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

16 de mayo de 2022

Índice general

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 este 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

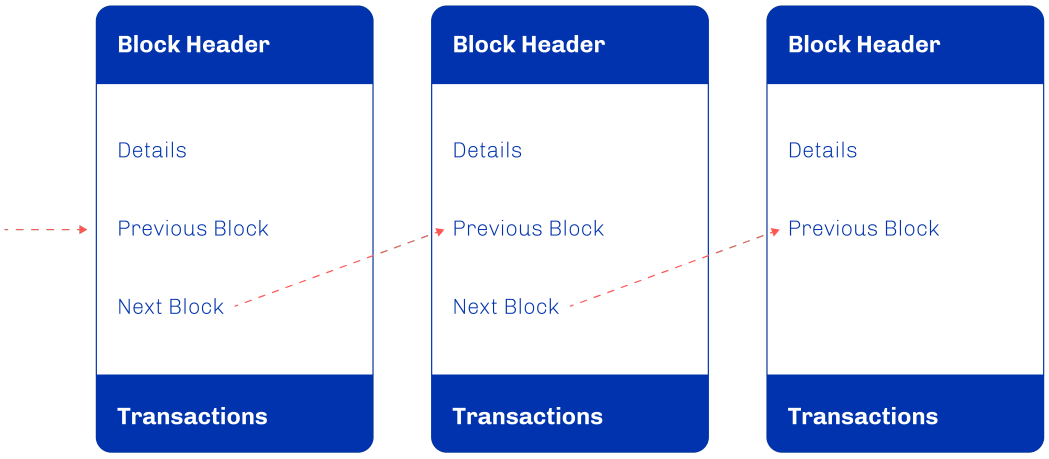

Figura 1.1: Representación simplificada de los datos en un bloque de la cadena. Extraída de [Brünjes and Vinogradova, 2019]

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 (‘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 trackea, verifica y ejecuta las transacciones vinculantes de un contrato entre varias partes. 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 autoejecutables 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, son distribuibles y a prueba de manipulaciones, rápidos y rentables, ya que no hay intermediarios, lo que ahorra dinero y tiempo, y 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

Una criptomoneda es un activo digital, que se almacena en el ledger y está diseñado 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). Además, 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 su página de Wikipedia

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

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

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. En un lenguaje funcional como Haskell, se fomenta la construcción de su sistema 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. $\boxed{1 \text{ ada} = 1{,}000{,}000 \text{ lovelaces}}$. 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.

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.

[2] Ampliamente conocido por su acrónimo en inglés NFT

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 de la red distribuida 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 significativamente 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. Dentro de la cadena se produce una especie de elección de líder. 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 poder 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'.

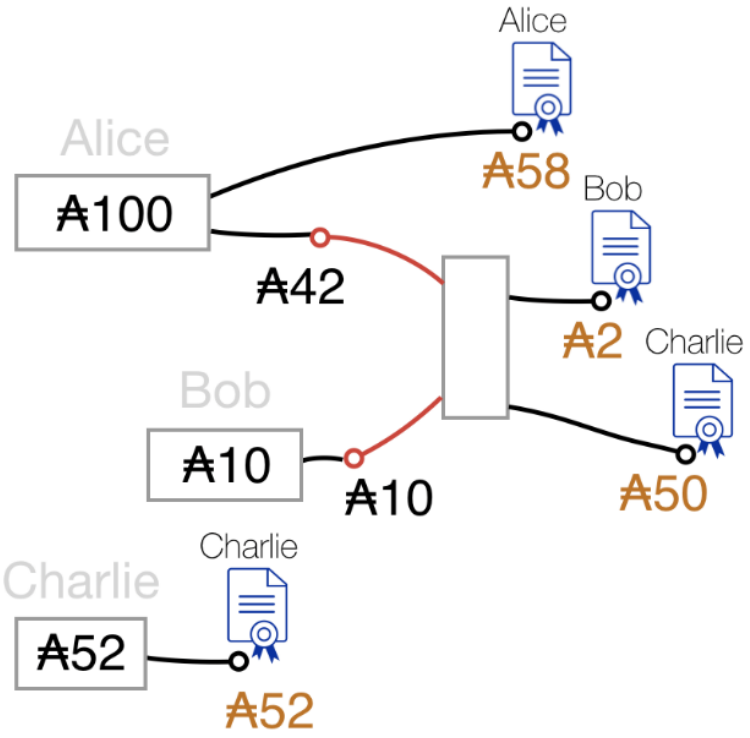

Figura 2.1: Flujo de dinero en el modelo UTXO. Extraído del libro '*Plutus: Writing reliable smart contracts*'

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:

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

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 transaccion 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 direccion, 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 escturctura de la transacción en sí, analicemos primero los 'outpus'. 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 contrable 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 terminos 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 ouputs 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 expícito 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 detras de escena cuando se genera una transacción en la blockchain.

