Solidity Documentation

Versión UNKNOWN

Ethereum

31 de octubre de 2017

Índice general

Solidity es un lenguaje de alto nivel orientado a contratos. Su sintaxis es similar a la de JavaScript y está enfocado específicamente a la Máquina Virtual de Etehreum (EVM).

Solidity está tipado de manera estática y acepta, entre otras cosas, herencias, librerías y tipos complejos definidos por el usuario.

Como verá, es posible crear contratos para votar, para el crowdfunding, para subastas a ciegas, para monederos muti firmas, y mucho más.

Nota: La mejor manera de probar Solidity ahora mismo es usando Remix (puede tardar un rato en cargarse, por favor sea paciente).

CAPÍTULO 1

Enlaces útiles

- Ethereum
- Registro de Cambios
- Trabajos Pendientes (Story Backlog)
- Código Fuente
- Ethereum Stackexchange
- Chat de Gitter

CAPÍTULO 2

Integraciones de Solidity disponibles

- **Remix** Entorno integrado de desarrollo (IDE) basado en un navegador que integra un compilador y un entorno en tiempo de ejecución para Solidity sin los componentes orientados al servidor.
- **Ethereum Studio** Entorno integrado de desarrollo (IDE) especializado que proporciona acceso a un entorno completo de Ethereum a través de un intérprete de comandos (shell).
- **Plugin IntelliJ IDEA** Plugin de Solidity para IntelliJ IDEA (y el resto de IDEs de JetBrains).
- **Extensión de Visual Studio** Plugin de Solidity para Microsoft Visual Studio que incluye un compilador de Solidity.
- **Paquete para SublimeText** Paquete para resaltar la sintáxis de Solidity en el editor SublimeText.
- **Etheratom** Plugin para el editor Atom que ofrece: resaltar la sintáxis, un entorno de compilación y un entorno en tiempo de ejecución (compatible con un nodo en segundo plano y con una máquina virtual).
- **Linter de Solidity para Atom** Plugin para el editor Atom que ofrece linting para Solidity.
- **Linter de Solium para Atom** Programa de linting de Solidity configurable para Atom que usa Solium como base.
- **Solium** Programa de linting de Solidity para la interfaz de línea de comandos que sigue estrictamente las reglas prescritas por la Guía de Estilo de Solidity.
- **Extensión para Visual Studio Code** Plugin de Solidity para Microsoft Visual Studio que incluye resaltar la sintáxis y el compilador de Solidity.
- **Emacs Solidity** Plugin para el editor Emacs que incluye resaltar la sintáxis y el reporte de los errores de compilación.
- **Vim Solidity** Plugin para el editor Vim que incluye resaltar la sintáxis. Plugin for the Vim editor providing syntax highlighting.
- **Vim Syntastic** Plugin para el editor Vim que incluye la verificación de la compilación.

Descontinuados:

- **Mix IDE** Entorno integrado de desarrollo (IDE) basado en Qt para el diseño, debugging y testeo de contratos inteligentes en Solidity.

CAPÍTULO 3

Herramientas para Solidity

- **Dapp** Herramienta de construcción, gestión de paquetes y asistente de despliegue para Solidity.
- **Solidity REPL** Prueba Solidity al instante gracias a una consola de línea de comandos de Solidity.
- **solgraph** Visualiza el flujo de control de Solidity y resalta potenciales vulnerabilidades de seguridad.
- **evmdis** Desensamblador de la Máquina Virtual de Ethereum (EVM) que realiza análisis estáticos sobre el bytecode y así proporcionar un mayor nivel de abstracción que las operaciones brutas del EVM.
- **Doxity** Generador de documentación para Solidity.

CAPÍTULO 4

Analizadores de sintáxis y de gramática alternativos para Solidity

- **solidity-parser** Analizador de sintáxis para JavaScript.
- **Solidity Grammar para ANTLR 4** Analizador de gramática de Solidity para el generador de sintáxis ANTLR 4.

CAPÍTULO 5

Documentación del lenguaje

A continuación, primero veremos un *contrato inteligente sencilo* escrito en Solidity, seguido de una introducción sobre *blockchains* y sobre la *Máquina Virtual de Ethereum*.

En la siguiente sección se explicarán distintas *características* de Solidity con varios *ejemplos de contratos*. Recuerde que siempre puede probar estos contratos en su navegador!.

La última sección (y también la más amplia) cubre, en profundidad, todos los aspectos de Solidity.

Si todavía tiene dudas o preguntas, puede buscar y preguntar en la web de Ethereum Stackexchange, o puede unirse a nuestro canal de gitter. ¡Ideas para mejorar Solidity o esta documentación siempre son bienvenidas!

También existe la versión rusa ( ).

CAPÍTULO 6

Contenidos

Índice de palabras clave, Página de búsqueda

Introducción a los Contratos Inteligentes

Un contrato inteligente simple

Vamos a comenzar con el ejemplo más básico. No pasa nada si no entiendes nada ahora, entraremos más en detalle posteriormente.

Almacenamiento

```
pragma solidity ^0.4.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) {
        storedData = x;
    }

    function get() constant returns (uint) {
        return storedData;
    }
}
```

La primera línea simplemente dice que el código fuente se ha escrito en la versión 0.4.0 de Solidity o en otra superior totalmente compatible (cualquiera anterior a la 0.5.0). Esto es para garantizar que el contrato no se va a comportar de una forma diferente con una versión más nueva del compilador. La palabra reservada `pragma` es llamada de esa manera porque, en general, los "pragmas" son instrucciones para el compilador que indican como este debe operar con el código fuente (p.ej.: pragma once). .

Un contrato para Solidity es una colección de código (sus *funciones*) y datos (su *estado*) que residen en una dirección específica en la blockchain de Ethereum. La línea `uint storedData;` declara una variable de estado llamada `storedData` del tipo `uint` (unsigned integer de 256 bits). Esta se puede entender como una parte única en una base de datos que puede ser consultada o modificada llamando a funciones del código que gestiona dicha base de datos. En el caso de Ethereum, este es siempre el contrato propietario. Y en este caso, las funciones `set` y `get` se pueden usar para modificar o consultar el valor de la variable.

Para acceder a una variable de estado, no es necesario el uso del prefijo `this.` como es habitual en otros lenguajes.

Este contrato no hace mucho todavía (debido a la infraestructura construída por Ethereum), simplemente permite a cualquiera almacenar un número accesible para todos sin un (factible) modo de prevenir la posibilidad de publicar este número. Por supuesto, cualquiera podría simplemente hacer una llamada `set` de nuevo con un valor diferente y sobreescribir el número inicial, pero este número siempre permanecería almacenado en la historia de la blockchain. Más adelante, veremos como imponer restricciones de acceso de tal manera que sólo tú puedas cambiar el número.

Ejemplo de Submoneda

El siguiente contrato va a implementar la forma más sencilla de una criptomoneda. Se pueden generar monedas de la nada, pero sólo la persona que creó el contrato estará habilitada para hacerlo (es trivial implementar un esquema diferente de emisión). Es más, cualquiera puede enviar monedas a otros sin necesidad de registrarse con usuario y contraseña - sólo hace falta un par de claves de Ethereum.

```
pragma solidity ^0.4.0;

contract Coin {
    // La palabra clave "public" hace que dichas variables
    // puedan ser leídas desde fuera.
    address public minter;
    mapping (address => uint) public balances;

    // Los eventos permiten a los clientes ligeros reaccionar
    // de forma eficiente a los cambios.
    event Sent(address from, address to, uint amount);

    // Este es el constructor cuyo código
    // sólo se ejecutará cuando se cree el contrato.
    function Coin() {
        minter = msg.sender;
    }

    function mint(address receiver, uint amount) {
        if (msg.sender != minter) return;
        balances[receiver] += amount;
    }

    function send(address receiver, uint amount) {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        Sent(msg.sender, receiver, amount);
    }
}
```

Este contrato introduce algunos conceptos nuevos que vamos a detallar uno a uno.

La línea `address public minter;` declara una variable de estado de tipo address (dirección) que es públicamente accesible. El tipo `address` es un valor de 160 bits que no permite operaciones aritméticas. Es apropiado para

almacenar direcciones de contratos o pares de claves pertenecientes a personas externas. La palabra reservada `public` genera automáticamente una función que permite el acceso al valor actual de la variable de estado. Sin esta palabra reservada, otros contratos no tienen manera de acceder a la variable. Esta función sería algo como esto:

```
function minter() returns (address) { return minter; }
```

Por supuesto, añadir una función exactamente como esa no funcionará porque deberíamos tener una función y una variable de estado con el mismo nombre, pero afortunadamente, has cogido la idea - el compilador se lo imagina por tí.

La siguiente línea, `mapping (address => uint) public balances;` también crea una variable de estado pública, pero se trata de un tipo de datos más complejo. El tipo mapea direcciones a enteros sin signo. Los mapeos (Mappings) pueden ser vistos como tablas hash hash tables que son virtualmente inicializadas de tal forma que cada clave candidata existe y es mapeada a un valor cuya representación en bytes es todo ceros. Esta anología no va mucho más allá, ya que no es posible obtener una lista de todas las claves de un mapeo, ni tampoco una lista de todos los valores. Por eso hay que tener en cuenta (o mejor, conservar una lista o usar un tipo de datos más avanzado) lo que se añade al mapping o usarlo en un contexto donde no es necesario, como este caso. La función getter creada mediante la palabra reservada `public` es un poco más compleja en este caso. De forma aproximada, es algo parecido a lo siguiente:

```
function balances(address _account) returns (uint) {
    return balances[_account];
}
```

Como se puede ver, se puede usar esta función para, de forma sencilla, consultar el balance de una única cuenta.

La línea `event Sent(address from, address to, uint amount);` declara un evento que es disparado en la última línea de la ejecución de `send`. Las interfaces de ususario (como las de servidor, por supuesto) pueden escuchar esos eventos que están siendo disparados en la blockchain sin mucho coste. Tan pronto son disparados, el listener también recibirá los argumentos `from`, `to` y `amount`, que hacen más fácil trazar las transacciones. Con el fin de escuchar este evento, se podría usar

```
Coin.Sent().watch({}, '', function(error, result) {
    if (!error) {
        console.log("Coin transfer: " + result.args.amount +
            " coins were sent from " + result.args.from +
            " to " + result.args.to + ".");
        console.log("Balances now:\n" +
            "Sender: " + Coin.balances.call(result.args.from) +
            "Receiver: " + Coin.balances.call(result.args.to));
    }
})
```

Es interesante como la función generada automáticamente `balances` es llamada desde la interfaz de usuario.

La función especial `Coin` es el constructor que se ejecuta durante la creación de un contrato y no puede ser llamada con posterioridad. Almacena permanentemente la dirección de la persona que crea el contrato: `msg` (junto con `tx` y `block`) es una variable global mágica que contiene propiedades que permiten el acceso a la blockchain. `msg.sender` es siempre la dirección desde donde se origina la llamada a la función actual (externa).

Finalmente, las funciones que realmente habrá en el contrato y que podrán ser llamadas por usuarios y contratos como son `mint` y `send`. Si se llama a `mint` desde una cuenta distinta a la del creador del contrato, no ocurrirá nada. Por otro lado, `send` puede ser usado por todos (los que ya tienen algunas de estas monedas) para enviar monedas a cualquier otro. Hay que tener en cuenta que si se usa este contrato para enviar monedas a una dirección, no se verá reflejado cuando se busque la dirección en un explorador de la blockchain por el hecho de enviar monedas, y que los balances sólo serán guardados en el almacenamiento de este contrato de moneda. Con el uso de eventos es relativamente sencillo crear un "explorador de la blockchain" que monitorice las transacciones y los balances de la nueva moneda.

Fundamentos de Blockchain

Las blockchains son un concepto no muy difícil de entender para desarrolladores. La razón es que la mayoría de las complicaciones (minería, hashes, criptografa de curva elíptica, redes P2P, etc.) están justo ahí para proveer un conjunto de funcionalidades y espectativas. Una vez que aceptas estas funcionalidades tal cual vienen dadas, no tienes que preocuparte por la tecnología que lleva inmersa - o, ¿tienes que saber realmente cómo funciona internamente Amazon AWS para poder usarlo?.

Transacciones

Una blockchain es una base de datos transaccional globalmente compartida. Esto quiere decir que todos pueden leer las entradas en la base de datos simplemente participando en la red. Si quieres cambiar algo en la base de datos, tienes que crear una transacción a tal efecto que tiene que ser aceptada por todos los demás. La palabra transacción implica que el cambio que quieres hacer (asumiendo que quieres cambiar dos valores al mismo tiempo) o se aplica por completo, o no se realiza. Es más, mientras tu transacción es aplicada en la base de datos, ninguna otra transacción puede modificarla.

Como ejemplo, imagine una tabla que lista los balances de todas las cuentas en una divisa electrónica. Si se solicita una transferencia de una cuenta a otra, la naturaleza transaccional de la base de datos garantiza que la cantidad que es sustraída de una cuenta, es añadida en la otra. Si por la razón que sea, no es posible añadir la cantidad a la cuenta de destino, la cuenta origen tampoco se modifica.

Yendo más allá, una transacción es siempre firmada criptográficamente por el remitente (creador). Esto la hace más robusta para garantizar el acceso a modificaciones específicas de la base de datos. En el ejemplo de divisas electrónicas, un simple chequeo asegura que sólo la persona que posee las claves de la cuenta puede transferir dinero desde ella.

Bloques

Un obstáculo mayor que sobrepasar es el que, en términos de Bitcoin, se llama ataque de “doble gasto”: ¿qué ocurre si dos transacciones existentes en la red quieren borrar una cuenta?, ¿un conflicto?.

La respuesta abstracta a esto es que no tienes de qué preocuparte. El orden de las transacciones se seleccionará por ti, las transacciones se aglutinarán en lo que es llamado “bloque” y entonces serán ejecutadas y distribuídas entre todos los nodos participantes. Si dos transacciones se contradicen, la que concluye en segundo lugar será rechazada y no formará parte del bloque.

Estos bloques forman una secuencia lineal en el tiempo de la que viene la palabra cadena de bloques o “blockchain”. Los bloques son añadidos a la cadena en intervalos regulares - para Ethereum esto viene a significar cada 17 segundos.

Como parte del “mecanismo de selcción de orden” (que se conoce como minería), tiene que pasar que los bloques sean revertidos de cuando en cuando, pero sólo en el extremo o “tip” de la cadena. Cuantos más bloques se añaden encima, menos probable es. En ese caso, lo que ocurre es que tus transacciones son revertidas e incluso borradas de la blockchain, pero cuanto más esperes, menos probable será.

Máquina Virtual de Ethereum

Introducción

La máquina virtual de Ethereum (EVM por sus siglas en inglés) es un entorno de ejecución de contratos inteligentes en Ethereum. Va más allá de una configuración tipo sandbox ya que se encuentra totalmente aislada, lo que significa que el código que se ejecuta en la EVM no tiene acceso a la red, ni al sistema de ficheros, ni a ningún otro proceso. Incluso los contratos inteligentes tienen acceso limitado a otros contratos inteligentes.

Cuentas

Hay dos tipos de cuentas en Ethereum que comparten el mismo espacio de dirección: **Cuentas externas** que están controladas por un par de claves pública-privada (p-ej.: humanos) y **Cuentas contrato** que están controladas por el código almacenado conjuntamente con la cuenta.

La dirección de una cuenta externa viene determinada por la clave pública mientras que la dirección de la cuenta contrato se define en el momento en que se crea dicho contrato (se deriva de la dirección del creador y del número de transacciones enviadas desde esa dirección, el llamado "nonce").

Independientemente de que la cuenta almacene código, los dos tipos se tratan de forma equitativa por la EVM.

Cada cuenta tiene un almacenamiento persistente clave-valor que mapea palabras de 256 bits a palabras de 256 bits llamado **almacenamiento**.

Además, cada cuenta tiene un **balance** en Ether (en "Wei" para ser exactos) que puede ser modificado enviando transacciones que incluyen Ether.

Transacciones

Una transacción es un mensaje que se envía de una cuenta a otra (que debería ser la misma o la especial cuenta-cero, ver más adelante). Puede incluir datos binarios (payload) y Ether.

Si la cuenta destino contiene código, este es ejecutado y el payload se provee como dato de entrada.

Si la cuenta destino es la cuenta-cero (la cuenta con dirección `0`), la transacción crea un **nuevo contrato**. Como se ha mencionado, la dirección del contrato no es la dirección cero, sino de una dirección derivada del emisor y su número de transacciones enviadas (el "nonce"). Los datos binarios de la transacción que crea el contrato son obtenidos como bytecode por la EVM y ejecutados. La salida de esta ejecución es permanentemente almacenada como el código del contrato. Esto significa que para crear un contrato, no se envía el código actual del contrato, realmente se envía código que nos devuelve ese código final.

