Especificación de Vyper en Maude
Specification of Vyper in Maude

Trabajo de Fin de Máster
Curso 2022–2023

Autor
Adrián Burillo Elmaleh

Director
Adrián Riesco Rodríguez

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Especificación de Vyper en Maude
Specification of Vyper in Maude

Trabajo de Fin de Máster en Ingeniería Informática
Departamento de Sistemas Informáticos y Computación

Autor
Adrián Burillo Elmaleh

Director
Adrián Riesco Rodríguez

Convocatoria: *Septiembre* 2023
Calificación: *5*

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

29 de Septimebre de 2023

Resumen

Especificación de Vyper en Maude

En este trabajo se ha utilizado el lenguaje Maude para especificar la sintaxis de Vyper y desarrollar un sistema de entrada/salida, en el que el usuario elige los contratos y las funciones a ejecutar con el objetivo de devolverle la salida de cada ejecución y el estado del contrato en ese momento. El desarrollo del proyecto se puede dividir en tres partes.

En la primera parte se ha desarrollado una primera forma de la gramática de Vyper que comprueba que cada línea de código tenga una forma correcta. Para poder desarrollar esta primera gramática, se ha utilizado la documentación de Vyper y se han tomado varios contratos de ejemplo donde se han podido ver distintas casuísticas que no se deducían de la documentación.

Tras recibir un contrato se ejecuta un preprocesamiento encargado de transformar distintas partes del código de Vyper con el objetivo de facilitar el diseño de la representación interna, que por la forma de programar en Maude daba lugar a distintos problemas. A continuación, el contrato se pasa por una función propia de Maude para darle forma uniforme y por otra función de implementación propia que corrige su forma y lo dispone para la fase siguiente.

En la segunda parte, se ha definido una segunda gramática de Vyper y se han desarrollado distintas estructuras de datos para almacenar variables y representarlas como se verían en Vyper. Para mantener una imagen del contrato durante la ejecución se ha desarrollado los conceptos de pila de memorias y memoria de funciones: el primero guarda una imagen del programa y almacena todas las variables simples y estructuras complejas, y la segunda guarda la definición de todas las funciones ejecutables dentro del contrato.

En la tercera parte se han definido reglas de reescritura para especificar la semántica del lenguaje. Asimismo, se ha definido un módulo funcional para conectar las reglas con las estructuras de datos y conseguir inicializar, introducir, extraer y comprobar condiciones útiles para la correcta ejecución del contrato.

Por último, se ha definido un módulo de sistema para especificar la entrada/salida para que el usuario interactúe con el programa e indicar qué contrato se quiere cargar, las funciones a ejecutar y los parámetros de estas. Por cada función ejecutada se muestra al usuario si la salida es correcta y la imagen del programa en ese momento.

Palabras clave

Vyper, Maude, Contrato inteligente, Especificación, Cadena de bloques, Meta-nivel, Ethereum, Sintaxis

Abstract

Specification of Vyper in Maude

In this work, the Maude language has been used to specify the syntax of Vyper and to develop an input/output system, in which the user chooses the contracts and the functions to be executed with the aim of returning the output of each execution and the state of the contract at that moment. The development of the project can be divided into three parts. In the first part, a first form of the Vyper grammar was developed, which checks that each line of code has a correct form. In order to develop this first grammar, the Vyper documentation has been used, and several example contracts have been taken where different cases that could not be deduced from the documentation could be seen. After receiving a contract, preprocessing is carried out to transform different parts of the Vyper code in order to facilitate the design of the internal representation, which, due to the way Maude was programmed, gave rise to various problems. The contract is then passed through Maude's own function to give it a uniform form and through another implementation function that corrects its form and prepares it for the next phase.

In the second part, a second Vyper grammar has been defined and different data structures have been developed to store variables and represent them as they would be seen in Vyper. To maintain an image of the contract during execution, the concepts of memory stack and function memory have been developed: the former stores an image of the program and stores all simple variables and complex structures, and the latter stores the definition of all executable functions within the contract.

In the third part, rewrite rules have been defined to specify the semantics of the language. A functional module has also been defined to connect the rules with the data structures and to initialize, introduce, extract and check conditions useful for the correct execution of the contract.

Finally, a system module has been defined to specify the input/output for the user to interact with the program and to indicate which contract is to be loaded, the functions to be executed and their parameters. For each function executed, the user is shown whether the output is correct and the image of the program at that moment.

Keywords

Vyper, Maude, Smart Contract, Specification, Blockchain, Meta-level, Ethereum, Syntax

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este capítulo presentamos los conceptos de cadena de bloques,[1] Ethereum y contrato inteligente, dando una visión general que ayudará a entender la motivación, objetivos y el desarrollo de este trabajo de fin de máster.

La *blockchain* (Guardeño et al.) se puede entender como una base de datos distribuida, sincronizada y segura, donde cada registro corresponde a un bloque y es necesario la existencia de nodos (usuarios) para verificar y validar distintas transacciones. Estas características hacen posible la eliminación de intermediarios en las transacciones realizadas, haciendo que la *blockchain* haya ganado mucha popularidad en los últimos años y plantee una revolución en distintos ámbitos, siendo la economía uno de los más relevantes.

Los bloques de la cadena (bybit learn) registran las transacciones en orden cronológico e irreversible. La concatenación de bloques se realiza gracias a procesos criptográficos, lo que garantiza la posibilidad de añadir nuevos bloques sin que los anteriores sean sustituidos o modificados. Con el paso del tiempo, la red será cada vez mayor. El uso de funciones *hash*, un algoritmo criptográfico que trasforma los datos de entrada en cadenas de longitud fija, permite autenticar las transacciones existentes, ya que no se puede duplicar esta cadena. Cada bloque se identifica con la información de la cabecera que se compone del número de la versión de la *blockchain*, el tiempo de Unix, los punteros de *hash*, el valor necesario de los mineros para crear un bloque (*nonce*)(explicado a continuación) y el *hash* de una raíz Merkle, que es una estructura dividida en capas para facilitar la verificación de grandes cantidades de datos gracias al nodo raíz. Este puntero es el que marca la dirección del siguiente bloque, qué funcionaria como una lista enlazada donde cada puntero marca la dirección correcta a otras variables.

En relación con la *blockchain* es importante mencionar el algoritmo de la prueba de trabajo (*proof of work*) (Guardeño et al.) que se utiliza para confirmar transacciones y generar nuevos bloques. Este algoritmo exige que un bloque se añada a la red solo si se obtiene el valor aleatorio (*nonce*, número utilizado solo una vez, de 4 bytes) que concatenado con la cabecera del bloque de lugar a un resumen que tenga un cierto número de ceros a la izquierda. Esta búsqueda de ceros a la izquierda es un problema de alto coste computacional.

La *blockchain* es una red *Peer to Peer* (P2P), donde todos los nodos que la componen tienen los mismos derechos y ejecutan las mismas funciones. Esto es lo que garantiza que todos los participantes operen bajo las mismas reglas, asegurando así que una operación se

[1] En el resto de la presente memoria usaremos el término inglés *blockchain* para referirnos a la cadena de bloques, ya que es el término más aceptado.

completa satisfactoriamente y que no existan datos inconsistentes. La tecnología *blockchain* crea *tokens*, que podemos entender como "monedas" y que, aunque no tienen valor por sí mismos, pueden adquirirlo dependiendo de quién lo utilice y con qué fin.

Ethereum (Wackerow et al., 2022) es una plataforma digital que adopta la tecnología *blockchain* y extiende su uso a gran variedad de aplicaciones. El ether es su criptomoneda nativa, es decir, es el token usado en las transacciones. Los datos que se almacenen en la red tiene que ser verificados por, al menos, el 51 % de los nodos. Cada transacción utiliza una cantidad de gas, que es la unidad de medida de cualquier trabajo ejecutado dentro de la red. Por tanto, cuanto más complejo sea el trabajo, más gas consume.

Ethereum ha sido diseñado para que las transacciones que se realicen cumplan distintas condiciones. Estas condiciones vienen definidas en los *contratos inteligentes* (Arvelo). Los contratos inteligentes son contratos que tienen la capacidad de ejecutarse de forma automática. Las diferencias principales con los contratos tradicionales en papel son que son programas informáticos, su cumplimiento no está sujeto a las interpretaciones de ninguna de las partes y no se requiere de ningún intermediario, como por ejemplo una notaría.

La primera idea de contrato inteligente data de los años 90 (Kemmoe et al., 2020), pero en esta década no estaba tan avanzada la tecnología como para asegurar que estos contratos fuesen seguros. Existían problemas como la manipulación del software implementando el contrato escrito, con lo que no se podía asegurar su seguridad y la fuerte regulación que tiene el dinero.

Es la *blockchain* la que puede dar solución a estos problemas al ser una base de datos inmutable y mantenida por una red de ordenadores donde se puede registrar cualquier cosa. Nos evita la manipulación informática y además aparece el concepto de criptomoneda que es un activo digital y descentralizado, no perteneciente a ninguna entidad que facilita la trasferencia de dinero.

Dentro de los contratos inteligentes, es fundamental mencionar el papel de los oráculos. Estos son instrumentos que permiten que los contratos puedan interactuar con el mundo real, actualizando estados internos con información del exterior. Estos mecanismos tienen problemas de centralización, que va en contra de la filosofía de *blockchain*, ya que si el oráculo que se está utilizando tiene información errónea o ha sido manipulada o ha fallado el servidor, depende solo de este oráculo centralizado. Para solucionar este tipo de problemas se está optando por combinar resultados de distintos proveedores y tomar la decisión en base a la mayoría y así descentralizar el resultado.

A continuación, presentamos las ventajas y desventajas de los contratos inteligentes:

Ventajas:

- Eliminar los costes de terceras partes.
- Mayor accesibilidad a todo tipo de público por la simplicidad y los bajos costes.
- Mayor seguridad, trasparencia y eficiencia.
- Reduce asimetrías de la información, la falta de conocimiento o confianza de alguna de las partes.
- Externalidades positivas, se crea, comparte y almacena información de las cadenas para su uso.

Desventajas:

- Posibles errores ante los futuros cambios tecnológicos.

- Conseguir que la cadena de bloques consiga entidad legal, esté validada y admitida por todos los intermediarios.
- Necesidad de estandarizar y regular casos de uso, como contratos P2P o B2B, que serían contratos entre particulares(P2P) y contratos entre entidades (B2B) que utilizan *tokens* como método de pago.

Los contratos inteligentes se pueden programar utilizando lenguajes relativamente fáciles para el programador. Dos claros ejemplos de este tipo de lenguajes son Solidity (Solidity) y Vyper (Vyper), que son los más utilizados (ethereum.org). Los desarrolladores más experimentados también usan Yul (Authors).

Solidity es un lenguaje de programación de alto nivel orientado a objetos tipado estáticamente, influenciado por C++, Python y JavaScript. Las características principales que lo definen son la herencia, la existencia de bibliotecas para generar código reutilizable y la definición de tipos complejos definidos por el usuario. A continuación se enumeran las ventajas y desventajas de este lenguaje:

Ventajas:

- Muchos recursos de aprendizaje, al ser el lenguaje más utilizado para Ethereum.
- Muchas herramientas para los desarrolladores, como el editor en línea Remix Remix.
- Admite cadena y matrices de tamaño dinámico.
- Manejo de excepciones.

Desventajas:

- Posibilidad de desbordamiento, si los valores a almacenar son demasiado grandes.
- Escalabilidad, no está preparado para manejar un gran número de transacciones.
- Problemas de seguridad.

El segundo lenguaje más utilizado es **Vyper**. Vyper es un lenguaje de programación "pythonico" desarrollado principalmente para solucionar los problemas de seguridad que existían en Solidity. Vyper ha eliminado algunos conceptos como la orientación a objetos y la herencia. También, ha añadido un tipado fuerte, es decir, no se permiten violaciones de los tipos de datos, no es posible usar un tipo como otro a menos que se haga una conversión. El objetivo de todos estos cambios era crear contratos que fueran auditables y menos propensos a errores.

A continuación se presentan las ventajas y desventajas de este lenguaje:

Ventajas:

- Lenguaje más sencillo para que los desarrolladores de Python comiencen con él.
- Lenguaje transparente para facilitar la seguridad y legibilidad.
- Aumento en las herramientas de desarrollo como Brownie y Etherscan, para desarrollar en línea y probar los contratos inteligentes.
- Limitación de los tamaños de cadena y matriz para evitar los ataques.
- Posibilidad de calcular un límite superior de gas para cualquier llamada de función.

Desventajas:

- No existe tanto apoyo de la comunidad como en Solidity.
- No existen modificadores, herencia, llamadas recursivas y tipos de datos dinámicos.
- Todavía en desarrollo, muchas funciones aún no están en Vyper.

1.1. Motivación

Dado que los contratos inteligentes que contienen errores suponen varios problemas (la inmutabilidad de la cadena nos impide cambiarlos) y conllevan riesgos económicos es necesario garantizar, en la medida de lo posible, que son correctos y se ajustan a ciertos límites de consumo. Para ello es conveniente formalizar su semántica y ser capaz de verificarlos. Esto lo haremos con un lenguaje de especificación como es Maude.

Maude (Clavel et al., 2007) es un lenguaje de especificación basado en lógica de reescritura. Además, Maude es un marco semántico que nos ofrece la posibilidad de crear un lenguaje que represente el lenguaje Vyper para poder comprobar la corrección de un contrato y localizar los posibles errores. El metanivel propio de este lenguaje ha facilitado la representación de instrucciones y bloques de código de Vyper.

Es necesario realizar un preprocesamiento de cada contrato, ajeno a Vyper, debido a distintos errores que aparecían por sintaxis de Maude que entraba en conflicto con la sintaxis de Vyper.

La definición de dos versiones de la sintaxis de Vyper, la primera para qué encaje en forma y orden. La segunda encaja en el tipo y la forma interna por instrucción del contrato.

Creación de una semántica partiendo del resultado anterior, que da funcionalidad a cada instrucción del contrato Vyper. Conectar con las estructuras de datos para almacenar la imagen del programa en cada punto de la ejecución.

Por último, un sistema de entrada/salida para que el usuario elija los contratos y el valor que aportar a cada ejecución.

1.2. Objetivos y plan de trabajo

En este apartado se van a repasar los objetivos desglosados que se plantearon al inicio de este trabajo. También, el plan de trabajo seguido para llegar al objetivo final de conseguir un método que permitiera especificar un contrato inteligente escrito en Vyper.

- **Estudio de la *blockchain* y los contratos inteligentes.** El conocimiento inicial de la *blockchain* y Ethereum era bastante reducido, por eso el primer paso fue un estudio de la definición, fundamentos, ventajas y objetivos de ambas tecnologías. Asimismo, se revisó en profundidad los contratos inteligentes, que es la base de este Trabajo de Fin de Máster.
- **Estudio de los lenguajes de programación.** Respecto a los lenguajes de programación, el conocimiento de Maude era a nivel usuario, por lo que resultaba insuficiente para realizar el trabajo. Un paso fundamental fue reforzar el conocimiento de este lenguaje, revisando otros ejemplos existentes de especificación de otros lenguajes en Maude. En el caso de Vyper, el desconocimiento era absoluto, pero al ser un lenguaje *pyhtonico* la curva de aprendizaje no fue muy pronunciada.

- **Preprocesamiento.** En lo que respecta a la implementación, el primer paso fue el desarrollo de un preprocesamiento para poder adecuar el código Vyper a un código que fuese más fácil de tratar. Este proceso se llevó a cabo en paralelo con otro proceso de lectura del contrato desde Maude y comprobación de la corrección de su forma general.
- **Desarrollo de la sintaxis y estructuras de datos.** El segundo paso fue el desarrollo de una sintaxis propia que nos permitiese trasformar la salida de la fase anterior a una sintaxis que describa los tipos y la forma interna de cada instrucción para poder aplicar las reglas. Además, se crearon las distintas estructuras de datos para almacenar información de un contrato, como las variables y funciones.
- **Desarrollo de la semántica.** Definición de reglas para implementar la semántica del lenguaje. En este apartado se usaron las estructuras de datos creadas anteriormente para imitar la ejecución real de un contrato.
- **Desarrollo de un módulo de I/O.** El módulo que relaciona todas las partes descritas anteriormente e incluye la comunicación con el usuario.

1.3. Estructura de la memoria

En este apartado se va a repasar las partes que forman la memoria, aportando un breve resumen de cada una:

- **Estado de la cuestión:** En este capítulo se explica la tecnología utilizada en el proyecto, Maude y Vyper para facilitar el entendimiento del código y su complejidad. También se expone un contrato real con el que se va a trabajar a lo largo de la memoria. Por último, se habla de trabajos relacionados que han inspirado a este.
- **Descripción del Trabajo:** En este capítulo se explica el trabajo desarrollado, diferenciado en fases ordenadas por su creación en el tiempo y mostrando ejemplos de código que facilitarán el entendimiento de cada parte. También se ha expuesto el proceso de ejecución de un contrato inteligente.
- **Ejecución y pruebas de un contrato:** En este apartado se han definido las instrucciones de Vyper que se han implementado en Maude y se han representado en un cuadro que indica en que contratos aparecen. Además, se ha profundizado en la ejecución del contrato 4, que es con el que se trabaja a lo largo de la memoria, explicando las reglas que desencadenan cada instrucción y su funcionamiento.
- **Conclusiones y Trabajo Futuro:** En este último apartado se han dado las conclusiones tras la finalización del código y las pruebas. Además, se han indicado ideas y que no se han podido realizar de cara a un futuro desarrollo.

1.4. Código

Todo el código y los ejemplos presentados en esta memoria están disponibles en `https://github.com/aburillo/Especificacion-y-Verificacion-de-Vyper-con-Maude`

Capítulo 2

Estado de la Cuestión

El trabajo realizado se basa en dos pilares fundamentales: **Maude** y **Vyper**. En este capítulo presentamos ambos para facilitar la comprensión de la parte práctica y entender la base del trabajo realizado.

2.1. Maude

Maude Clavel et al. (2007, 1996) es un lenguaje declarativo basado en la lógica de reescritura. Asimismo, Maude es un marco semántico donde otros lenguajes pueden ser especificados. Por último, aprovechando que la lógica de reescritura es reflexiva, Maude implementa un potente meta-nivel, donde es posible razonar sobre aspectos definidos en el propio Maude.

A continuación se va a repasar tres partes que son la base de un sistema en Maude, **sintaxis y módulos funcionales**; **reglas y módulos de sistema** ; y el **metanivel** para facilitar el código presentado.

2.1.1. Sintaxis y módulos funcionales

En esta sección se va a explicar las instrucciones de programación básicas con la que construir un módulo, y qué módulos las incluyen.

- **Declaración de tipos y subtipos:**

  A continuación se expone las instrucciones con las que definir tipos en un módulo funcional.

  ```
  sort T .
  sorts T1 T2 T3 .
  subsort T < T1 .
  subsort T1 < T2 < T3 .
  ```

  En este ejemplo se puede observar que mediante la palabra `sort`, se puede crear nuevos tipos. Estos tipos se pueden subrelacionar con otros a través de `subsort`.

  En este caso, tenemos cuatro tipos distintos: el tipo `T` está contenido en `T1` que a su vez está contenido en `T2` y este en `T3`.

- **Operaciones:**

  Se muestran operaciones presentes en módulos funcionales.

```
op s : -> T .
ops s1 s2 : -> T .
op _s3_ : T1 T2 -> T .
op s4 : T1 T2 -> T .
```

  En los dos primeros casos, las operaciones `s`, `s1` y `s2` no reciben ningún argumento y devuelven un tipo `T`.

  En la tercera y cuarta función `s3` y `s4` reciben dos argumentos y devuelven un tipo `T`. El carácter `_` se utiliza para indicar explícitamente dónde deben aparecer los argumentos.

  También es posible sobreescribir operadores ya definidos en módulos estándar de Maude.

  A las funciones se las puede etiquetar con diversos atributos, encerrados entre corchetes al final de la definición. El atributo `ctor` indica que una operación es constructora del tipo resultado.

  También existen los atributos `assoc`, `comm` y `prec` para indicar si es una operación asociativa, conmutativa y su precedencia.

  Ejemplo: `op _x_ : T T -> T2 [ctor assoc comm prec 10] .`

  También se puede indicar la etiqueta `id` para indicar el término identidad. En el siguiente ejemplo, el tipo `T` se puede construir de dos formas, como `x[zero]` o `s`.

```
op zero : -> T1 .
op s : -> T  [ctor] .
op x[_] : T1 -> T [ctor id: s] .
```

- **Declaración de ecuaciones:**

  El comportamiento de las operaciones(no constructoras) se definen mediante ecuaciones. Las transacciones se ejecutan de izquierda a derecha, teniendo múltiples opciones de resolución.

  En este apartado podemos distinguir dos tipos de ecuaciones, las ecuaciones simples (o no condicionales) y las ecuaciones condicionales.

```
op sum : Nat Nat -> Nat [comm] .
op res : Nat Nat -> Nat .
eq sum(N , N1) = N + N1 .
ceq res(N , N1) = if N1 > N
                  then 0
                  else N - N1
                  fi
```

  En este ejemplo aparece la función `sum`, que recibe dos argumentos de tipo `Nat` (el `sort` predefinido para los números naturales) y devuelve otro `Nat`. La ecuación `sum` devuelve los dos números sumados.

En el caso de la operación `res` se define igual que la suma, recibe dos naturales y devuelve otro. Esta ecuación es condicional, pues no devuelve un valor fijo, sino que realiza una comprobación; si el segundo número es mayor que el primero devuelve 0 y si no devuelve la resta.