2.2. Marlowe como DSL

Marlowe [Lamela Seijas et al., 2020] [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 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, 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. Contratos 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], con contratos definidos por el tipo de contrato:

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

| **Assert** Observation Contract

Marlowe tiene seis maneras de construir contratos. Cinco de esos métodos — `Pay`, `Let`, `If`, `When`, and `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**: 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**: 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**: 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. 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 montos fijos o variables, 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 recompra, 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, 2019d] 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.

2.3.1. 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 Transicion 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 terminos 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 analoga 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.

2.3.2. 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

PAM: Contract Schedule

Event	Schedule	Comments
AD	$\vec{t}^{AD} = (t_0, t_1, ..., t_n)$	With $t_i, i = 1, 2, ...$ a custom input
IED	$t^{IED} = \texttt{IED}$	
MD	$t^{MD} = \textbf{Tmd}_{t_0}$	
PP	$\vec{t}^{PP} = \begin{cases} \varnothing & \text{if } \texttt{PPEF} = \text{'N'} \\ (\vec{u}, \vec{v}) & \text{else} \end{cases}$ where $\vec{u} = S(s, \texttt{OPCL}, T^{MD})$ $\vec{v} = O^{ev}(\texttt{CID}, \texttt{PP}, t)$	with $s = \begin{cases} \varnothing & \text{if } \texttt{OPANX} = \varnothing \wedge \texttt{OPCL} = \varnothing \\ \texttt{IED} + \texttt{OPCL} & \text{else if } \texttt{OPANX} = \varnothing \\ \texttt{OPANX} & \text{else} \end{cases}$
PY	$\vec{t}^{PY} = \begin{cases} \varnothing & \text{if } \texttt{PYTP} = \text{'O'} \\ \vec{t}^{PP} & \text{else} \end{cases}$	
FP	$\vec{t}^{FP} = \begin{cases} \varnothing & \text{if } \texttt{FER} = \varnothing \vee \texttt{FER} = 0 \\ S(s, \texttt{FECL}, T^{MD}) & \text{else} \end{cases}$	with $s = \begin{cases} \varnothing & \text{if } \texttt{FEANX} = \varnothing \wedge \texttt{FECL} = \varnothing \\ \texttt{IED} + \texttt{FECL} & \text{else if } \texttt{FEANX} = \varnothing \\ \texttt{FEANX} & \text{else} \end{cases}$
PRD	$t^{PRD} = \texttt{PRD}$	
TD	$t^{TD} = \texttt{TD}$	
IP	$\vec{t}^{IP} = \begin{cases} \varnothing & \text{if } \texttt{IPNR} = \varnothing \\ S(\ldots, T^{MD}) & \text{else} \end{cases}$	with

Figura 2.2: Cronograma del contrato PAM para algunos eventos. Extraído de [ACTUS, 2018]

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:

PAM: State Variables Initialization

State	Initialization per t_0	Comments
Tmd	$\mathbf{Tmd}_{t_0} = \texttt{MD}$	
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	$\mathbf{Ipnr}_{t_0} = \begin{cases} 0.0 & \text{if} \quad \texttt{IED} > t_0 \\ \texttt{IPNR} & \text{else} \end{cases}$	

Continued on next page

Figura 2.3: Inicialización de algunas variables del contrato PAM.Extraído de [ACTUS, 2018]

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	$O^{rf}(\texttt{CURS}, t)R(\texttt{CNTRL})(-1)(\texttt{NT} + \texttt{PDIED})$	$\mathbf{Nt}_{t+} = R(\texttt{CNTRL})\texttt{NT}$ $\mathbf{Ipnr}_{t+} = \begin{cases} 0.0 & \text{if} \quad \texttt{IPNR} = \varnothing \\ \texttt{IPNR} & \text{else} \end{cases}$ $\mathbf{Ipac}_{t+} = \begin{cases} \texttt{IPAC} & \text{if} \quad \texttt{IPAC} \neq \varnothing \\ y\mathbf{Nt}_{t+}\mathbf{Ipnr}_{t+} & \text{if} \quad \texttt{IPANX} \neq \varnothing \wedge \texttt{IPANX} < t \\ 0.0 & \text{else} \end{cases}$ $\mathbf{Sd}_{t+} = t$ with $y = Y(\texttt{IPANX}, t)$
MD	$O^{rf}(\texttt{CURS}, t)(\mathbf{Nsc}_{t-}\mathbf{Nt}_{t-} + \mathbf{Isc}_{t-}\mathbf{Ipac}_{t-} + \mathbf{Feac}_{t-})$	$\mathbf{Nt}_{t+} = 0.0$ $\mathbf{Ipac}_{t+} = 0.0$ $\mathbf{Feac}_{t+} = 0.0$ $\mathbf{Sd}_{t+} = t$
PP	$O^{rf}(\texttt{CURS}, t)f(O^{ev}(\texttt{CID}, \texttt{PP}, t))$	$\mathbf{Ipac}_{t+} = \mathbf{Ipac}_{t-} + Y(\mathbf{Sd}_{t-}, t)\mathbf{Ipnr}_{t-}\mathbf{Nt}_{t-}$ $\mathbf{Fac}_{t+} = \begin{cases} \mathbf{Fac}_{t-} + Y(\mathbf{Sd}_{t-}, t)\mathbf{Nt}_{t-}\texttt{FER} & \text{if} \quad \texttt{FEB} = \text{'N'} \\ \frac{Y(t^{FP-}, t)}{Y(t^{FP-}, t^{FP+})} R(\texttt{CNTRL})\texttt{FER} & \text{else} \end{cases}$ $\mathbf{Nt}_{t+} = f(O^{ev}(\texttt{CID}, \texttt{PP}, t))$