Gas

En cuanto se crean, cada transacción se carga con una determinada cantidad de **gas**, cuyo propósito es limitar la cantidad de trabajo que se necesita para ejecutar la transacción y pagar por esta ejecución. Mientras la EVM ejecuta la transacción, el gas se gasta gradualmente según unas reglas específicas.

El **precio del gas** (gas price) es un valor establecido por el creador de la transacción, quien tiene que pagar el `gas_price * gas` desde la cuenta de envío. Si queda algo de gas después de la ejecución, se le reembolsa.

Si se ha gastado todo el gas en un punto (p.ej.: es negativo), se lanza una excepción de out-of-gas, que revierte todas las modificaciones hechas al estado en el contexto de la ejecución actual.

Almacenamiento, Memoria y la Pila

Cada cuenta tiene un área de memoria persistente que se llama **almacenamiento**. El almacenamiento es un almacén clave-valor que mapea palabras de 256 bits con palabras de 256 bits. No es posible enumerar el almacenamiento interno desde un contrato y es comparativamente costoso leer y, más todavía, modificar el almacenamiento. Un contrato no pueder leer ni escribir en otro almacenamiento que no sea el suyo.

La segunda área de memoria se conoce como **memoria**, de la que un contrato obtiene de forma ágil una instancia clara de cada message call (llamada de mensaje). La memoria es lineal y puede ser tratada a nivel de byte, pero las lecturas están limitadas a un ancho de 256 bits, mientras que las escrituras puden ser tanto de 8 bits como de 256 bits de ancho. La memoria se expande por palabras (256 bits), cuando se accede (tanto para leer o escribir) a una palabra de memoria

sin modificar previamente (p.ej.: cualquier offset de una palabra). En el momento de expansión, se debe pagar el coste en gas. La memoria es más costosa cuanto más crece (escala cuadráticamente).

La EVM no es una máquina de registro, es una máquina de pila por lo que todas las operaciones se hacen en un área llamada la **pila**. Tiene un espacio máximo de 1024 elementos y contiene palabras de 256 bits. El acceso a la pila está limitado a su cima de la siguiente manera: Es posible copiar uno de los 16 elementos superiores a la cima de la pila o intercambiar el elemento superior justo después de uno de los 16 elementos superiores. El resto de operaciones cogen los dos elementos más superiores (o uno, o más, dependiendo de la operación) de la pila y ponen el resultado en ella. Por supuesto, es posible mover elementos de la pila al almacenamiento o a la memoria, pero no es posible acceder simplemente a elementos arbitrarios más profundos dentro de la pila sin, primeramente, borrar los que ya están encima.

Conjunto de instrucciones

El conjunto de instrucciones de la EVM se mantiene mínimo con el objetivo de evitar implementaciones incorrectas que podrían causar problemas de consenso. Todas las instrucciones operan con el tipo de datos básico, palabras de 256 bits. Las operaciones de aritmética habitual, bit, lógica y de comparación están presentes. Se permiten tanto los saltos condicionales como los no condicionales. Es más, los contratos pueden acceder a propiedades relevantes del bloque actual como su número y timestamp.

Message Calls

Los contratos pueden llamar a otros contratos o enviar Ether a cuentas que no sean de contratos usando message calls. Los Message calls son similares a las transacciones, en el sentido de que tienen un origen, un destino, datos, Ether, gas y datos de retorno. De hecho, cada transacción consiste en un message call de alto nivel que de forma consecutiva puede crear message calls posteriores.

Un contrato puede decidir cuánto de su **gas** restante podría ser enviado con el message call interno y cuánto quiere retener. Si ocurre una excepción de out-of-gas durante la llamada interna (o cualquier otra excepción), se mostrará como un valor de error introducido dentro de la pila. En este caso, sólo se gasta el gas enviado junto con la llamada. En Solidity, el contrato que hace la llamada causa una excepción manual por defecto en estas situaciones, por lo que esas excepciones ascienden en la pila de llamada.

Como se ha mencionado, el contrato llamado (que podría ser el mismo que el que hace la llamada) recibirá una instancia de memoria vacía y tendrá acceso a los datos de la llamada - que serán provistos en un área separada que se llama **calldata**. Después de finalizar su ejecución, puede devolver datos que serán almacenados en una localización en la memoria del que hace la llamada que éste ha reservado previamente.

Las llamadas están **limitadas** a la profundidad de 1024, lo que quiere decir que para operaciones más complejas, se debería preferir bucles sobre llamadas recursivas.

Delegatecall / Callcode y librerías

Existe una variante especial de message call llamada **delegatecall** que es idéntica a un message call con la excepción de que el código en la dirección destino se ejecuta en el contexto del que hace la llamada y `msg.sender` y `msg.value` no cambian sus valores.

Esto significa que un contrato puede cargar código dinámicamente desde una dirección diferente en tiempo de ejecución. El almacenamiento, la dirección actual y el balance siguen referenciando al contrato que realiza la llamada, sólo se coge el código desde la dirección llamada.

Esto hace posible implementar la funcionalidad de "librería" en Solidity: Código de librería reusable que se puede aplicar a un almacenamiento de contrato, por ejemplo, con el fin de implementar una estructura de datos compleja.

Logs

Es posible almacenar datos en una estructura de datos indexada que mapea todo el recorrido hasta el nivel de bloque. Esta funcionalidad llamada **logs** se usa en Solidity para implementar **eventos**. Los contratos no pueden acceder a los datos del log después de crearse, pero pueden ser accedidos desde fuera de la blockchain de forma eficiente. Como parte de los datos del log se guardan en bloom filters, es posible buscar estos datos eficientemente y criptográficamente de manera segura, por lo que los otros miembros de la red que no se han descargado la blockchain entera ("light clients") todavía pueden buscarlos.

Creación

Los contratos pueden incluso crear otros contratos usando un opcode especial (p.ej.: ellos no llaman simplemente a la dirección cero). La única diferencia entre estos **create calls** y los message calls normales es que los datos son ejecutados y el resultado almacenado como código y el llamador / creador recibe la dirección del nuevo contrato en la pila.

Auto-destrucción

La única posibilidad de borrar el código de la blockchain es cuando un contrato en esa dirección realiza una operación de `selfdestruct`. Los Ether restantes almacenados en esa dirección son enviados al destinatario designado y, entonces, se borran el almacenamiento y el código del estado.

Advertencia: Aunque un contrato no contenga una llamada a `selfdestruct`, todavía podría hacer esa operación mediante `delegatecall` o `callcode`.

Nota: La eliminación de contratos antiguos puede, o no, ser implementada en clientes de Ethereum. Adicionalmente, los nodos de archivo podrían elegir mantener el almacenamiento del contrato y el código de forma indefinida.

Nota: Actualmente las **cuentas externas** no se pueden borrar del estado.

Instalando Solidity

Control de Versiones

Las versiones de Solidity siguen un versionado semántico, y además de los releases, también hay **nightly developement builds**. El funcionamiento de los nightly builds no está garantizado y puede que contengan cambios no documentados o que no funcionen. Recomendamos usar la última release. Los siguientes instaladores de paquetes usarán la release más actual.

Remix

Si sólo quieres probar Solidity para pequeños contratos, puedes usar Remix que no necesita instalación. Si quieres usarlo sin conexión a internet, puedes ir a https://github.com/ethereum/browser-solidity/tree/gh-pages y bajar el .ZIP como se explica en esa página.

npm / Node.js

Esta es probablemente la manera más portable y conveniente de instalar Solidity localmente.

Se proporciona una librería Javascript independiente de plataforma mediante la compilación del código C++ a Javascript usando Emscripten. Puede ser usado en proyectos directamente (como Remix). Visita el repositorio solc-js para ver las instrucciones.

También contiene una herramienta de línea de comandos llamada *solcjs* que puede ser instalada vía npm:

```
npm install -g solc
```

Nota: Las opciones de línea de comandos de *solcjs* no son compatibles con *solc*, y las herramientas (tales como *geth*) que esperen el comportamiento de *solc* no funcionarán con *solcjs*.

Docker

Proveemos builds de docker actualizadas para el compilador. El repositorio `stable` contiene las versiones publicadas mientras que el `nightly` contiene cambios potencialmente inestables de la rama de desarrollo.

```
docker run ethereum/solc:stable solc --version
```

Actualmente, la imagen de docker contiene el compilador ejecutable, así que tendrás que enlazar las carpetas de código y de output.

Paquetes Binarios

Los paquetes binarios de Solidity están disponibles en solidity/releases.

También tenemos PPAs para Ubuntu. Para la versión estable más reciente.

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install solc
```

Si quieres la versión en desarrollo más reciente:

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo add-apt-repository ppa:ethereum/ethereum-dev
sudo apt-get update
sudo apt-get install solc
```

Arch Linux también tiene paquetes, pero limitados a la versión de desarrollo más reciente:

```
pacman -S solidity-git
```

Homebrew aún no tiene paquetes preconstruidos, siguiendo una migración de Jenkins a TavisCI, pero Homebrew todavía debería funcionar para construir desde el código. Pronto se agregarán los paquetes preconstruidos.

```
brew update
brew upgrade
brew tap ethereum/ethereum
brew install solidity
brew linkapps solidity
```

Si necesitas una versión específica de Solidity, puedes instalar una fórmula Homebrew desde Github.

Ver solidity.rb commits on Github.

Sigue los enlaces de historia hasta que veas un enlace a un fichero de un commit específico de `solidity.rb`.

instalar con `brew`:

```
brew unlink solidity
# Install 0.4.8
brew install https://raw.githubusercontent.com/ethereum/homebrew-ethereum/
↪77cce03da9f289e5a3ffe579840d3c5dc0a62717/solidity.rb
```

Gentoo también provee un paquete Solidity que puede instalarse con `emerge`:

```
demerge ev-lang/solidity
```

Construir desde el código

Clonar el Repositorio

Para clonar el código fuente, ejecuta el siguiente comando:

```
git clone --recursive https://github.com/ethereum/solidity.git
cd solidity
```

Si quieres ayudar a desarrollar Solidity, debes hacer un fork de Solidity y agregar tu fork personal como un remoto secundario:

```
cd solidity
git remote add personal git@github.com:[username]/solidity.git
```

Solidity tiene submódulos de git. Asegúrate que están cargados correctamente:

```
git submodule update --init --recursive
```

Prerrequisitos - macOS

Para macOS, asegúrate de tener la versión más reciente de Xcode instalada. Esto contiene el compilador Clang C++, las herramientas que se necesitan para construir aplicaciones C++ en OS X. Si estás instalando Xcode por primera vez, necesitarás aceptar las condiciones de uso antes de poder hacer builds de línea de comandos:

```
sudo xcodebuild -license accept
```

Nuestras builds OS X requieren instalar el gestor de paquetes Homebrew para instalar dependencias externas. Aquí puedes ver cómo desinstalar Homebrew, si alguna vez quieres empezar de nuevo.

Prerrequisitos - Windows

Necesitarás instalar las siguientes dependencias para los builds de Solidity en Windows:

Software	Notas
Git para Windows	Herramienta de línea de comandos para repositorios git.
CMake	Generador de build multi plataforma.
Visual Studio 2015	Compilador C++ y entorno de desarrollo.

Dependencias Externas

Ahora tenemos un script fácil de usar que instala todas las dependencias externas en macOS, Windows y varias distros Linux. Esto solía ser un proceso manual de varias etapas, pero ahora es una sola línea:

```
./scripts/install_deps.sh
```

o, en Windows:

```
scripts\install_deps.bat
```

Build en línea de comandos

Construir Solidity es bastante similar en Linux, macOS y otros sistemas Unix:

```
mkdir build
cd build
cmake .. && make
```

o aún más fácil:

```
#nota: esto instalará los binarios de solc y soltest en usr/local/bin
./scripts/build.sh
```

Incluso para Windows:

```
mkdir build
cd build
cmake -G "Visual Studio 14 2015 Win64" ..
```

Estas últimas instrucciones deberían resultar en la creación de **solidity.sln** en ese directorio de build. Hacer doble click en ese archivo debería abrir Visual Studio. Sugerimos construir la configuración **RelWithDebugInfo**, pero todas funcionan.

O si no, puedes construir para Windows en la línea de comandos de la siguiente manera:

```
cmake --build . --config RelWithDebInfo
```

La cadena de versión en detalle

La cadena de versión de Solidity está compuesta por 4 partes:

- el número de la versión
- etiqueta pre-release, en general en formato `develop.YYYY.MM.DD` o `nightly.YYYY.MM.DD`
- commit en formato `commit.GITHASH`
- la plataforma tiene número arbitrario de ítems, contiene detalles de la plataforma y el compilador

Si hay modificaciones locales, el commit tendrá el sufijo `.mod`.

Éstas partes se combinan como es requerido por Semver, donde la etiqueta pre-release de Solidity equivale al pre-release de Semver y el commit Solidity y plataforma combinadas hacen el metadata del build de Semver.

Un ejemplo de release: `0.4.8+commit.60cc1668.Emscripten.clang`.

Un ejemplo pre-release: `0.4.9-nightly.2017.1.17+commit.6ecb4aa3.Emscripten.clang`

Información importante sobre versiones

Tras hacer un release, se incrementa el nivel de versión patch, porque asumimos que sólo siguen cambios de nivel de patch. Cuando se integran los cambios, la versión se aumenta de acuerdo a la versión Semver y la urgencia de los cambios. Finalmente, un release siempre se hace con la versión de la build nightly actual, pero sin el especificador `prerelease`.

Ejemplo:

0. se hace el release 0.4.0
1. el nightly build tiene versión 0.4.1 a partir de ahora
2. se introducen cambios compatibles - no hay cambio en versión
3. se introduce cambios no compatibles - la versión se aumenta a 0.5.0
4. se hace el release 0.5.0

Este comportamiento funciona bien con la *versión pragma*.

Solidity mediante ejemplos

Votación

El siguiente contrato es bastante complejo, pero muestra muchas de las características de Solidity. Es un contrato de votación. Uno de los principales problemas de la votación electrónica es la asignación de los derechos de voto a las personas correctas para evitar la manipulación. No vamos a resolver todos los problemas, pero por lo menos vamos a ver cómo se puede hacer un sistema de votación delegado que permita contar votos de forma **automática y completamente transparente** al mismo tiempo.

La idea es crear un contrato por cada votación, junto con un breve nombre para cada opción. El creador del contrato, que ejercerá de presidente, será el encargado de dar derecho a voto a cada dirección una a una.

Las personas que controlen estas direcciones podrán votar o delegar su voto a una persona de confianza.

Cuando finalice el periodo de votación, `winningProposal()` devolverá la propuesta más votada.

```
pragma solidity ^0.4.11;

/// @title Votación con voto delegado
contract Ballot {
    // Declara un nuevo tipo de dato complejo, que será
    // usado para almacenar variables.
    // Representará a un único votante.
    struct Voter {
        uint weight; // el peso del voto se acumula mediante la delegación de votos
        bool voted;  // true si esa persona ya ha votado
        address delegate; // persona a la que se delega el voto
        uint vote;   // índice de la propuesta votada
    }

    // Representa una única propuesta.
    struct Proposal {
        bytes32 name;   // nombre corto (hasta 32 bytes)
        uint voteCount; // número de votos acumulados
    }

    address public chairperson;
```

```
    // Declara una variable de estado que
    // almacena una estructura de datos `Voter` para cada posible dirección.
    mapping(address => Voter) public voters;

    // Una matriz dinámica de estructuras de datos de tipo `Proposal`.
    Proposal[] public proposals;

    // Crea una nueva votación para elegir uno de los `proposalNames`.
    function Ballot(bytes32[] proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // Para cada nombre propuesto
        // crea un nuevo objeto de tipo Proposal y lo añade
        // al final del array.
        for (uint i = 0; i < proposalNames.length; i++) {
            // `Proposal({...})` crea una nuevo objeto de tipo Proposal
            // de forma temporal y se añade al final de `proposals`
            // mediante `proposals.push(...)`.
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    // Da a `voter` el derecho a votar en esta votación.
    // Sólo puede ser ejecutado por `chairperson`.
    function giveRightToVote(address voter) {
        // Si el argumento de `require` da como resultado `false`,
        // finaliza la ejecución y revierte todos los cambios
        // producidos en el estado y los balances de Ether.
        // A veces es buena idea usar esto por si las funciones
        // están siendo ejecutadas de forma incorrecta. Pero ten en cuenta
        // que de esta forma se consumirá todo el gas enviado
        // (está previsto que esto cambie en el futuro).
        require((msg.sender == chairperson) && !voters[voter].voted);
        voters[voter].weight = 1;
    }

    /// Delega tu voto a `to`.
    function delegate(address to) {
        // Asignación por referencia
        Voter sender = voters[msg.sender];
        require(!sender.voted);

        // No se permite la delegación a uno mismo.
        require(to != msg.sender);

        // Propaga la delegación en tanto que `to` también delegue.
        // Por norma general, los bucles son muy peligrosos
        // porque si tienen muchas iteracciones puede darse el caso
        // de que empleen más gas del disponible en un bloque.
        // En este caso, eso implica que la delegación no será ejecutada,
        // pero en otros casos puede suponer que un
        // contrato se quede completamente bloqueado.
        while (voters[to].delegate != address(0)) {
```

```
            to = voters[to].delegate;

            // Encontramos un bucle en la delegación. No está permitido.
            require(to != msg.sender);
        }

        // Dado que `sender` es una referencia, esto
        // modifica `voters[msg.sender].voted`
        sender.voted = true;
        sender.delegate = to;
        Voter delegate = voters[to];
        if (delegate.voted) {
            // Si la persona en la que se ha delegado el voto ya ha votado,
            // se añade directamente al número de votos.
            proposals[delegate.vote].voteCount += sender.weight;
        } else {
            // Si la persona en la que se ha delegado el voto
            // todavía no ha votado, se añade al peso de su voto.
            delegate.weight += sender.weight;
        }
    }

    /// Da tu voto (incluyendo los votos que te han delegado)
    /// a la propuesta `proposals[proposal].name`.
    function vote(uint proposal) {
        Voter sender = voters[msg.sender];
        require(!sender.voted);
        sender.voted = true;
        sender.vote = proposal;

        // Si `proposal` está fuera del rango de la matriz,
        // esto lanzará automáticamente una excepción y
        // se revocarán todos los cambios
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev Calcula la propuesta ganadora teniendo en cuenta
    // todos los votos realizados.
    function winningProposal() constant
            returns (uint winningProposal)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal = p;
            }
        }
    }

    // Llama a la función winningProposal() para obtener
    // el índice de la propuesta ganadora y así luego
    // devolver el nombre.
    function winnerName() constant
            returns (bytes32 winnerName)
    {
        winnerName = proposals[winningProposal()].name;
    }
```

```
}
```

Posibles mejoras

Actualmente, hacen falta muchas transacciones para dar derecho de voto a todos los participantes. ¿Se te ocurre una forma mejor?

