argumento de terminación tan simple. Para ellos, podemos especificar la relación de terminación manualmente.

El método *lexicographic_order* falla en el *loop* del contrato *Auction*, porque ninguno de los argumentos disminuye en la llamada recursiva con respecto al ordenamiento de tamaño estándar. Para probar la terminación manualmente, debemos proporcionar una relación fundamentada personalizada.

Además, como las funciones fueron definidas mediante *recursión mutua* 4.4.2, se debe probar la terminación para las funciones *contractLoop*, *handleDeposit* y *handleChoose* en forma conjunta.

Debemos entonces encontrar una medida que sea decreciente (similar a *evalBound* en 3.9), para cada ejecución de nuestras funciones. Teniendo en cuenta los requerimientos, se utilizaron las siguientes propiedades del contrato para la funcion *evalBoundAuction*.

- Las funciones auxiliares (por ejemplo *settle*, *choice* y *remove*) terminan trivialmente.
- La función *contractLoop* termina trivialmente cuando las listas *ps* y *qs* están vacías.
- La función *handleChoose* remueve un elemento p de qs (si se encuentra en la lista) y lo agrega a *ps*.
- La función *handleChoose* es llamada en *contractLoop*, siempre con elementos de *qs*. Consecuentemente siempre se cumple que $p \in qs$, sin embargo, Isabelle no es capaz de determinar dicha propiedad, y nuestra función de medida debe decrecer incluso el caso en el que $p \notin qs$.
- La función *handleDeposit* remueve un elemento p de ps (si se encuentra en la lista).
- La función *handleDeposit* es llamada en *contractLoop*, siempre con elementos de *ps*. Consecuentemente siempre se cumple que $p \in ps$. De forma análoga a lo sucedido con *handleChoose*, Isabelle no es capaz de determinar dicha propiedad, y nuestra función de medida debe decrecer incluso el caso en el que $p \notin ps$.

Teniendo en cuenta la sintaxis de recursión mutua definimos los tipos necesarios y la funcion *evalBoundAuction*, que respeta las propiedades deseadas.

Como Isabelle transforma nuestra definición triple en una funcion de tipo *suma de tres funciones*, el mismo será de la forma $\tau_1 + (\tau_2 + \tau_3)$ (para mas funciones, los tipos de las nuevas definiciones se anidan a derecha).

Para compactar la sintaxis, definimos los tipos especificos para nuestras funciones:

```
type_synonym contractLoopType = "AuctionWinner × Party list × Party list
    × AuctionTerms"
type_synonym handleChooseType = "AuctionWinner × Party list × Party list
    × AuctionTerms × Party"
type_synonym handleDepositType = "AuctionWinner × Party list × Party
   list × AuctionTerms × Party"
```

Código 4.26: Tipos de datos recibidos por la funciones mutuamente recursivas.

Para finalmente definir nuestra función de medida:

```
fun evalBoundAuction :: "(contractLoopType + (handleChooseType +
   handleDepositType)) ⇒ nat" where

"evalBoundAuction (Inl (_, ps, qs, _)) =
        2 * length ps + 4 * length qs + 1" |
"evalBoundAuction (Inr (Inl (_, ps, qs, _, p))) =
        2 * length ps + 4 * length qs + (if p ∈ set qs then 0 else 8)" |
"evalBoundAuction (Inr (Inr (_, ps, qs, _, p))) =
        2 * length ps + 4 * length qs + (if p ∈ set ps then 0 else 8)"
```

Código 4.27: Función que será utilizada por *measure* para probar la terminación del contrato *Auction*.

Como se puede apreciar, utilizamos `Inl` y `Inr`[10] para extraer las diferentes funciones durante el pattern matching. Según el orden en el que las definimos, `(Inl)` correponde a *contractLoop*, `(Inr (Inl))` a *handleChoose* y `(Inr (Inr))` a *handleDeposit*.