Figura 2.4: Funciones de cambio de estado y pago del contrato PAM.Extraído de [ACTUS, 2018]

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, metodologías

3.2. Verificación formal

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.

3.2.1. 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].

3.3. 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 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 coon 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.

3.3.1. 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.

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.

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 cierto tipo de

búsqueda está involucrada, no solo son más lentos que el simplificador y *auto*, los mismos no brindan ninguna pista cuando no logran demostrar el objetivo.

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 basado 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 3.3.2.

3.3.2. 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 (entre los que me incluyo) 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. (Esto se debía a que el kernel de inferencia de Isabelle vuelve a verificar las demostraciones, pero la solidez no era crucial). 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 iterativamente 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 combinador 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[1], E, SPASS, y Z3 son ejecutados local-

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

mente, 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 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.

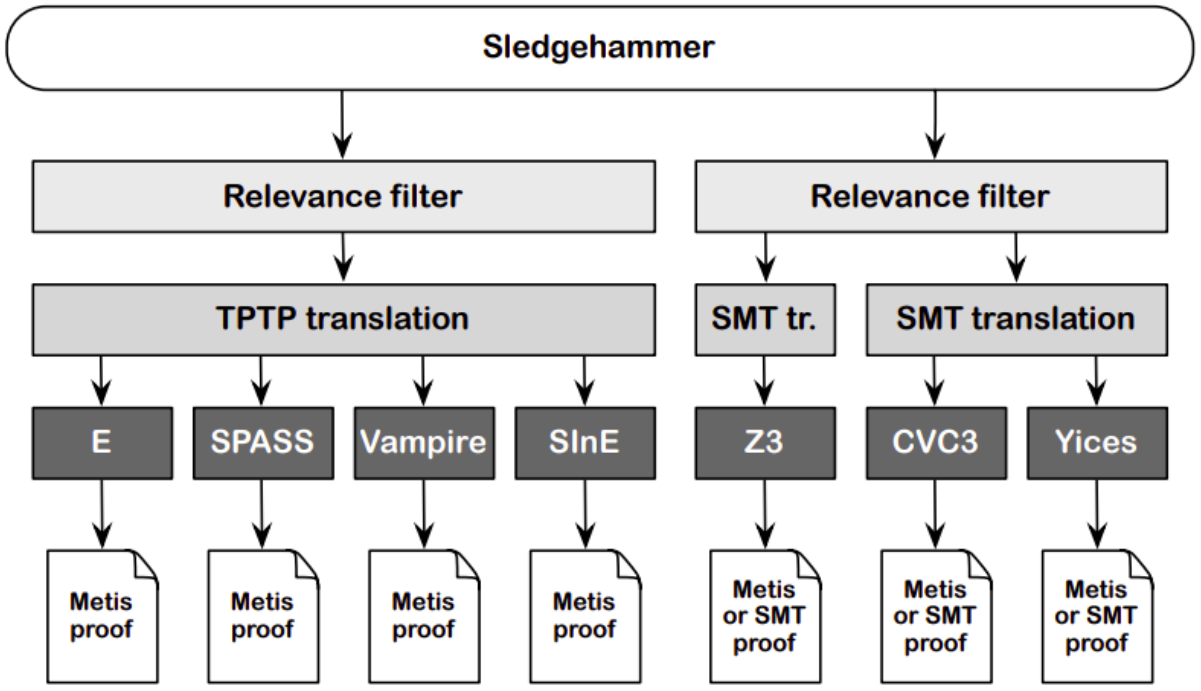

Figura 3.1: Arquitectura de Sledgehammer. Extraída de [Blanchette et al., 2011]

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 procesador, 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].

Capítulo 4

Desarrollo: Verificación de contratos financieros usando Isabelle

4.1. Escritura de contratos ACTUS para Cardano

4.2. sss

4.3. sss

Capítulo 5

Conclusión

Capítulo 6

Apéndice

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 taxonomy. https://www.actusfrf.org/taxonomy.

[Blanchette et al., 2011] Blanchette, J., Bulwahn, L., and Nipkow, T. (2011). Automatic proof and disproof in isabelle/hol. pages 12–27.

[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.

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

[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*, 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/.

[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., 2020] Lamela Seijas, P., Nemish, A., Smith, D., and Thompson, S. (2020). 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.

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

[Paulson, 1997] Paulson, L. C. (1997). Generic automatic proof tools. pages 23–47.

[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.

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

[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.