Subasta a ciegas

En esta sección vamos a ver lo fácil que es crear un contrato para hacer una subasta a ciegas en Ethereum. Comenzaremos con una subasta en donde todo el mundo pueda ver las pujas que se van haciendo. Posteriormente, ampliaremos este contrato para convertirlo en una subasta a ciegas en donde no sea posible ver las pujas reales hasta que finalice el periodo de pujas.

Subasta abierta sencilla

La idea general del siguiente contrato de subasta es que cualquiera puede enviar sus pujas durante el periodo de pujas. Como parte de la puja se envía el dinero / ether para ligar a los pujadores con sus pujas. Si la puja más alta es superada, la anterior puja más alta recupera su dinero. Tras el periodo de pujas, el contrato tiene que ser llamado manualmente por parte de los beneficiarios para recuperar su dinero - los contratos no pueden activarse por sí mismos.

```
pragma solidity ^0.4.11;

contract SimpleAuction {
    // Parámetros de la subasta. Los tiempos son
    // o timestamps estilo unix (segundos desde 1970-01-01)
    // o periodos de tiempo en segundos.
    address public beneficiary;
    uint public auctionStart;
    uint public biddingTime;

    // Estado actual de la subasta.
    address public highestBidder;
    uint public highestBid;

    // Retiradas de dinero permitidas de las anteriores pujas
    mapping(address => uint) pendingReturns;

    // Fijado como true al final, no permite ningún cambio.
    bool ended;

    // Eventos que serán emitidos al realizar algún cambio
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // Lo siguiente es lo que se conoce como un comentario natspec,
    // se identifican por las tres barras inclinadas.
    // Se mostrarán cuando se pregunte al usuario
    // si quiere confirmar la transacción.

    /// Crea una subasta sencilla con un periodo de pujas
    /// de `_biddingTime` segundos. El beneficiario de
    /// las pujas es la dirección `_beneficiary`.
```

```
    function SimpleAuction(
        uint _biddingTime,
        address _beneficiary
    ) {
        beneficiary = _beneficiary;
        auctionStart = now;
        biddingTime = _biddingTime;
    }

    /// Puja en la subasta con el valor enviado
    /// en la misma transacción.
    /// El valor pujado sólo será devuelto
    /// si la puja no es ganadora.
    function bid() payable {
        // No hacen falta argumentos, toda
        // la información necesaria es parte de
        // la transacción. La palabra payable
        // es necesaria para que la función pueda recibir Ether.

        // Revierte la llamada si el periodo
        // de pujas ha finalizado.
        require(now <= (auctionStart + biddingTime));

        // Si la puja no es la más alta,
        // envía el dinero de vuelta.
        require(msg.value > highestBid);

        if (highestBidder != 0) {
            // Devolver el dinero usando
            // highestBidder.send(highestBid) es un riesgo
            // de seguridad, porque la llamada puede ser evitada
            // por el usuario elevando la pila de llamadas a 1023.
            // Siempre es más seguro dejar que los receptores
            // saquen su propio dinero.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
        HighestBidIncreased(msg.sender, msg.value);
    }

    /// Retira una puja que fue superada.
    function withdraw() returns (bool) {
        var amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // Es importante poner esto a cero porque el receptor
            // puede llamar de nuevo a esta función como parte
            // de la recepción antes de que `send` devuelva su valor.
            pendingReturns[msg.sender] = 0;

            if (!msg.sender.send(amount)) {
                // Aquí no es necesario lanzar una excepción.
                // Basta con reiniciar la cantidad que se debe devolver.
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
```

```
    }

    /// Finaliza la subasta y envía la puja más alta al beneficiario.
    function auctionEnd() {
        // Es una buena práctica estructurar las funciones que interactúan
        // con otros contratos (i.e. llaman a funciones o envían ether)
        // en tres fases:
        // 1. comprobación de las condiciones
        // 2. ejecución de las acciones (pudiendo cambiar las condiciones)
        // 3. interacción con otros contratos
        // Si estas fases se entremezclasen, otros contratos podrían
        // volver a llamar a este contrato y modificar el estado
        // o hacer que algunas partes (pago de ether) se ejecute multiples veces.
        // Si se llama a funciones internas que interactúan con otros contratos,
        // deben considerarse como interacciones con contratos externos.

        // 1. Condiciones
        require(now >= (auctionStart + biddingTime)); // la subasta aún no ha acabado
        require(!ended); // esta función ya ha sido llamada

        // 2. Ejecución
        ended = true;
        AuctionEnded(highestBidder, highestBid);

        // 3. Interacción
        beneficiary.transfer(highestBid);
    }
}
```

Subasta a ciegas

A continuación, se va a extender la subasta abierta anterior a una subasta a ciegas. La ventaja de una subasta a ciegas es que conforme se acaba el plazo de pujas, no aumenta la presión. Crear una subasta a ciegas en una plataforma de computación transparente puede parecer contradictorio, pero la criptografía lo hace posible.

Durante el **periodo de puja**, un pujador no envía su puja como tal, sino una versión hasheada de la misma. Puesto que en la actualidad se considera que es prácticamente imposible encontrar dos valores (suficientemente largos) cuyos hashes son iguales, el pujador realiza la puja de esa forma. Tras el periodo de puja, los pujadores tienen que revelar sus pujas. Para ello, envían los valores descrifrados y el contrato comprueba que el valor del hash se corresponde con el proporcionado durante el periodo de puja.

Otra complicación es cómo hacer la subasta **vinculante y ciega** al mismo tiempo: la única manera de evitar que el pujador no envíe el dinero tras ganar la subasta, es haciendo que lo envíe junto con la puja. Puesto que las transferencias de valor no pueden ser ocultadas en Ethereum, cualquiera podrá ver la cantidad.

El siguiente contrato soluciona este problema al aceptar cualquier valor que sea al menos tan alto como lo pujado. Puesto que esto sólo se podrá comprobar durante la fase de revelación, algunas pujas podrán ser **inválidas**. Esto es así a propósito (incluso sirve para prevenir errores en caso de envíar pujas con valores muy altos). Los pujadores pueden confundir a su competencia realizando multiples pujas inválidas con valores altos o bajos.

```
pragma solidity ^0.4.11;

contract BlindAuction {
    struct Bid {
        bytes32 blindedBid;
        uint deposit;
    }
```

```
    address public beneficiary;
    uint public auctionStart;
    uint public biddingEnd;
    uint public revealEnd;
    bool public ended;

    mapping(address => Bid[]) public bids;

    address public highestBidder;
    uint public highestBid;

    // Retiradas permitidas de pujas previas
    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);

    /// Los modificadores son una forma cómoda de validar los
    /// inputs de las funciones. Abajo se puede ver cómo
    /// `onlyBefore` se aplica a `bid`.
    /// El nuevo cuerpo de la función pasa a ser el del modificador,
    /// sustituyendo `_` por el anterior cuerpo de la función.
    modifier onlyBefore(uint _time) { require(now < _time); _; }
    modifier onlyAfter(uint _time) { require(now > _time); _; }

    function BlindAuction(
        uint _biddingTime,
        uint _revealTime,
        address _beneficiary
    ) {
        beneficiary = _beneficiary;
        auctionStart = now;
        biddingEnd = now + _biddingTime;
        revealEnd = biddingEnd + _revealTime;
    }

    /// Efectúa la puja de manera oculta con `_blindedBid`=
    /// keccak256(value, fake, secret).
    /// El ether enviado sólo se recuperará si la puja se revela de
    /// forma correcta durante la fase de revelacin. La puja es
    /// válida si el ether junto al que se envía es al menos "value"
    /// y "fake" no es cierto. Poner "fake" como verdadero y no enviar
    /// la cantidad exacta, son formas de ocultar la verdadera puja
    /// y aún así realizar el depósito necesario. La misma dirección
    /// puede realizar múltiples pujas.
    function bid(bytes32 _blindedBid)
        payable
        onlyBefore(biddingEnd)
    {
        bids[msg.sender].push(Bid({
            blindedBid: _blindedBid,
            deposit: msg.value
        }));
    }

    /// Revela tus pujas ocultas. Recuperarás los fondos de todas
    /// las pujas inválidas ocultadas de forma correcta y de
    /// todas las pujas salvo en aquellos casos en que sea la más alta.
```

```
    function reveal(
        uint[] _values,
        bool[] _fake,
        bytes32[] _secret
    )
        onlyAfter(biddingEnd)
        onlyBefore(revealEnd)
    {
        uint length = bids[msg.sender].length;
        require(_values.length == length);
        require(_fake.length == length);
        require(_secret.length == length);

        uint refund;
        for (uint i = 0; i < length; i++) {
            var bid = bids[msg.sender][i];
            var (value, fake, secret) =
                    (_values[i], _fake[i], _secret[i]);
            if (bid.blindedBid != keccak256(value, fake, secret)) {
                // La puja no ha sido correctamente revelada.
                // No se recuperan los fondos depositados.
                continue;
            }
            refund += bid.deposit;
            if (!fake && bid.deposit >= value) {
                if (placeBid(msg.sender, value))
                    refund -= value;
            }
            // Hace que el emisor no pueda reclamar dos veces
            // el mismo depósito.
            bid.blindedBid = 0;
        }
        msg.sender.transfer(refund);
    }

    // Esta función es "internal", lo que significa que sólo
    // se podrá llamar desde el propio contrato (o contratos
    // que deriven de él).
    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != 0) {
            // Devolverle el dinero de la puja
            // al anterior pujador con la puja más alta.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBid = value;
        highestBidder = bidder;
        return true;
    }

    /// Retira una puja que ha sido superada.
    function withdraw() returns (bool) {
        var amount = pendingReturns[msg.sender];
        if (amount > 0) {
```

```
            // Es importante poner esto a cero porque el receptor
            // puede llamar a esta función de nuevo como parte
            // de la recepción antes de que `send` devuelva su valor.
            // (ver la observacin de arriba sobre condiciones -> efectos
            // -> interacción).
            pendingReturns[msg.sender] = 0;

            if (!msg.sender.send(amount)){
                // Aquí no es necesario lanzar una excepción.
                // Basta con reiniciar la cantidad que se debe devolver.
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// Finaliza la subasta y envía la puja más alta
    /// al beneficiario.
    function auctionEnd()
        onlyAfter(revealEnd)
    {
        require(!ended);
        AuctionEnded(highestBidder, highestBid);
        ended = true;
        // Enviamos todo el dinero que tenemos, porque
        // parte de las devoluciones pueden haber fallado.
        beneficiary.transfer(this.balance);
    }
}
```

Compra a distancia segura

```
pragma solidity ^0.4.11;

contract Purchase {
    uint public value;
    address public seller;
    address public buyer;
    enum State { Created, Locked, Inactive }
    State public state;

    function Purchase() payable {
        seller = msg.sender;
        value = msg.value / 2;
        require((2 * value) == msg.value);
    }

    modifier condition(bool _condition) {
        require(_condition);
        _;
    }

    modifier onlyBuyer() {
        require(msg.sender == buyer);
        _;
```

```
    }

    modifier onlySeller() {
        require(msg.sender == seller);
        _;
    }

    modifier inState(State _state) {
        require(state == _state);
        _;
    }

    event Aborted();
    event PurchaseConfirmed();
    event ItemReceived();

    /// Aborta la compra y reclama el ether.
    /// Sólo puede ser llamado por el vendedor
    /// antes de que el contrato se cierre.
    function abort()
        onlySeller
        inState(State.Created)
    {
        Aborted();
        state = State.Inactive;
        seller.transfer(this.balance);
    }

    /// Confirma la compra por parte del comprador.
    /// La transacción debe incluir la cantidad de ether
    /// multiplicada por 2. El ether quedará bloqueado
    /// hasta que se llame a confirmReceived.
    function confirmPurchase()
        inState(State.Created)
        condition(msg.value == (2 * value))
        payable
    {
        PurchaseConfirmed();
        buyer = msg.sender;
        state = State.Locked;
    }

    /// Confirma que tú (el comprador) has recibido el
    /// artículo. Esto desbloqueará el ether.
    function confirmReceived()
        onlyBuyer
        inState(State.Locked)
    {
        ItemReceived();
        // Es importante que primero se cambie el estado
        // para evitar que los contratos a los que se llama
        // abajo mediante `send` puedan volver a ejecutar esto.
        state = State.Inactive;

        // NOTA: Esto permite bloquear los fondos tanto al comprador
        // como al vendedor - debe usarse el patrón withdraw.

        buyer.transfer(value);
```

```
        seller.transfer(this.balance);
    }
}
```

Canal de micropagos

Por escribir.

Solidity a fondo

Esta sección debería proveerte de todo lo que necesitas saber sobre Solidity. Si falta algo, por favor, ponte en contacto con nosotros en Gitter o haz un pull request en Github.

Composición de un fichero fuente de Solidity

Los ficheros fuente pueden contener un número arbitrario de definiciones de contrato, incluir directivas y directivas de pragma.

Versión de Pragma

Es altamente recomendable anotar los ficheros fuente con la versión de pragma para impedir que se compilen con una version posterior del compilador que podría introducir cambios incompatibles. Intentamos mantener este tipo de cambios al mínimo y que los cambios que introduzcamos y modifiquen la semántica requieran también un cambio en la sintáxis, pero esto no siempre es posible. Por ese motivo, siempre es buena idea repasar el registro de cambios por lo menos para las nuevas versiones que contienen cambios de ruptura. Estas nuevas versiones siempre tendrán un número de versión del tipo `0.x.0` o `x.0.0`.

La versión de pragma se usa de la siguiente manera:

    pragma solidity ^0.4.0;

Un fichero como este no se compilará con un compilador con una versión anterior a `0.4.0` y tampoco funcionará con un compilador que tiene una versión posterior a `0.5.0` (se especifíca esta segunda condición usando el `^`). La idea detrás de esto es que no va a haber un cambio de ruptura antes de la versión `0.5.0`, así que podemos estar seguro de que nuestro código se compilará de la manera en que nosotros esperamos. No fijamos la versión exacta del compilador, de manera que podemos liberar nuevas versiones que corrigen bugs.

Se puede especificar reglas mucho más complejas para la versión del compilador, la expresión sigue las utilizadas por npm.

Importar otros ficheros fuente

Sintáxis y semántica

Solidity soporta la importación de declaraciones, que son muy similares a las que se hacen en JavaScript (a partir de ES6), aunque Solidity no conoce el concepto de “default export”.

A nivel global, se puede usar la importación de declaraciones de la siguiente manera:

```
import "filename";
```

Esta declaración importa todos los símbolos globales de "filename" (junto con los símbolos importados desde allí) en el alcance global actual (diferente que en ES6 pero compatible con versiones anteriores de Solidity).

```
import * as symbolName from "filename";
```

...crea un nuevo símbolo global `symbolName` cuyos miembros son todos los símbolos globales de `"filename"`.

```
import {symbol1 as alias, symbol2} from "filename";
```

...crea un nuevo símbolo global `alias` y `symbol2` que referencian respectivamente `symbol1` y `symbol2` desde `"filename"`.