- **Declaración de variables:**

  Se muestra la declaración de variables de tipos previamente definidos que utilizan las ecuaciones y las reglas.

```
var N: Nat .
vars N1 N2: Nat .
```

  Las variables se declaran con la palabra reservada `var`, un nombre o nombres de variable, dos puntos y el tipo que se le va a asignar a esa variable.

- **Módulo funcional:**

  En el módulo funcional se define los tipos, sus elementos y funciones que trabajan sobre estos tipos, para crear un módulo funcional utilizamos las palabras reservadas `fmod` y `endfm`.

```
fmod nombreMóduloFuncional is
...
endfm
```

  Ejemplo de un módulo funcional completo.

```
fmod NATURAL is
      pr BOOL .
      sort Ntl .
      op zero : -> Ntl [ctor]
      op suc : Ntl -> Ntl [ctor] .
      op sum : Ntl Ntl -> Ntl [comm] .
      op equ : Ntl Ntl -> Bool .
      vars A B : Ntl .
      eq sum(zero, A) = A .
      eq sum(suc(A),B) = suc(sum(A, B)) .
      ceq equ(A,B) = if A =/= B
                     then false
                     else true
                     fi .
endfm
```

  En este módulo inicialmente se ha importado el módulo predefinido `Bool`, para permitir a las ecuaciones utilizar elementos booleanos.

  Se ha declarado el tipo `Ntl` (números naturales), con sus dos operaciones constructoras: `zero` y `suc` (sucesor).

  `zero` es un natural en sí mismo, en cambio, `suc` recibe un natural y devuelve otro valor de tipo natural. Además, se han definido dos operaciones no constructoras, con sus consiguientes ecuaciones: `sum` y `equ`.

La función `sum` es no condicional, por ajuste de patrones se cumple la forma de la izquierda, se infiere en la parte derecha de la regla. En el primer caso, si la suma es `zero` y cualquier natural, devolvería el natural; y en el segundo, con el constructor `suc` de un natural y otro natural cualquiera, se devuelve el sucesor de la suma de ambos.

La segunda función es condicional, pues se pueden inferir varias soluciones a la parte izquierda de la ecuación, en el caso de `equ` recibe dos naturales cualquiera y retorna verdadero o falso según si son iguales o diferentes.

- **Importación de módulos:** Cada uno de los módulos explicados puede importar otros módulos en forma de submódulos, con lo que se consigue cargar al módulo inicial tipos y operaciones definidas en otro lugar del código. Existen tres formas de importar módulos: `protecting`, `extending` e `including`. Las diferencias de ambos están fuera del rango de este trabajo.

2.1.2. Reglas y módulos de sistema

Este tipo de módulos incluyen las reglas, para transicionar sobre el sistema definido.

- **Módulo de sistema:** Este tipo de módulo es de estilo declarativo utilizando la sintaxis definida previamente en los módulos funcionales, se utiliza las palabras reservadas `mod` y `endm` para su declaración.

```
mod NombreModuloSistema is
...
endm
```

- **Reglas:** Las reglas especifican transacciones concurrentes locales donde si el sistema en un punto concreto coincide con el lado izquierdo de la regla, este transaccionaría a la parte derecha modificando el estado. Estas reglas vienen incluidas en los módulos de sistema. Existen dos tipos de reglas; incondicionales y condicionales.

  - **Reglas incondicionales:** Estas reglas describen una transición local, donde solo busca encajar con el estado del sistema.
  - **Reglas condicionales:** Las reglas condicionales pueden tener condiciones muy generales que involucran ecuaciones y otras reescrituras. Las condiciones indicadas se evalúan de derecha a izquierda.

A continuación se va a mostrar un ejemplo de un módulo de sistema.

```
mod ModuloSistema is
     pr NATURAL.
     sort OPERACION .
     subsort Ntl < OPERACION .
     vars A B C : Ntl .
     OP OPER : Ntl Ntl Ntl -> OPERACION [ctor] .
     rl [add] : OPER(A,B,C) => OPER(A,B,sum(A,B)) .
     crl[add2] :  OPER(A,B,C) => OPER(A,B,sum(A,B))
        if sum(A,B) > C.
endm
```

En este módulo, que importa el módulo funcional `NATURAL`, se han definido dos reglas, a partir del tipo `operación` que está compuesto por tres naturales, la primera regla incondicional devuelve los dos primeros naturales sumados en la tercera posición. La segunda regla condicional, comprueba que la suma de los dos naturales sea mayor que el tercero y devuelve esta suma en caso de que se cumpla.

2.1.3. Metanivel

Las teorías especificadas en `Maude` se pueden representar dentro del propio `Maude` aprovechando que la lógica de reescritura es reflexiva. Así, los propios módulos de Maude pueden usarse como datos en otras especificaciones usando el meta-nivel, implementado en el módulo `META-LEVEL`, que define tipos para representar términos (sort `Term`) y módulos (sort `Module`), así como todos los elementos que contienen.

Este módulo define, entre otras, las funciones `upModule`, `upTerm` y `downTerm` que permiten moverse entre niveles de reflexión. Los términos distinguen entre variables, constantes y términos compuestos. Las constantes son identificadores precedidos de una comilla, que contienen el nombre de la constante y su tipo separados por un '`.`', como por ejemplo `'5.Nat`. Las variables contienen su nombre y tipo separados por un '`:`', como por ejemplo `'N:Nat`. Los términos compuestos usan notación prefija, usando el operador que construye el término y una lista de términos entre corchetes y separados por comas como argumentos, como por ejemplo (`'_+_['5.Nat, 'N:Nat]`), que meta-representa `5 + N:Nat`.[1].

Además de definir formas de representar los términos, es posible usar las siguientes funciones

- `metaParse`, que recibe un módulo y una secuencia de `Qid` y trata de interpretar la secuencia como un término en dicho módulo.
- `metaPrettyPrint`, que dados un módulo y un término genera una secuencia de `Qid` correspondiente al término.
- `metaReduce`, que dado un módulo y un término aplica las ecuaciones en el módulo al término.
- `metaRewrite`, que dado un módulo y un término aplica las reglas en el módulo al término.

Encontraremos un tipo especial de término meta-representado en nuestras gramáticas cuando usemos el tipo `@Token@`, el cual encajará, como es habitual en las gramáticas, con un elemento de texto que no corresponda con un separador (como unos paréntesis o una coma) ni a una palabra reservada (que será necesario definir para cada lenguaje). Además de esto, no se requiere que los términos de este tipo estén construidos siguiendo operador alguno, lo que nos permite usar luego operaciones como `metaParse` para comprobar su adecuación a gramáticas más concretas.

2.2. Vyper

`Vyper` (Team) es un lenguaje de programación *pythonico* orientado a contratos, cuya finalidad es dirigirse a la máquina virtual de Ethereum. Los objetivos que plantea `Vyper` son

[1] Nótese el uso del tipo `Qid`, *quoted identifiers*, es decir, identificadores precedidos por una comilla

crear contratos inteligentes, seguros, auditables, con facilidad de lenguaje y del compilador. El código en `Vyper` es sencillo para facilitar su lectura y evitar la aparición de código engañoso, se entiende como código engañoso que pueda tener perdidas de gas o funciones de ejecución infinita que no se reconocen a simple vista.

Para cumplir estos objetivos, este nos ofrece:

- Comprobación de límites y desbordamientos en el acceso a *arrays* y matrices.
- Soporte para enteros con signo y números decimales fijos.
- Decibilidad, para calcular el límite de consumo de gas de cualquier llamada a una función.
- Código pequeño y comprensible para evitar código engañoso y bucles infinitos.
- Los elementos marcados como constantes no pueden cambiar el estado.

Existen diferencias entre Vyper y otros lenguajes orientados a contratos como `Solidity`. A continuación se van a describir las distintas características que no se pueden implementar en `Vyper`, debido a que no se cumplirían los objetivos finales.

- **Modificadores**: En lenguajes como `Solidity` se puede definir una verificación, que aparece en otro lugar del código, que se ejecutará antes y después de una función, esto en `Vyper` no es posible porque sería sencillo crear código engañoso.
- **Herencia de clases**: La herencia obliga al lector a moverse entre archivos y tener clara las reglas de precedencia en caso de conflictos.
- **Sobrecarga de funciones**: Genera mucha confusión sobre qué función se llama en cada momento.
- **Sobrecarga de operadores**: Facilita la existencia de código engañoso al no tener claro que operadores se están ejecutando.
- **Llamadas recursivas y bucles infinitos**: Ambos no permiten establecer un límite superior, por lo que podrían aparecer ataques de límite de gas.
- **Punto fijo binario**: En muchos casos el punto fijo binario requiere aproximaciones, esto genera resultados poco intuitivos.

2.2.1. Estructura de un contrato

Un archivo en `Vyper` contiene exactamente un contrato, a continuación se presentan las partes que lo componen:

1. **Versión Pragma**: Para garantizar que un contrato solo se compile en una versión del compilador en un rango de versiones.
2. **Variables de estado**: Se almacenan permanentemente, se declaran fuera de cualquier cuerpo de función e inicialmente tienen el valor predeterminado de su tipo, a excepción de las constantes.
3. **Estructuras**: Tipo personalizado que permite agrupar varias variables.

4. **Funciones**: Unidades ejecutables de código dentro de un contrato, pueden aceptar argumentos de entrada y devolver variables, además de tener una visibilidad interna o externa.

5. **Eventos**: Proporcionan una interfaz para las funciones de registro de EVM, los eventos se registran con estructuras de datos.

6. **Interfaces**: Proporcionan un conjunto de definiciones de funciones para permitir llamadas entre contratos inteligentes. Al importar una interfaz, el contrato puede llamar a funciones de otros contratos.

2.2.2. Análisis de la sintaxis

En este apartado se trata la sintaxis de un contrato, para facilitar el entendimiento del código explicado en el siguiente apartado.

2.2.2.1. Tipos

`Vyper` al ser un lenguaje estático, el tipo de cada variable tiene que conocerse en tiempo de compilación. Los tipos existentes son:

- **Booleano**: La palabra clave es `bool` y los valores posibles `true` y `false`.
- **Enteros con signo de 256 bits**: La palabra clave es `int256` y los valores posibles entre -2^{255} y $2^{255} - 1$.
- **Enteros sin signo de 256 bits**: La palabra clave es `uint256` y los valores posibles entre 0 y $2^{255} - 1$.
- **Enteros con signo de 128 bits**: La palabra clave es `int128` y los valores posibles entre -2^{127} y $2^{127} - 1$.
- **Decimal**: La palabra clave es `decimal`, este tipo de dato tiene una precisión de 10 decimales entre -2^{127} y $2^{127} - 1$.
- **Dirección**: La palabra clave es `address`, contiene dirección de Ethereum de 20 bytes.
- **Matriz de bytes de 32 bits** : La palabra clave es `bytes32`, contiene dirección de 32 bytes.
- **Matriz de bytes de tamaño fijo** : La palabra clave es `bytes[maxLen]`, contiene dirección del número bytes indicados como longitud.
- **String**: Palabra clave es `string[maxLen]`, contiene una cadena de tamaño como máximo, el indicado como longitud inicial.
- **Listas**: Existen listas de diferentes tipos, declarándolas como `nombre: tipo[longitud]`
- **Estructuras**: Las estructuras pueden agrupar distintas variables, matrices u otras estructuras, pero no declaraciones. Se declaran como:

```
struct MiEstructura:
    variable1: tipo1
    variable2: tipo2
```

- **Hashmap**: Son tablas donde inicialmente existen todos las claves y asignan un valor inicial de cero, se declaran dando el tipo a la clave y al valor. La clave pueden ser cualquier tipo base y el valor puede ser cualquier tipo, incluido estructuras y variables multidimensionales.

  ```
  mapa: hashamp[clave, valor]
  ```

2.2.2.2. Variables y constantes de entorno

En Vyper existen diferentes palabras reservadas para su uso en el contrato sin necesidad de una declaración previa.

- **Variables de entorno**: Siempre existen en el espacio de nombres y se utilizan para proporcionar información de la cadena de bloques y la transacción actual. Son once:

  `block.coinbase, block.difficulty, block.number, block.prevhash, block.timestamp, chain.id, tx.origin, msg.gas, msg.data, msg.sender, msg.value.`

- **Variable self:** Es una variable de entorno utilizada para hacer referencia a un contrato en sí mismo, ya sea a funciones del propio contrato o a variables declaradas estáticamente. Además, `self.` tiene la dirección del contrato actual y `self.balance`, el saldo.

- **Constantes:** `Vyper` tiene algunas constantes incorporadas:

  `ZERO_ADDRESS, EMPTY_BYTES32, MAX_INT128, MIN_INT128, MAX_DECIMAL, MIN_DECIMAL, MAX_UINT256.`

  Se pueden crear contantes personalizadas con el término `constant`:

  ```
  MICONSTANTE: constant(unit256) = 1000
  ```

2.2.2.3. Declaraciones

En declaraciones se incluyen instrucciones predefinidas en Vyper. Las declaraciones se pueden distinguir en tres grupos:

- **Flujo de control:** En este apartado existen cuatro declaraciones de control de flujo diferentes: *break* termina el *for* que lo envuelve más cercano, *continue* pasa la siguiente vuelta del *for* más cercano que lo envuelve, *pass* es una operación nula, al ejecutarla no sucede nada, *return* termina la función y puede retornar un valor o no regresar nada.

- **Registro de eventos:** la declaración `log` se utiliza para declarar eventos que han sido registrados previamente. `log MiEvento(parametros)`

- **Afirmaciones y excepciones:** En este caso existen dos declaraciones, `raise mensaje error` que desencadena una excepción y `assert condición, mensaje error` si la condición se evalúa como cierto se continúa la ejecución, si no la excepción se activa. Cuando una excepción se activa, el código se detiene, el estado del contrato se revierte y el gas restante se devuelve al remitente de la transacción.

2.2.2.4. Estructuras de control:

En este apartado se incluyen las estructuras que tienen su propio cuerpo de instrucciones.

- **Funciones:** Las funciones son unidades ejecutables de código que se declaran dentro del alcance de un contrato. Todas las funciones tienen que incluir un decorador de Visibilidad: `@external` forman parte de la interfaz del contrato y se pueden llamar mediante transacciones o desde otros contratos y `@internal` solo son accesibles desde otras funciones del mismo contrato.

  Las funciones opcionalmente pueden tener un decorador de mutabilidad, hay cuatro niveles: `@pure` la función no lee el estado del contrato y de ninguna variable de entorno, `@view` lee el estado del contrato, pero no lo modifica. `@nonpayable` lee y escribe el estado del contrato, pero no recibe Ether, `@payable` lee y escribe el estado del contrato y recibe Ether. Si la función no tiene ningún decorador de mutabilidad explicito, tiene el valor de `@nonpayable`.

  Existen dos funciones especiales en `Vyper`: `__default__` esta función se ejecuta si se ha llamado a una función del contrato cuyo identificador no se conoce, es una función `@external` y `@payable` que no acepta argumentos ni devuelve valores. La otra función es `__init__` solo se puede llamar a esta función en el momento de implementar el contrato, desde esta función no se puede llamar a otras funciones del contrato.

- **Construcciones IF:** La condición se evalúa de izquierda a derecha, según si es cierta o no se ejecuta un cuerpo de la declaración o se continúa la ejecución sin ejecutar nada. Puede contener declaraciones `elif` y `else`.

```
if condicion:
    ...
elif otra_condicion:
    ...
else:
```

- **Construcciones For:** Utilizadas para iterar sobre un valor. Existen diferentes tipos de construcciones `for`.

  `For` que iteran sobre una lista o una variable que contiene una lista. La lista no se puede modificar mientras se itera y no puede ser multidimensional.

```
for i in [1,2,3]:
    ...
```

```
list: int128[3] = [1,2,3]
for i in lista:
    ...
```

  `for` que iteran sobre la función `range`, esta función acepta dos opciones: Se le puede pasar un entero mayor que 0 e itera de 0 al entero pasado, o se le pasan dos enteros (o dos expresiones que se reducen a un entero) e itera de menor al mayor.

```
for i in range(entero):
    ...
```

```
for i in range(entero1/exp1 ,
               entero2/exp2):
    ...
```

2.2.2.5. Alcance y declaraciones:

- **Declaración de variables**: En todas las declaraciones de variables es necesario indicar el tipo específico, existen cuatro formas de declarar variables:

  1. `var1: tipo` Representa una variable global, no se le puede establecer un valor inicial.
  2. `var2: public(tipo)` Solo se pueden nombrar como públicas las variables globales, indicarla como publica hace que se cree automáticamente un `getter`, como nombre de función sería el nombre de la variable, en caso de ser listas, aceptaría un argumento para retornar solo un valor.
  3. `var3: inmutable(tipo)` Las variables marcadas como inmutables se les debe asignar un valor en el constructor, este no se puede modificar en ninguna otra parte del contrato.
  4. `var4: constant(tipo) = valor` Las variables constantes solo puede ser globales, es obligatorio que tengan un valor inicial que no se puede modificar en ninguna otra parte del contrato.

- **Alcance:** son los lugares donde las variables son alcanzables, se pueden diferenciar en tres tipos:

  1. **Alcance del módulo**: Las variables se declaran fuera de cualquier bloque de código, y son accesibles desde todas las funciones, a través del objeto `self`.
  2. **Alcance de función**: Las variables se declaran dentro de una función o se pasan como parámetros, por lo tanto, estas variables solo se pueden usar en este bloque de función, al terminar de ejecutar la función se eliminarían de la memoria.
  3. **Alcance de bloque**: Las variables se declaran dentro de un `for` o un `if`, y solo tiene validez dentro de estos bloques.

2.2.2.6. Funciones predefinidas:

`Vyper` ofrece una colección de funciones disponibles en el espacio de nombres global de todos los contratos, las funciones se pueden diferenciar en seis categorías:

Operaciones bit a bit, interacción de cadena, criptografía, manejo de datos, matemáticas y utilidades. Las funciones extendidas se encuentran en lala documentación de `Vyper`.

2.2.2.7. Interfaces:

Una interfaz es un conjunto de definiciones de funciones que permiten la comunicación entre contratos inteligentes.

- **Declaración y uso de interfaces:** La agregación de una interfaz se puede realizar desde un archivo independiente o en línea, en ambos casos para usar alguna de las funciones importadas es necesario pasarle como parámetro una dirección de contrato. La palabra `interface` se utiliza para declarar en línea:

```
Declaración:
interface Interfaz:
    def fun1() -> uint128: view
    def fun2(): nonpayable
```

```
Uso:
def fun(direcciom: address):
    Interfaz(direccion).calculate()
```

En el caso de importarlo desde un archivo, es necesario crear una variable del tipo de la interfaz, y asignar a esa variable la dirección del contrato para acceder al contrato:

```
Declaración:
var: Interfaz
```

```
Uso:
 def __init__(direccion: address):
    self.var = Interfaz(direccion)
 def fun1():
    self.var.calculate()
```

Especificar `payable` o `nonpayable` indica que la llamada al contrato externo podrá modificar el almacenamiento, el caso de `view` y `pure` garantiza que no se puede alterar durante la ejecución.

2.2.2.8. Eventos:

En `Vyper` se pueden registrar eventos para ser capturados y mostrados en las interfaces de usuario.

- **Declaración de eventos:** son parecidas a la declaración de estructuras, contienen dos tipos de argumentos: `Argumentos indexados` que los oyentes pueden buscar, pero no se pasan directamente a los oyentes, se marcan con la palabra `indexed` y `Argumentos de valor` que se pasan a los oyentes, puede tener un número ilimitado de argumentos, pero cada uno limitado a 32 bytes. También se pueden crear eventos sin argumentos, con la sentencia `pass`.

```
Evento con argumentos:
event evento1:
    var1: indexed(address)
    var2: indexed(uint128)
    var3: uint256
```

```
Evento vacio:
 event evento2:
       pass
```

- **Registro de eventos:** Con un evento declarado se pueden enviar eventos, tantas veces como se quiera, los eventos enviados no ocupan almacenamiento y no gastan gas. Los eventos no están disponibles para los contratos, solo para los clientes. Registrar(enviar) eventos se realiza mediante la sentencia `log`, el orden y tipos de los argumentos proporcionados deben coincidir con los argumentos declarados.

```
log evento1(msg.sender, var_uint128, var_uint256)
```

- **Escuchar eventos:** Los clientes pueden escuchar y manejar eventos utilizando la biblioteca `web3.js`.

2.2.3. Ejemplo de un contrato real

Con la estructura de un contrato y la sintaxis explicada, en este apartado se va a estudiar un contrato real, para presentarlo en conjunto y que se entienda perfectamente como es un contrato inteligente en `Vyper`.

El contrato seleccionado es una financiación colectiva, en la que los participantes contribuyen con fondos a una campaña, si la contribución total supera el objetivo, los fondos se envían a la campaña y si no se reembolsa a los contribuidores. Además, este contrato es uno de los seleccionados para el procesamiento con Maude, se encuentra en el *Github*, con nombre `contrato4.txt`.