Luego de aplicar la función de medida y un *auto* para simplicar (entre otras cosas), nos encontramos con los siguientes *subgoals*:

[10]Definidos, junto con otras funciones para el manejo de estos tipos, en https://isabelle.in.tum.de/library/HOL/HOL/Sum_Type.html

```
proof (prove)
goal (8 subgoals):
 1. ⋀qs p.
       p ∈ set qs ⇒
       3 +
       4 *
       length (filter ((≠) p) qs)
       < 4 * length qs
 2. ⋀qs p.
       p ∉ set qs ⇒
       4 *
       length (filter ((≠) p) qs)
       < 5 + 4 * length qs
 3. ⋀ps p.
       p ∈ set ps ⇒
       Suc (2 *
            length
             (filter ((≠) p) ps))
       < 2 * length ps
 4. ⋀ps p.
       p ∉ set ps ⇒
       2 *
       length (filter ((≠) p) ps)
       < 7 + 2 * length ps
 5. ⋀ps p.
       p ∈ set ps ⇒
       Suc (2 *
            length
             (filter ((≠) p) ps))
       < 2 * length ps
 6. ⋀ps p.
       p ∉ set ps ⇒
       2 *
       length (filter ((≠) p) ps)
       < 7 + 2 * length ps
 7. ⋀ps p.
       p ∈ set ps ⇒
       Suc (2 *
            length
             (filter ((≠) p) ps))
       < 2 * length ps
```

```
8. ⋀ps p.
      p ∉ set ps ⇒
      2 *
      length (filter ((≠) p) ps)
      < 7 + 2 * length ps
```

Código 4.28: Subgoals parciales en la prueba de terminación de *contractLoop* y sus funciones auxiliares.

Todos los subgoals se solucionan observando que la funcion *filter* cumple con las siguientes propiedades:

- Si $p \in ps$, entonces `(filter ((≠) p) ps)` tiene menor longitud que `ps`.
- Si $p \notin ps$, entonces `(filter ((≠) p) ps)` tiene la misma longitud que `ps`.

Probamos dichos *lemmas* de forma auxiliar:

```
lemma removePresentElementReducesSize :
    "p ∈ set ls ⇒ length (filter ((≠) p) ls) < length ls"
  by (simp add: length_filter_less)

lemma removeAbsentElementMantainsSize :
    "p ∉ set ls ⇒ length (filter ((≠) p) ls) = length ls"
  by (metis (mono_tags, lifting) filter_True)
```

Código 4.29: Lemas auxiliares sobre *filter* utilizados para la prueba de terminación.

Para incluir dichos *lemmas* a un probador automático, usamos *using*. Con esto, podemos probar todos los *subgoals*. La prueba de terminación queda completada:

```
termination
  apply (relation "measure (evalBoundAuction)")
  apply auto
  using removePresentElementReducesSize apply fastforce
  using removeAbsentElementMantainsSize apply fastforce
  using removePresentElementReducesSize apply fastforce
  using removeAbsentElementMantainsSize apply fastforce
  using removePresentElementReducesSize apply fastforce
  using removeAbsentElementMantainsSize apply fastforce
```

```
    using removePresentElementReducesSize apply fastforce
    using removeAbsentElementMantainsSize by fastforce
```

Código 4.30: Prueba de terminación de *contractLoop* y sus funciones recursivas.

Todo lo referido a mi implementación y pruebas en el contrato *Auction* puede encontrarse en https://github.com/julianferres/marlowe/blob/master/isabelle/Auction.thy (visto por última vez el 29 de junio de 2022).

4.4.4. Ausencia de *warnings*

Actualmente, el único contrato que posee dicha prueba es el contrato *Close*, la misma se puede encontrar en el archivo `CloseSafe.thy`, en [IOHK, 2019b]. El objetivo del mismo es probar el siguiente teorema:

```
theorem closeIsSafe : "computeTransaction tra sta Close =
    TransactionOutput trec ⟹ txOutWarnings trec = []"
```

Código 4.31: Ausencia de *warnings* para el contrato `Close` en sintaxis de Isabelle/HOL.