Esta otra sintaxis no forma parte de ES6 pero es probablemente conveniente:

```
import "filename" as symbolName;
```

lo que es equivalente a `import * as symbolName from "filename";`.

Ruta

En lo que hemos visto más arriba, `filename` siempre se trata como una ruta con el `/` como separador de directorio, `.` como el directorio actual y `..` como el directorio padre. Cuando `.` o `..` es seguido por un carácter excepto `/`, no se considera como el directorio actual o directorio padre. Se tratan todos los nombres de ruta como rutas absolutas, a no ser que empiecen por `.` o el directorio padre `...`

Para importar un fichero `x` desde el mismo directorio que el fichero actual, se usa *import "./x" as x;*'. Si en lugar de esa expresión se usa `import "x.as x;`, podría ser que se referencie un fichero distinto (en un "include directory" global).

Cómo se resuelve la ruta depende del compilador (ver más abajo). En general, la jerarquía de directorios no necesita mapear estrictamente su sistema local de ficheros, también puede mapear recursos que se descubren con por ejemplo ipfs, http or git.

Uso en compiladores actuales

Cuando se invoca el compilador, no sólo se puede especificar la manera en que se descubre el primer elemento de la ruta, también es posible especificar un prefijo de ruta de remapeo, de tal manera que por ejemplo `github.com/ethereum/dapp-bin/library` se remapee por `/usr/local/dapp-bin/library` y el compilador lea el fichero desde allí. Si bien se pueden hacer múltiples remapeos, se intentará primero con el remapeo con la clave más larga. Esto permite "fallback-remapping" con, por ejemplo, mapea a `/usr/local/include/solidity`. Además, estos remapeos pueden depender del contexto, lo que permite configurar paquetes para importar por ejemplo diferentes versiones de una librería con el mismo nombre.

solc:

Para solc (el compilador de línea de comando), los remapeos se proporcionan como argumentos `context:prefix=target`, donde tanto la parte `context:` como la parte `=target` son opcionales (en este caso target toma por defecto el valor de prefix). Todos los valores que remapean que son ficheros estándares son compilados (incluyendo sus dependencias). Este mecanismo es completamente compatible con versiones anteriores (siempre y cuando ningún nombre de fichero contenga = o :) y por lo tanto no es un cambio de ruptura. Todas las importaciones en los ficheros en el directorio `context` (o debajo de él) que importa un fichero que empieza con un `prefix` están redireccionados reemplazando `prefix` por `target`.

Como ejemplo, si clonas `github.com/ethereum/dapp-bin/` en tu local a `/usr/local/dapp-bin`, puedes usar lo siguiente en tu código fuente:

```
import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol" as it_mapping;
```

y luego se corre el compilador de esa manera:

```
solc github.com/ethereum/dapp-bin/=/usr/local/dapp-bin/ source.sol
```

Como ejemplo un poco más complejo, imaginemos que nos basamos en algún módulo que utiliza una versión muy antigua de dapp-bin. Esta versión anterior de dapp-bin se comprueba en `/usr/local/dapp-bin_old`, y luego se puede usar:

```
solc module1:github.com/ethereum/dapp-bin/=/usr/local/dapp-bin/ \
     module2:github.com/ethereum/dapp-bin/=/usr/local/dapp-bin_old/ \
     source.sol
```

así, todas las importaciones en `module2` apuntan a la versión anterior pero las importaciones en `module1` consiguen la nueva versión.

Nótese que solc sólo permite incluir ficheros desde algunos directorios. Tienen que estar en el diretorio (o subdirectorio) de uno de los ficheros fuente explícitamente especificado o en el directorio (o subdirectorio) de un target de remapeo. Si se desea permitir inclusiones directas absolutas, sólo hace falta añadir el `=/` de remapeo.

Si hay múltiples remapeos que conducen a un fichero válido, se elige el remapeo con el prefijo común más largo.

Remix:

Remix proporciona un remapeo automático para github y también recupera automáticamente el fichero desde la red: se puede importar el mapeo iterable con por ejemplo `import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol.ªs it_mapping;`.

A futuro se podrían añadir otros proveedores de código fuente.

Comentarios

Se aceptan los comentarios de línea simple (`//`) y los comentarios de múltiples líneas (`/*...*/`).

```
// Esto es un comentario de línea simple.

/*
Esto es un comentario
de múltiples líneas.
*/
```

Adicionalmente, existe otro tipo de comentario llamado natspec, pero la documentación todavía no está escrita. Estos comentarios se escriben con tres barras ***oblícuas (‘‘///‘‘) o como un bloque de comentarios con doble asterísco (‘‘/** ... */‘‘) y se deben usar justo arriba de las declaraciones de función o instrucciones. Para documentar funciones, anotar condiciones para la verificación formal y para proporcionar un **texto de confirmación** al usuario cuando intenta invocar una función, se puede usar etiquetas del tipo Doxygen dentro de estos comentarios.

En el siguiente ejemplo, documentamos el título del contrato, la explicación para los dos parametros de entrada y para los dos valores de retorno.

```
   pragma solidity ^0.4.0;

/** @title Shape calculator.*/
contract shapeCalculator{
    /**@dev Calculates a rectangle's surface and perimeter.
     * @param w Width of the rectangle.
     * @param h Height of the rectangle.
```

```
     * @return s The calculated surface.
     * @return p The calculated perimeter.
     */
    function rectangle(uint w, uint h) returns (uint s, uint p) {
        s = w * h;
        p = 2 * (w + h);
    }
}
```

Estructura de un contrato

Los Contratos en Solidity son similares a las clases de los lenguajes orientados a objetos. Cualquier contrato puede contener declaraciones del tipo *variables de estado*, *funciones*, *modificadores de función*, *eventos*, *structs* y *enums*. Además, los contratos pueden heredar de otros contratos.

Variables de estado

Las variables de estado son valores que están permanentemente almacenados en una parte del contrato conocida como storage del contrato.

```
pragma solidity ^0.4.0;

contract SimpleStorage {
    uint storedData; // Variable de estado
    // ...
}
```

Véase la sección *tipos* para conocer los diferentes tipos válidos de variables de estado y *visibilidad-y-getters* para conocer las distintas posibilidades de visibilidad que pueden tener las variables de estado.

Funciones

Las funciones son las unidades ejecutables del código dentro de un contrato.

```
pragma solidity ^0.4.0;

contract SimpleAuction {
    function bid() payable { // Función
        // ...
    }
}
```

Las *llamadas a una función* pueden ocurrir dentro o fuera de la misma. Una función puede tener varios niveles de *visibilidad* con respecto a otros contratos.

Modificadores de función

Los modificadores de función se usan para enmendar de un modo declarativo la semántica de las funciones (véase *modificadores* en la sección sobre contratos).

```
pragma solidity ^0.4.11;

contract Purchase {
    address public seller;

    modifier onlySeller() { // Modificador
        require(msg.sender == seller);
        _;
    }

    function abort() onlySeller { // Uso de modificador
        // ...
    }
}
```

Eventos

Los eventos son interfaces de conveniencia con los servicios de registro de la EVM (Máquina Virtual de Ethereum).

```
pragma solidity ^0.4.0;

contract SimpleAuction {
    event HighestBidIncreased(address bidder, uint amount); // Evento

    function bid() payable {
        // ...
        HighestBidIncreased(msg.sender, msg.value); // Lanzamiento del evento
    }
}
```

Véase *eventos* en la sección sobre contratos para tener más información sobre cómo se declaran los eventos y cómo se pueden usar dentro de una dapp.

Tipos de structs

Las estructuras de datos (Structs) son tipos definidos por el propio usuario y pueden agrupar mútiples variables (véase *structs* en la sección sobre tipos).

```
pragma solidity ^0.4.0;

contract Ballot {
    struct Voter { // Structs
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

Tipos de enum

Los enumerados (Enums) se usan para crear tipos con un conjunto de valores finitos y están definidos por el propio usuario (véase *enums* en la sección sobre tipos).

```
pragma solidity ^0.4.0;

contract Purchase {
    enum State { Created, Locked, Inactive } // Enum
}
```

s.. index:: type

Tipos

Solidity es un lenguaje de tipado estático, que significa que cada tipo de variable (estado y local) tiene que ser especificada (o al menos conocida - ver *Deducción de tipo* abajo) en tiempo de compilación. Solidity proporciona varios tipos elementales que pueden ser combinados para crear tipos más complejos.

Además de eso, los tipos pueden interactuar el uno con el otro en expresiones conteniendo operadores. Para una lista rápida de referencia de los operadores, ver *Orden de preferencia de operadores*.

Tipos de valor

Los siguientes tipos también son llamados tipos de valor porque las variables de este tipo serán siempre pasadas como valores, ej. siempre serán copiados cuando son usados como argumentos de funciones o en asignaciones.

Booleanos

`bool`: Los posibles valores son las constantes `true` y `false`.

Operadores:

- `!` (negación lógica)
- `&&` (conjunción lógica, "y")
- `||` (disyunción lógica, "or")
- == (igualdad)
- `!=` (inigualdad)

Los operadores `||` y `&&` aplican las reglas comunes de corto circuitos. Esto significa que en la expresión `f(x) || g(y)`, si `f(x)` evalúa a `true`, `g(y)` no será evaluado incluso si tuviera efectos secundarios.

Enteros

`int` / `uint`: Enteros con y sin signo de varios tamaños. Las palabras clave `uint8` a `uint256` en pasos de `8` (sin signo de 8 hasta 256 bits) y `int8` a `int256`. `uint` y `int` son alias para `uint256` y `int256`, respectivamente.

Operadores:

- Comparaciones: <=, <, ==, `!=`, >=, > (evalúa a `bool`)
- Operadores bit: `&`, `|`, ^ (OR exclusivo a nivel de bit), ~ (negación a nivel de bit)
- Operadores aritméticos: +, −, − unario, + unario, `*`, `/`, `%` (restante), `**` (exponenciales), << (desplazamiento a la izquierda), >> (desplazamiento a la derecha)

La división siempre trunca (está compilada a la opcode DIV de la EVM), pero no trunca si los dos operadores son *literales* (o expresiones literales).

División por cero y módulos con cero arrojan una excepción en tiempo de ejecución.

El resultado de una operación de desplazamiento es del tipo del operador izquierdo. La expresión `x << y` es equivalente a `x * 2**y` y `x >> y` es equivalente a `x / 2**y`. Esto significa que hacer un desplazamiento de números negativos extiende en signo. Hacer un desplazamiento por un número negativo arroja una excepción en tiempo de ejecución.

Advertencia: Los resultados producidos por desplazamientos a la derecha de valores negativos de tipos enteros con signo son diferentes de los producidos por otros lenguajes de programación. En Solidity, el desplazamiento a la derecha mapea la división para que los valores negativos del desplazamiento a la derecha sean redondeados hacia cero (truncado). En otros lenguajes de programación el desplazamiento a la derecha de valores negativos funciona como una división con redondeo hacia abajo (hacia infinito negativo).

Address

`address`: Contiene un valor de 20 bytes (tamaño de una dirección de Ethereum). Los tipos address también tienen miembros y sirven como base para todos los contratos.

Operadores:

- <=, <, ==, !=, >= y >

Miembros de Address

- `balance` y `transfer`

Para una referencia rápida, ver *En relación a las direcciones*.

Es posible consultar el monto de una dirección usando la propiedad `balance` y de enviar Ether (en unidades de wei) a una dirección usando la función `transfer`:

```
address x = 0x123;
address myAddress = this;
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

Nota: Si `x` es una dirección de contrato, su código (específicamente: su función de fallback, si es que está presente) será ejecutada con el llamado `transfer` (esta es una limitación de la EVM y no puede ser prevenida). Si esa ejecución agota el gas o falla de cualquier forma, el Ether transferido será revertido y el contrato actual se detendrá con una excepción.

- `send`

Send es la contrapartida de bajo nivel de `transfer`. Si la ejecución falla, el contrato actual no se detendrá con una excepción, sino que `send` devolverá `false`.

Advertencia: Hay algunos peligros en utilizar `send`: La transferencia falla si la profundidad de la llamada es de 1024 (esto puede ser forzado por el llamador) y también falla si al recipiente se le acaba el gas. Entonces para hacer

transferencias de Ether seguras, siempre revisar el valor devuelto por `send`, usar `transfer` o incluso mejor: usar un patrón donde el recipiente retira el dinero.

- `call`, `callcode` y `delegatecall`

Además, para interactuar con contratos que no se adhieren al ABI, la función `call` es prevista que tome un número arbitrario de argumentos de cualquier tipo. Estos argumentos son acolchados a 32 bytes y concatenados. Una excepción es el caso donde el primer argumento es codificado a exactamente 4 bytes. En este caso, no está acolchado para permitir el uso de firmas de función.

```
address nameReg = 0x72ba7d8e73fe8eb666ea66babc8116a41bfb10e2;
nameReg.call("register", "MyName");
nameReg.call(bytes4(keccak256("fun(uint256)")), a);
```

`call` devuelve un booleano indicando si la función llamada terminó (`true`) o causó una excepción de la EVM (`false`). No es posible acceder a los datos reales devueltos (para esto necesitaremos saber de antemano el tamaño de codificación).

`delegatecall` puede ser usado de forma similar: la diferencia es que sólo se usa el código de la dirección dada, todos los demás aspectos (almacenamiento, saldo, ...) salen directamente del contrato actual. El propósito de `delegatecall` es usar el código de librería que está almacenado en otro contrato. El usuario tiene que asegurarse de que el layout del almacenamiento en ambos contratos es correcto para usar `delegatecall`. Antes de homestead, sólo una versión limitada llamada `callcode` estaba disponible pero no daba acceso a los valores `msg.sender` y `msg.value` originales.

Las tres funciones `call`, `delegatecall` y `callcode` son funciones de muy bajo nivel y deben usarse sólo como medida de último recurso ya que rompen la seguridad de tipo de Solidity.

La opción `.gas()` está disponible en los 3 métodos, mientras que la opción `.value()` no se admite para `delegatecall`.

Nota: Todos los contratos heredan los miembros de address, así que es posible consultar el saldo del contrato actual usando `this.balance`.

Advertencia: Todas estas funciones son funciones de bajo nivel y deben usarse con cuidado. Específicamente, cualquier contrato desconocido puede ser malicioso y si se le llama, se le da el control a ese contrato, que luego puede llamar de vuelta a tu contrato, así que prepárate para cambios a tus variables de estado cuando la llamada retorna el valor.

Arrays de bytes de tamaño fijo

`bytes1`, `bytes2`, `bytes3`, ..., `bytes32`. `byte` es un alias para `bytes1`.

Operadores:

- Comparaciones: <=, <, ==, !=, >=, > (evalúa a `bool`)
- Operadores Bit: `&`, `|`, `^` (OR exclusivo a nivel de bits), ~ (negación a nivel de bits), << (desplazamiento a la izquierda), >> (desplazamiento a la derecha)
- Acceso por índice: Si `x` es de tipo `bytesI`, entonces `x[k]` para `0 <= k < I` devuelve el byte `k` (sólo lectura).

El operador de desplazamiento funciona con cualquier entero como operador derecho (pero devuelve el tipo del operador izquierdo, que denota el número de bits a desplazarse. Desplazarse por un número negativo arroja una excepción en tiempo de ejecución.

Miembros:

- `.length` devuelve el largo fijo del array byte (sólo lectura).

Arrays de bytes de tamaño dinámico

`bytes`: Array bytes de tamaño dinámico, ver *Arrays*. ¡No un tipo de valor!

`string`: Cadena de caracteres UTF-8-codificado de tamaño dinámico, ver *Arrays*. ¡No un tipo de valor!

Como regla general, usa `bytes` para data raw byte de tamaño arbitrario y `string` para una cadena de caracteres (UTF-8) de tamaño arbitrario. Si puedes limitar el tamaño a un cierto número de bytes, siempre usa una de `bytes1` a `bytes32` porque son muchas más baratas.

Números de punto fijo

PRÓXIMAMENTE...

Address literales

Literales hexadecimales que pasan el test checksum, por ejemplo `0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF` es de tipo `address`. Literales hexadecimales que están entre 39 y 41 dígitos de largo y no pasan el test de checksum producen una advertencia y son tratados como números racionales literales regulares.

Literales racionales y enteros

Literales enteros son formados por una secuencia de números en el rango 0-9. Son interpretados como decimales. Por ejemplo, `69` significa sesenta y nueve. Literales octales no existen en Solidity y los ceros a la izquierda son inválidos.

Literales de fracciones decimales son formados por un `.` con al menos un número en un lado. Ejemplos incluyen `1.`, `.1` y `1.3`.

La notación científica está también soportada, donde la base puede tener fracciones, mientras que el exponente no puede. Ejemplos incluyen `2e10`, `-2e10`, `2e-10`, `2.5e1`.