Se puede diferenciar el contrato en dos partes:

- **Declaración de variables de estado**: Se muestran todas las variables de estado, que son comunes a todas las funciones hasta la finalización del contrato. Se va a analizar por partes:

```
struct Funder:
  sender: address
  value: uint256
funders: HashMap[int128, Funder]
nextFunderIndex: int128
beneficiary: address
deadline: public(uint256)
goal: public(uint256)
refundIndex: int128
timelimit: public(uint256)
```

  1. **Estructuras**: Hay una estructura `Funder` con dos variables propias `sender y value`.
  2. **Hashmap**: Hay un hashmap `funders` que tiene como claves números incluidos en int128 y como valores el tipo `Funder` que es una estructura.
  3. **Variables privadas**: Hay tres variables privadas `nextFunderIndex, beneficiary y refundIndex` de tipos propios de Vyper, en el caso de que el usuario quisiese acceder a ellas sería necesario hacer el *getter* a mano.
  4. **Variables públicas**: Hay tres variables públicas `deadline, goal y timelimit` de tipos propios de Vyper. En estas variables el *getter* se genera automáticamente al declararlas públicas.

- **Declaración de funciones**: Este contrato tiene cuatro funciones, que se van a repasar de forma independiente:

  1. **__init__**: Es la función constructora, obligatoriamente tiene que tener el decorador `@external`, en este caso se pasa por parámetros tres valores, que se asignan a tres variables globales y una cuarta variable se calcula con un parámetro pasado y el tiempo del bloque actual `block.timestamp`.

```
@external
def __init__(_beneficiary: address, _goal: uint256, _timelimit: uint256):
    self.beneficiary = _beneficiary
    self.deadline = block.timestamp + _timelimit
    self.timelimit = _timelimit
    self.goal = _goal
```

  2. **Participate**: Esta función es `@external y @payable` por lo que se puede llamar desde fuera, modifica el estado del contrato y gasta Ether. La primera línea que se ejecuta es un `assert` que hace una comprobación si el tiempo actual es menor que la fecha limite, si se cumple continua la ejecución si no salta la excepción.
     La segunda instrucción crea una variable de tipo `int128` solo accesible dentro de esta función. La tercera instrucción crea una nueva clave-valor en la posición guardada en la variable nfi, donde se almacena un tipo `Funder` al que se le pasan dos valores, el identificador del usuario que financia y la cantidad. La cuarta instrucción incrementa el contador de siguiente Financiador, para evitar que se sobreescriba.

```
@external
@payable
def participate():
    assert block.timestamp < self.deadline, "deadline not met (yet)"
    nfi: int128 = self.nextFunderIndex
    self.funders[nfi] = Funder({sender: msg.sender, value: msg.value})
    self.nextFunderIndex = nfi + 1
```

3. **Finalize**: Es una función `@external` que realiza dos comprobaciones mediante la instrucción `assert`, que el tiempo actual sea mayor o igual que el tiempo limite y que lo recaudado sea mayor que la meta. Si ambas condiciones se cumplen, elimina el contrato de la cadena de bloques con la función `selfdestruct`.

```
@external
def finalize():
    assert block.timestamp >= self.deadline, "deadline has passed"
    assert self.balance >= self.goal, "the goal has been reached"
    selfdestruct(self.beneficiary)
```

4. **Refund**: También es una función `@external`, que comprueba si el tiempo actual ha superado la fecha límite, pero no se ha llegado a la meta recaudada.
   Si esto se cumple, crea un `for`, que recorre todos los Financiadores y les devuelve lo donado con la función `send` donde le pasa el identificador y la cantidad. Además, libera cada clave-valor del *hashamp* y actualiza el índice de financiadores existentes.

```
@external
def refund():
    assert block.timestamp >= self.deadline
                        and self.balance < self.goal
    ind: int128 = self.refundIndex
    for i in range(ind, ind + 30):
        if i >= self.nextFunderIndex:
            self.refundIndex = self.nextFunderIndex
            return
        send(self.funders[i].sender, self.funders[i].value)
        self.funders[i] = empty(Funder)
    self.refundIndex = ind + 30
```

2.3. Trabajo relacionado

En este apartado exploramos la relación de trabajos similares con la implementación en este trabajo.

Consideramos en primer lugar el artículo (Meseguer y Rosu, 2007). Este trabajo surge por la búsqueda de una reducción de la brecha entre la parte teórica y práctica dentro de la informática. Es con la lógica de reescritura con la que se intenta reducir esta brecha de la diferencia entre la semántica formal y la implementación de un lenguaje. Incluso aunque ambas estén alineadas, la inexistencia de una semántica a alto nivel o a nivel de hardware crea la necesidad de una solución que elimine esta distancia.

El problema con el que se encontraban es que la semántica formal existente para los lenguajes puede no ser común a todos los compiladores. Esto hace que cada compilador ofrezca diferentes comportamientos para un mismo lenguaje. La falta de una semántica formal afecta además en el análisis de los lenguajes, escondiendo posibles errores de diseño difíciles de localizar.

La semántica especificada en lógica de reescritura tiene como objetivo ser la base de todas las implementaciones de un lenguaje. En la práctica se puede usar para dar definiciones de lenguajes tan complejos como Java y C.

En este trabajo también se nombra la herramienta K que sirve como traductor de `Maude`, para convertir gramáticas libres de contexto a especificaciones en `Maude`. `Maude` es un lenguaje de programación perfecto para este trabajo, ya que como marco semántico facilita no solo la definición, también sirve como lenguaje ejecutable, intérprete y facilitador de análisis de resultados.

Otro trabajo reseñable en este sentido es (Verdejo Lopez, 2003), donde se extiende la idea de `José Meseguer` de una lógica de reescritura y utiliza Maude como un marco semántico ejecutable.

Es importante dar relevancia a los métodos formales, que son la base para especificar y verificar distintos sistemas. Especificar, aporta un conocimiento profundo del sistema que permite derivar distintas propiedades(reglas de inferencia) y comprobar su consistencia.

En este trabajo, también se habla de las varias posibilidades de analizar formalmente una lógica de reescritura debido a que esta tiene una semántica operacional y otra denotacional bien definidas. Al ser una lógica ejecutable, se puede utilizar como lenguaje propiamente dicho. Para esto es muy útil el lenguaje `Maude`, ya que el intérprete utiliza técnicas avanzadas de compilación y los módulos son teorías en las lógicas de reescritura.

También se habla del concepto de `semánticas operacionales estructuradas`, donde presentan de forma unificada distintos aspectos de los lenguajes de programación de una forma lógica como semánticas estáticas, dinámicas, traducciones y reglas que caractericen las distintas expresiones del lenguaje. Dentro de este apartado, se enfoca en `las semánticas de paso largo` donde el predicado inductivo define el valor de un cómputo hasta su ejecución completa. Este tipo de semántica es la que se ha utilizado para el desarrollo de la semántica en `Vyper`.

Otro punto muy importante de este trabajo es la elección otra vez de `Maude` como marco semántico ejecutable y su metanivel. Al describir en `Maude` un lenguaje, se consigue que esta representación sea ejecutable desde dos puntos: como interprete de este lenguaje y como demostrador y corrector mediante reglas semánticas que derivan del lenguaje.

El siguiente trabajo estudiado es (Serbanuta y Rosu, 2010) que introduce la necesidad de crear `K-Maude` explicando previamente K. Es un marco de trabajo para la lógica de reescritura, especializado y optimizado para lenguajes de programación. El uso principal de K, ha sido para la enseñanza y verificación de programas de forma manual, pero como `Maude` no está especialmente optimizado para lenguajes de programación, surge esta idea de superponer K sobre este.

`K-Maude` para el usuario, no es más que una interfaz que internamente conecta módulos de K a módulos de `Maude`. K produce definiciones del lenguaje ejecutables que recoge `Maude` y sirve como interprete para los lenguajes definidos o como base para un análisis. Además, traduce a archivos de Latex que servirán como documentación.

La herramienta K, utiliza una visión abstracta de la sintaxis, donde representa un árbol de etiquetas aplicadas a listas que serían subárboles. Para lograr esto, K es capaz de reducir toda la sintaxis a etiquetas, distinguiendo entre cálculos de valor y sin valor. En el caso de la semántica, K utiliza el término de configuración para resolver el contexto y posteriormente, agrega variables anónimas que remplazará por variables del tipo adecuado y por último las reglas de K se trasforman en reglas y ecuaciones de reescritura.

El siguiente trabajo que se va a analizar es (Hildenbrandt et al., 2018), donde se presenta una introducción a Ethereum, los contratos inteligentes y muestra ejemplos de fallos en distintos contratos que han provocado cuantiosas pérdidas de dinero. Los contratos son

objetos ideales de verificación debido a que son cortos, deterministas y finitos.

Verificar formalmente los contratos, aunque conlleva un esfuerzo extra para el programador, permite reforzar la seguridad de los contratos, algo que hubiese ahorrado gran cantidad de pérdidas por los distintos ataques a contratos debido a la cantidad de dinero que manejan.

En este trabajo se introduce el concepto de `KEVM`, una especificación ejecutable de una máquina virtual que se consigue añadiendo una capa con el marco K (explicado anteriormente) sobre la máquina virtual de Ethereum `EVM`. `K` representa el estado de ejecución de un programa utilizando una configuración, que es una lista desordenada de células (una notación similar a XML). Esto permite al usuario especificar solo las partes necesarias dentro de un contrato.

La configuración se divide en dos partes, una máquina virtual activa para ejecutar transacciones y contratos, y el estado de la red en un momento determinado.

Este trabajo demuestra que `KEVM` proporciona una semántica completa, fiel y eficaz donde cada cambio ejecutado en EVM crea un prototipo en `KEVM`. Además, `KEVM` sirve como plataforma para generar herramientas de análisis y de verificación de contratos. Una de las implementaciones que se verifican es en Vyper con la finalidad de demostrar la modularidad.

Otro artículo a considerar es (Albert et al., 2021) donde se introduce el concepto de gas en Ethereum y el análisis realizado en este trabajo mediante la herramienta *Gastap, Gas-Aware Smart Contract Analysis Platform*. Para entender este estudio, es necesario tener claro que `EVM` tiene tres partes donde almacenar: el estado del contrato, la memoria qué es más barata de usar que el estado, pues se borra en las transacciones, y la pila que realiza operaciones y es de uso gratuito.

`EVM` proporciona una estimación correcta de los límites de gas de las operaciones de bajo nivel, pero los lenguajes como Solidity o Python son de alto nivel, por esto el cálculo de gas y el tiempo de ejecución es una tarea compleja.

El de los límites superiores de gas se basan en cuatro procesos complejos: (1) la construcción de flujos de control y descompilación del código de alto nivel, (2) tratamiento especial de los tipos de datos, su almacenamiento y la gestión de la memoria, (3) generar un modelo de cálculo estático de gastos consumido por el programa, (4) aprovechar las herramientas para calcular los límites de gas para el programa de entrada.

En este trabajo se ha generado una representación basada en reglas de EVM, con un análisis previo de direcciones para calcular los posibles saltos, una descompilación de `EVM` para poder realizar un cálculo estático posterior y poder representar las variables de estado y locales, la pila de operaciones y los datos de la cadena de bloques.

También, se ha creado un modelo de datos que permiten manejar las características concretas de distintas funciones y poder realizar un análisis de gas de las distintas cadenas y bytes mediante un estudio del tamaño de los datos, su visibilidad y el cálculo de los accesos a memoria. Con esto, se puede crear un marco de un análisis del límite superior de gas de cada instrucción de EVM en bytes.

Para inferir el gas que gasta la memoria, se obtiene la dirección más alta de memoria a la que se puede acceder y se calcula esta dirección obteniendo la dirección de memoria a la que se accede y sumándole un pico máximo de acceso. En este trabajo se han analizado más de 300.000 funciones pertenecientes a 30.000 contratos distintos.

El último trabajo relacionado es (Albert et al., 2020). En este trabajo se presenta `Gasol`, una aplicación que permite deducir los gastos de gas que tendrá una instrucción, las veces que una instrucción será ejecutada y permite estimar el gasto de una función dentro de un contrato.

En EVM las instrucciones de almacenamiento son muy costosas, por eso, es obligatorio la medición del consumo de gas de cada operación para pagar en el momento lo que se va a consumir y evitar que los mineros realicen un trabajo sin evaluar. Ver la tarifa de gas, motiva a los usuarios a no consumir demasiado almacenamiento y establece un límite de cantidad de instrucciones a ejecutar, lo que evita ataques `DoS` basados en ejecuciones que no terminan.

La información que ofrece `Gasol` en el análisis de líneas o fragmentos de código, es útil para detectar posibles optimizaciones. El objetivo es conseguir reducir el número de accesos a memoria a los mismos datos globales. Para cada variable, se busca el costo y si se encuentra que el límite de gas es distinto de uno, es que existen múltiples accesos a memoria, lo que sería otro indicativo para aplicar una optimización. Una de las optimizaciones más utilizadas es la creación de una variable local (muy bajo coste de gas de acceso), con el mismo nombre que la variable de estado, e introduciendo `getter` y `setter` para acceder a la variable de almacenamiento.

Existe una nueva tendencia a definir nuevos lenguajes como `Scilla` o `Michelson` que tienen un consumo de gas predecible al no tener ni bloques ni ser recursivos, pero aun así el lenguaje más utilizado es `Solidity` que permite todo esto y también es aplicable a `Vyper` que también ofrece todas estas posibilidades.

Capítulo 3

Desarrollo

En este apartado se describe el trabajo realizado, la especificación de un contrato en Vyper en Maude y su procesamiento que será solicitado al usuario. Se usará el contrato explicado en el capítulo anterior para ilustrar todas las fases por las que pasa un contrato, y se podrán repasar las dificultades encontradas, las soluciones y el resultado final.

El desarrollo consta de tres fases: una primera fase donde se ha desarrollado un preprocesamiento que modifica el código para unificarlo y facilitar su tratamiento, y una gramática que presenta a la sintaxis de Vyper con los cambios previamente aplicados y usando un tipo común.

Una segunda fase donde se ha desarrollado la sintaxis del apartado anterior con tipos sin utilizar el tipo anterior mencionado y generando las estructuras de datos para gestionar los datos que tenga el contrato.

En la tercera fase se han desarrollado un módulo de sistema que da funcionalidad al contrato introducido por el usuario, utilizando la sintaxis del apartado anterior y las estructuras de datos. Además, se ha creado un módulo de entrada/salida donde se han conectado todas las partes anteriores para generar un flujo con sentido desde la elección de contrato, función y parámetros, pasando por todas las demás funcionalidades hasta la salida de la función para el usuario.

Se repasarán las tres fases en profundidad.

3.1. Primera fase

Esta primera fase el desarrollo se centra en el preprocesamiento de un contrato de `Vyper` para adaptarlo a una sintaxis que `Maude` acepte y en distintas transformaciones que faciliten las siguientes fases. Además, se ha especificado la primera semántica y la función que transforma el contrato entrante mediante esta semántica diseñada.

3.1.1. Preprocesamiento

Antes de poder metarepresentar la sintaxis de `Vyper`, es necesario tener un código común y uniforme. Incluso hacer distintas transformaciones que no existen en el código propio de `Vyper`, esto es porque `Maude` no almacena información sobre tabulaciones, saltos de línea y subrayados, por lo que es indispensable pasar este proceso.

La función *preparse* inicialmente separa un contrato en dos partes para tratarlas de forma distinta: una primera donde se acumulen todas las variables de estado y otra donde

aparezcan todas las funciones. Todas las líneas de código tienen que ocupar una línea real del contrato, es decir, fallaría por ejemplo si un `assert` ocupa dos líneas. El código del *preparse* se encuentra en el archivo `preparse.maude`.

El *preparse* se divide en tres fases, que analizamos a continuación.

3.1.1.1. Pre-preparse

En esta fase se aplican distintas transformaciones al contrato completo sin todavía tener en cuenta las dos partes diferenciadas. Los cambios aplicados son:

1. La instrucción `{assert operación, excepción}`: utiliza la coma para separar la operación booleana de la excepción. Si la operación devuelve cierta continua y si devuelve falso, lanza la excepción. Se ha optado por reemplazar la coma por la barra vertical(|).

2. El flujo de control `continue`: se sustituye por `continue` debido a que es una palabra reservada de `Maude`.

3. Nombres de variables en `snake_case`: se sustituye la barra baja por guion al ser un signo propio de `Maude`.

4. Cambios en las expresiones numéricas: se añade un espacio en blanco a los signos operacionales al ir junto a las variables.

En este ejemplo se muestran algunas de las líneas modificadas en el `pre-preparse`.

Ejemplo 3.1.1 *Ejemplo de líneas modificadas en la primera fase del preparse:*

```
x=x+1                    ->      x = x + 1
y=ZERO_ADDRESS           ->      y = ZERO-ADDRESS
assert z>1 , "fallo"     ->      assert z > 1 | "fallo"
```

3.1.1.2. Preparse del espacio de nombres

Tras la primera fase del `preparse`, se toma la primera parte del contrato, el espacio de nombres donde se declaran todas las variables. En este caso se sustituye los dos puntos (:) que tienen todas las variables en su declaración por dos dobles puntos (::).

Este reemplazo se realiza porque al aplicar la siguiente fase, algunas asignaciones daban problemas con el doble punto al ser un signo que aparece en las instrucciones y en las declaraciones globales de Vyper, por eso una decisión de diseño fue reemplazarlo en las variables globales. En este ejemplo se muestran variables modificadas por estar declaradas en el espacio de nombres.

Ejemplo 3.1.2 *Ejemplo de líneas modificadas en la segunda fase del preparse:*

```
x : int128                    ->   x :: int128
y : Hashmap[int128, bool]     ->   y :: Hashmap[int128, bool]
```

3.1.1.3. Preparse de funciones

Este `preparse` se aplica en la segunda parte del contrato, a todas las funciones e instrucciones pertenecientes a cada función. El objetivo principal de esta fase es añadir un punto y coma (;) al final de cada instrucción en un bloque. Los únicos sitios sin punto

y coma son: cabecera de función, final de la función, decoradores, cabecera de `for, if, else y elif`.

Las instrucciones internas de estos bloques y los finales de bloque, sí que reciben el punto y coma, esto se añadió porque al aplicar la siguiente fase aparecían problemas de pertenencia de instrucciones a bloques sin saber si una determinada instrucción pertenecía a una función o a un bloque interno de esta como un `if` o un `for`.

Además de eliminar los punto y coma, en la ejecución de contrato se observó que las funciones que no reciben parámetros daban problemas por los paréntesis vacíos, por lo que se introdujo entre los paréntesis por la palabra `PaV`.

En este ejemplo se muestra las instrucciones que han sufrido cambios, añadiendo punto y coma a cada una que no pertenezca a final de bloque y la sustitución de la coma.

Ejemplo 3.1.3 *Ejemplo de líneas modificadas en la tercera fase del preparse.*

```
assert z > 1 , "fallo"          ->  assert z > 1 | "fallo" ;
x : int128 = 0                  ->  x : int128 = 0 ;
if x > 1:                       ->  if x > 1:
   send(msg.sender, msg.value)          send(msg.sender, msg.value);
   return                               return
end                                   end ;

def participate():             -> def participate(PaV);
```

3.1.2. Metaparse

En este apartado se presenta la gramática de un contrato de `Vyper` en Maude, teniendo en cuenta los cambios aplicados por el `preparse` que es lo primero que se ejecuta. Este preprocesamiento facilitará el desarrollo de la gramática y eliminará los distintos errores que pudiesen aparecer.

Esta gramática se encuentra definida a partir de los tipos básicos `@Token@` y `@Name@`, se usan además tipos que se han creado como representación del tipo real cuando la gramática se transforme en términos y que es un supertipo de los dos anteriores. Estos tipos representan cualquier `Qid` (*quoted identifier*, es decir, cualquier cadena precedida por una comilla) que aparezca en las posiciones de la gramática señaladas por estas palabras, exceptuando las propias palabras reservadas de Vyper. Se utiliza la función `metaParse` del meta-nivel para transformar una secuencia de `token` en un término, que se corresponderá con un programa en la sintaxis previamente definida. Esta gramática está desarrollada en al archivo `vyper.maude` en el módulo `vyper-grammar`.

Ejemplo de operaciones que definen una variable simple, una variable constante y un `hashmap` basado en `@Token@` y operaciones de acceso:

Ejemplo 3.1.4 *Ejemplo de la sintaxis de distintas operaciones basadas en `@Token@`*

```
sorts @statement@ @statementHashMap@ @statementConstant@
sorts @Access@ @Exp@ .
subsort @Token@ < @Access@ < @Exp@ .
op _[_] : @Exp@ @Exp@ -> @Access@ [ctor prec 6] .
op _._ :  @Exp@ @Exp@ ->  @Access@ [ctor prec 7] .
op _::_ : @Token@ @Access@ ->  @statement@ [ctor prec 22 ] .
op _:: public( _ ) : @Token@ @Access@ -> @statement@ [ctor prec 22] .
```

```
op _:: constant( _ ) = _ : @Token@ @Token@ @Token@ ->
                    @statementConstant@ [ctor prec 22] .
op _:: HashMap[_,_] :  @Token@ @Token@ @Access@ ->
                              @statementHashMap@ [ctor prec 22] .
op _:: public(HashMap[_,_]) :  @Token@ @Token@ @Access@ ->
                                @statementHashMap@ [ctor prec 22] .
```

Es fundamental dar la precedencia correcta a cada operación, para el momento en el que se aplique el `metaparse` lo haga en el orden correcto. En este ejemplo inicialmente se procesa el `@access@` que distingue entre una variable normal (de tipo `Qid` que es la base de los términos), una estructura (`_._`) o un *array* (`_[_]`) para posteriormente reducir cada declaración al conjunto de términos que lo representan.