Capítulo 5

Conclusión

En esta tesis hemos estudiado cómo validar propiedades de contratos financieros escritos en Marlowe y Haskell para la blockchain Cardano. Para ello ha sido necesario traducir contratos al lenguaje de Isabelle, y en este contexto hemos propuesto distintos pasos y técnicas para la conversión de tipos de dato, funciones y estructuras características de los lenguajes a traducir.

En particular, hemos probado formalmente las siguientes propiedades:

- Terminación de todos los contratos traducidos, entre ellos el contrato *Pay*, *Swap*, algunos contratos COM y PAM y el contrato *Auction*.
- Prueba de *minSlot* no decreciente para los contratos COM y PAM.
- Ausencia de *warnings* para el contrato *Auction*. Este el primer contrato no simple (distinto de `Close`) en el que se ha verificado formalmente la ausencia de *warnings*.

Además, hemos escrito contratos financieros basados en el estándar ACTUS, que ya se encuentran disponibles en la rama principal del repositorio de Marlowe Cardano. Dichos contratos podrán ser ejecutados tanto en Marlowe, como en las demás alternativas de lenguajes ofrecidas por el *Playground* [IOHK, 2021]. En particular, durante el desarrollo de esta tesis hemos escrito:

- Contrato COM(*Commodity*) en 4.1.3: Realiza el seguimiento de los movimientos de commodities como el petroleo, gas o incluso viviendas. Dichas commodities pueden utilizarse servir como garantía subyacente de futuros en materias primas, garantías o simplemente posiciones de activos.
- Contrato CSH(*Cash*) en 4.1.4: Representa tenencias de dinero en efectivo.
- Contrato COM(*Call Money*) en 4.1.5: Modela préstamos que se renuevan, mientras no sea notificado. Una vez realizada, debe devolverse después del período de notificación estipulado.

Dificultades encontradas

Debido a que la escritura de los contratos ACTUS en Haskell sigue en etapa de desarrollo por parte de IOHK, no se factible tratar de traducir los mismos a Isabelle, debido a que sería necesario traducir completamente la estructura del generador

Otra limitación aparente surge de la traducción de los contratos completos desde Haskell a Marlowe (y consecuentemente a Isabelle). Debido a la complejidad de algunos contratos ACTUS y a la naturaleza de Marlowe de anidar sub-contratos, las traducciones suelen tener decenas (y hasta cientos) de niveles de indentación.

Esto dificulta no solo la lectura, sino la traducción manual a Isabelle de decenas de términos del contrato, haciendo compleja extraer propiedades para posteriores verificaciones.

Líneas de trabajo futuro

Con respecto a la escritura de contratos ACTUS en Cardano, actualmente no se han terminado de escribir todos los contratos especificados por el estándar.

En cuanto a la verificación de programas, es posible tomar algunos contratos (por ejemplo entre los ejemplos del *Playground*) en los que no se esperan warnings y verificarlos mediante Isabelle. Dichas verificaciones podrán beneficiarse de los lemas probados previamente para el contrato *Auction* y el contrato *Close*.

Una última linea de trabajo podría abarcar la traducción automática de algunas sentencias de código fuente Haskell a Isabelle. Parte de esta idea es abarcada en [Torrini et al., 2007], aunque manteniendo la traducción manual.

Bibliografía

[ACTUS, 2018] ACTUS (2018). Actus technical specification. https://www.actusfrf.org/techspecs.

[ACTUS, 2019a] ACTUS (2019a). Actus dictionary events. https://github.com/actusfrf/actus-dictionary/blob/master/actus-dictionary-event-types.json. Online; accessed 17 April 2022.

[ACTUS, 2019b] ACTUS (2019b). Actus dictionary states. https://github.com/actusfrf/actus-dictionary/blob/master/actus-dictionary-states.json.