Expresiones de números literales retienen precisión arbitraria hasta que son convertidas a un tipo no literal (ej. usándolas juntas con una expresión no literal). Esto significa que las computaciones no se desbordan y las divisiones no se truncan en expresiones de números literales.

Por ejemplo, `(2**800 + 1) - 2**800` resulta en la constante `1` (de tipo `uint8`) aunque resultados intermedios ni siquiera serían del tamaño de la palabra. Además, `.5 * 8` resulta en el entero `4` (aunque no se hayan usado enteros entremedias).

Si el resultado no es un entero, un tipo apropiado `ufixed` o `fixed` es usado del cual el número de bits fraccionales es tan grande como se necesite (aproximando el número racional en el peor de los casos).

En `var x = 1/4;`, `x` recibirá el tipo `ufixed0x8` mientras que en `var x = 1/3` recibirá el tipo `ufixed0x256` porque `1/3` no es finitamente representable en binario y entonces será aproximado.

Cualquier operador que puede ser aplicado a enteros también puede ser aplicado a una expresión de número literal con tal que los operadores sean enteros. Si cualquiera de los dos es fraccional, las operaciones de bit no son permitidas y la exponenciación no es permitida si el exponente es fraccional (porque eso puede resultar en un número no racional).

Nota: Solidity tiene un tipo literal de número para cada número racional. Literales enteros y números racionales literales pertenecen a los tipos de números literales. Por otra parte, todas las expresiones literales (p.ej. las expresiones que contienen sólo números literales y operadores) pertenecen a tipos de números literales. Entonces las expresiones de números literales `1 + 2` y `2 + 1` ambas pertenecen al mismo tipo de número literal para el número racional tres.

Nota: La mayoría de fracciones decimales finitas como `5.3743` no son finitamente representables en binario. El tipo correcto para `5.3743` es `ufixed8x248` porque permite la mejor aproximación del número. Si quieres usar el número junto con tipos como `ufixed` (ej. `ufixed128x128`), tienes que especificar la precisión buscada de forma explícita: `x + ufixed(5.3743)`.

Advertencia: La división de enteros literales se solía truncar en versiones anteriores, pero ahora se convertirá en un número racional, ej. `5 / 2` no es igual a `1`, más bien a `2.5`.

Nota: Expresiones de números literales son convertidas en tipos no literales tan pronto como sean usadas con expresiones no literales. Aunque sabemos que el valor de la expresión asignada a `b` en el siguiente ejemplo evalúa a un entero, sigue usando tipos de punto fijo (y no números literales racionales) entremedio y entonces el código no compila.

```
uint128 a = 1;
uint128 b = 2.5 + a + 0.5;
```

String literales

Los strings literales se escriben con comillas simples o dobles (`"foo"` or `'bar'`). No hay ceros implícitos como en C; `"foo"` representa tres bytes, no cuatro. Como con literales enteros, su tipo puede variar, pero son implícitamente convertibles a `bytes1`, ..., `bytes32`, si caben a `bytes` y a `string`.

Los strings literales soportan caracteres de escape, tales como `\n`, `\xNN` y `\uNNNN`. `\xNN` toma un valor e inserta el byte apropiado, mientras que `\uNNNN` toma un codepoint Unicode e inserta una secuencia UTF-8.

Literales hexadecimales

Los literales hexadecimales son prefijos con la palabra clave `hex` y son cerrados por comillas simples o dobles (`hex"001122FF"`). Su contenido debe ser una cadena hexadecimal y su valor será la representación binaria de esos valores.

Los literales hexadecimales se comportan como los string literales y tienen las mismas restricciones de convertibilidad.

Enums

Los Enums son una manera para el usuario de crear sus propios tipos en Solidity. Son explícitamente convertibles a y desde todos los tipos de enteros, pero la conversión implícita no se permite. Las conversiones explícitas revisan los

valores de rangos en tiempo de ejecución y un fallo causa una excepción. Los Enums necesitan al menos un miembro.

```
pragma solidity ^0.4.0;

contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight() {
        choice = ActionChoices.GoStraight;
    }

    // Ya que los tipos enum no son parte del ABI, la firma de "getChoice"
    // será automáticamente cambiada a "getChoice() returns (unit8)"
    // para todo lo externo a Solidity. El tipo entero usado es apenas
    // suficientemente grande para guardar todos los valores enum, p.ej. si
    // tienes más valores, `unit16` será utilizado y así sucesivamente.
    function getChoice() returns (ActionChoices) {
        return choice;
    }

    function getDefaultChoice() returns (uint) {
        return uint(defaultChoice);
    }
}
```

Función

Los tipos función son tipos de función. Variables de tipo función pueden ser asignados desde funciones y parámetros de funciones de tipo función pueden ser usadas para pasar funciones y retornar funciones de llamados de funciones. Los tipos de función, los hay de dos tipos: *internas* y *externas*:

Las funciones internas sólo pueden ser usadas dentro del contrato actual (específicamente, dentro de la unidad de código actual, que también incluye funciones de librerías internas y funciones heredadas) porque no pueden ser ejecutadas fuera del contexto del contrato actual. La llamada a una función interna se realiza saltando a su label de entrada, tal como cuando se llama a una función interna del contrato actual.

Las funciones externas están compuestas de una dirección y una firma de función y pueden ser pasadas y devueltas desde una llamada de función externa.

Los tipos de funciones son notadas como sigue:

```
function (<parameter types>) {internal|external} [constant] [payable] [returns (
↪<return types>)]
```

A diferencia de los tipos de parámetros, los tipos de retorno no pueden estar vacíos - si el tipo función no debe retornar nada, la parte `returns (<return types>)` tiene que ser omitida.

Por defecto, las funciones son de tipo interna, así que la palabra clave `internal` puede ser omitida.

Hay dos formas de acceder una función en el contrato actual: o bien directamente con su nombre, `f`, o usando `this.f`. Usando el nombre resultará en una función interna, y con `this` habrá una función externa.

Si una variable de tipo función no es inicializada, llamarla resultará en una excepción. Lo mismo ocurre si llamas una función después de usar `delete` en ella.

Si funciones externas son usadas fuera del contexto de Solidity, son tratadas como tipo `function`, que codifica la dirección seguida por el identificador de la función junto con un tipo `bytes24`.

Nótese que las funciones públicas del contrato actual pueden ser usadas tanto como una función interna como externa. Para usar `f` como función interna, sólo se le llama como `f`, y si se quiere usar como externa, usar `this.f`.

Ejemplo que muestra como usar tipos de función internas:

```
pragma solidity ^0.4.5;

library ArrayUtils {
  // las funciones internas pueden ser usadas en funciones de librerías
  // internas porque serán parte del mismo contexto de código.
  function map(uint[] memory self, function (uint) returns (uint) f)
    internal
    returns (uint[] memory r)
  {
    r = new uint[](self.length);
    for (uint i = 0; i < self.length; i++) {
      r[i] = f(self[i]);
    }
  }
  function reduce(
    uint[] memory self,
    function (uint x, uint y) returns (uint) f
  )
    internal
    returns (uint r)
  {
    r = self[0];
    for (uint i = 1; i < self.length; i++) {
      r = f(r, self[i]);
    }
  }
  function range(uint length) internal returns (uint[] memory r) {
    r = new uint[](length);
    for (uint i = 0; i < r.length; i++) {
      r[i] = i;
    }
  }
}

contract Pyramid {
  using ArrayUtils for *;
  function pyramid(uint l) returns (uint) {
    return ArrayUtils.range(l).map(square).reduce(sum);
  }
  function square(uint x) internal returns (uint) {
    return x * x;
  }
  function sum(uint x, uint y) internal returns (uint) {
    return x + y;
  }
}
```

Otro ejemplo que usa tipos de función externa:

```
pragma solidity ^0.4.11;

contract Oracle {
  struct Request {
    bytes data;
```

```
    function(bytes memory) external callback;
  }
  Request[] requests;
  event NewRequest(uint);
  function query(bytes data, function(bytes memory) external callback) {
    requests.push(Request(data, callback));
    NewRequest(requests.length - 1);
  }
  function reply(uint requestID, bytes response) {
    // Aquí se revisa que la respuesta viene de una fuente de confianza
    requests[requestID].callback(response);
  }
}

contract OracleUser {
  Oracle constant oracle = Oracle(0x1234567); // contrato conocido
  function buySomething() {
    oracle.query("USD", this.oracleResponse);
  }
  function oracleResponse(bytes response) {
    require(msg.sender == address(oracle));
    // Usar los datos
  }
}
```

Nótese que las funciones lambda o inline están planeadas pero no están aún implementadas.

Tipos de referencia

Tipos complejos, ej. tipos que no siempre caben en 256 bits tienen que ser manejados con más cuidado que los tipos de valores que ya hemos visto. Ya que copiarlos puede ser muy caro, tenemos que pensar sobre si queremos que se almacenen en **memory** (que no es persistente) o en **storage** (donde las variables de estado se guardan).

Ubicación de datos

Cada tipo complejo, ej. *arrays* y *structs*, tienen anotaciones adicionales, la "data location", con respecto a si es almacenado en memoria o en almacenamiento. Dependiendo del contexto, siempre hay un valor por defecto, pero puede ser reemplazado añadiendo o bien `storage` o *memory'* al tipo. Por defecto para tipos parámetros de función (incluyendo parámetros de retorno) es `memory`, por defecto para variables locales es `storage` y la ubicación es forzada a `storage` para variables de estado (obviamente).

Hay una tercera ubicación de datos, "calldata", un área que no es modificable ni persistente donde argumentos de función son almacenados. Parámetros de función (no parámetros de retorno) de funciónes externas son forzados a "calldata" y se comportan casi como memoria.

Las ubicaciones de datos son importantes porque cambian cómo las asignaciones se comportan: Las asignaciones entre almacenamiento y memoria y también de variables de estado (incluso desde otras variable de estado) siempre crean una copia independiente. Asignaciones a almacenamiento variable de almacenamiento local sólo asignan una referencia, y esta referencia siempre apunta a la variable de estado aunque la referencia cambie entretanto. En cambio, asignaciones de la referencia almacenada en memoria a otro tipo de referencia no crea una copia.

```
pragma solidity ^0.4.0;

contract C {
    uint[] x; // la ubicación de los datos de x es storage
```

```
    // la ubicación de datos de memoryArray es memory
    function f(uint[] memoryArray) {
        x = memoryArray; // funciona, copia el array entero al almacenamiento
        var y = x; // funciona, asigna una referencia, ubicación de datos de y es␣
↪almacenamiento
        y[7]; // bien, devuelve el octavo elemento
        y.length = 2; // bien, modifica x a través de y
        delete x; // bien, limpia el array, también modifica y
        // Lo siguiente no funciona; debería crear un nuevo array temporal/sin nombre
        // en almacenamiento, pero el almacenamiento es asignado "estáticamente":
        // y = memoryArray;
        // Esto no funciona tampoco, ya que resetearía el apuntador, pero no hay
        // ubicación donde podría apuntar
        // delete y;
        g(x); // llama g, dando referencia a x
        h(x); // llama h y crea una copia independiente y temporal en la memoria
    }

    function g(uint[] storage storageArray) internal {}
    function h(uint[] memoryArray) {}
}
```

Resumen

Ubicación de datos forzada:

- parámetros (no de retorno) de funciones externas: calldata
- variables de estado: almacenamiento

Ubicación de datos por defecto:

- parámetros (también de retorno) de funciones: memoria
- todas otras variables: almacenamiento

Arrays

Los array pueden tener tamaño fijo en compilación o pueden ser dinámicos. Para arrays de almacenamiento, el tipo del elemento puede ser arbitrario (ej. también otros arrays, mapeos o structs). Para arrays de memoria, no puede ser un mapping y tiene que ser un tipo ABI si es que es un argumento de una función públicamente visible.

Un array de tamaño fijo `k` y elemento tipo `T` es escrito como `T[k]`, un array de tamaño dinámico como `T[]`. Como ejemplo, un array de 5 arrays dinámicos de `uint` es `uint[][5]` (nótese que la notación es invertida comparada a otros lenguajes). Para acceder al segundo uint en el tercer array dinámico, se utiliza `x[2][1]` (los índices comienzan en 0 y el acceso funciona de forma opuesta a la declaración. i.e. `x[2]` reduce un nivel en el tipo desde la derecha)

Variables de tipo `bytes` y `string` son arrays especiales. Un `bytes` es similar a `byte[]`, pero está junto en el calldata. `string` es igual a `bytes` pero no permite el acceso a la longitud o mediante índice (por ahora).

De modo que `bytes` siempre será preferible a `byte[]` ya que es más barato.

Nota: Si quieres acceder a la representación en bytes de un string `s`, usa `bytes(s).length` / `bytes(s)[7] = 'x';`. ¡Ten en cuenta que estás accediendo a los bytes a bajo nivel de la representación en UTF-8, y no a los caracteres

individualmente!

Es posible marcar arrays como `public` y dejar que Solidity cree un getter. El índice numérico se convertirá en un parámetro requerido por el getter.

Asignación de memoria en Arrays

Crear arrays con longitud variable en memoria se puede hacer usando la palabra clave `new`. Al contrario que con los arrays en storage, no es posible redimensionar los arrays en memoria mediante asignación al miembro `.length`.

```
pragma solidity ^0.4.0;

contract C {
    function f(uint len) {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        // Aquí tenemos a.length == 7 y b.length == len
        a[6] = 8;
    }
}
```

Array Literales / Arrays en linea

Los Array literales son arrays que se escriben como una expresión y no están asignados a una variable al momento.

```
pragma solidity ^0.4.0;

contract C {
    function f() {
        g([uint(1), 2, 3]);
    }
    function g(uint[3] _data) {
        // ...
    }
}
```

El tipo de array literal es un array de memoria de tamaño fijo del cual el tipo base es el tipo común de los elementos dados. El tipo de `[1, 2, 3]` es `uint[3] memory`, porque el tipo de cada una de estas constantes es `uint8`. Por eso, fue necesario convertir el primer elemento en el ejemplo arriba a `uint`. Nótese que actualmente, los arrays de memoria de tamaño fijo no pueden ser asignados a arrays de memoria de tamaño dinámico, ej. lo siguiente no es posible:

```
pragma solidity ^0.4.0;

contract C {
    function f() {
        // La próxima línea crea un tipo error porque uint[3] memory
        // no puede ser convertido a uint[] memory.
        uint[] x = [uint(1), 3, 4];
}
```

Esta restricción está planeada para ser eliminada en el futuro pero actualmente crea complicaciones por cómo los arrays son pasados en el ABI.

Miembros

length: Los arrays tienen un miembro `length` para guardar su número de elementos. Arrays dinámicos pueden ser modificados en almacenamiento (no en memoria) cambiando el miembro `.length`. Ésto no ocurre automáticamente cuando se intenta acceder a los elementos fuera de la longitud actual. El tamaño de arrays de memoria es fijo (pero dinámico, ej. puede depender de parámetros en tiempo de ejecución) cuando son creados.

push: Los arrays de almacenamiento dinámico y `bytes` (no `string`) tienen una función miembro llamada `push` que puede ser usada para agregar un elemento al final del array. La función devuelve el nuevo length.

Advertencia: Aún no es posible usar arrays en funciónes externas.

Advertencia: Debido a las limitaciones de la EVM, no es posible retornar contenido dinámico de las funciónes externas . La función `f` en `contract C { function f() returns (uint[]) { ... } }` devolverá algo si es llamado desde web3.js, pero no si se llama desde Solidity.