3.1.3. Parse

La función `parse` recibe un conjunto de términos (expresiones del sort `Term`) y devuelve un conjunto de `Qid` tras aplicarle la gramática definida en el apartado anterior. El objetivo principal es eliminar los `@Token@` y `@Name@` que no van a ser utilizados en el siguiente proceso y eliminar todos los tipos creados que contienen `@`.

La idea principal es que los términos con tipos `@Token@` y `@Name@`, que en una primera pasada de análisis sintáctico nos sirvieron para ver que había tókenes/identificadores en ciertas posiciones, se evalúen ahora completamente como parte del contrato. Cuando un contrato es generado por esta función, ya está preparado para aplicarle la siguiente fase, porque cumple la gramática anterior de forma satisfactoria. Esta función se encuentra en el archivo `parse.maude` en el módulo `PARSING`.

Ejemplo 3.1.5 *Ejemplo de parseo del decorador de una función*

```
ceq parse(T) =  Q
 if S := string(T) /\
     isDecorador(S) /\e
     N := find(S, ".", 0) /\
     S1 := substr(S, 0, N) /\
     S2 := S1 + ".Decorator" /\
     Q := qid(S2) .
```

Un ejemplo de entrada a la ecuación anterior sería `@external.@Decorator@` y devuelve un `Qid` procesado como `@external.decorator`. Como se ha anticipado, elimina el tipo contenido con `@` para dar lugar a un término que será procesado por la siguiente gramática.

Analizando el código línea por línea:

1. El término que recibe la ecuación lo transforma en un alfanumérico.
2. Llama a una función que devuelve true en caso de que el alfanumérico pasado sea un decorador del estilo `@external.@Decorator@`.
3. La función *find* devuelve el número del carácter pasado por parámetro, en este caso el punto.
4. La función *substr* devuelve la primera parte del decorador hasta el punto `@external`.
5. Concatenemos `@external` con `.Decorator` que será el tipo utilizado en la siguiente fase.

6. Se trasforma el alfanumérico en un *qid* y se retorna.

Ejemplo 3.1.6 *Ejemplo de* parseo *de la cabecera de una función y sus parámetros*

```
ceq parse('_def_‘(_‘)[T1,T2,TL]) = 'headerD[T1', T2', TL']
    if  T1' := parse (T1) /\
        T2' := parse (T2) /\
        TL' := parse (TL) .
ceq parse('_:_[T1,T2]) =  'P[T1', T2']
     if  T1' := parse (T1) /\
         T2' := parse (T2) .
```

Una instrucción posible para la ejecución de estas reglas:

```
@external def suma (num:int256):
```

La salida tras aplicarle el `metaparse`:

```
'_def_‘(_‘)['@external.@Decorator@,'token['suma.Qid],
                ['_:_['token['num.Qid],'token['int256.Qid]]]]
```

La salida tras aplicarle el `parse`:

```
headerD['@external.Decorator,'suma.Qid","P['num.Qid, 'int256.type]]
```

A continuación se ha realizado un análisis desde la elección de ecuación hasta lograr la salida:

1. Por ajuste de patrones, la ecuación que encaja sería la primera. Como se ve en la entrada, la cabecera es divisible en tres partes: Los decoradores, el nombre de la función y los parámetros. El *parse* al ser una conjunto de reglas recursivas, se procesarían cada una de las partes anteriores de forma independiente y en caso de ser correcto finalmente se concatenarían.

2. La instrucción 1 ejecutaría la ecuación del ejemplo anterior y devolvería como resultado `@external.Decorator`.

3. La instrucción 2 trasformaría `'token['suma.Qid]`, intentando dar un tipo concreto al `Qid` que tiene dentro, pero como no lo encuentra por defecto, entiende que es una palabra no reservada y retorna `'suma.Qid`.

4. La instrucción 3 trasformaría `_:_['token[num.Qid],'token[int256.Qid]]`, con lo que por ajuste de patrones encaja con la ecuación 2.

5. Como en la ecuación 1, se llamaría al *parse* en la instrucción 4 con `'token[num.Qid]` y la 5 con `'token[ínt256.type]`. En el primer caso(como con el nombre de la función) al no encontrar un tipo reservado devolvería `num.Qid`. En el segundo caso, sí que es una palabra reservada de Vyper y devolvería `int256.type`. Al ser ambas funciones satisfactorias, resolvería el parámetro `'P[num.Qid, ínt256.type]`.

6. Componiendo todo lo anterior daría como resultado la salida completa:

   `'headerD['@external.Decorator,'suma.Qid,'P['num.Qid, 'int256.type]]`.

3.2. Segunda fase

En esta fase se ha desarrollado una semántica definiendo tipos concretos para dar forma con sentido a un contrato. Se han creado distintas estructuras de datos con las que trabajarán las reglas para almacenar todos los datos de un contrato y darle una estructura lo más sencilla posible para facilitar su futura ejecución.

3.2.1. Definición sintaxis interna

En este apartado se ha desarrollado una sintaxis con las operaciones obtenidas de ejecutar la función *parse* sobre su versión metarepresentada. El concepto de esta sintaxis es igual que la del módulo `vyper-grammar`, ya que se basan en el mismo lenguaje, pero teniendo en cuenta que esta sintaxis no está meta-representada, sino que se definen tipos internos propios que dan la forma final al contrato y se realiza una comprobación de la forma de cada instrucción y los elementos que la componen. Estos tipos son los que se van a utilizar para trabajar con las estructuras de datos y a partir de los cuales se van a generar las reglas.

El resultado de la función *parse* es el comienzo para el desarrollo de esta sintaxis, sin olvidarse de la importancia de los tipos. El código se encuentra en el archivo `vyper.maude` en el módulo `MY-PROG`.

En el siguiente código se muestra el mismo ejemplo que en el apartado 3.1.2 y se pueden observar las diferencias claras de una gramática *metarepresentada* y una gramática que utiliza tipos.

Ejemplo 3.2.1 *Ejemplo de la sintaxis de distintas operaciones y definición de tipos básicos de la gramática*

```
subsort Qid Float VarEnt bool type Bool constant stringVar NoneValue < Value .
subsort Value < Access .
subsort Access Call constant < Exp .
op -> : Exp Exp  -> Access [ctor] .
op . :  Exp Exp  -> Access [ctor] .
op D : Qid Access -> statement [ctor] .
op Dc : Qid type Value String -> statementConstant [ctor] .
op Dhm :  Qid type Access -> statementHashMap [ctor] .
op DhmP :  Qid type Access String -> statementHashMap [ctor] .
```

En el ejemplo anterior, el tipo `Value` está compuesto por distintos subtipos, sirve como base para todas las operaciones y declaraciones. A su vez, siendo este subtipo de *Access* que se crea como resultado de una operación de acceso al interior de distintas estructuras de datos. Teniendo estos tipos, se puede crear las distintas declaraciones de variables (simple, constante y mapas), marcando en cada una el tipo más restrictivo posible que forma la instrucción.

3.2.2. Estructuras de datos

En este apartado se han desarrollado distintas estructuras de datos, con el objetivo de almacenar las declaraciones del programa, las direcciones de las funciones y los datos con sus contenedores. Hay estructuras de datos que son comunes en todos los contratos, como la pila y la memoria de funciones, y otras que dependen de las variables que utilice

el contrato. En muchos casos pueden llegar a aparecer tipos de estructuras contenidas en otras. Todas las estructuras están en el archivo `Vyper.maude`

A continuación se va a repasar en profundidad cada estructura de datos desarrollada:

3.2.2.1. Hashmap

Se explicarán las decisiones de diseño tras mostrar el código.

Ejemplo 3.2.2 *Diseño de un `hashmap` en Maude*

```
sorts Mapa Map KeyValue TMapas .
subsort KeyValue < Map .
op D( _ -> _ )  : Value Value -> KeyValue [ctor] .
op D( _ -> _ )  : Value Array -> $ KeyValue [ctor] .
op D( _ -> _ )  : Value Memory -> KeyValue [ctor] .
op mapv : -> Map [ctor] .
op __ : Map Map -> Map [ctor assoc comm id: mapv] .
op [_,_,_] : type Value Float -> TMapas [ctor] .
op M|_,_| : TMapas Map -> Mapa [ctor]  .
```

El tipo final `Mapa` tiene dos partes diferenciadas:

Una primera parte `Tmapas` que indica la forma y tipo que tendrá el `hashmap`, con tres atributos; el tipo de la clave, el tipo del valor que puede ser un tipo propio de Vyper, una estructura o un *array*, y un tercer valor que indica el tamaño del *array* en caso de ser multidimensional.

La segunda parte `Map` encargada de almacenar los datos, está conformada por un conjunto de pares clave-valor (`KeyValue`), se representa como `D(_->_)` donde el primer parámetro es siempre un valor fijo, y el segundo puede ser un valor fijo, un conjunto de valores en forma de *array* o una estructura(`struct`) que internamente es una memoria. Un `Map` puede ser vacío(`mapv`) o tener una o varias clave-valor.

Ejemplo 3.2.3 *Ejemplo de un `Hashmap` con datos de un contrato real*

```
M|[int128, 'Funder, 0.0],D(0.0 ->['sender, address, 3.0,
0.0, private, normal]['value, uint256, 1.0, 0.0, private, normal])|
```

El *Hashmap* que se presenta tiene la clave de tipo `int128` y los valores son tipo `'Funder` que es una estructura con dos variables `sender y value`. En este caso solo tiene un valor que para obtenerlo habría que buscarlo por la clave 0 y devolvería una estructura donde el primer valor es 3.0 y el segundo valor 1.0. El código se encuentra en el módulo `MEMORY`.

3.2.2.2. Array

Se ha desarrollado el tipo `Array` para contener un conjunto de valores simples o un conjunto de estructuras, como se presenta en el contrato 2.

Ejemplo 3.2.4 *Definición de un `Array` en Maude*

```
sorts ArraySample Array .
subsort ArraySample < Array .
op arrayS( _ | _ ) : Memory Value -> ArraySample [ctor] .
op arrayS( _ | _ ) : Value Value -> ArraySample [ctor] .
op arrayEmpty : -> Array [ctor] .
op __ : Array Array -> Array [ctor assoc comm id: arrayEmpty] .
```

El tipo `Array` puede ser vacío (`arrayEmpty`) o estar compuesto por una o varias `ArraySample`, que representa cada celda de un array. Donde se diferencian dos partes (`arrayS( _|_ )`): un valor numérico, que funciona como `index` del array y otro valor o estructura según del tipo que se haya declarado que funciona como el contenido de la celda. El código de esta estructura se encuentra en el módulo `Memory`.

3.2.2.3. Pila y memoria

Se ha definido la memoria como un conjunto de variables con nombre único, en cada contrato existe una memoria principal con las variables globales que es accesible desde todas los puntos del contrato.

Además, existen memorias auxiliares por cada llamada a una función, donde se almacena las variables locales y se elimina al finalizar la llamada.

El conjunto de variables pertenecientes a la memoria están formadas por seis características:

- Nombre identificativo único.
- El tipo: `String`, `Array`, `Qid`, `Bool`, `Float`, estructura y mapa.
- Valor propio de la variable en concordancia al tipo.
- Un numérico que indica el tamaño en caso de ser multidimensional y cero en caso de ser una variable simple.
- Una característica que indica si la variable es constante, indexada o normal.
- La visibilidad de la variable (pública o privada).

Ejemplo 3.2.5 *Definición de los tipos `variable y memoria`*

```
subsort Variable < Memory .
ops public private : -> Visibilidad [ctor] .
ops constante indexed normal : -> ConsIndexNorm [ctor] .
op [_,_,_,_,_,_] : Qid type Value Float Visibilidad ConsIndexNorm
                                                  -> Variable [ctor] .
op mv : -> Memory [ctor] .
op __ : Memory Memory -> Memory [ctor assoc comm id: mv] .
```

En el código se muestra la definición de una variable con todos los atributos explicados anteriormente, y la definición de una memoria que puede ser vacía (`mv`) o estar compuesta por una o varias variables.

Aparte de la definición de la memoria, para la ejecución de los contratos se ha definido también el concepto de `Pila`. La pila tiene como base una memoria con todas las variables globales y sobre esta se apilaría o desapilaría memorias auxiliares, según las funciones referenciadas.

Ejemplo 3.2.6 *Definición de la `Pila`*

```
sort Stack .
subsort Memory < Stack .
op stackE : -> Stack [ctor] .
```

```
op push : Memory Stack -> Stack [ctor] .
op pop : Stack ->  Stack .
op top : Stack ->  Memory .
op isStackEmpty : Stack -> Bool .
eq pop(push(M,S)) = S .
eq top(push(M,S)) = M .
eq isStackEmpty(stackE) = true .
eq isStackEmpty(push(M,S)) = false .
```

Se han definido distintas funciones para trabajar con la pila: `push` para añadir una memoria, `pop` para eliminar una memoria, `top` que devuelve la cima de la pila e `isStackEmpty` para ver si la pila está vacía o no.

Una pila puede estar vacía(`stackE`) o estar compuesta por una o varias memorias. En la ejecución del contrato, la base de la pila siempre será la memoria global.

Ejemplo 3.2.7 *Ejemplo de una pila real*

```
push(['deadline, uint256, 1.00003e+5, 0.0, public, normal]
['goal,uint256, 2.0, 0.0, public, normal], stackE)
```

En este ejemplo aparece una pila con únicamente una memoria que tiene dos variables: `deadline` y `goal` ambas públicas, normales y de tipo `uint256`. En el módulo `Memory` está declarado todo el código de variable, memoria y pila.

3.2.2.4. Struct

El concepto de `estructura` es muy parecido al de memoria, es un conjunto de variables que van ligadas, por eso internamente declarando un `struct` lo que realmente se tiene es una memoria inmutable a la que no se puede añadir ni eliminar variables.

Esto hace que una variable simple del tipo `struct` en el campo valor tenga una memoria, aunque esto sea invisible para el usuario.

Ejemplo 3.2.8 *Código de un `struct` en un contrato de Vyper y su representación en Maude.*

```
struct Funder:
sender: address
['Funder, struct,['sender, address, 0.0, 0.0, private, normal]
, 0.0, private, normal]
```

En este ejemplo tenemos un *struct* `Funder` privado y normal que en el campo valor tiene una variable `sender` que no se puede borrar, solo modificar su valor. Se ha descrito dos versiones de la estructura, en la primera aparece el código en `Vyper` de un *struct* y en el segundo se presenta como sería esa estructura implementada en Maude.

3.2.2.5. Body

`Body` sería un híbrido entre una estructura datos y propiamente sintaxis, porque está definida en esta, pero su creación tiene como objetivo contener el conjunto de instrucciones de un bloque concreto.

Los bloques posibles son las instrucciones contenidas en una función, en un `if-elif-else` y en un `for`.

Ejemplo 3.2.9 *Definición del tipo `Body`*

```
subsort For StatementControl assig assigTuple RegisterEvent statementValue
ifElse statementValue Call controlFlow printP Parameters < Body .
 op bv : -> Body [ctor] .
 op BodyF : Body Body -> Body [ctor id: bv] .
```

En el código mostrado se marca `body` como una superclase de todas las instrucciones que pueden aparecer en un contrato, pudiendo ser este vacío (`bv`) o estar compuesto por una o varias de ellas.

3.2.2.6. Memoria de funciones

La memoria de funciones se ha creado para contener la información de todas las funciones existentes dentro de un contrato. Las funciones se pueden diferenciar según sus características:

- Init: Función para inicializar el contrato
- Default: Función que se llama en caso de que el nombre introducido no corresponda con ninguna función del contrato.
- Función con una o dos decoradores.
- Función que puede devolver o no un valor.

Ejemplo 3.2.10 *Ejemplos de la definición de parámetros de entrada y una función.*

```
sorts DataFunction Entry LEntry MemoryFunctions.
subsort DataFunction < MemoryFunctions .
```

Parámetros de entrada

```
op (_,_,_) : Qid Value Float -> Entry [ctor] .
op pV : -> LEntry [ctor] .
op __ : LEntry LEntry -> LEntry [ctor assoc id: pV] .
```

Función con dos decoradores:

```
op FDDR[_,_,_,_,_,_] : Qid Decorator Decorator LEntry Value Body
```

Memoria de funciones:

```
op mfv : -> MemoryFunctions [ctor] .
op __ : MemoryFunctions MemoryFunctions
                    -> MemoryFunctions [ctor assoc comm id: mfv] .
```

En este ejemplo se puede observar el tipo `Entry` que representa los parámetros de entrada de una función, con tres partes: el nombre del parámetro, el tipo del parámetro y un número que indica el tamaño en caso de ser un `array`. Una función puede tener varios parámetros `LEntry` o ser vacío `pV`.

`DataFunction` del ejemplo representa una función con dos decoradores y que devuelve un valor, en este caso contiene: el nombre, los dos decoradores, los parámetros de entrada (vacíos o no), el tipo que devuelve la función y el `body` que son las instrucciones internas.

Durante la ejecución de un contrato lo que se genera es un `MemoryFunctions` que contiene todas las funciones ejecutables, podría ser vacío `mfv`, pero en ningún contrato aparece este caso. La memoria de funciones es inmutable durante la ejecución, se carga al principio y se puede consultar pero no modificar.

El código de esta estructura de datos, acompañado de distintas funciones, están en el módulo `MEMORYFUNCTIONS`.

3.2.2.7. Expresiones

Las expresiones no son estructuras de datos, pero se han creado a partir de la sintaxis para poder representar operaciones complejas que van a ser resueltas mediante reglas en un único dato que se puede utilizar para ser almacenado, pasado por parámetro o retornado. Mostremos el código de la definición de expresiones:

Ejemplo 3.2.11 *Definición de expresiones y tipos de las variables.*

```
subsort Qid Float VarEnt bool type Bool constant stringVar NoneValue < Value .
subsort Value < Access .
op -> : Exp Exp -> Access [ctor] .
op . :  Exp Exp -> Access [ctor] .
subsort Access Call  < Exp .
```

Llamadas a funciones

```
op CallP : Qid ArgList -> Call [ctor] .
op Call : Qid -> Call [ctor] .
```

Resolución de operaciones numéricas

```
op PA_PC : Exp -> Exp [ctor prec 8] .
op -._ : Exp -> Exp [ctor prec 10] .
op not._ : Exp -> Exp [ctor prec 10] .
op _*._ : Exp Exp -> Exp [ctor comm assoc prec 14] .
op _._ : Exp Exp -> Exp [ctor prec 14] .
op _%._ : Exp Exp -> Exp [ctor prec 14] .
op _+._ : Exp Exp -> Exp [ctor comm assoc prec 16] .
op _-._ : Exp Exp -> Exp [ctor assoc prec 16] .
op _>._ : Exp Exp -> Exp [ctor prec 18] .
op _>=._ : Exp Exp -> Exp [ctor prec 18] .
op _<._ : Exp Exp -> Exp [ctor prec 18] .
op _<=._ : Exp Exp -> Exp [ctor prec 18] .
op _==._ : Exp Exp -> Exp [ctor assoc prec 19] .
op _!=._ : Exp Exp -> Exp [ctor prec 19] .
op _and._ : Exp Exp -> Exp [ctor prec 20] .
op _or._ : Exp Exp -> Exp [ctor assoc prec 21] .
```

Todos los tipos básicos que pueden tener las variables se fusionan en un único tipo `Value`, que las reglas desmigaran hasta llegar al tipo mínimo reducible. A su vez, el tipo `Value` ha sido definido como subtipo de `Access` cuya finalidad es acceder a las estructuras para extraer los datos necesarios.

La definición de acceso tiene dos expresiones como parámetros por si fuese necesario realizar un cálculo para obtener la dirección final.

A continuación se muestra un ejemplo de acceso(diferenciado por el punto) cuyo primer parámetro es un `hashmap(hmp)` y como segundo parámetro `var2` que representa el nombre de la variable de la estructura:

```
hmp[var1 + 40].var2
```

El tipo `exp` que representa una expresión incluye a `Call y Access` que son subtipos. El tipo `call` representa la llamada a una función del contrato, con o sin parámetros. Para poder operar dentro de una expresión, la función a la que se llama tiene que devolver un tipo de los definidos como subtipos de `Value`.

Aparte de lo visto hasta ahora, todas las operaciones numéricas tienen como parámetros dos expresiones y retornan otro tipo expresión. Esta decisión tiene como objetivo poder componer expresiones dentro de otras expresiones o incluso llamadas a funciones y accesos a memoria con la que crear operaciones de alta complejidad. Siempre devolviendo una expresión que será reducible en última instancia al tipo más bajo posible. Las operaciones numéricas tienen definidas una prioridad para ser resueltas en el orden correcto.

Ejemplo 3.2.12 *Ejemplos de expresiones*

```
var1 *. var2 ; 8.0 -. var4; var5 >=. var6 ;true or var6.
```

Expresiones contenidas en otras expresiones:

```
var1 *. var2 +. var3;
PA var4 -. var5  PC * var6,  PA var6 +. var7 PC >. var8 or. var9;
```

Expresiones complejas con accesos y llamadas a otras funciones:

```
hmp[var1 + 40].var2 *. var3 ;
PA call(fun1) +. var3 PC /. var4
```

3.3. Tercera fase

Esta fase se puede dividir en dos subfases, una primera subfase donde se han desarrollado las ecuaciones y las reglas que dan funcionalidad al programa y una segunda, donde se conectan todos los componentes de las partes anteriores junto a la entrada salida.