[ACTUS, 2019c] ACTUS (2019c). Actus dictionary terms. https://github.com/actusfrf/actus-dictionary/blob/master/actus-dictionary-terms.json.

[ACTUS, 2019d] ACTUS (2019d). Actus dictionary terms. https://github.com/actusfrf/actus-dictionary/blob/master/actus-dictionary.json. Online; accessed 26 May 2022.

[ACTUS, 2019e] ACTUS (2019e). Actus taxonomy. https://www.actusfrf.org/taxonomy.

[ACTUS, 2019f] ACTUS (2019f). Actus test cases. https://github.com/actusfrf/actus-tests/.

[Bahr et al., 2015] Bahr, P., Berthold, J., and Elsman, M. (2015). Certified symbolic management of financial multi-party contracts. *ACM SIGPLAN Notices*, 50(9):315–327.

[Blanchette et al., 2011] Blanchette, J. C., Bulwahn, L., and Nipkow, T. (2011). Automatic proof and disproof in isabelle/hol. In *International Symposium on Frontiers of Combining Systems*, pages 12–27. Springer.

[Blanchette and Krauss, 2011] Blanchette, J. C. and Krauss, A. (2011). Monotonicity inference for higher-order formulas. *Journal of Automated Reasoning*, 47(4):369–398.

[Böhme and Nipkow, 2010] Böhme, S. and Nipkow, T. (2010). Sledgehammer: Judgement day. In Giesl, J. and Hähnle, R., editors, *Automated Reasoning*, pages 107–121, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Brünjes and Gabbay, 2020] Brünjes, L. and Gabbay, M. J. (2020). Utxo- vs account-based smart contract blockchain programming paradigms. *CoRR*, abs/2003.14271.

[Brünjes and Vinogradova, 2019] Brünjes, L. and Vinogradova, P. (2019). *Plutus: Writing reliable smart contracts*. IOHK.

[Bulwahn et al., 2007] Bulwahn, L., Krauss, A., and Nipkow, T. (2007). Finding lexicographic orders for termination proofs in isabelle/hol. In *International Conference on Theorem Proving in Higher Order Logics*, pages 38–53. Springer.

[Cardano, 2019] Cardano (2019). Cardano official website. https://cardano.org/.

[Claessen and Hughes, 2000] Claessen, K. and Hughes, J. (2000). Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA. Association for Computing Machinery.

[Corduan et al., 2019] Corduan, J., Vinogradova, P., and Gudemann, M. (2019). A formal specification of the cardano ledger.

[de Moura and Bjørner, 2008] de Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Fowler, 2010] Fowler, M. (2010). *Domain-specific languages*. Pearson Education.

[Harrison, 2003] Harrison, J. (2003). Formal verification at intel. In *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.*, pages 45–54.

[Harrison et al., 2014] Harrison, J., Urban, J., and Wiedijk, F. (2014). History of interactive theorem proving. In *Computational Logic*, volume 9, pages 135–214.

[HaskellWiki, 2020] HaskellWiki (2020). Algebraic data type. https://wiki.haskell.org/Algebraic_data_type. Online; accessed 30 April 2022.

[IOHK, 2015] IOHK (2015). Documentation for the cardano ecosystem. https://docs.cardano.org/.

[IOHK, 2016] IOHK (2016). Marlowe cardano repository. https://github.com/input-output-hk/marlowe-cardano. Online; accessed 26 May 2022.

[IOHK, 2019a] IOHK (2019a). Auction contract, written in haskell for the cardano blockchain. https://github.com/input-output-hk/marlowe/blob/master/src/Language/Marlowe/Examples/Auction.hs.

[IOHK, 2019b] IOHK (2019b). Marlowe's proofs in isabelle, github repository. https://github.com/input-output-hk/marlowe/tree/master/isabelle.

[IOHK, 2020] IOHK (2020). Marlowe-actus github repository. https://github.com/input-output-hk/marlowe-cardano/tree/main/marlowe-actus/src/.