La única alternativa por ahora es usar grandes arrays de tamaño estático.

```
pragma solidity ^0.4.0;

contract ArrayContract {
    uint[2**20] m_aLotOfIntegers;
    // Nótese que el siguiente no es un par de arrays dinámicos, sino un
    // array dinámico de pares (ej. de arrays de tamaño fijo de length 2).
    bool[2][] m_pairsOfFlags;
    // newPairs es almacenado en memoria - por defecto para argumentos de función

    function setAllFlagPairs(bool[2][] newPairs) {
        // asignación a un array de almacenamiento reemplaza el array completo
        m_pairsOfFlags = newPairs;
    }

    function setFlagPair(uint index, bool flagA, bool flagB) {
        // acceso a un index que no existe arrojará una excepción
        m_pairsOfFlags[index][0] = flagA;
        m_pairsOfFlags[index][1] = flagB;
    }

    function changeFlagArraySize(uint newSize) {
        // si el tamaño nuevo es más pequeño, los elementos eliminados del array␣
↪serán limpiados
        m_pairsOfFlags.length = newSize;
    }

    function clear() {
        // éstos limpian los arrays completamente
        delete m_pairsOfFlags;
        delete m_aLotOfIntegers;
        // efecto idéntico aquí
        m_pairsOfFlags.length = 0;
    }

    bytes m_byteData;
```

```
    function byteArrays(bytes data) {
        // byte arrays ("bytes") son diferentes ya que no son almacenados sin padding,
        // pero pueden ser tratados idénticamente a "uint8[]"
        m_byteData = data;
        m_byteData.length += 7;
        m_byteData[3] = 8;
        delete m_byteData[2];
    }

    function addFlag(bool[2] flag) returns (uint) {
        return m_pairsOfFlags.push(flag);
    }

    function createMemoryArray(uint size) returns (bytes) {
        // Arrays de memoria dinámicos son creados usando `new`:
        uint[2][] memory arrayOfPairs = new uint[2][](size);
        // Crear un byte array dinámico:
        bytes memory b = new bytes(200);
        for (uint i = 0; i < b.length; i++)
            b[i] = byte(i);
        return b;
    }
}
```

Structs

Solidity provee una manera de definir nuevos tipos con structs, que es mostrado en el siguiente ejemplo:

```
pragma solidity ^0.4.11;

contract CrowdFunding {
    // Define un nuevo tipo con dos campos.
    struct Funder {
        address addr;
        uint amount;
    }

    struct Campaign {
        address beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping (uint => Funder) funders;
    }

    uint numCampaigns;
    mapping (uint => Campaign) campaigns;

    function newCampaign(address beneficiary, uint goal) returns (uint campaignID) {
        campaignID = numCampaigns++; // campaignID es variable de retorno
        // Crea un nuevo struct y lo guarda en almacenamiento. Dejamos fuera el tipo␣
↪mapping.
        campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);
    }

    function contribute(uint campaignID) payable {
```

```
        Campaign c = campaigns[campaignID];
        // Crea un nuevo struct de memoria temporal, inicializado con los valores␣
↪dados
        // y lo copia al almacenamiento.
        // Nótese que también se puede usar Funder(msg.sender, msg.value) para␣
↪inicializarlo
        c.funders[c.numFunders++] = Funder({addr: msg.sender, amount: msg.value});
        c.amount += msg.value;
    }

    function checkGoalReached(uint campaignID) returns (bool reached) {
        Campaign c = campaigns[campaignID];
        if (c.amount < c.fundingGoal)
            return false;
        uint amount = c.amount;
        c.amount = 0;
        c.beneficiary.transfer(amount);
        return true;
    }
}
```

El contrato no provee la funcionalidad total de un contrato crowdfunding, pero contiene los conceptos básicos necesarios para entender structs. Los tipos struct pueden ser usados dentro de mappings y arrays, y ellos mismos pueden contener mappings y arrays.

No es posible para un struct contener un miembro de su propio tipo, aunque el struct puede ser el tipo valor de un miembro mapping. Esta restricción es necesaria, ya que el tamaño del struct tiene que ser finito.

Nótese como en todas las funciónes, un tipo struct es asignado a la variable local (de la ubicación por defecto del almacenamiento). Esto no copia el struct pero guarda una referencia para que las asignaciones a miembros de la variable local realmente escriban al estado.

Por supuesto, puedes diréctamente acceder a los miembros del struct sin asignarlos a la variable local, como en `campaigns[campaignID].amount = 0`.

Mappings

Tipos mapping son declarados como `mapping(_KeyType => _ValueType)`. Aquí `_KeyType` puede ser casi cualquier tipo excepto mapping, un array de tamaño dinámico, un contrato, un enum y un struct. `_ValueType` puede ser cualquier tipo, incluyendo mappings.

Mappings pueden verse como 'tablas hash <https://en.wikipedia.org/wiki/Hash_table>'_ que son virtualmente inicializadas ya que cada posible clase existe y es mapeada a un valor que su representación byte es todo ceros: el valor *por defecto* de un tipo. Aunque la similitud termina aquí: los datos clave no son realmente almacenados en el mapping, sólo su hash `keccak256` usado para buscar el valor.

Por esto, los mappings no tienen un length o un concepto de "fijar" clave o valor.

Los mappings sólo son permitidos para variables de estado (o como tipos de referencia en funciones internas).

Es posible marcar los mappings `public` y hacer que Solidity cree un getter. El `_KeyType` será un parámetro requerido para el getter y devolverá `_ValueType`.

El `_ValueType` puede ser un mapping también. El getter tendrá un parámetro para cada `_KeyType`, recursivamente.

```
pragma solidity ^0.4.0;

contract MappingExample {
    mapping(address => uint) public balances;
```

```
    function update(uint newBalance) {
        balances[msg.sender] = newBalance;
    }
}

contract MappingUser {
    function f() returns (uint) {
        return MappingExample(<address>).balances(this);
    }
}
```

Nota: Los mappings no son iterables, pero es posible implementar una estructura de datos encima de ellos. Por ejemplo, ver iterable mapping.

Operadores con LValues

Si `a` es un LValue (ej. una variable o algo que puede ser asignado), los siguientes operadores son abreviaturas posibles:

`a += e` es equivalente a `a = a + e` . Los operadores `-=`, `*=`, `/=`, `%=`, `a |=`, `&=` y `^=` son todos definidos de esa manera. `a++` y `a--` son equivalentes a `a += 1` / `a -= 1` pero la expresión en sí todavía tiene el valor anterior de `a`. En contraste, `--a` y `++a` tienen el mismo efecto en `a` pero devuelven el valor después del cambio.

delete

`delete a` asigna el valor inicial para el tipo a `a`. Ej. para enteros, el equivalente es `a = 0`, pero puede ser usado en arrays, donde se asigna un array dinámico de length cero o un array estático del mismo length con todos los elementos reseteados. Para structs, se asigna a struct con todos los miembros reseteados.

`delete` no tiene efecto en mappings enteros (ya que las claves de los mappings pueden ser arbitrarias y generalmente desconocidas). Así que si se hace delete a un struct, reseteará todos los miembros que no son mappings y también recursivamente a los miembros al menos que sean mappings. Sin embargo, las claves individuales y lo que pueden mapear pueden ser eliminados.

Es importante notar que `delete a` en realidad se comporta como una asignación a `a`, ej. almacena un nuevo objeto en `a`.

```
pragma solidity ^0.4.0;

contract DeleteExample {
    uint data;
    uint[] dataArray;

    function f() {
        uint x = data;
        delete x; // setea x to 0, no afecta a los datos
        delete data; // setea data a 0, no afecta a x que aún tiene una copia
        uint[] y = dataArray;
        delete dataArray; // esto setea dataArray.length a cero, pero como uint[] es␣
↪un objecto complejo,
        // también y es afectado que es un alias al objeto de almacenamiento
        // Por otra parte: "delete y" no es válido, ya que asignaciones a variables␣
↪locales
        // haciendo referencia a objetos de almacenamiento sólo pueden ser hechas de
```

```
        // objetos de almacenamiento existentes.
    }
}
```

Conversión entre tipos elementales

Conversiones implícitas

Si un operador es aplicado a diferentes tipos, el compilador intenta implícitamente convertir uno de los operadores al tipo del otro (lo mismo es verdad para asignaciones). En general, una conversión implícita entre tipos de valores es posible si tiene sentido semanticamente y no hay información perdida: `uint8` es convertible a `uint16` y `int128` a `int256`, pero `int8` no es convertible a `uint256` (porque `uint256` no puede contener `-1`). Además, enteros sin signo pueden ser convertidos a bytes del mismo tamaño o más grande pero no vice-versa. Cualquier tipo que puede ser convertido a `uint160` puede también ser convertido a `address`.

Conversiones explícitas

Si el compilador no permite conversión implícita pero sabes lo que estás haciendo, una conversión explícita de tipo es a veces posible. Nótese que esto puede darte un comportamiento inesperado, ¡así que asegúrate de probar que el resultado es el que querías! Este ejemplo es para convertir de un negativo `int8` a `uint`:

```
int8 y = -3;
uint x = uint(y);
```

Al final de este snippet de código, `x` tendrá el valor `0xfffff..fd` (64 caracteres hex), que es -3 en la representación de 256 bits de los complementos de dos.

Si un tipo es explícitamente convertido a un tipo más pequeño, los bits de orden mayor son eliminados:

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b será 0x5678 ahora
```

Deducción de tipo

Por conveniencia, no es siempre necesario especificar explícitamente el tipo de una variable, el compilador infiere automáticamente el tipo del la primera expresión a la cual es asignada esa variable:

```
uint24 x = 0x123;
var y = x;
```

Aquí, el tipo de `y` será `uint24`. No es posible usar `var` para parámetros de función o parámetros de retorno.

Advertencia: El tipo es deducido sólo de la primera asignación, así que el bucle del siguiente snippet es infinito, ya que `i` tendrá el tipo `uint8` y cualquier valor de este tipo es más pequeño que `2000`. `for (var i = 0; i < 2000; i++) { ... }`

Unidades y variables disponibles globalmente

Unidades de Ether

Un numero literal puede tomar un sufijo como el `wei`, el `finney`, el `szabo` o el `ether` para convertirlo entre las subdenominaciones del Ether. Se asume que un numero sin sufijo para representar la moneda Ether está expresado en Wei, por ejemplo, `2 ether == 2000 finney` devuelve `true`.

Unidades de tiempo

Sufijos como `seconds`, `minutes`, `hours`, `days`, `weeks` y `years` utilizados después de numeros literales pueden usarse para convertir unidades de tiempo donde los segundos son la unidad de base. Las equivalencias son las siguientes:

- `1 == 1 seconds`
- `1 minutes == 60 seconds`
- `1 hours == 60 minutes`
- `1 days == 24 hours`
- `1 weeks == 7 days`
- `1 years == 365 days`

Ojo si utilizan estas unidades para realizar cálculos de calendario, porque no todos los años tienen 365 días y no todos los días tienen 24 horas, por culpa de los _segundos intercalares. Debido a que los segundos intercalares no son predecibles, la librería del calendario extacto tiene que estar actualizada por un oráculo externo.

Estos sufijos no pueden aplicarse a variables. Si desea interpretar algunas variables de entrada, como por ejemplo días, puede hacerlo de siguiente forma:

```
function f(uint start, uint daysAfter) {
    if (now >= start + daysAfter * 1 days) { ... }
}
```

Variables y funciones especiales

Existen variables y funciones especiales de ámbito global que siempre están disponibles y que se usan principalmente para proporcionar información sobre la blockchain.

Bloque y propiedades de las transacciones

- `block.blockhash(uint blockNumber) returns (bytes32)`: el hash de un bloque dado - sólo funciona para los 256 bloques más recientes, excluyendo el actual
- `block.coinbase` (`address`): devuelve la dirección del minero que está procesando el bloque actual
- `block.difficulty` (`uint`): devuelve la dificultad del bloque actual
- `block.gaslimit` (`uint`): devuelve el límite de gas del bloque actual
- `block.number` (`uint`): devuelve el número del bloque actual
- `block.timestamp` (`uint`): devuelve el timestamp del bloque actual en forma de segundos siguiendo el tiempo universal de Unix (Unix epoch)
- `msg.data` (`bytes`): datos enviados en la transacción (calldata)
- `msg.gas` (`uint`): devuelve el gas que queda

- `msg.sender` (`address`): devuelve remitente de la llamada actual
- `msg.sig` (`bytes4`): devuelve los primeros cuatro bytes de los datos enviados en la transacción (i.e. el identificador de la función)
- `msg.value` (`uint`): devuelve el numero de Wei enviado con la llamada
- `now` (`uint`): devuelve el timestamp del bloque actual (es un alias de `block.timestamp`)
- `tx.gasprice` (`uint`): devuelve el precio del gas de la transacción
- `tx.origin` (`address`): devuelve el emisor original de la transacción

Nota: Los valores de todos los elementos de `msg`, incluido `msg.sender` y `msg.value` pueden cambiar para cada llamada a una función **externa**. Incluyendo las llamadas a funciones de una librería.

Si desea implementar restricciones de acceso para funciones de una librería utilizando `msg.sender`, tiene que proporcionar manualmente el valor de `msg.sender` como argumento.

Nota: Los hashes de los bloques no están disponibles para todos los bloques por motivos de escalabilidad. Sólo se puede acceder a los hashes de los 256 bloques más recientes. El valor del hash para bloques más antiguos será cero.

Funciones matemáticas y criptográficas

`assert(bool condition)`: lanza excepción si la condición no está satisfecha. `addmod(uint x, uint y, uint k) returns (uint)`: computa `(x + y) % k` donde la suma se realiza con una precisión arbitraria y no se desborda en `2**256`. `mulmod(uint x, uint y, uint k) returns (uint)`: computa `(x * y) % k` donde la multiplicación se realiza con una precisión arbitraria y no se desborda en `2**256`. `keccak256(...) returns (bytes32)`: computa el hash de Ethereum-SHA-3 (Keccak-256) de la unión (compactada) de los argumentos. `sha3(...) returns (bytes32)`: equivalente a `keccak256()`. `sha256(...) returns (bytes32)`: computa el hash de SHA-256 de la unión (compactada) de los argumentos. `ripemd160(...) returns (bytes20)`: computa el hash de RIPEMD-160 de la unión (compactada) de los argumentos. `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`: recupera la dirección asociada a la clave pública de la firma de tipo curva elíptica o devuelve cero si hay un error (ejemplo de uso). `revert()`: aborta la ejecución y revierte los cambios de estado a como estaban.

Más arriba, "compactado" significa que los argumentos están concatenados sin relleno (padding). Esto significa que las siguientes llamadas son todas identicas:

```
keccak256("ab", "c")
keccak256("abc")
keccak256(0x616263)
keccak256(6382179)
keccak256(97, 98, 99)
```

Si hiciera falta relleno (padding), se pueden usar las conversiones explícitas de tipo: `keccak256("\x00\x12")` es lo mismo que `keccak256(uint16(0x12))`.

Ten en cuenta que las constantes se compactarán usando el mínimo numero de bytes requeridos para almacenarlas. Eso significa por ejemplo que `keccak256(0) == keccak256(uint8(0))` y `keccak256(0x12345678) == keccak256(uint32(0x12345678))`.

Podría pasar que le falte gas para llamar a las funciones `sha256`, `ripemd160` or `ecrecover` en *blockchain privadas*. La razón de ser así se debe a que esas funciones están implementadas como contratos precompilados y estos contratos sólo existen depués de recibir el primer mensaje (a pesar de que el contrato está hardcodeado). Los mensajes

que se envían a contratos que todavía no existen son más caros y por lo tanto su ejecución puede llevar a un error por una falta de gas (Out-of-Gas error). Una solución a este problema consiste por ejemplo en mandar 1 Wei a todos los contratos antes de empezar a usarlos. Note que esto no llevará a un error en la red oficial o en la red de testeo.

En relación a las direcciones

`<address>.balance (uint256)`: balance en Wei de la *dirección*. `<address>.transfer(uint256 amount)`: envía el importe deseado en Wei a la *dirección* o lanza excepción si falla. `<address>.send(uint256 amount) returns (bool)`: envía el importe deseado en Wei a la *dirección* o devuelve `false` si falla. `<address>.call(...) returns (bool)`: crea una instrucción de tipo `CALL` a bajo nivel o devuelve `false` si falla. `<address>.callcode(...) returns (bool)`: crea una instrucción de tipo `CALLCODE` a bajo nivel o devuelve `false` si falla. `<address>.delegatecall(...) returns (bool)`: crea una instrucción de tipo `DELEGATECALL` a bajo nivel o devuelve `false` si falla.

Para más información, véase la sección *dirección*.

Advertencia: Existe un peligro a la hora de usar `send`: la transferencia falla si la profundidad de la pila de llamadas es de 1024 (esto siempre lo puede forzar el que hace la llamada); también falla si el destinatario se queda sin gas. Entonces, para asegurarse de hacer transferencias seguras en Ether, fíjese siempre en el valor devuelto por `send`, use `transfer` en lugar de `send`, o mejor aún, use un patrón donde es el destinatario quien retira los fondos.

En relación a los contratos

`this` (el tipo del contrato actual): el contrato actual, explicitamente convertible en *Address*. `selfdestruct(address recipient)`: destruye el contrato actual y envía los fondos que tiene a una *dirección* especificada.

Además, todas las funciones del contrato actual se pueden llamar directamente, incluida la función actual.

Expresiones y estructuras de control

Parámetros de entrada y de salida

Al igual que en Javascript, las funciones obtienen parámetros como entrada. Por otro lado, al contrario que en Javascript y C, estas también deberían devolver un número aleatorio de parámetros como salida.

Parámetros de entrada

Los parámetros de entrada se declaran de la misma forma que las variables. Como una excepción, los parámetros no usados pueden omitir el nombre de la variable. Por ejemplo, si quisiéramos que nuestro contrato acepte un tipo de llamadas externas con dos enteros, el código quedaría similar a este:

```
contract Simple {
    function taker(uint _a, uint _b) {
        // hace algo con _a y _b.
    }
}
```

Parámetros de salida

Los parámetros de salida se pueden declarar con la misma sintaxis después de la palabra reservada `returns`. Por ejemplo, supongamos que deseamos devolver dos resultados: la suma y el producto de dos valores dados. Entonces, escribiríamos un código como este:

```
contract Simple {
    function arithmetics(uint _a, uint _b) returns (uint o_sum, uint o_product) {
        o_sum = _a + _b;
        o_product = _a * _b;
    }
}
```

Los nombres de los parámetros de salida se pueden omitir. Los valores de salida se pueden especificar también usando declaraciones `return`. Las declaraciones `return` también son capaces de devolver múltiples valores, ver *Devolver múltiples valores*. Los parámetros de retorno se inicializan a cero; si no se especifica explícitamente su valor, permanecen con dicho valor cero.

Los parámetros de entrada y salida se pueden usar como expresiones en el cuerpo de la función. En este caso, también pueden ir en el lado izquierdo de una asignación.

Estructuras de control

La mayoría de las estructuras de control disponibles en JavaScript, también lo están en Solidity exceptuando `switch` y `goto`. Esto significa que tenemos: `if`, `else`, `while`, `do`, `for`, `break`, `continue`, `return`, `? :`, con la semántica habitual conocida de C o JavaScript.

Los paréntesis no se pueden omitir para condicionales, pero sí las llaves alrededor de los cuerpos de las declaraciones sencillas.

Hay que tener en cuenta que no hay conversión de tipos desde non-boolean a boolean como hay en C y JavaScript, por lo que `if (1) { ... }` *no* es válido en Solidity.

Devolver múltiples valores

Cuando una función tiene múltiples parámetros de salida, `return (v0, v1, ..., vn)` puede devolver múltiples valores. El número de componentes debe ser el mismo que el de parámetros de salida.

Llamadas a funciones

Llamadas a funciones internas

Las funciones del contrato actual pueden ser llamadas directamente ("internamente") y, también, recursivamente como se puede ver en este ejemplo sin sentido funcional:

```
contract C {
    function g(uint a) returns (uint ret) { return f(); }
    function f() returns (uint ret) { return g(7) + f(); }
}
```

Estas llamadas a funciones son traducidas en simples saltos dentro de la máquina virtual de Ethereum (EVM). Esto tiene como consecuencia que la memoria actual no se limpia, así que pasar referencias de memoria a las funciones llamadas internamente es muy eficiente. Sólo las funciones del mismo contrato pueden ser llamadas internamente.

Llamadas a funciones externas

Las expresiones `this.g(8);` and `c.g(2);` (donde `c` es la instancia de un contrato) son también llamadas válidas, pero en esta ocasión, la función se llamará "externamente" mediante un message call y no directamente por saltos. Por favor, es importante tener en cuenta que las llamadas a funciones en `this` no pueden ser usadas en el constructor, ya que el contrato en cuestión no se ha creado todavía.

Las funciones de otros contratos se tienen que llamar de forma externa. Para una llamada externa, todos los argumentos de la función tienen que ser copiados en memoria.

Cuando se llama a funciones de otros contratos, la cantidad de Wei enviada con la llamada y el gas pueden especificarse con las opciones especiales `.value()` y `.gas()`, respectivamente:

```
contract InfoFeed {
    function info() payable returns (uint ret) { return 42; }
}


contract Consumer {
    InfoFeed feed;
    function setFeed(address addr) { feed = InfoFeed(addr); }
    function callFeed() { feed.info.value(10).gas(800)(); }
}
```

El modificador `payable` se tiene que usar para `info`, porque de otra manera la opción *.value()* no estaría disponible.

Destacar que la expresión `InfoFeed(addr)` realiza una conversión de tipo explícita afirmando que "sabemos que el tipo de contrato en la dirección dada es `InfoFeed`", sin ejecutar un constructor. Las conversiones de tipo explícitas tienen que ser gestionadas con extrema precaución. Nunca se debe llamar a una función en un contrato donde no se sabe con seguridad cuál es su tipo.

También se podría usar `function setFeed(InfoFeed _feed) { feed = _feed; }` directamente. Hay que tener cuidado con el hecho de que `feed.info.value(10).gas(800)` sólo (localmente) establece el valor y la cantidad de gas enviada con la llamada a la función. Sólo tras el último paréntesis se realiza realmente la llamada.

Las llamadas a funciones provocan excepciones si el contrato invocado no existe (en el sentido de que la cuenta no contiene código) o si el contrato invocado por sí mismo dispara una excepción o se queda sin gas.

Advertencia: Cualquier interacción con otro contrato supone un daño potencial, especialmente si el código fuente del contrato no se conoce de antemano. El contrato actual pasa el control al contrato invocado y eso potencialmente podría suponer que haga cualquier cosa. Incluso si el contrato invocado hereda de un contrato padre conocido, el contrato del que hereda sólo requiere tener una interfaz correcta. La implementación del contrato, sin embargo, puede ser totalmente aleatoria y, por ello, crear un perjuicio. Además, hay que estar preparado en caso de que llame dentro de otros contratos del sistema o, incluso, volver al contrato que lo llama antes de que la primera llamada retorne. Esto significa que el contrato invocado puede cambiar variables de estado del contrato que le llama mediante sus funciones. Escribir tus funciones de manera que realicen, por ejemplo, llamadas a funciones externas ocurridas después de cualquier cambio en variables de estado en tu contrato, hace que este contrato no sea vulnerable a un ataque de reentrada.

Llamadas con nombre y parámetros de funciones anónimas

Los argumentos de una llamada a una función pueden venir dados por el nombre, en cualquier orden, si están entre `{ }` como se puede ver en el siguiente ejemplo. La lista de argumentos tiene que coincidir por el nombre con la lista de parámetros de la declaración de la función, pero pueden estar en orden aleatorio.

```
pragma solidity ^0.4.0;

contract C {
    function f(uint key, uint value) { ... }

    function g() {
        // argumentos con nombre
        f({value: 2, key: 3});
    }
}
```

Nombres de parámetros de función omitidos

Los nombres de parámetros no usados (especialmente los de retorno) se pueden omitir. Esos nombres estarán presentes en la pila, pero serán inaccesibles.

```
pragma solidity ^0.4.0;

contract C {
    // Se omite el nombre para el parámetro
    function func(uint k, uint) returns(uint) {
        return k;
    }
}
```

Creando contratos mediante `new`

Un contrato puede crear un nuevo contrato usando la palabra reservada `new`. El código completo del contrato que se está creando tiene que ser conocido de antemano, por lo que no son posibles las dependencias de creación recursivas.

```
pragma solidity ^0.4.0;

contract D {
    uint x;
    function D(uint a) payable {
        x = a;
    }
}


contract C {
    D d = new D(4); // Se ejecutará como parte del constructor de C

    function createD(uint arg) {
        D newD = new D(arg);
    }

    function createAndEndowD(uint arg, uint amount) {
        // Envía Ether junto con la creación
        D newD = (new D).value(amount)(arg);
    }
}
```

Como se ve en el ejemplo, es posible traspasar Ether a la creación usando la opción `.value()`, pero no es posible

limitar la cantidad de gas. Si la creación falla (debido al desbordamiento de la pila, falta de balance o cualquier otro problema), se dispara una excepción.

Orden de la evaluación de expresiones

El orden de evaluación de expresiones no se especifica (más formalmente, el orden en el que los hijos de un nodo en el árbol de la expresión son evaluados no es especificado. Eso sí, son evaluados antes que el propio nodo). Sólo se garantiza que las sentencias se ejecutan en orden y que se hace un cortocircuito para las expresiones booleanas. Ver *Orden de preferencia de operadores* para más información.

Asignación

Asignaciones para desestructurar y retornar múltiples valores

Solidity internamente permite tipos tupla, p.ej.: una lista de objetos de, potencialmente, diferentes tipos cuyo tamaño es constante en tiempo de compilación. Esas tuplas pueden ser usadas para retornar múltiples valores al mismo tiempo y, también, asignarlos a múltiples variables (o lista de valores en general) también al mismo tiempo:

```
contract C {
    uint[] data;

    function f() returns (uint, bool, uint) {
        return (7, true, 2);
    }

    function g() {
        // Declara y asigna variables. No es posible especificar el tipo de forma␣
↪explícita.
        var (x, b, y) = f();
        // Asigna a una variable pre-existente.
        (x, y) = (2, 7);
        // Truco común para intercambiar valores -- no funciona con tipos de␣
↪almacenamiento sin valor.
        (x, y) = (y, x);
        // Los componentes se pueden dejar fuera (también en declaraciones de␣
↪variables).
        // Si la tupla acaba en un componente vacío,
        // el resto de los valores se descartan.
        (data.length,) = f(); // Establece la longitud a 7
        // Lo mismo se puede hacer en el lado izquierdo.
        (,data[3]) = f(); // Sets data[3] to 2
        // Los componentes sólo se pueden dejar en el lado izquierdo de las␣
↪asignaciones, con
        // una excepción:
        (x,) = (1,);
        // (1,) es la única forma de especificar una tupla de un componente, porque␣
↪(1)
        // equivale a 1.
    }
}
```

Complicaciones en Arrays y Structs

La sintaxis de asignación es algo más complicada para tipos sin valor como arrays y structs. Las asignaciones *a* variables de estado siempre crean una copia independiente. Por otro lado, asignar una variable local crea una copia independiente sólo para tipos elementales, como tipos estáticos que encajan en 32 bytes. Si los structs o arrays (incluyendo `bytes` y `string`) son asignados desde una variable de estado a una local, la variable local se queda una referencia a la variable de estado original. Una segunda asignación a la variable local no modifica el estado, sólo cambia la referencia. Las asignaciones a miembros (o elementos) de la variable local *hacen* cambiar el estado.

Scoping y declaraciones

Una variable cuando se declara tendrá un valor inicial por defecto que, representado en bytes, será todo ceros. Los valores por defecto de las variables son los típicos "estado-cero" cualquiera que sea el tipo. Por ejemplo, el valor por defecto para un `bool` es `false`. El valor por defecto para un `uint` o `int` es `0`. Para arrays de tamaño estático y `bytes1` hasta `bytes32`, cada elemento individual será inicializado a un valor por defecto según sea su tipo. Finalmente, para arrays de tamaño dinámico, `bytes``y ``string`, el valor por defecto es un array o string vacío.

Una variable declarada en cualquier punto de una función estará dentro del alcance de *toda la función*, independientemente de donde se haya declarado. Esto ocurre porque Solidity hereda sus reglas de scoping de JavaScript. Esto difiere de muchos lenguajes donde las variables sólo están en el alcance de donde se declaran hasta que acaba el bloque semántico. Como consecuencia de esto, el código siguiente es ilegal y hace que el compilador devuelva un error porque el identificador se ha declarado previamente, `Identifier already declared`:

```
pragma solidity ^0.4.0;

contract ScopingErrors {
    function scoping() {
        uint i = 0;

        while (i++ < 1) {
            uint same1 = 0;
        }

        while (i++ < 2) {
            uint same1 = 0; // Ilegal, segunda declaración para same1
        }
    }

    function minimalScoping() {
        {
            uint same2 = 0;
        }

        {
            uint same2 = 0; // Ilegal, segunda declaración para same2
        }
    }

    function forLoopScoping() {
        for (uint same3 = 0; same3 < 1; same3++) {
        }

        for (uint same3 = 0; same3 < 1; same3++) { // Ilegal, segunda declaración␣
↪para same3
        }
```

```
    }
}
```

Como añadido a esto, si la variable se declara, se inicializará al principio de la función con su valor por defecto. Esto significa que el siguiente código es legal, aunque se haya escrito de manera un tanto pobre:

```
function foo() returns (uint) {
    // baz se inicializa implícitamente a 0
    uint bar = 5;
    if (true) {
        bar += baz;
    } else {
        uint baz = 10;// Nunca se ejecuta
    }
    return bar;// devuelve 5
}
```

Excepciones

Hay algunos casos en los que las excepciones se lanzan automáticamente (ver más adelante). Se puede usar la instrucción `throw` para lanzarlas manualmente. La consecuencia de una excepción es que la llamada que se está ejecutando en ese momento se para y se revierte (todos los cambios en los estados y balances se deshacen) y la excepción también se genera mediante llamadas de función de Solidity (las excepciones `send` y las funciones de bajo nivel `call`, `delegatecall` y `callcode`, todas ellas devuelven `false` en caso de una excepción).

Todavía no es posible capturar excepciones.

En el siguiente ejemplo, se enseña como `throw` se puede usar para revertir fácilmente una transferencia de Ether y, además, se enseña como comprobar el valor de retorno de `send`:

```
pragma solidity ^0.4.0;

contract Sharer {
    function sendHalf(address addr) payable returns (uint balance) {
        if (!addr.send(msg.value / 2))
            throw; // También revierte la transferencia de Sharer
        return this.balance;
    }
}
```

Actualmente, Solidity genera automáticamente una excepción en tiempo de ejecución en las siguientes situaciones:

1. Si se accede a un array en un índice demasiado largo o negativo (ejemplo: `x[i]` donde `i >= x.length` o `i < 0`).
2. Si se accede a un `bytesN` de longitud fija en un índice demasiado largo o negativo.
3. Si se llama a una función con un message call, pero no finaliza adecuadamente (ejemplo: se queda sin gas, no tiene una función de matching, o dispara una excepción por sí mismo), exceptuando el caso en el que se use una operación de bajo nivel `call`, `send`, `delegatecall` o `callcode`. Las operaciones de bajo nivel nunca disparan excepciones, pero indican fallos devolviendo `false`.
4. Si se crea un contrato usando la palabra reservada `new`, pero la creación del contrato no finaliza correctamente (ver más arriba la definición de "no finalizar correctamente").
5. Si se divide o se hace módulo por cero (ejemplos: `5 / 0` o `23 % 0`).
6. Si se hace un movimiento por una cantidad negativa.

7. Si se convierte un valor muy grande o negativo en un tipo enum.
8. Si se realiza una llamada de función externa apuntando a un contrato que no contiene código.
9. Si un contrato recibe Ether mediante una función sin el modificador `payable` (incluyendo el constructor y la función de fallback).
10. Si un contrato recibe Ether mediante una función getter pública.
11. Si se llama a una variable inicializada a cero de un tipo de función interna.
12. Si un `.transfer()` falla.
13. Si se invoca con `assert` junto con un argumento que evalúa a falso.

Un usuario genera una excepción en las siguientes situaciones:

1. Llamando a `throw`.
2. Llamando a `require` junto con un argumento que evalúa a `false`.

Internamente, Solidity realiza una operación de revertir (revert, instrucción `0xfd`) cuando una excepción provista por un usuario se lanza o la condición de la llamada `require` no se satisface. Por contra, realiza una operación inválida (instrucción `0xfe`) si una excepción en tiempo de ejecución aparece o la condición de una llamada `assert` no se satisface. En ambos casos, esto ocasiona que la EVM revierta todos los cambios de estado acaecidos. El motivo de todo esto es que no existe un modo seguro de continuar con la ejecución debido a que no sucedió el efecto esperado. Como se quiere mantener la atomicidad de las transacciones, lo más seguro es revertir todos los cambios y hacer que la transacción no tenga ningún efecto en su totalidad o, como mínimo, en la llamada.

En el caso de que los contratos se escriban de tal manera que `assert` sólo sea usado para probar condiciones internas y `require` se use en caso de que haya una entrada malformada, una herramienta de análisis formal que verifique que el opcode inválido nunca pueda ser alcanzado, se podría usar para chequear la ausencia de errores asumiendo entradas válidas.

Contratos

Los contratos en Solidity son similares a las clases en los lenguajes orientados a objeto. Los contratos contienen datos persistentes almacenados en variables de estados y funciones que pueden modificar estas variables. Llamar a una función de un contrato diferente (instancia) realizará una llamada a una función de la EVM (Máquina Virtual de Ethereum) para que cambie el contexto de manera que las variables de estado no estén accesibles.

Crear contratos

Los contratos pueden crearse "desde fuera" o desde contratos en Solidity. Cuando se crea un contrato, su constructor (una función con el mismo nombre que el contrato) se ejecuta una sola vez.

El constructor es opcional. Se admite un solo constructor, lo que significa que la sobrecarga no está soportada.

Desde `web3.js`, es decir la API de JavaScript, esto se hace de la siguiente manera:

```
// Es necesario especificar alguna fuente, incluido el nombre del contrato para los␣
↪parametros de abajo
var source = "contract CONTRACT_NAME { function CONTRACT_NAME(uint a, uint b) {} }";

// El array json del ABI generado por el compilador
var abiArray = [
    {
        "inputs":[
            {"name":"x","type":"uint256"},
            {"name":"y","type":"uint256"}
```

```
        ],
        "type":"constructor"
    },
    {
        "constant":true,
        "inputs":[],
        "name":"x",
        "outputs":[{"name":"","type":"bytes32"}],
        "type":"function"
    }
];

var MyContract_ = web3.eth.contract(source);
MyContract = web3.eth.contract(MyContract_.CONTRACT_NAME.info.abiDefinition);

// Desplegar el nuevo contrato
var contractInstance = MyContract.new(
    10,
    11,
    {from: myAccount, gas: 1000000}
);
```

Internamente, los argumentos del constructor son transmitidos después del propio código del contrato, pero no se tiene que preocupar de eso si utiliza `web3.js`.

Si un contrato quiere crear otros contratos, el creador tiene que conocer el código fuente (y el binario) del contrato a crear. Eso significa que la creación de dependencias cíclicas es imposible.