3.3.1. Primera subfase

Se ha desarrollado en paralelo los módulos `Semantics` y `AuxiliarFunctions` que se encuentran en al archivo `vyper.maude`, en este primero se encuentra las reglas que ejecutan las funciones del contrato Vyper y en el segundo módulo se han desarrollado una gran cantidad de ecuaciones para facilitar el uso de las estructuras de datos según va surgiendo la necesidad en las reglas.

3.3.1.1. Módulo de reglas

Ejemplo 3.3.1 *Definición del tipo que almacena la información del contrato.*

```
op <_,_,_,_,_> : Body MemoryFunctions Stack Value Memory
                                        -> ExecutionTuple [ctor] .
```

ExecutionTuple vacio:

```
<bv,mfv,stackE,noneV,mv>
```

En este módulo se ha creado el tipo `ExecutionTuple` sobre el que trabajan las reglas y que se crea al iniciar la ejecución de un contrato, inicialmente estaría vacío. Como se observa en el código está compuesto por cinco parámetros

1. Body: Contiene las líneas de código a ejecutar, este es el parámetro que permite la ejecución continuada de las reglas, ya que no se finaliza la ejecución de una función hasta que el `body` no es vacío (`bv`).
2. Memoria de funciones: Almacena todas las funciones existentes en el contrato al inicio y no se puede modificar durante la ejecución.
3. Pila de memoria: Tiene como base de la pila la memoria global y se va apilando o desapilando memorias según las funciones que se ejecuten.
4. Parámetro de retorno de valores individuales, `noneV` si la función no devuelve nada.
5. Parámetro de retorno de valores multidimensionales, `mv` si la función no devuelve nada.

Para que se entienda el funcionamiento de las reglas, el usuario seleccionaría una función a ejecutar con los parámetros deseados, se cargaría de la memoria de funciones en el parámetro `body` y se ejecutaría instrucción a instrucción hasta acabar la función mostrando la memoria resultante de ejecutar la función y el valor devuelto en caso de existir.

Se han creado varias reglas para cada tipo de sentencia desarrollada en la sintaxis, a continuación se muestra un ejemplo de código de la regla que crea una variable local.

Ejemplo 3.3.2 *Ejemplo de la regla que crea una variable local.*
Código en Vyper: `x: int128 = 2.0`
ExecutionTuple: `< BodyF( Dv( 'x, int128, 2.0 ), B), MF, S, V1, M >`
Regla:

```
crl [newVariable] :
    < BodyF( Dv( Q, V, E ), B), MF, S, V1, M > =>  < B, MF, S1, V1, M >
      if < E, MF, S, V1, M > =>  < bv, MF, S, V2, M > /\
    S1 := addNewVariableStack(S, Q , V , V2) .
```

Tras el ajuste de patrones, inicialmente procesa la expresión `E` hasta reducirla a un valor simple, en este caso las reglas no realizarían ninguna operación compleja, ya que 2.0 no es reducible. Con este valor, se llama a la función `addNewVariableStack` que tiene como parámetros, la pila, el nombre de la nueva variable, el tipo y el valor recogido anteriormente y devuelve la pila con la variable nueva añadida en la memoria de la cima.

En el final de la regla se pasa a la siguiente instrucción en el caso de que body no fuese vacío (no habría más instrucciones), con la pila modificada y sin la instrucción que se acaba de procesar y los demás valores iguales.

```
< B ,  MF , S1 , V1 , M >
```

3.3.1.2. Módulo de ecuaciones

En este módulo se han creado un gran número de ecuaciones para facilitar el funcionamiento de las reglas y su conectividad con las estructuras de datos.

- **Inicializadoras**: Que genera nuevas variables, funciones, declaraciones, etc.
- **Procesamiento**: En el caso de los parámetros de las funciones, se extraen utilizando este tipo de funciones.
- **Sobre estructuras de datos**: Ejecutan distintas acciones sobre las estructuras de datos: almacenar, extraer, modificar, verificar e inicializar.
- **Tipado**: Realizan una comprobación sencilla de tipos sobre los que se ejecutan acciones y los que devuelven valores booleanos.
- **Trasformar a cadena**: Para poder imprimir el resultado de las ejecuciones de las funciones, describen las estructuras de datos para presentarlas al usuario.

3.3.2. Segunda subfase

En esta subfase se ha desarrollado un módulo de sistema de reglas enfocado a la entrada salida, para relacionar el programa con el usuario, este módulo se encuentra en el archivo `parse.maude` de nombre `MTP`.

Estas reglas iteran en torno a distintos estados que se han definido para marcar en que momento se encuentra el programa. A continuación se muestran los estados y qué acción se realiza en cada uno:

1. **Cargar contrato**: En este estado se solicita al usuario el nombre del contrato a cargar que tiene que estar en la misma carpeta desde la cual se está lanzando las reglas de entrada/salida. Si el nombre del contrato es correcto, se pasa al siguiente estado.
2. **Abrir contrato**: En este estado se abre el contrato que se ha leído en el apartado anterior, si se abre correctamente se pasa al siguiente estado.
3. **Leer contrato**: Con el contrato ya abierto, se lee línea a línea hasta que sea vacío, se concatena punto y coma a cada línea para evitar confusiones en las siguientes operaciones que apliquemos al contrato.
4. **Procesamiento**: En este estado se aplican los distintos procesamientos para dejar el contrato en un estado que permita la aplicación de las distintas reglas.
5. **Cargar función**: Solicita al usuario la función que quiere ejecutar.
6. **Cargar parámetros**: En este estado se solicitan parámetros si la función los necesita. Es necesario pasar un espacio para pasar al estado de ejecución.
7. **Ejecutar función**: En este estado se ejecuta la función seleccionada por el usuario con los parámetros pasados(podría ser vacío si la función no tiene parámetros de entrada). Se muestra al usuario la salida de ejecutar la función y se pasa al estado que solicita la siguiente función a procesar.
8. **Salida**: En caso de que no introduzcamos nombre de función, se pasa al estado que finaliza la ejecución del contrato.
9. **Error**: Si alguno de los estados por los que se atraviesan no tienen una salida satisfactoria, se pasaría a un estado de error en el que se describe el problema.

3.4. Cómo ejecutar un contrato

En este apartado se va a presentar la ejecución de un contrato(el explicado en el apartado anterior), basándose en los estados por los que pasa y explicando la salida producida por cada uno de ellos. Como ya se ha explicado anteriormente, el contrato tiene que estar en la misma carpeta desde la cual lanzamos las reglas que ejecutan los estados.

1. **Cargar programa**: Es necesario posicionarse en la carpeta donde estén las reglas y cargar el programa con el comando `"load archivo a cargar"`(en este caso *load parse.maude*). Se muestra en pantalla los paquetes y operadores importados, y los avisos que pueden causar error. Tras haber cargado el programa se solicita el nombre del archivo a abrir.

   ```
   Maude> load parse.maude
   ```

```
Maude> load parse.maude
Advisory: "vyper.maude", line 531 (fmod FOO): sort Zero has been imported from both "prelude.maude", line 101 (fmod NAT)
    and "prelude.maude", line 101 (fmod NAT).
Advisory: "vyper.maude", line 531 (fmod FOO): sort NzNat has been imported from both "prelude.maude", line 101 (fmod
    NAT) and "prelude.maude", line 101 (fmod NAT).
Advisory: "vyper.maude", line 531 (fmod FOO): sort Nat has been imported from both "prelude.maude", line 101 (fmod NAT)
    and "prelude.maude", line 101 (fmod NAT).
Advisory: "vyper.maude", line 531 (fmod FOO): sort String has been imported from both "prelude.maude", line 678 (fmod
    STRING) and "prelude.maude", line 678 (fmod STRING).
Advisory: "vyper.maude", line 531 (fmod FOO): sort Char has been imported from both "prelude.maude", line 678 (fmod
    STRING) and "prelude.maude", line 678 (fmod STRING).
```

Figura 3.1: Cargar el programa en Maude

2. **Abrir contrato**: Se introduce el nombre del contrato a cargar y se imprimen las tres fases de procesamiento por las que pasa el contrato seleccionado.

   ```
   erewrite in MTP : run .
   Introduce file name> contrato4.txt
   ```

```
erewrite in MTP : run .
Introduce file name> contrato4.txt
regla1
regla3
```

Figura 3.2: Abrir contrato

   a) **Preparse**:

   ```
   'struct 'Funder ':: 'sender ':: 'address 'value ':: 'uint256 'end
   ...
   ...
   '= 'empty '`( 'Funder '`) 'end '; 'refundIndex '= 'ind '+ '30 'end
   ```

   b) **Metaparse**:

   ```
   METAPARSE '__['__['struct_::_end['token[''Funder.Qid],'__['_::_,
   ...
   ...
   ,'_+_['token[''ind.Qid],'token[''30.Qid]]]]]]]]]]
   ```

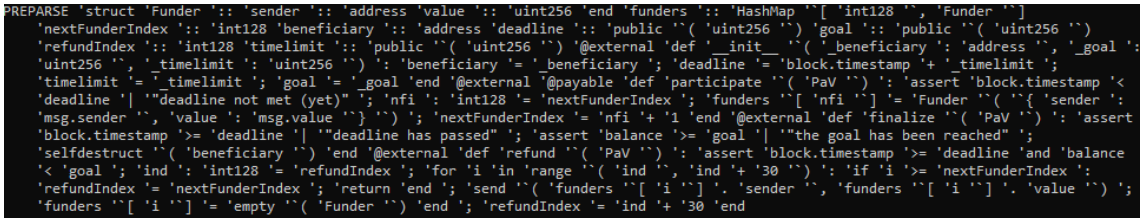

Figura 3.3: Preparse aplicado al contrato

Figura 3.4: Metaparse aplicado al contrato

c) **Parse**:

```
PARSE 'Contract['ListS['Ds['Funder.Qid,'ListS['D['sender.Qid,'address.type],
...
...
'=['refundIndex.Qid,'_+._['ind.Qid,'3.0e+1.FiniteFloat]]]]]]]]]]
```

Figura 3.5: Parse aplicado al contrato

3. **Leer función y parámetros**: Se introduce el nombre de la función a ejecutar, y los parámetros necesarios en caso de que la función los solicite. En esta ejecución introducimos *init* como nombre de la función y tres valores numéricos, para indicar que se comience a ejecutar la función se pasa un espacio cuando se solicita el siguiente parámetro.

```
Introduce function name> __init__
rule asking parameters
```

```
Introduce arguments> 1.0
rule asking parameters T ’1.0.FiniteFloat TY ’FiniteFloat
Introduce parameters> 2.0
rule asking parameters T ’2.0.FiniteFloat TY ’FiniteFloat
Introduce parameters> 3.0
rule asking parameters T ’3.0.FiniteFloat TY ’FiniteFloat
Introduce parameters>
```

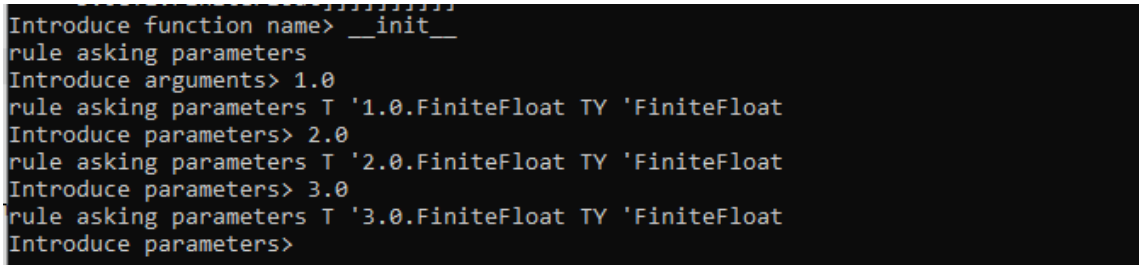

Figura 3.6: Llamada a la función init, con tres parametros

Si la función se ejecuta satisfactoriamente, se muestra la pila de memorias tras los cambios, y el valor devuelto en caso de que exista. Al ejecutar la función se puede ver las reglas que se han aplicado para llegar al resultado obtenido, que es una buena forma de trazabilidad en busca de fallos.

Ejemplo 3.4.1 *Ejemplo de la devolución al ejecutar una función.*
Pila con las variables modificadas:

```
push([’Funder, struct,[’sender, address, 0.0, 0.0,
private, normal][’value,uint256, 0.0, 0.0, private, normal], 0.0, private,
normal][’balance, uint256, 1.0e+2, 0.0, public, normal][’beneficiary, address,
1.0, 0.0, private, normal][’deadline, uint256, 1.00003e+5, 0.0, public, normal]
[’goal,uint256, 2.0, 0.0, public, normal][’nextFunderIndex,
int128, 0.0, 0.0, private, normal][’refundIndex, int128,
0.0,0.0, private, normal][’timelimit, uint256, 3.0, 0.0,
public, normal][’funders, HashMap, M|[int128, ’Funder, 0.0],
mapv |, 0.0, private, normal], stackE)
```

Valores de retorno:

```
Valor devuelto por la función(noneV si no hay) noneV
Struct devuelto por la funcion(mv si no hay): mv
```

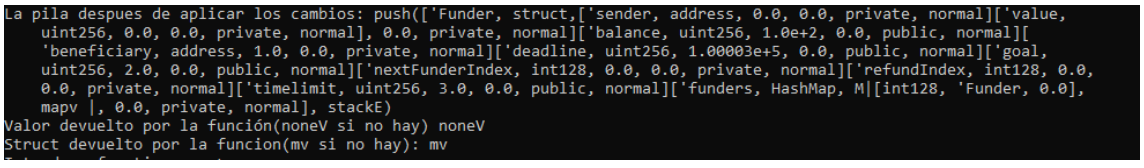

Figura 3.7: Resultado de ejecutar la función init

Capítulo 4

Ejecución y pruebas de un contrato

En este capítulo se llevará a la práctica todo lo explicado en los capítulos anteriores, repasando las características que aparecen en el conjunto de ejemplos, señalando distintas instrucciones según donde se presenten y la formas de tratarlas. Además, se va a ejecutar y analizar de forma precisa el contrato 4, que es con el que se ha operado a lo largo de la memoria, estudiando sus características principales respecto al entorno desarrollado en `Maude` que da sentido a su ejecución.

Repasando en profundidad las entradas, variables, la pila, funciones y reglas por las que pasa, evaluando su salida, instrucciones y los resultados.

Inicialmente, se va a repasar las distintas características de Vyper (véase la sección 2.2) que aparecen a lo largo de los contratos del conjunto de pruebas, facilitando al usuario llevar a la práctica alguna en concreto. Todo lo que aparece en la siguiente lista se presenta en al menos un contrato y tiene funcionalidad implementada y probada en Maude.

- Variables globales:
  - Públicas: `nombre: public(tipo)`
  - Privadas: `nombre: (tipo)`
  - `Hashmap`: `nombre: Hashmap[tipo,tipo]`
  - Estructuras: `struct nombre: nombre1 : tipo1 ... nombreN : tipoN`
  - Constantes: `nombre: nombre1 : constant(tipo) = valor`
  - Eventos: `event nombre: nombre1 : tipo1 ... nombreN : tipoN(tipo)`
- Funciones
  - Con parámetros de entrada:

    `def nombre_funcion(nombre1:tipo1,nombre2:tipo2 ... nombreN:tipoN)`
  - Sin parámetros de entrada: `def nombre_funcion()`
  - Devuelve valor
  - Sin devolver valor
- Instrucciones
  - Expresión simple: `var1 = var2`
  - Expresion compleja: exp1 representa una expresión numérica `var1 = exp1`

- Expresión += / -= : `var1 (+/-)= var2`
- Acceso a memoria(struct, array o hashmap): `acceso1 = exp1`
- Crear struct:

  ```
  acceso1 = Nombre_struct{nombre1: var1, nombre2:var2... nombren:varN }
  ```

- Assert: `assert Exp1(de tipo boolean)`
- Variable local: `nombre: tipo = exp1`
- Llamada a función: `nombre_funcion(var1,var2...varN) / nombre_funcion()`
- if: `if exp(tipo bool) : cuerpo`
- if else: `if exp(tipo bool): instrucciones else: instrucciones2`
- if elif else:

  ```
  if exp(tipo bool): instrucciones elif exp2(tipo bool): instrucciones2
          else: instrucciones3
  ```

- For in range: `for var in range(exp1)/range(exp1,exp2) : instrucciones`

La tabla 4.1 muestra un resumen de las características, según en que contratos dentro de los probados en este trabajo se presentan.

4.1. Ejecución de un contrato

En esta sección se presenta la ejecución del contrato 4, con el objetivo de clarificar y ejemplificar todo lo estudiado en el apartado 3 de la memoria. El apartado se dividirá en subsecciones según el punto de ejecución en el que se encuentre el contrato.

4.1.1. Análisis del contrato parseado

Se parte del punto donde ha pasado de forma satisfactoria la verificación de la sintaxis, tras aplicar el `preparse`, `metaparse` y `parse`. La forma del contrato que ofrece el `parse`:

- **Variables globales**: Es la primera parte de un contrato, que engloba el conjunto de declaraciones.

  ```
  'ListS['Ds[''Funder.Qid,'ListS['D[''sender.Qid,'address.type],
  'D[''value.Qid,'uint256.type]]],'ListS['Dhm[''funders.Qid,
  'int128.type,''Funder.Qid],'ListS['D[''nextFunderIndex.Qid,
  'int128.type],'ListS['D[''beneficiary.Qid,'address.type]
  ,'ListS['Dp[''deadline.Qid,'uint256.type,'"public".String],'ListS[
  'Dp[''goal.Qid,'uint256.type,'"public".String],'
  ListS['D[''refundIndex.Qid,'int128.type],'Dp[''timelimit.Qid,'uint256.type,
  '"public".String]]]]]]]
  ```

  La lista de declaraciones se forma como `ListS[Declaración, ListS]` o de la forma `ListS[Declaracion, Declaracion]` en el caso de que solo quede una. Las declaraciones del contrato 4 son:

  1. Struct: `Ds[Nombre, Lists]` con dos variables, `sender` que es una dirección y `value` un número. `Funder` es la estructura de este contrato.

Contrato	Var pub	Var priv	Mapa	Estructuras	Eventos	Constantes	Fun-param	Fun-no-param
1	X		X				X	X
2	X		X	X	X	X	X	X
3	X							X
4	X	X		X			X	X
5	X		X	X			X	X
6	X	X	X				X	X
7	X	X	X	X			X	X
8	X	X		X			X	X
9	X	X		X	X		X	X
10	X	X	X	X	X		X	X
11	X	X	X	X	X		X	X

Tabla 4.1: Tabla de características presentes en los contratos

Contrato	Return	Exp	Exp Compleja	Crear-Struct	Assert	Acceso	Var local	Llamada	If	If-Else	if-elif	For
1		X	X		X	X	X					
2	X	X	X	X	X	X		X	X			
3		X	X		X			X				
4		X	X			X	X	X	X			X
5	X	X	X	X	X	X		X	X			X
6	X	X	X	X	X	X	X	X				
7	X	X	X		X	X	X	X	X	X		X
8	X	X	X		X			X	X	X		
9		X	X	X	X			X		X		
10	X	X	X	X	X	X		X	X	X	X	
11		X	X	X	X	X	X	X	X	X		X

Tabla 4.2: Tabla de características presentes en los contratos

2. Hashmap: `Dhm[Nombre, tipo de la clave, tipo de los valores]` donde la clave es un número y el valor es la estructura *Funder*. En este ejemplo se forma un hashmap de estructuras.
3. Variables públicas: `Dp[Nombre, tipo, "public".string]` se han definido varias variables públicas, `deadline, goal y timelimit`.
4. Variables privadas: `D[Nombre, tipo]` se han definido las variables `nextFunderIndex`, `beneficiary` y `refundIndex`.

- **Funciones**: Es la segunda parte de un contrato, formado por el conjunto de funciones.

```
1.  __init__:
'ListF['Fun['init['@external.Decorator,'LParam['P
[''_beneficiary.Qid,'address.type],'LParam['P[''_goal.Qid,'uint256.type]
,'P[''_timelimit.Qid,'uint256.type]]]],'BodyF['=[''beneficiary.Qid,
''_beneficiary.Qid],'BodyF['=[''deadline.Qid,'_+._
['block.timestamp.VarEnt,''_timelimit.Qid]],'BodyF[
'=[''timelimit.Qid,''_timelimit.Qid],'=[''goal.Qid,''_goal.Qid]]]]],

2.  participate:
'ListF['Fun['headerDD['@external.Decorator,'@payable.Decorator,
''participate.Qid,'PaV.Parameters],'BodyF['Assert['_<._
['block.timestamp.VarEnt,''deadline.Qid],'"deadline not met (yet)
".String],'BodyF['Dv[''nfi.Qid,'int128.type,''nextFunderIndex.Qid]
,'BodyF['=St['->[''funders.Qid,''nfi.Qid],''Funder.Qid,
'LParam['P[''sender.Qid,'msg.sender.VarEnt],'P[''value.Qid,
'msg.value.VarEnt]]],'=[''nextFunderIndex.Qid
,'_+._[''nfi.Qid,'1.0.FiniteFloat]]]]]],


3.  finalize:
'ListF['Fun['headerD['@external.Decorator,''finalize.Qid,'PaV.Parameters],
'BodyF['Assert['_>=._['block.timestamp.VarEnt,''deadline.Qid],
'"deadline has passed".String],'BodyF['Assert['_>=._[''balance.Qid,''goal.Qid],
'"the goal has been reached".String],'CallP[
''selfdestruct.Qid,''beneficiary.Qid]]]],

4.  refund:
'Fun['headerD['@external.Decorator,''refund.Qid,'PaV.Parameters],
'BodyF['Assert['_and._['_>=._['block.timestamp.VarEnt,''deadline.Qid]
,'_<._[''balance.Qid,''goal.Qid]]],'BodyF['Dv[''ind.Qid,'int128.type,
''refundIndex.Qid],'BodyF['ForR[''i.Qid,'ArgL[''ind.Qid,
'_+._[''ind.Qid,'3.0e+1.FiniteFloat]],'BodyF['If['_>=._[''i.Qid,
''nextFunderIndex.Qid],'BodyF['=[''refundIndex.Qid,''nextFunderIndex.Qid],
'return.controlFlow]],'BodyF['CallP[''send.Qid,'ArgL['
.['->[''funders.Qid,''i.Qid],''sender.Qid],'.['->[''funders.Qid,
''i.Qid],''value.Qid]]],'=['->[''funders.Qid,''i.Qid]
,'CallP[''empty.Qid,''Funder.Qid]]]]],'=[''refundIndex.Qid,
'_+._[''ind.Qid,'3.0e+1.FiniteFloat]]]]]]]]]
```

Las funciones parseadas toman la forma `ListF[Función, ListF]` o `ListF[Función, Función]` en el caso de que solo quede una función. Las funciones también tienen dos partes: una cabecera y un cuerpo con distintas instrucciones `Fun[cabecera, cuerpo]`. La cabecera es diferente según del tipo de función que sea y el cuerpo se forma como `BodyF[Instrucción, BodyF]` o `BodyF[Instrucción, Instrucción]`.