[IOHK, 2021] IOHK (2021). Playground for the simulations of contracts written in marlowe. https://play.marlowe-finance.io/doc/marlowe/tutorials/escrow-ex.html.

[Kiayias et al., 2017] Kiayias, A., Russell, A., David, B., and Oliynykov, R. (2017). Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In *Advances in Cryptology – CRYPTO 2017*, volume 10401, pages 357–388. Springer International Publishing, Cham. Series Title: Lecture Notes in Computer Science.

[Kondratiuk et al., 2021] Kondratiuk, D., Seijas, P. L., Nemish, A., and Thompson, S. (2021). Standardized crypto-loans on the cardano blockchain. In *5th Workshop on Trusted Smart Contracts, Financial Cryptography and Data Security 2021*.

[Lamela Seijas et al., 2020a] Lamela Seijas, P., Nemish, A., Smith, D., and Thompson, S. (2020a). Marlowe: Implementing and analysing financial contracts on blockchain. In Bernhard, M., Bracciali, A., Camp, L. J., Matsuo, S., Maurushat, A., Rønne, P. B., and Sala, M., editors, *Financial Cryptography and Data Security*, pages 496–511, Cham. Springer International Publishing.

[Lamela Seijas et al., 2020b] Lamela Seijas, P., Smith, D., and Thompson, S. (2020b). Efficient static analysis of marlowe contracts. In *International Symposium on Leveraging Applications of Formal Methods*, pages 161–177. Springer.

[Nair and Dorai, 2021] Nair, P. R. and Dorai, D. R. (2021). Evaluation of performance and security of proof of work and proof of stake using blockchain. In *2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)*, pages 279–283. IEEE.

[Nakamoto, 2008] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. Accessed: 2015-07-01.

[Nipkow, 2013] Nipkow, T. (2013). *Programming and proving in Isabelle/HOL*.

[Paulson, 1997] Paulson, L. C. (1997). Generic automatic proof tools. *arXiv preprint cs/9711106*, pages 23–47.

[Peyton Jones et al., 2000] Peyton Jones, S., Eber, J.-M., and Seward, J. (2000). Composing contracts: an adventure in financial engineering (functional pearl). *ACM SIGPLAN Notices*, 35(9):280–292.

[Ranise and Tinelli, 2006] Ranise, S. and Tinelli, C. (2006). The smt-lib standard: Version 1.2. Technical report, Technical report, Department of Computer Science, The University of Iowa ...

[Russinoff, 1999] Russinoff, D. (1999). A mechanically checked proof of correctness of the amd k5 floating point square root microcode. *Formal Methods in System Design*, 14:75–125.

[Sutcliffe, 2008] Sutcliffe, G. (2008). The cade-21 automated theorem proving system competition. *AI Commun.*, 21(1):71–81.

[Torrini et al., 2007] Torrini, P., Lueth, C., Maeder, C., and Mossakowski, T. (2007). Translating haskell to isabelle. *CADE-21*, page 14.

[Ullman, 1994] Ullman, J. D. (1994). *Elements of ML Programming*. Prentice-Hall, Inc., USA.

[Vashchuk and Shuwar, 2018] Vashchuk, O. and Shuwar, R. (2018). Pros and cons of consensus algorithm proof of stake. difference in the network safety in proof of work and proof of stake. *Electronics and Information Technologies*, 9(9):106–112.

[Wenzel, 2012] Wenzel, M. (2012). Asynchronous proof processing with isabelle/scala and isabelle/jedit. *Electronic Notes in Theoretical Computer Science*, 285:101–114.

[Zahnentferner, 2018] Zahnentferner, J. (2018). Chimeric ledgers: Translating and unifying utxo-based and account-based cryptocurrencies. Cryptology ePrint Archive, Report 2018/262. https://ia.cr/2018/262.

[Zhang and Chan, 2020] Zhang, R. and Chan, W. K. V. (2020). Evaluation of energy consumption in block-chains with proof of work and proof of stake. *Journal of Physics: Conference Series*, 1584(1):012023.