A continuación el desglose explicado por función:

1. **Función init**:
   `Fun[init[decorador, parámetros], instrucciones]` Al ser la función inicializadora, el único decorador que puede tener es `external` y en el caso de los parámetros de entrada esta función tiene tres, se presentan como `LParam[Parámetro, LParam]` o `LParam[Parámetro, Parámetro]` y a su vez un parámetro como `P[nombre, tipo]`. Esta función recibe `_beneficiary, _goal y _timelimit` que es una dirección y dos números.
   En cuanto a las instrucciones, aparecen tres asignaciones (`=[variable, expresión]`): dos de ellas sencillas que asignan el valor recibido por parámetro a una variable global y la tercera que asigna una expresión, en este caso es la suma de un parámetro de entrada de la función al `timestamp` actual `=[var,_+_[expresión, expresión]]`.

   ```
   '=[''deadline.Qid,'_+._['block.timestamp.VarEnt,''_timelimit.Qid]]
   ```

   Las variables del programa y de entorno son subtipo de expresiones, por lo que en este ejemplo la expresión que se asigna a la variable `deadline`, se reduce a la suma de dos variables.
2. **Función participate**:
   `Fun[headerDD[decorador, decorador,'participate, parámetros], instrucciones]` los decoradores en este caso son `@external y @payable` y no tiene parámetros de entrada por lo que la cabecera está marcada con `PaV.Parameters`. Esta función está compuesta por cuatro instrucciones:
   Un assert `assert[expresión, mensaje(string)]`, donde la `expresión` tiene que ser booleana, compara una variable global con una variable de entorno, si la variable `deadline` es menor al `timestamp` del bloque, mostraría el error y revertiría el contrato. Si no, continuaría la ejecución con la siguiente instrucción.
   La definición de una variable local, solo visible para esta función `'Dv[variable, tipo, expresión]`, `nfi` de tipo `int128` que se le asigna el valor de una variable simple `nextFunderIndex`.

   ```
   '=St['->[''funders.Qid,''nfi.Qid],''Funder.Qid,
   'LParam['P[''sender.Qid,'msg.sender.VarEnt],
   'P[''value.Qid,'msg.value.VarEnt]]]
   ```

   Creación de una estructura que esta dentro de un `hashmap '=St[acceso, estructura, parámetros]`, los accesos se utilizan para referenciar estructuras de datos complejas `->[variable, expresión]` donde la variable indica el nombre del hashmap y la expresión que en este caso se reduce a una variable para marcar la clave donde se va a almacenar.
   En estructura se indica el nombre del `struct` que se va a crear y en parámetros se pasan los datos con los que se va a inicializar la estructura que internamente tienen la misma forma que los parámetros de entrada de una función `LPARAM[parámetro, parámetro]` siendo un parámetro `P[nombre, valor]`. En este caso se pasa como valores el `msg.sender y msg.value` que son variables de entorno.
   La última instrucción es una asignación en la que a la variable `nextFunderIndex` se le da el valor de la variable `nfi + 1`.

3. **Función finalize**:
   `Fun[headerD[decorador,'finalize, parámetros], instrucciones]` esta función solo tiene un decorador `@external` y no tiene parámetros de entrada. Contiene tres instrucciones: dos `assert` y una llamada a una función propia de `Vyper`.
   Ambos `assert` realizan una operación booleana con dos expresiones que se reducen en dos variables simples. La llamada a una función `callP[función, lista de parámetros]`, que contiene el nombre y una lista de parámetros de la forma `ArgL[expresión, expresión]` si existen varias variables y en caso de solo tener una variable, aparecería una expresión simple. En este caso se llama a la función `'selfdestruct` con `'beneficiary` como único parámetro.

   ```
   'CallP[''selfdestruct.Qid,''beneficiary.Qid]
   ```

4. **Función refund**:
   `Fun[headerD[decorador,'refund, parámetros], instrucciones]` función con un solo decorador(`@external`) y sin parámetros de entrada. Esta función contiene cuatro instrucciones:
   Un `assert` que tiene una expresión booleana compleja
   `and.[>=.[block.timestamp.VarEnt, deadline.Qid], <.[balance.Qid, goal.Qid]]`
   donde por precedencia de operadores se ejecutaría `and` sobre las dos expresiones resueltas, en el primer caso comparando `block.timestamp y deadline` y en el segundo caso `balance y goal`.

   ```
   'ForR[i.Qid,'ArgL[ind.Qid,'_+._[ind.Qid,'3.0e+1.FiniteFloat],
                                                    BodyF[...]]
   ```

   La instrucción `ForR[iterador, argumentos, instrucciones]` crea una variable que funcionara como iterador del bucle, en argumentos se indica el inicio y fin, el inicio lo marca la variable `ind` y el final `ind + 30`. El bucle a su vez contiene tres instrucciones:

   ```
   If['>=.[i.Qid,nextFunderIndex.Qid],
    BodyF[=[refundIndex.Qid,nextFunderIndex.Qid],return.controlFlow]]
   ```

   Una instrucción `if` es de la forma `if[expresión, instrucciones]`, donde la expresión devuelve un valor booleano y en caso de ser cierto ejecuta el bloque de instrucciones y en caso de ser falso pasa a la instrucción. En esta expresión utiliza la variable `i`, solo visible dentro del bloque de instrucciones contenidas en el `for`, comparándola con una variable global. Este `if` contiene dos instrucciones: una asignación de variables de `nextFunderIndex a refundIndex` y `return` que es una palabra reservada de tipo `controlFlow`, que produce el fin de la ejecución de la función.

   ```
   CallP[send.Qid,ArgL[.[->[funders.Qid,i.Qid],sender.Qid],
   .[->[funders.Qid,''i.Qid],value.Qid]]]
   ```

   La segunda instrucción del bloque `for` es una llamada a la función propia de `Vyper send`, con dos parámetros que son dos accesos. Los accesos se resuelven de dentro a fuera, por lo que en este caso devolvería la estructura que indica la posición `i` mediante el acceso `->` y a través de este, recuperamos la variable interna con el operador `..`

```
=[->[funders.Qid,i.Qid], CallP[empty.Qid,Funder.Qid]]
```

La tercera instrucción es una asignación compleja, de la función `empty` propia de `Vyper`, cuya funcionalidad es cargar con los datos vacíos una estructura del tipo pasado por parámetro, en este caso `Funder`.

```
=[refundIndex.Qid,_+._[ind.Qid,'3.0e+1.FiniteFloat]]
```

La última instrucción de la función es una asignación a la variable `refundIndex`, el valor se obtiene resolviendo la expresión suma de la variable `ind` y el valor numérico `30`.

4.1.2. Análisis de reglas aplicadas al contrato

Se va a repasar la ejecución mediante reglas en tres situaciones: al cargar el contrato, en la función `init` y la función `refund` con dos posibles resultados. Con el fin de clarificar la aplicación, elección y reducción del conjunto de reglas.

1. **Carga del contrato**: Al introducir la primera función a ejecutar, el módulo de entrada/salida comprueba si existe un entorno del contrato ejecutable por las reglas y en caso de que no exista, antes de llamar a la función ejecuta la regla `exec`

```
crl[exec] : exec(C,B) => < B, MF1,
                           push(initMemory(C), stackE), noneV, mv >
    if  MF := initMemoryFunctions(C) /\
  MF1 := addMemoryBuiltFunctions(MF) .
```

Esta regla recibe dos parámetros: `C` que representa un `contrato` tras pasar el `preparse, metaparse y parse` de forma satisfactoria y también un tipo body (`B`) que es la instrucción a ejecutar, con la llamada a la función y los parámetros introducidos por el usuario. En este ejemplo se ha llamado a la función `init` con `1.0` en los tres parámetros, de la forma:

```
exec(Contract[ListS[...],ListF[...]], call(init, ArgL(1.0,ArgL(1.0,1.0))))
```

Tras el ajuste de patrones se ejecuta en esta regla la función `initMemoryFunctions`, que es la encargada de trasformar las funciones del contrato parseadas en el tipo `MemoryFunctions` definido como una estructura de datos. Si se resuelve de forma correcta esta función se ejecuta la siguiente `addMemoryBuiltFunctions` que recoge la `MemoryFunctions` anterior y le añade dos funciones propias de `Vyper` como son `send` y `selfdestruct`.

Otra función que se ejecuta es `initMemory`, que recibe como parámetro el contrato y devuelve la estructura de datos `Memory` con todas las variables globales, a partir de esta se genera la pila usándola como base.

Si ambas funciones se resuelven de forma correcta, la regla devolvería un tipo

`ExecutionTuple`(Entorno de ejecución) que da comienzo a la ejecución de las demás reglas.

```
salida: <call(init, ArgL(1.0,ArgL(1.0,1.0))), MemoryFunctions, Stack,
        NoneV(valor de retorno vacio), mv(la estructura de retorno vacio)>
```

2. **Llamada a la función init**: La salida de la regla anterior es la que desencadena la ejecución de las siguientes, en este caso la función que encaja por ajuste de parámetros es `callParameters`:

```
crl [callParameters] :
< BodyF(CallP(Q,AL), B),  MF, S, V, M > => < B,  MF, S1, V, M >
      if not FunHasReturn?(MF,Q) /\
           LE := getParametersFunction(Q , MF) /\
           < AL, MF, S, V, M > => < bv, MF, S, AL1, M > /\
           adjustParameters(LE , AL1)  /\
           B1 := getF(MF, Q) /\
           M1 := addMemory(LE, AL1, mv) /\
          < B1, MF, push(M1, S), V, M > => < bv, MF, push(M2, S1), V, M > .
```

Para entrar a esta regla lo inicial que tiene que encajar es que la siguiente instrucción a procesar sea una llamada a una función con parámetros `CallP(Q,AL)` teniendo `Q` como el `Qid` que representa el nombre de la función y `AL` como la lista de argumentos.

Posteriormente, se llama a la función `FunHasReturn?` con el nombre de la función que se quiere ejecutar y la memoria de funciones para comprobar si la función introducida por el usuario devuelve algún parámetro, en caso de que si no encajaría esta regla.

`getsParametersFuncions` devuelve `LE` que son los parámetros que tiene la función descritos y que posteriormente comprobaremos si se corresponden con los introducidos por el usuario.

En el siguiente caso creamos otro tipo `executionTuple` donde en el `body` a ejecutar le pasamos la lista de argumentos de la función, esto genera otra rama de ejecución en la que se ejecuta dos reglas `ExpIsArglRecursive` y `ExpIsArgv`.

```
crl [ExpIsArglRecursive] :
< ArgL(E, AL), MF, S, V, M > => < bv, MF, S1, ArgL(V1, AL'), M1 >
        if  AL =/= argV /\
            < E, MF, S, V, M > => < bv, MF, S1, V1, M1 > /\
            < AL, MF, S, V, M > => < bv, MF, S1, AL', M1 >  .

rl [ExpIsArgV] :
< argV, MF, S, V, M > => < bv, MF, S, argV, M > .
```

El objetivo de estas reglas es parsear uno a uno los parámetros introducidos por el usuario y devolverlos todos procesados en el cuarto parámetro del `executionTuple`. En la regla `callParameters` se recoge el procesamiento anterior en la variable`AL1` y ejecuta la función `adjustParameters` con los parámetros de entrada definidos en la función y los parámetros procesados, esta devolvería cierto si ambas se tienen una correspondencia correcta.

Si los parámetros son correctos, se ejecuta la función `getF(MF, Q)`, con la memoria de funciones y el nombre de la función deseada y devuelve el conjunto de instrucciones contenidas en el parámetro B1(`tipo body`). Por último, ejecuta la función `addMemory(LE, AL1, mv)` que recibe los parámetros de la función con sus valores y devuelve la memoria local con los nuevos parámetros solo visible en las instrucciones de la función a ejecutar.

```
< B1, MF, push(M1, S), V, M >
```

Se construye el `executionTuple` con las instrucciones de la función a ejecutar (`B1`), la memoria de funciones (`MF`) inmutable en todo el proceso, la pila añadiéndole a la cima la memoria local (`M1`) y los parámetros de retorno si la función devolviese algo, en este caso no se utilizan (`V` y `M`).

```
crl [assig] :
< BodyF( =(E,E1), B), MF, S, V, M > =>  < B, MF, S1, noneV, M >
   if isQid(E) /\
     < E1, MF, S, V, M > =>  < bv, MF, S, V1, M > /\
     V1 =/= noneV /\
     S1 := storeMemoryStack(S, E, V1) .
```

Las cuatro instrucciones de la función init entran por la misma regla `assig`. Inicialmente, se comprueba que la primera expresión sea un `Qid` para distinguir donde se va a almacenar el valor calculado, si en una variable o una estructura de datos.

Posteriormente, se crea un *executionTuple* para procesar la expresión que se quiere asignar (`E1`) reduciéndola a un valor concreto (`V1`) que se recupera en el parámetro de retorno y comprueba que no es vacía (`noneV`). En el caso de la primera, tercera y cuarta instrucción, la regla con la que encaja es `ExpIsQid`.

```
crl [ExpIsQid] :
    < E ,  MF , S , noneV , M > =>  < bv ,  MF , S , V , M >
    if not isAcces(E) /\
    isQid(E) /\
    V := getValueStack(S , E) .
```

Esta regla es utilizada para reducir expresiones, se comprueba que lo recibido no sea un acceso, ya que encajaría con otra regla, y verifica que sea un `Qid`. Por último se llama a la función `getValueStack(S, E)` con la pila y el `Qid` que identifica el nombre de la variable del que se va a obtener el valor.

```
crl [add] :
    < E +. E1, MF, S, V, M > => < bv, MF, S, V3, M >
    if < E, MF, S, V, M > => < bv, MF, S, V1, M > /\
       < E1, MF, S, V, M > => < bv, MF, S, V2, M > /\
       V3 := V1 + V2  .
```

En el caso de la segunda instrucción por ajuste de parámetros encaja con la regla `add` que procesa las dos expresiones de entrada (`E1`,`E2`) hasta reducirlas en dos valores(`V1`,`V2`) que suma y devuelve en el parámetro de retorno (`V3`). En este caso, al ser dos `Qid` ejecutaría dos veces la regla `ExpIsQid` para obtener el valor de la pila de cada variable.

Por último se llama a la función `storeMemoryStack(S, E, V1)` con la pila, el `Qid` del nombre de la variable a la que se va a cargar el valor calculado y el valor propiamente dicho. Esta función devuelve la pila con las modificaciones realizadas (`S1`).

`Fin de ejecución de instrucciones: < bv , MF , push(M2 , S1) , V , M >`

Tras ejecutar las cuatro instrucciones, no puede continuar con ninguna regla y regresa a la regla que ha generado la ejecución que es `callParameters` con el body vacío (`bv`), la memoria de funciones y los parámetros de retorno iguales que en la entrada, ya que no es una función que devuelve un valor, y la pila modificada.

Para finalizar la ejecución de esta función se suprime la cima de la pila (`M2`) y la devuelve con los cambios aplicados.

Se ha diseñado un diagrama con un resumen de las instrucciones de la función y las reglas principales que desencadena.

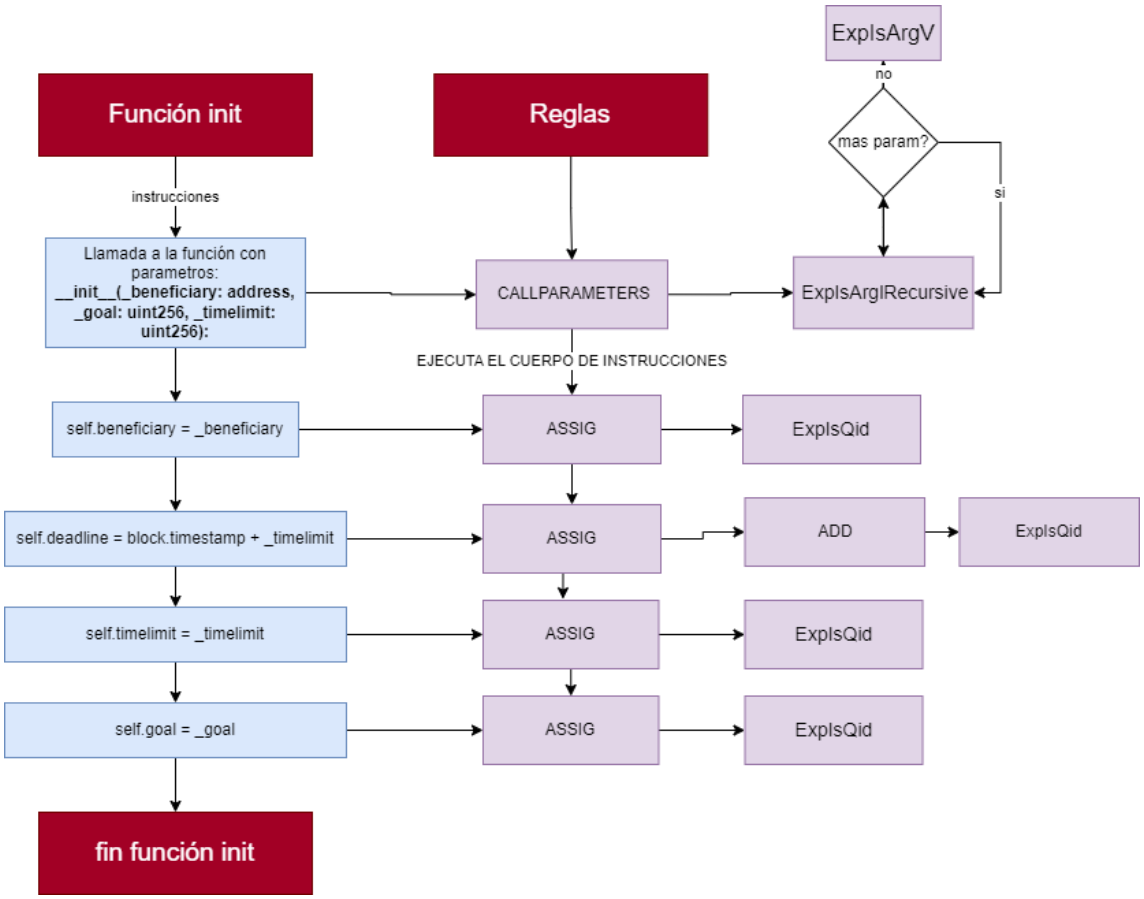

Figura 4.1: Diagrama de ejecución de instrucciones y sus consiguientes reglas de la función init

3. **Función refund**: Se llama a esta función tras la salida de la función `init`, en este caso la primera regla ejecutada es `callEnd`.

```
crl [callEnd] :
< BodyF(Call(Q), B), MF, S, V, M > => < B, MF, S, V, M >
   if not FunHasReturn?(MF,Q) /\
      B1 := getF(MF, Q) /\
     < B1, MF, push(mv, S), V, M > =>
                    < bv, MF, push(M1, S1), ["end"], M > .
```

La función `refund` no tiene parámetros, por lo que esta regla solo comprueba que la función no devuelva ningún valor(`FunHasReturn?`) y obtiene las instrucciones a ejecutar mediante la función `getF`.

A partir del body (`B1`) devuelto por la función y añadiendo a la pila una memoria vacía (`mv`) solo visible en esta función, se crea la rama de ejecución que da inicio a la ejecución de las instrucciones.

```
crl [assert] :
    < BodyF( Assert( E ), B), MF, S, V, M > => < B, MF, S, noneV, M >
    if < E, MF, S, V, M > => < bv, MF, S, V1, M > .
```

La primera instrucción encaja con la regla `assert` que resuelve la expresión (`E`) mediante otra rama de ejecución. En el código desarrollado no se comprueba que el valor reducido (`V1`) sea `True` para facilitar pruebas, pero solo en este caso podría continuar con la ejecución.

Regla para >=.

```
crl [greatherEqualThan]:
< E >=. E1, MF, S, V, M > => < bv, MF, S, V3, M >
    if < E, MF, S, V, M > => < bv, MF, S, V1, M > /\
       < E1, MF, S, V, M > => < bv, MF, S, V2, M > /\
       V1 :: Float /\ V2 :: Float /\
       V3 := V1 >= V2 .
```

Regla para <.

```
crl [lessThan] :
< E <. E1, MF, S, V, M > => < bv, MF, S, V3, M >
  if < E, MF, S, V, M > => < bv, MF, S, V1, M > /\
     < E1, MF, S, V, M > => < bv, MF, S, V2, M > /\
     V1 :: Float /\ V2 :: Float /\
     V3 := V1 < V2  .
```

Regla para and.

```
crl [and] :
< E and. E1, MF, S, V, M > => < bv, MF, S, V3, M >
    if < E, MF, S, V, M > =>  < bv, MF, S, V1, M > /\
       < E1, MF, S, V, M > => < bv, MF, S, V2, M > /\
       V3 := V1 and V2  .
```

La expresión por prioridad de operadores ejecutaría la regla `and`, que resuelve a su vez las dos expresiones (`E`, `E1`) que la forman. La primera expresión(E) encaja con la regla `greatherEqualThan` obteniendo los dos valores de la memoria a través de la regla `expIsQid`, comprobando que sean numéricos y que el primer valor sea mayor o igual que el segundo.

La segunda expresión (`E1`) hace lo mismo que la primera, pero pasando por la regla `lesthan` y comprobando que el primer valor sea menor que el segundo.

Con los valores booleanos ya resueltos (`V1`,`V2`) la regla realiza una operación `and` entre ambos y lo devuelve en el parámetro `V3`, que posteriormente utiliza `assert` para comprobar que sea cierto o falso.

```
crl [newVariable] :
    < BodyF(Dv(Q, V, E), B),  MF, S, V1, M > => < B, MF, S1, V1, M >
        if < E, MF, S, V1, M > =>  < bv, MF, S, V2, M > /\
        S1 := addNewVariableStack(S, Q, V, V2) .
```

La segunda instrucción de la función se ajusta a la regla `newVariable` donde recibe el nombre de la variable(Q), el tipo(V) y una expresión(E) que simplifica a un valor(V2), en este caso se recupera el valor de una variable simple de la memoria(con la regla `expIsQid`). Con todo calculado, ejecuta la función `addNewVariableStack(S, Q, V, V2)`, que devuelve la pila modificada(S1), con la nueva variable.

```
crl [ForRangeTrueFirstReturnEnd] :
< BodyF(ForR(Q, ArgL(E ,E1), B), B1),  MF, S, V, M > =>
    < bv,  MF, S3, ["end"], M >
  if  < E, MF, S, V, M > => < bv, MF, S, V1, M > /\
      < E1, MF, S, V, M > => < bv, MF, S, V2, M > /\
      V1 < V2 /\
     < Dv(Q, int128, V1), MF, S, V, M > => < bv, MF, S1, V, M > /\
     < B, MF, S1, V, M > => < bv, MF, S2, ["end"], M > /\
     S3 := delVariableStack(S2, Q)  .
```

La regla que encaja con la instrucción `for` que aparece en esta función es la etiquetada como `ForRangeTrueFirstReturnEnd` que está enfocada en bucles que acaban su ejecución de forma abrupta.

Inicialmente, se utilizan dos ramas de ejecución para reducir las dos expresiones (`E`, `E1`) que representan los índices inicio y fin del bucle. Tras su reducción (`V1`, `V2`) comprueba que valor inicio sea menor que el valor final.

Se define otra rama de ejecución con la instrucción `newVariable(DV)` para añadir a la cima de la pila la variable (`Q`) con el valor del índice inició (`V1`) con el fin de poder utilizarla dentro del bloque de instrucciones del `for`.

```
< B, MF, S1, V, M > => < bv, MF, S2, ["end"], M >
```

A través del último `ExecutionTuple`, se ejecuta el cuerpo (`B`) del `for` hasta que no queden instrucciones (`bv`) y devuelva un tipo `string`([.$^{\text{en}}$d"]). La regla etiquetada como `ForRangeTrueFirstReturnEnd` impone devolver un `string`, si esto no pasase, la rama de ejecución buscaría el encaje con otra regla distinta que no le obligue a devolver este tipo.

```
crl [ifBasicTREnd] :
< BodyF(If(E, B), B1), MF, S, V, M > => < bv, MF, S1, ["end"], M >
if < E, MF, S , V , M > => < bv, MF, S, V1, M >  /\
    V1 /\
    < B , MF , S , V , M > => < bv , MF , S1 , ["end"] , M > .
```

`ifBasicTREnd` es la primera regla por la que pasa el cuerpo del `for`, donde se reduce la expresión(E) para obtener el valor booleano que indicara si se ejecuta o no el cuerpo del `if(B)`.

En la prueba que se está analizando entraría a las instrucciones del if, puesto que compara el valor de inicio con el valor de la variable `nextFunderIndex` y ambas son

0. En el caso de que la evaluación de la expresión fuese falsa, ejecutaría la instrucción posterior al `if(B1)`.

```
rl [ExpIsReturn] :
    < BodyF(return, B), MF, S, V, M > => < B, MF, S, ["end"], M > .
```

El cuerpo del `if(B)` está formado por dos instrucciones, una asignación simple que ejecuta la regla `assig` donde le da a la variable `refundIndex` el valor de `nextFunderIndex`.

La segunda instrucción ejecuta la regla `expIsReturn` que comprueba si aparece el `controlFlow` *return* y anula las distintas instrucciones que pudieses existir después. Por último, devuelve el valor `[end]` para indicar a las reglas de la misma rama de ejecución previas, la finalización de la función. En esta función también se ha diseñado un diagrama para representar las reglas e instrucciones ejecutadas. ExecutionTuple

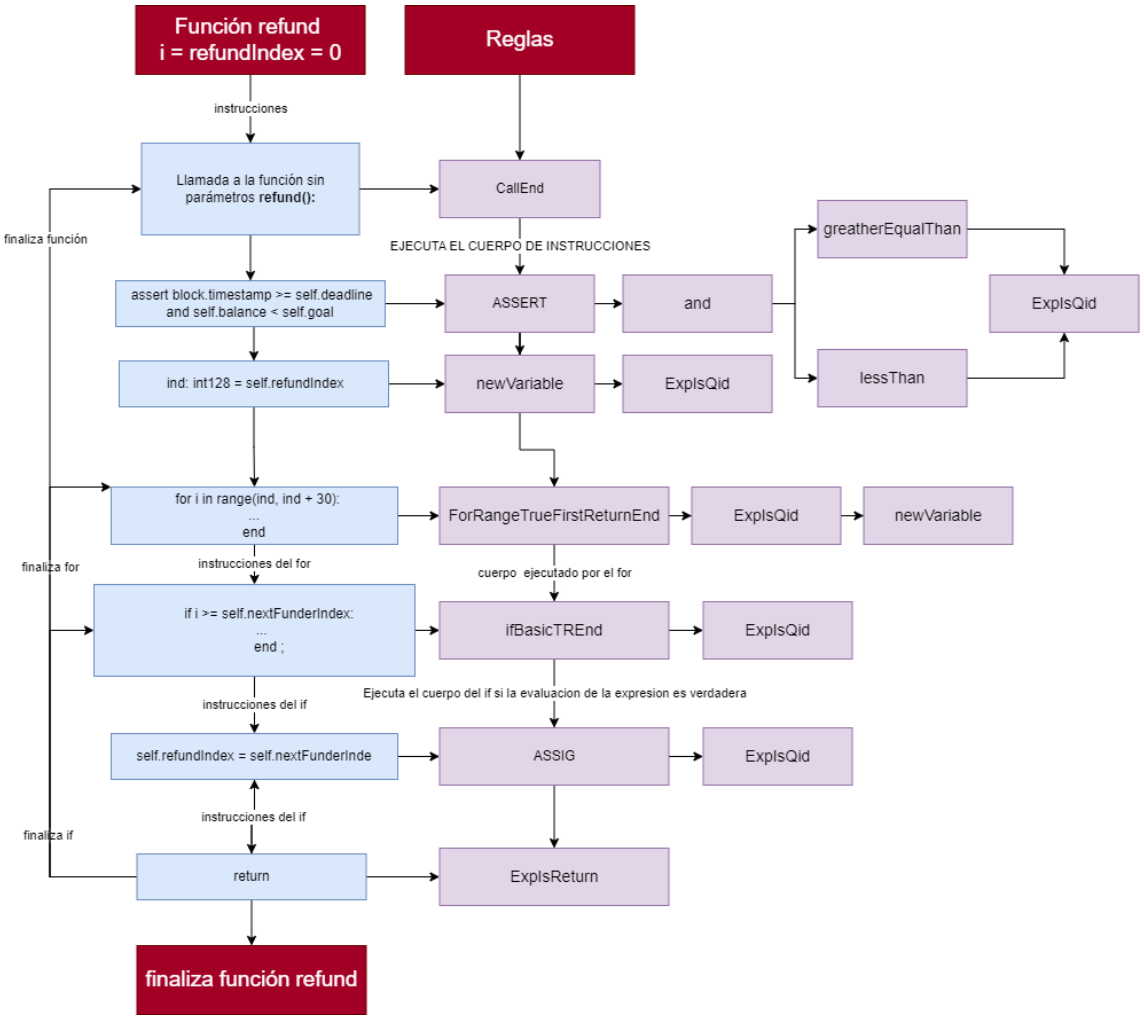

Figura 4.2: Diagrama de ejecución de instrucciones y sus consiguientes reglas de la función refund, con salida abrupta

de final del if y de retorno al for

```
< bv, MF, S1, ["end"], M >
```

Al aparecer esta sentencia, el `if` se finalizaría, ya que el valor recibido por parámetro es `[.end"]`, por lo que proyectamos esta variable en el `executionTuple` de retorno, para que la vayan recogiendo los padres (ejecutores de la llamada).

Por último, al llegar esta misma variable al `for` se realiza la misma comprobación y se finaliza la vuelta actual que devolvería la variable, por si hubiese ejecutado varias vueltas que coinciden con varias reglas y vaya finalizando una a una hasta llegar a la primera regla de todas, eliminando en cada una la variable `i` de la cima de la pila.

Esta última regla siempre es un `call` que terminaría también sin devolver nada, ya que la instrucción `return` indica fin de ejecución sin retorno de valores.

4. **Función refund versión 2**: En este apartado se va a repasar la función refund, pero en el caso de que la expresión evaluada por el `if` devolviese falsa porque la variable `nextFunderIndex` tiene valor 1. En este caso las reglas `call`, `assert` y `newVariable` se ejecutan de la misma forma que el ejemplo anterior. Pero la instrucción `for` encajaría con la regla `ForRangeTrueReturnEnd`.

```
crl [ForRangeTrueReturnEnd] :
< BodyF( ForR(Q, ArgL(E, E1), B), B1), MF, S, V, M >
                                    => < bv,  MF, S4, ["end"], M >
  if  < E, MF, S, V, M > => < bv, MF, S, V1, M > /\
      < E1, MF, S, V, M > => < bv, MF, S, V2, M > /\
      V1 < V2 /\
      < Dv( Q, int128, V1 ), MF, S, V, M > => < bv, MF, S1, V, M > /\
      < B, MF, S1, V, M > => < bv, MF, S2, V, M > /\
      S3 := delVariableStack(S2, Q) /\
      V3 := V1 + 1.0 /\
      < ForR(Q, ArgL(V3, E1), B), MF, S3, V, M > => < bv, MF, S4, ["end"], M > .
```

Esta regla encajaría en el caso que el cuerpo del `for(B)` se ejecuta de forma satisfactoria, devolviendo un cuerpo sin instrucciones(bv) y la pila modificada(S2). Para que se logre llegar a este fin de instrucciones se pasa por la regla `ifBasicF`, `callParameters`, `assigAccessMemory` y `CallnewStruct`.

```
crl [ifBasicF] :
< BodyF( If(E, B), B1),  MF, S, V, M > => < B1,  MF, S, V, M >
     if < E, MF, S, V, M > => < bv, MF, S, V1, M > /\
        V1 == false  .

crl [assigAccessMemory] :
< BodyF( =(E,E1), B), MF, S, V, M > => < B, MF, S1, V, M >
if isAcces(E) /\
< E1, MF, S, V, M > => < bv, MF, S, V, M1 > /\
Q := getAccessQid(E) /\ E2 := getAccessExp(E) /\
< E2, MF, S, V, M > => < bv, MF, S, V1, M > /\
S1 := storeMemoryHMStructStackWQ1(S, Q, V1, M1) .

 crl [CallnewStruct] :
< CallP('empty, AL) , MF, S, V, M > => < bv, MF, S, V, M1 >
    if M1 := getMemoryStack(S, AL)  .
```

```
crl [ExpIs->.] :
< .(->(Q, E), V1), MF, S, V, M > => < bv, MF, S, V3, M >
if < E, MF, S, V, M > => < bv, MF, S, V2, M > /\
    V3 := getValue.->Stack(S, Q, V2, V1) .
```

La regla `ifBasicF` evalúa la expresión (`E`) donde la variable `i` es cero y `nextFunderIndex` es uno, por lo que devolvería falso y no se ejecutaría el cuerpo del `if(B)` y se pasaría directamente a la siguiente instrucción (`B1`).

La regla `callParameters` ejecuta la función `send` propia de `Vyper` utilizando la regla `ExpIs->.` para parsear los parámetros de entrada, obteniendo de la pila el valor de la variable de una estructura contenida en un `hashmap` mediante la función `getValue.->Stack`.

La última instrucción del `for` ejecuta la regla `assigAccessMemory` que tiene como objetivo almacenar una estructura dentro de un `hashmap`.

Esta regla comprueba que la expresión(E) sea un acceso(contiene la dirección de almacenaje), lo descompone para obtener el nombre del hashmap(Q) y la clave de almacenamiento(V1).

Por otro lado, genera una rama de ejecución con la segunda expresión(E1) que a partir de la regla `CallnewStruct` devuelve la estructura vacía, ya que se llama a la función `empty` para inicializarla(M1).

Con las tres variables procesadas, ejecuta la función `storeMemoryHMStructStackWQ1` para modificar la pila(S) con los nuevos datos.

```
Final de la regla ForRangeTrueReturnEnd
< B, MF, S1, V, M > => < bv, MF, S2, V, M > /\
S3 := delVariableStack(S2, Q) /\
V3 := V1 + 1.0 /\
< ForR(Q, ArgL(V3, E1), B), MF, S3, V, M > => < bv, MF, S4, [Str], M >
```

Al finalizar todas las instrucciones del `for(bv)`, elimina de la cima de la pila el iterador (`Q`), aumenta en 1 el valor de la variable menor del `for(V1)` y construye otra regla `forR` para continuar con la siguiente iteración. En la siguiente iteración ocurrirá lo explicado en el apartado anterior, que entraría al cuerpo del `if` y finalizaría el bucle de forma abrupta mediante la instrucción `return`.

El diagrama de esta función, es parecida al anterior, pues es la misma función, pero con una ejecución de instrucciones y reglas distintas. En esta función también se ha diseñado un diagrama para representar las reglas e instrucciones ejecutadas.

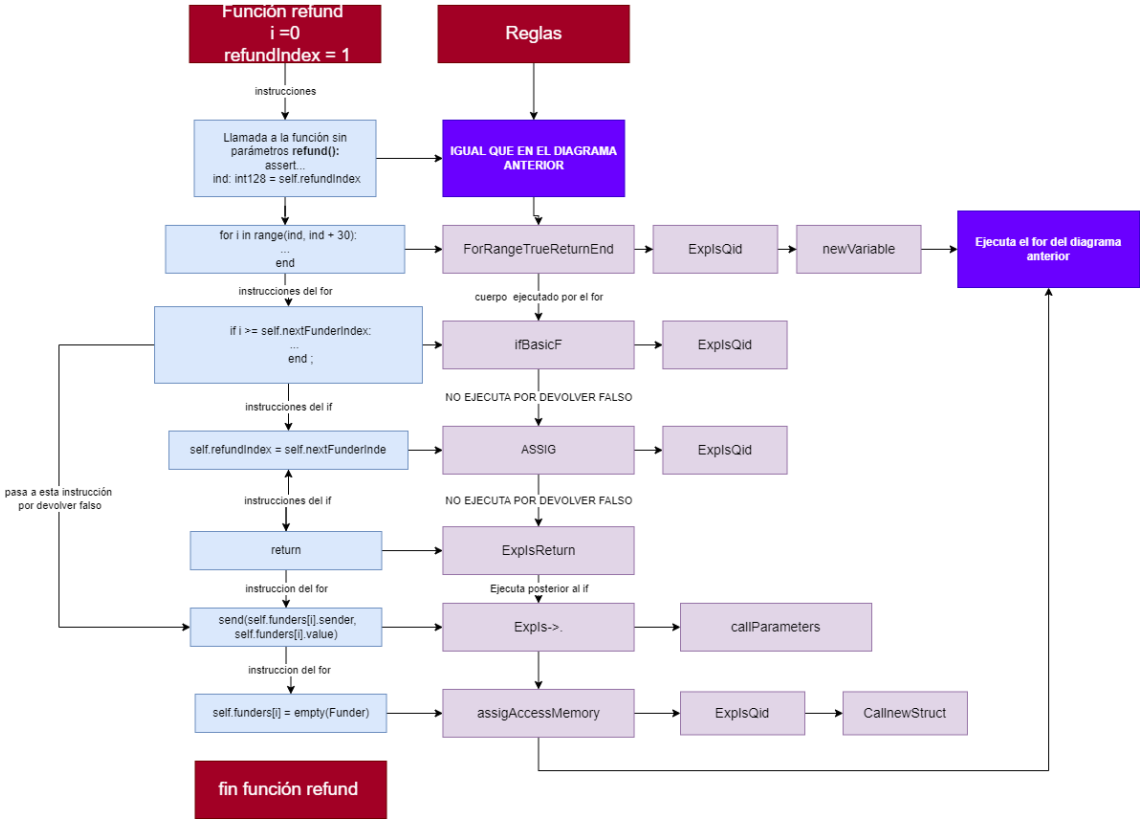

Figura 4.3: Diagrama de ejecución de instrucciones y sus consiguientes reglas de la función refund, con una vuelta del bucle y salida abrupta

Capítulo 5

Conclusiones y Trabajo Futuro

Se han desarrollado las conclusiones del trabajo, partiendo de los objetivos planteados como referencia inicial. Las partes que se plantearon y no se han podido realizar, por falta de tiempo o por falta de conocimiento, se ha dejado como trabajo futuro.

5.1. Conclusiones

Se ha incrementado la base de conocimiento centrado en **blockchain**, entendiendo la aparición de esta tecnología con el objetivo de la verificación de datos, que trajo consigo la necesidad de los `contratos inteligentes` para verificar la seguridad de las partes participantes en este tipo de contratos. También entendiendo a nivel de código, ya sea para **Solidity** o **Vyper** los recursos que tienen que cumplir los contratos para asegurar la seguridad y legibilidad.

Aun teniendo una base en **Maude**, donde se ha desarrollado el código, y de `Python`, que es el lenguaje del que parte **Vyper**, se han aprendido conceptos avanzados de ambos lenguajes para permitir la codificación de todo el programa. Iniciando con un estudio de las definiciones de gramáticas que usan el meta-nivel de otros lenguajes en Maude para observar las distintas posibilidades que nos ofrece un lenguaje de estas características. Con Vyper se ha estudiado el lenguaje al completo repasando distintos contratos a través de los cuales se logra entender las diferencias principales con `Python` y peculiaridades que hacen que estos sean `contratos inteligentes`.

En cuanto al desarrollo de código, se ha logrado crear un *preparse* que adapta los contratos a una forma que pueda ser procesada por **Maude** que tiene sus peculiaridades que hace que la forma básica de un contrato no pueda ser *metaparseada* directamente. No todas las adaptaciones han sido posibles y se han dejado explicadas en el trabajo futuro. Muchas de las definiciones del *preparse* han surgido según se iban ejecutando los distintos contratos de ejemplo. La definición de este `preparse` fue compleja por intentar a través de todos los medios posibles que `Maude` aceptase la forma en la que está escrito **Vyper** y que en la mayoría de casos no fue posible y se resolvió adaptando los contratos.

Al adecuar un contrato para que sea aceptado por el *metaparse* propio de **Maude** retorna un contrato preparado para pasarlo por el *parse* que se ha definido, y que nos devuelve si la gramática recibida tiene la forma esperada que tienen que cumplir todos los contratos de `Vyper`.

Esto sería un primer verificador de la forma del contrato y logra asignarle a cada instrucción el tipo teórico definido en el `metaparse` que se ha desarrollado. Este `parse`

ha sido retocado en todo el desarrollo por la aparición en distintos contratos de formas complejas de utilización de la sintaxis que no se había tenido en cuenta.

Este *metaparse* propio es un gran avance porque da la visión de una primera forma de un contrato de Vyper en `Maude`, en este punto la gramática esta basada en **@token@ y @name@**.

También se ha definido una gramática similar basada en tipos con los que trabajará el futuro módulo de reglas que va a servir para dar un siguiente paso en el tratamiento del contrato. En este punto se tendrá una sintaxis tipada, que distingue entre los tipos básicos de `Vyper` con los que se construyen sentencias complejas. Aunque existan tipos básicos que son los que conforman el programa, internamente en Maude no se han definido todos por la complejidad de la inferencia entre ellos, ya que en Vyper existen diversidad de tipos y las trasformaciones entre ellos que complicaban la sintaxis.

A partir de la segunda gramática, se ha definido un **módulo de reglas** que es el que aporta funcionalidad a toda la sintaxis. Este módulo es el que ejecuta funciones y reglas al encontrarse definido los tipos de la sintaxis, ha sido probado con todos los contratos de ejemplo y se han obtenido resultados satisfactorios.

Este módulo ha sido el más complicado de desarrollar debido a todas las posibilidades que existen dentro de la ejecución de un programa y la diferenciación de reglas muy parecidas para evitar la ejecución infinita en caso de que no se ajuste la siguiente instrucción con ninguna regla existente o que tome caminos que no llegan a una salida correcta.

Se han desarrollado **estructuras de datos** para que las reglas las utilicen a lo largo de su ejecución y para representar la forma interna que tendrían las variables complejas en Vyper como *hashmap* o estructuras. También almacenan direcciones, funciones y variables simples, todo lo necesario para la correcta ejecución de un contrato. Muchas de estas estructuras han surgido por la necesidad que aparecía al desarrollar el módulo de reglas. La dificultad está en distinguir todos los tipos posibles que se pueden dar dentro de las variables y como trabajar según los que aparezcan, muchas funciones han sido desarrolladas para poder trabajar con estas estructuras de datos, que permiten: buscar, almacenar, borrar, insertar... La mayor parte del tiempo en esta parte ha sido en la creación de estructuras de datos complejas que están compuestas por otras más simples, pero que tienen que interactuar entre ellas.

La última parte desarrollada ha sido el **módulo de entrada/salida** que comunica el programa con el usuario, esto nos facilita elegir un contrato, una función y los parámetros que se quieran pasar. Tener esta unión de las dos partes hace que la prueba de los contratos sea mucho más rápida y que se detecten de forma más sencilla los errores, ya sean errores de forma al llamar a algún contrato o función, y errores en la propia sintaxis del contrato cargado.

Con todo lo anterior completado y conectado, se puede concluir que existe un sistema desarrollado en Maude que se comunica con el usuario y es capaz de cargar contratos inteligentes escritos en Vyper, comprobar su sintaxis, la corrección de su forma y tipado e incluso recrear ejecuciones de estos contratos con sus salidas correspondientes.

Además, existen **diez contratos de ejemplo**, varios con instrucciones y funciones de alta complejidad, que ejecutan de forma satisfactoria. Por lo que se tiene una gran cantidad de ejemplos que hace que los módulos desarrollados hayan pasado por un conjunto de pruebas y corrección muy extenso.

Por último, cabe mencionar la utilidad de la ejecución por reglas, ecuaciones y funciones que ha permitido que a lo largo del programa se pueda tener una trazabilidad profunda de lo que se está ejecutando para verificar su corrección o en caso contrario, el punto exacto de error.

5.2. Trabajo futuro

Se han desarrollado ideas que se plantearon en un principio y no se han podido realizar, ideas que han ido apareciendo a lo largo del desarrollo y se han dejado para un futuro e incluso algunas que no se plantearon en ningún momento y no se sabe si se pudiesen implantar, pero harían que el trabajo continuase por caminos con muchas posibilidades:

- **Preparse**: La forma en la que `Maude` ajusta por parámetros al leer un contrato, hace que este tuviese confusiones, mezclando distintas instrucciones dentro del mismo bloque, e incluso mezclando bloques como por ejemplo una sentencia *for* o *if* dentro de una función. Al final de cada instrucción el preparse añade punto y coma, esto soluciona gran parte del problema, pero no en su totalidad, por eso se ha optado por añadir manualmente la palabra `end` al final de cada bloque de instrucciones. Sería muy productivo conseguir que el preprocesamiento o un programa previo detectase los bloques y añadiese esta palabra.

- **Instrucciones dobles**: Otro problema relacionado con lo anterior, es que una instrucción se puede dividir en dos líneas, esto `Maude` no lo procesa, traduce como si fueran dos instrucciones. La solución sería entender el fin de las instrucciones y reducirla a una línea o conseguir que `Maude` entienda instrucciones en líneas sucesivas.

- **Nombres de variables**: La denominación de variables con el mismo nombre cambiando mayúsculas y minúsculas, esto pasa en el contrato 8 donde una estructura y una variable local se llama de la misma forma y da problemas de ejecución.

- **Función predefinida** Se ha definido las funciones `send y selfdestruct` que se utilizan en `Vyper` para enviar gas y borrar el contrato, pero no se ha implementado el cuerpo para estas funciones, aunque se llamen, no tendrían ningún efecto interno.

- **Variable msg.sender y msg.value**: Estas variables representan la dirección del usuario que envía el gas y la cantidad enviada. En este caso se han dado unos valores por defecto, pero podrían ser parametrizadas según quien ejecuta el contrato.

- **Definición de tipos**: En las reglas y funciones se hacen comprobaciones para diferenciar números (solo **float**), string y booleanos, pero no se han definido todos los tipos de números existentes en `Vyper` ni direcciones que actualmente han sido representadas por números.

- **Interfaces**: El uso de interfaces para la comunicación entre contratos inteligentes no ha sido implementada, por falta de pruebas y por exceder la dimensión del planteamiento inicial.

- **Errores en archivo**: Durante toda la ejecución de un contrato se puede visualizar en la consola, todas las reglas utilizadas, para localizar errores y ver la trazabilidad. Se podría escribir toda la ejecución de un contrato en archivos para repasarlo y extraer información analizable con mayor facilidad.

- **Interfaz gráfica**: Modificar la ejecución de consola con una interfaz gráfica para seleccionar los contratos, ver ejecuciones anteriores, comprobación de errores y ver el valor de las variables del contrato durante de la ejecución.

- **Calculadora de gas**: Al ser un prototipo de verificación de contratos, no se ha implementado una construcción interna para trabajar con el concepto de gas y tener en

cuenta la finalización de ejecución por falta de gas o mantener la cuenta en cualquier punto de ejecución con el gas invertido.

- **Pruebas extra**: Se han ejecutado de forma satisfactoria diez contratos, pero sería positivo extender las pruebas para observar distintas casuísticas que no se hayan tenido en cuenta.
- **Verificación de propiedades**: Inicialmente, se planteó a aparte de la especificación del sistema, la definición de un conjunto de propiedades para verificar que comprobasen fórmulas, ecuaciones y reglas de la lógica. Por tamaño y dificultad del código no ha sido posible realizar esta definición de propiedades.

Chapter 6

Introduction

In this chapter we introduce the concepts of blockchain, Ethereum and smart contract, giving an overview that will help to understand the motivation, objectives and development of this final projects.

The blockchain (Guardeño et al.) can be understood as a distributed, synchronized and secure database, where each record corresponds to a block and the existence of nodes (users) is necessary to verify and validate different transactions. These characteristics make it possible to eliminate intermediaries in the transactions carried out, making blockchain very popular in the last few years and creating a revolution in different areas, with the economy being one of the most relevant.

The blocks in the blockchain (bybit learn) record transactions in chronological and irreversible order. The concatenation of blocks is done through cryptographic processes, which ensures that new blocks can be added without the previous ones being replaced or modified. Over time, the network will become larger and larger. The use of hash functions, a cryptographic algorithm that transforms input data into fixed-length chain, makes it possible to authenticate existing transactions, as this chain cannot be duplicated.

Each block is identified by header information consisting of the blockchain version number, the Unix time, the hash pointers, the value required by the miners to create a block (nonce) (explained below) and the hash of a Merkle root, which is a layered structure to facilitate the verification of large amounts of data thanks to the root node. This pointer is the one that marks the address of the next block, which would function as a linked list where each pointer marks the correct address to other variables.

In relation to the blockchain, it is important to mention the *proof of work* algorithm, which is used to confirm transactions and generate new blocks. This algorithm requires that a block is added to the network only if it obtains the random value (*nonce*, a one-time, 4-byte number) that concatenated with the block header results in a digest that has a certain number of leading zeros. This searching for leading zeros is a computationally expensive problem.

The blockchain is a *Peer to Peer*(P2P) network, where all nodes have the same rights and perform the same functions. This is what ensures that all participants operate under the same rules, thus ensuring that a transaction is completed satisfactorily and that there is no inconsistent data. Blockchain technology creates *tokens*, which we can understand as 'currencies' and which, although they have no value in themselves, can acquire value depending on who uses them and for what purpose.

Ethereum (Wackerow et al., 2022) is a digital platform that adopts blockchain technol-

ogy and extends its use to a wide variety of applications. Ether is its native cryptocurrency, i.e. it is the token used in transactions. Data stored in the network has to be verified by at least 51% of the nodes. Each transaction uses a quantity of gas, which is the unit of measurement for any work performed within the network. Therefore, the more complex the job, the more gas it consumes.

Ethereum has been designed so that the transactions that are carried out meet different conditions. These conditions are defined in smart contracts (Arvelo). Smart contracts are contracts that have the ability to be executed automatically. The main differences with traditional paper-based contracts are that they are computer programs, their performance is not subject to the interpretations of either party and no intermediary, such as a notary's office, is required.

The first idea of a smart contract dates back to the 1990s (Kemmoe et al., 2020), But in this decade the technology was not advanced enough to ensure that these contracts were secure. There were problems such as the manipulation of the software implementing the written contract, which could not ensure its security, and the strong regulation of money.

It is the blockchain that can provide a solution to these problems, as it is an immutable database maintained by a network of computers where anything can be recorded. It avoids computer manipulation and also introduces the concept of cryptocurrency, which is a digital and decentralized asset that does not belong to any entity and facilitates the transfer of money.

Within smart contracts, it is essential to mention the role of oracles. These are instruments that allow contracts to interact with the real world, updating internal states with information from outside the contract. These mechanisms have problems of centralization, which goes against the philosophy of blockchain, because if the oracle being used has wrong information or has been manipulated, or the server has failed, it depends only on this centralized oracle. To solve this type of problem, there is a move to combine results from different providers and make the decision on the basis of the majority and thus decentralize the result.

Here are the advantages and disadvantages of smart contracts:

Advantages:

- Eliminate third party costs.
- Greater accessibility to all audiences due to simplicity and low costs.
- Increased security, transparency and efficiency.
- Reduce information asymmetries, lack of knowledge or trust of either party.
- Positive externalities, chain information is created, shared and stored for use.

Disadvantages:

- Potential errors in the face of future technological change.
- To ensure that the blockchain becomes a legal entity, validated and accepted by all intermediaries.
- Need to standardise and regulate use cases, such as P2P or B2B contracts, which would be contracts between individuals (P2P) and contracts between entities (B2B) that use tokens as a payment method.

Smart contracts can be programmed using languages that are relatively easy for the programmer to use. Two clear examples of such languages are Solidity (Solidity) and Vyper (Vyper) which are the most commonly used (ethereum.org). More experienced developers also use Yul (Authors).

Solidity is a statically typed, high-level object-oriented programming language influenced by C++, Python and JavaScript. The main characteristics that define it are inheritance, the existence of libraries to generate reusable code and the definition of complex user-defined types. The advantages and disadvantages of this language are listed below:

Advantages:

- Many learning resources, as it is the most frequently used language for Ethereum.
- Many tools for developers, such as the online editor Remix Remix.
- Accepts string and dynamically sized arrays.
- Exception handling.

Disadvantages:

- Possibility of overflow, if the values to be stored are too large.
- Scalability, not ready to handle a large number of transactions.
- Security problems.

The second most frequently used language is **Vyper**. Vyper is a "pythonic" programming language developed principally to solve the security problems that existed in Solidity. Vyper has eliminated some concepts, such as object orientation and inheritance. Also, it has added strong typing, i.e. no data type violations are allowed, it is not possible to use one type as another unless a conversion is done. The objective of all these changes was to create contracts that are auditable and less prone to errors. The advantages and disadvantages of this language are also presented below:

Advantages:

- Simpler language for Python developers to get started with.
- Transparent language to facilitate security and legibility.
- Increased use of development tools such as Brownie and Etherscan, to develop online and test smart contracts.
- Limiting chain and array sizes to prevent attacks.
- Possibility to calculate an upper gas limit for any function call.

Disadvantages:

- There is not as much community support as in Solidity.
- There are no modifiers, inheritance, recursive calls and dynamic data types.
- Still under development, many functions are not yet in Vyper.

Chapter 7

Conclusions and Future Work

In this section, we will present the conclusions obtained from the evaluation of the finalized work, based on the objectives set as an initial reference of what we wanted to achieve and what has been achieved. On the other hand, what we have not managed to develop, either due to lack of time or lack of means and knowledge, and we have decided to leave as future work.

7.1. Conclusions

The knowledge base focused on blockchain has increased, understanding the emergence of this technology for the purpose of data verification, which brought with it the need for smart contracts to verify the security of the parties participating in this type of contract. Also, understanding at the code level, either for `Solidity` or `Vyper` the resources that contracts have to comply with to ensure security and legibility.

Even having a base in `Maude`, where the code has been developed, and in `Python`, which is the language from which `Vyper` starts, advanced concepts of both languages have been learned to allow the coding of the whole program. Beginning with a study of the definitions of meta-level grammars of other languages in Maude to observe the different possibilities offered by such a language. With Vyper, the language has been studied in full by reviewing different contracts, through which the main differences with Python and peculiarities that make these contracts smart contracts can be understood.

In relation of code development, it has been possible to create a preprocessing that adapts contracts to a form that can be processed by Maude. has managed to create a preprocessing that adapts the contracts to a form that can be processed by Maude, which has its own peculiarities that mean that the basic form of a contract cannot be meta-processed directly. Not all adaptations have been possible and have been explained in future work. Many of the definitions of pre-processing have emerged as the various example contracts have been executed. The definition of this preparse was complicated by trying in every possible way to get Maude to accept the form in which Vyper is written, which in most cases was not possible and was solved by adapting the contracts.

When a contract is adapted to be accepted by Maude's own meta-parse, it returns a contract ready to be passed through the parse that has been defined, and which returns whether the received grammar has the expected form that all Vyper contracts have to comply with. This would be a first checker of the contract shape and manages to assign to each instruction the theoretical type defined in the meta-parse that has been developed.

This parse has been adjusted throughout the development due to the emergence in various contracts of complex forms of syntax usage that had not been taken into consideration.

This meta-parse itself is a breakthrough because it gives the vision of a first form of a Vyper contract in Maude, stressing that at this point, the grammar is meta-represented from which further work is to be done.

A similar type-based grammar has been defined for the future rules module that will serve as a next step in the treatment of the contract. At this point you have a typed syntax, which distinguishes between the basic Vyper types with which complex sentences are constructed. Although there are basic types that make up the programme, internally in Maude not all of them have been defined due to the complexity of the inference between them, since in Vyper there is a diversity of types and the transformations between them that complicate the syntax.

From the second grammar, a rules module has been defined which provides functionality to the whole syntax. This module is the one that executes functions and rules as the syntax types are defined, it has been tested with all the example contracts and satisfactory results have been obtained. This module has been the most complicated to develop due to all the possibilities that exist within the execution of a program and the differentiation of very similar rules to avoid infinite execution in case the next instruction does not match any existing rule or takes paths that do not lead to a correct output.

Data structures have been developed for rules to use throughout their execution and to represent the internal form that complex variables would have in Vyper as hashmaps or structures. They also store addresses, functions and simple variables, everything necessary for the correct execution of a contract. Many of these structures have arisen out of the need that arose when developing the rules module. The difficulty is to distinguish all the possible types that can be given within the variables and how to work according to the ones that appear, many functions have been developed to be able to work with these data structures that allow you to: search, store, delete, insert. Most of the time in this part has been spent on creating complex data structures that are composed of simpler ones, but which have to interact with each other. Most of the time in this part has been spent on the creation of complex data structures that are composed of simpler ones but have to interact with each other.

The last part developed has been the input/output module that communicates the program with the user, which makes it easier to choose a contract, a function and the parameters to be passed. Having this union of the two parts makes the testing of contracts much faster and makes it easier to detect errors, whether they are form errors when calling a contract or function, or errors in the syntax of the loaded contract itself.

With all the above completed and connected, it can be concluded that there is a system developed in Maude that communicates with the user and is able to load smart contracts written in Vyper, check their syntax, the correctness of their form and typing and even recreate executions of these contracts with their corresponding outputs.

In addition, there are ten example contracts, some with complex instructions and functions, which execute satisfactorily, so that many examples can be seen, which means that the modules developed have undergone a very extensive set of tests and corrections, as well as certain cases that are difficult to see at first glance have been thought out and tested.

Finally, it is worth mentioning the utility of the execution by rules, equations and functions, which has made it possible to have a deep tracking of what is being executed throughout the program in order to verify its correction or, if not, the exact point where it has failed.

7.2. Future work

In this section we will explain ideas that were originally proposed but could not be implemented, ideas that have appeared during the development and have been left for the future and even some that were not considered at any time, and it is not known if they could be implemented but would allow the work to continue along paths with many possibilities:

- **Preparse**: The form in which Maude adjusts by parameters when reading a contract, makes it confusing, mixing different instructions within the same block, and even mixing blocks such as a for or if statement within a function. At the end of each instruction the preparse adds a semicolon, this solves a large part of the problem, but not all of it, so we have chosen to manually add the word end at the end of each block of instructions. It would be very productive to get the preparse or a previous program to detect the blocks and add this word.

- **Double instructions**: Another problem related to the above, is that an instruction can be split into two lines, this is not understood by Maude, it translates as if they were two instructions. The solution would be to understand the end of the instructions and reduce it to one line, or get Maude to understand instructions in successive lines.

- **Variable names**: The naming of variables with the same name is case-sensitive, this happens in contract 8 where a structure and a local variable are named the same way and gives execution problems.

- **Predefined function**: The functions **send and selfdestruct** which are used in Vyper to send gas and delete the contract have been defined, but the body for these functions has not been implemented, even if they were called they would have no internal effect.

- **Variable msg.sender and msg.value**: These variables represent the address of the user sending the gas and the quantity sent. In this case, default values have been given, but they could be parameterized according to who executes the contract.

- **Definition of types**: Checks are made in the rules and functions to differentiate between numbers (float only), strings and booleans, but not all existing number types have been defined in the text and addresses that are currently represented by numbers.

- **Interfaces**: The part of interfaces for communication between smart contracts has not been implemented, due to lack of testing and because it exceeds the dimensions of the initial approach.

- Errors in file: During the execution of a contract, all the rules used can be visualized in the console, in order to locate errors and see the traceability. The entire execution of a contract can be written in files to review it and extract information that can be studied more easily.

- **User Interface**: Modify the console execution with a graphical interface to select contracts, view previous executions, check for errors and view the value of contract variables during execution.

- **Gas calculator**: As a contract verification prototype, no internal construct has been implemented to work with the gas concept and to account for termination of execution due to lack of gas or to maintain the account at any execution point with gas invested.
- **Extra tests**: Ten contracts have been satisfactorily implemented, but it would be good to extend the testing to look at different cases that have not been taken into account.
- **Property verification**: Initially, apart from the specification of the system, the definition of a set of properties to verify that they check formulas, equations and logic rules was proposed. Because of the size and difficulty of the code, it has not been possible to carry out this definition of properties.

Bibliografía

Albert, E., Correas, J., Gordillo, P., Román-Díez, G. y Rubio, A. GASOL: gas analysis and optimization for Ethereum smart contracts. En *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II* (editado por A. Biere y D. Parker), vol. 12079 de *Lecture Notes in Computer Science*, páginas 118–125. Springer, 2020.

Albert, E., Correas, J., Gordillo, P., Román-Díez, G. y Rubio, A. *Don't run on fumes* - parametric gas bounds for smart contracts. *J. Syst. Softw.*, vol. 176, página 110923, 2021.

Arvelo, P. M. M. Los contratos inteligentes y su incorporacion en ordenamiento juridico. ????

Authors, T. S. Yul documentation. ????

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. y Talcott, C. L., editores. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, vol. 4350 de *Lecture Notes in Computer Science*. Springer, 2007. ISBN 978-3-540-71940-3.

Clavel, M., Eker, S., Lincoln, P. y Meseguer, J. Principles of maude. vol. 4, páginas 65–89, 1996.

ethereum.org. Lenguajes de contrato inteligente. ????

Guardeño, D. A., diaz Vico, J. y Encinas, L. H. ¿que sabemos de blockchain? ????

Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B. M., Park, D., Zhang, Y., Stefanescu, A. y Rosu, G. KEVM: A complete formal semantics of the Ethereum virtual machine. páginas 204–217, 2018.

Kemmoe, V. Y., Stone, W., Kim, J., Kim, D. y Son, J. Recent advances in smart contracts: A technical overview and state of the art. *IEEE Access*, vol. 8, páginas 117782–117801, 2020.

bybit learn. ¿qué es el hash en blockchain? ????

Meseguer, J. y Rosu, G. The rewriting logic semantics project. *Theor. Comput. Sci.*, vol. 373(3), páginas 213–237, 2007.

Remix. Remix. 2023.

Serbanuta, T. y Rosu, G. K-maude: A rewriting based tool for semantics of programming languages. vol. 6381, páginas 104–122, 2010.

Solidity. Solidity. 2023.

Team, V. Vyper documentation. ????

Verdejo Lopez, J. A. Maude como marco semantico ejecutable. vol. 1, página 289, 2003.

Vyper. Vyper. 2020.

Wackerow, P., Joshua, Smith, C. y Moya, T. Introducción a Ethereum. 2022.