```
pragma solidity ^0.4.0;

contract OwnedToken {
    // TokenCreator es un contrato que está definido más abajo.
    // No hay problema en referenciarlo, siempre y cuando no esté
    // siendo utilizado para crear un contrato nuevo.
    TokenCreator creator;
    address owner;
    bytes32 name;

    // Esto es el constructor que registra el creador y el nombre
    // que se le ha asignado
    function OwnedToken(bytes32 _name) {
        // Se accede a las variables de estado por su nombre
        // y no, por ejemplo, por this.owner. Eso también se aplica
        // a las funciones y, especialmente en los constructores,
        // solo está permitido llamarlas de esa manera ("internal"),
        // porque el propio contrato no existe todavía.
        owner = msg.sender;
        // Hacemos una conversión explícita de tipo, desde `address`
        // a `TokenCreator` y asumimos que el tipo del contrato que hace
        // la llamada es TokenCreator, ya que realmente no hay
        // formas de corroborar eso.
        creator = TokenCreator(msg.sender);
        name = _name;
    }

    function changeName(bytes32 newName) {
        // Solo el creador puede modificar el nombre --
        // la comparación es posible ya que los contratos
```

```
        // se pueden convertir a direcciones de forma implícita.
        if (msg.sender == address(creator))
            name = newName;
    }

    function transfer(address newOwner) {
        // Solo el creador puede transferir el token.
        if (msg.sender != owner) return;
        // También vamos a querer preguntar al creador
        // si la transferencia ha salido bien. Note que esto
        // tiene como efecto llamar a una función del contrato
        // que está definida más abajo. Si la llamada no funciona
        // (p.ej si no queda gas), la ejecución se para aquí inmediatamente.
        if (creator.isTokenTransferOK(owner, newOwner))
            owner = newOwner;
    }
}

contract TokenCreator {
    function createToken(bytes32 name)
       returns (OwnedToken tokenAddress)
    {
        // Crea un contrato para crear un nuevo Token.
        // Del lado de JavaScript, el tipo que se nos devuelve
        // simplemente es la dirección ("address"), ya que ese
        // es el tipo más cercano disponible en el ABI.
        return new OwnedToken(name);
    }

    function changeName(OwnedToken tokenAddress, bytes32 name) {
        // De nuevo, el tipo externo de "tokenAddress"
        // simplemente es "address".
        tokenAddress.changeName(name);
    }

    function isTokenTransferOK(
        address currentOwner,
        address newOwner
    ) returns (bool ok) {
        // Verifica una condición arbitraria
        address tokenAddress = msg.sender;
        return (keccak256(newOwner) & 0xff) == (bytes20(tokenAddress) & 0xff);
    }
}
```

Visibilidad y getters

Ya que Solidity sólo conoce dos tipos de llamadas a una función (las internas que no generan una llamada a la EVM (también llamadas “message calls”) y las externas que sí generan una llamada a la EVM), hay cuatro tipos de visibilidad para las funciones y las variables de estado.

Una función puede especificarse como `external`, `public`, `internal` o `private`. Por defecto una función es `public`. Para las variables de estado, el tipo `external` no es posible y el tipo por defecto es `internal`.

`external`: Las funciones externas son parte de la interfaz del contrato, lo que significa que pueden llamarse desde otros contratos y vía transacciones. Una función externa `f` no puede llamarse internamente (por ejemplo `f()` no

funciona, pero `this.f()` funciona). Las funciones externas son a veces más eficientes cuando reciben grandes arrays de datos.

`public`: Las funciones públicas son parte de la interfaz del contrato y pueden llamarse internamente o vía mensajes. Para las variables de estado públicas, se genera una función getter automática (ver más abajo).

`internal`: Estas funciones y variables de estado sólo pueden llamarse internamente (es decir, desde dentro del contrato actual o desde contratos de derivan del mismo), sin poder usarse `this`.

`private`: Las funciones y variables de estado privadas sólo están visibles para el contrato en el que se han definido y no para contratos de derivan del mismo.

Nota: Todo lo que está definido dentro de un contrato es visible para todos los observadores externos. Definir algo como `private` sólo impide que otros contratos puedan acceder y modificar la información, pero esta información siempre será visible para todo el mundo, incluso fuera de la blockchain.

El especificador de visibilidad se pone después del tipo para las variables de estado y entre la lista de parámetros y la lista de parámetros de retorno para las funciones.

```
pragma solidity ^0.4.0;

contract C {
    function f(uint a) private returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    uint public data;
}
```

En el siguiente ejemplo, `D`, puede llamar a `c.getData()` para recuperar el valor de `data` en el almacén de estados, pero no puede llamar a `f`. El contrato `E` deriva de `C` y, por lo tanto, puede llamar a `compute`.

```
pragma solidity ^0.4.0;

contract C {
    uint private data;

    function f(uint a) private returns(uint b) { return a + 1; }
    function setData(uint a) { data = a; }
    function getData() public returns(uint) { return data; }
    function compute(uint a, uint b) internal returns (uint) { return a+b; }
}


contract D {
    function readData() {
        C c = new C();
        uint local = c.f(7); // error: el miembro "f" no es visible
        c.setData(3);
        local = c.getData();
        local = c.compute(3, 5); // error: el miembro "compute" no es visible
    }
}


contract E is C {
    function g() {
        C c = new C();
        uint val = compute(3, 5);  // acceso a un miembro interno (desde un contrato␣
↪derivado a su contrato padre)
```

```
    }
}
```

Funciones getter

El compilador crea automáticamente funciones getter para todas las variables de estado **públicas**. En el contrato que se muestra abajo, el compilador va a generar una función llamada `data` que no lee ningún argumento y devuelve un `uint`, el valor de la variable de estado `data`. La inicialización de las variables de estado se puede hacer en el momento de la declaración.

```
pragma solidity ^0.4.0;

contract C {
    uint public data = 42;
}


contract Caller {
    C c = new C();
    function f() {
        uint local = c.data();
    }
}
```

Las funciones getter tienen visibilidad externa. Si se accede al símbolo internamente (es decir sin `this.`), entonces se evalúa como una variable de estado. Si se accede al símbolo externamente, (es decir con `this.`), entonces se evalúa como una función.

```
pragma solidity ^0.4.0;

contract C {
    uint public data;
    function x() {
        data = 3; // acceso interno
        uint val = this.data(); // acceso externo
    }
}
```

El siguiente ejemplo es un poco más complejo:

```
pragma solidity ^0.4.0;

contract Complex {
    struct Data {
        uint a;
        bytes3 b;
        mapping (uint => uint) map;
    }
    mapping (uint => mapping(bool => Data[])) public data;
}
```

Nos va a generar una función de la siguiente forma:

```
function data(uint arg1, bool arg2, uint arg3) returns (uint a, bytes3 b) {
    a = data[arg1][arg2][arg3].a;
```

```
    b = data[arg1][arg2][arg3].b;
}
```

Notese que se ha omitido el mapeo en el struct porque no hay una buena manera de dar la clave para hacer el mapeo.

Modificadores de funciones

Se pueden usar los modificadores para cambiar el comportamiento de las funciones de una manera ágil. Por ejemplo, los modificadores son capaces de comprobar automáticamente una condición antes de ejecutar una función. Los modificadores son propiedades heredables de los contratos y pueden ser sobrescritos por contratos derivados.

```
pragma solidity ^0.4.11;

contract owned {
    function owned() { owner = msg.sender; }
    address owner;

    // Este contrato sólo define un modificador pero no lo usa, se va a utilizar en un contrato derivado.
    // El cuerpo de la función se inserta donde aparece el símbolo especial "_;" en la definición del modificador.
    // Esto significa que si el propietario llama a esta función, la función se ejecuta, pero en otros casos devolverá una excepción.
    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }
}


contract mortal is owned {
    // Este contrato hereda del modificador "onlyOwner" desde "owned" y lo aplica a la función "close", lo que tiene como efecto que las llamadas a "close" solamente tienen efecto si las hace el propietario registrado.
    function close() onlyOwner {
        selfdestruct(owner);
    }
}


contract priced {
    // Los modificadores pueden recibir argumentos:
    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
    }
}


contract Register is priced, owned {
    mapping (address => bool) registeredAddresses;
    uint price;

    function Register(uint initialPrice) { price = initialPrice; }
```

```
    // Aquí es importante facilitar también la palabra clave "payable", de lo
↪contrario la función rechazaría automáticamente todos los ethers que le mandemos.
    function register() payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }

    function changePrice(uint _price) onlyOwner {
        price = _price;
    }
}

contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(!locked);
        locked = true;
        _;
        locked = false;
    }

/// Esta función está protegida por un mutex, lo que significa que llamadas reentrantes desde den
/// La declaración `return 7` asigna 7 al valor devuelto, pero aún así ejecuta la declaración `lo
    function f() noReentrancy returns (uint) {
        require(msg.sender.call());
        return 7;
    }
}
```

Se pueden aplicar varios modificadores a una misma función especificándolos en una lista separada por espacios en blanco. Serán evaluados en el orden presentado en la lista.

Advertencia: En una versión anterior de Solidity, declaraciones del tipo `return` dentro de funciones que contienen modificadores se comportaban de otra manera.

Lo que se devuelve explícitamente de un modificador o del cuerpo de una función solo sale del modificador actual o del cuerpo de la función actual. Las variables que se devuelven están asignadas y el control de flujo continúa después del "_" en el modificador que precede.

Se aceptan expresiones arbitrarias para los argumentos del modificador y en ese contexto, todos los símbolos visibles desde la función son visibles en el modificador. Símbolos introducidos en el modificador no son visibles en la función (ya que pueden cambiar por sobreescritura).

Variables de estado constantes

Las variables de estado pueden declarase como `constant`. En este caso, se tienen que asignar desde una expresión que es una constante en tiempo de compilación. Las expresiones que acceden al almacenamiento, datos sobre la blockchain (p.ej `now`, `this.balance` o `block.number`), datos sobre la ejecución (`msg.gas`) o que hacen llamadas a contratos externos, están prohibidas. Las expresiones que puedan tener efectos colaterales en el reparto de memoria están permitidas, pero las que puedan tener efectos colaterales en otros objetos de memoria no lo están. Las funciones por defecto `keccak256`, `sha256`, `ripemd160`, `ecrecover`, `addmod` y `mulmod` están permitidas (aunque hacen llamadas a contratos externos).

Se permiten efectos colaterales en el repartidor de memoria porque debe ser posible construir objetos complejos como p.ej lookup-tables. Esta funcionalidad todavía no se puede usar tal cual.

El compilador no guarda un espacio de almacenamiento para estas variables, y se remplaza cada ocurrencia por su respectiva expresión constante (que puede ser compilada como un valor simple por el optimizador).

En este momento, no todos los tipos para las constantes están implementados. Los únicos tipos implementados por ahora son los tipos de valor y las cadenas de texto (string).

```
pragma solidity ^0.4.0;

contract C {
    uint constant x = 32**22 + 8;
    string constant text = "abc";
    bytes32 constant myHash = keccak256("abc");
}
```

Funciones constantes

En el caso en que una función se declare como constante, promete no modificar el estado.

```
pragma solidity ^0.4.0;

contract C {
    function f(uint a, uint b) constant returns (uint) {
        return a * (b + 42);
    }
}
```

Nota: Los métodos getter están marcados como constantes.

Advertencia: El compilador todavía no impone que un método constante no modifique el estado.

Función fallback

Un contrato puede tener exactamente una sola función sin nombre. Esta función no puede tener argumentos ni puede devolver nada. Se ejecuta si, al llamar al contrato, ninguna de las otras funciones del contrato se corresponde al identificador de función proporcionado (o si no se hubiera proporcionado ningún dato).

Además, esta función se ejecutará siempre y cuando el contrato sólo reciba Ether (sin datos). En este caso en general hay muy poco gas disponible para una llamada a una función (para ser preciso, 2300 gas), por eso es importante hacer las funciones fallback lo más baratas posible.

En particular, las siguientes operaciones consumirán más gas de lo que se da como estipendio para una función fallback.

- Escribir en almacenamiento
- Crear un contrato
- Llamar a una función externa que consume una cantidad de gas significativa
- Mandar Ether

Asegúrese por favor de testear su función fallback meticulosamente antes de desplegar el contrato para asegurarse de que su coste de ejecución es menor de 2300 gas.

Advertencia: Los contratos que reciben Ether directamente (sin una llamada a una función, p.ej usando `send` o `transfer`) pero que no tienen definida una función fallback, van a lanzar una excepción, devolviendo el Ether (nótese que esto era diferente antes de la versión v0.4.0 de Solidity). Por lo tanto, si desea que su contrato reciba Ether, tiene que implementar una función fallback.

```
pragma solidity ^0.4.0;

contract Test {
    // Se llama a esta función para todos los mensajes enviados a este contrato (no
↪hay otra función). Enviar Ether a este contrato lanza una excepción, porque la
↪función fallback no tiene el modificador "payable".
    function() { x = 1; }
    uint x;
}


// Este contrato guarda todo el Ether que se le envía sin posibilidad de recuperarlo.
contract Sink {
    function() payable { }
}


contract Caller {
    function callTest(Test test) {
        test.call(0xabcdef01); // el hash no existe
        // resulta en que test.x se vuelve == 1.

        // La siguiente llamada falla, devuelve el Ether y devuelve un error:
        test.send(2 ether);
    }
}
```

Eventos

Los eventos permiten el uso conveniente de la capacidad de registro del EVM, que a su vez puede "llamar" a callbacks de JavaScript en la interfaz de usuario de una dapp que escucha a esos eventos.

Los eventos son miembros heredables de los contratos. Cuando se les llama, hacen que los argumentos se guarden en el registro de transacciones - una estructura de datos especial en la blockchain. Estos registros están asociados con la dirección del contrato y serán incorporados en la blockchain y allí permanecerán siempre que un bloque esté accesible (eso es: para siempre con Frontier y con Homestead, pero puede cambiar con Serenity). Los datos de registros y de eventos no están disponibles desde dentro de los contratos (ni siquiera desde el contrato que los ha creado).

Se pueden hacer pruebas SPV para los registros, de manera que si una entidad externa proporciona un contrato con dicha prueba, se puede comprobar que el registro realmente existe en la blockchain. Dicho esto, tenga en cuenta que las cabeceras de bloque deben proporcionarse porque el contrato sólo lee los últimos 256 hashes de bloque.

Hasta tres parámetros pueden recibir el atributo `indexed`, lo que hará que se busque por los respectivos parámetros. En la interfaz de usuario, es posible filtrar por los valores específicos de argumentos indexados.

Si se utilizan arrays como argumentos indexados (incluyendo `string` y `bytes`), en su lugar se guarda su hash Keccak-256 como asunto.

El hash de la firma de un evento es uno de los asuntos, excepto si declaras el evento con el especificador `anonymous`. Esto significa que no es posible filtrar por eventos anónimos específicos por su nombre.

Todos los argumentos no indexados se guardarán en la parte de datos del registro.

Nota: No se guardan los argumentos indexados propiamente dichos. Uno sólo puede buscar por los valores, pero es imposible recuperar los valores en sí.

```
pragma solidity ^0.4.0;

contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );

    function deposit(bytes32 _id) payable {
        // Cualquier llamada a esta función (por muy anidada que sea) puede ser␣
↪detectada desde la API de JavaScript con un filtro para que se llame a `Deposit`.
        Deposit(msg.sender, _id, msg.value);
    }
}
```

Su uso en la API de JavaScript sería como sigue:

```
var abi = /* abi generado por el compilador */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at(0x123 /* dirección */);

var event = clientReceipt.Deposit();

// mirar si hay cambios
event.watch(function(error, result){
    // el resultado contendrá varias informaciones incluyendo los argumentos␣
↪proporcionados en el momento de la llamada a Deposit.
    if (!error)
        console.log(result);
});

// O ejecutar una funcin callback para empezar a escuchar de inmediato
var event = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result);
});
```

Interfaz a registros de bajo nivel

También es posible acceder al mecanismo de logging a través de la interfaz de bajo nivel mediante las funciones `log0`, `log1`, `log2`, `log3` y `log4`. `logi` toma `i + 1` parámetros del tipo `bytes32`, donde el primer argumento se utiliza para la parte de datos del log y los otros como asuntos. La llamada al evento de arriba puede realizarse de una manera similar a esta:

```
log3(
    msg.value,
```

```
    0x50cb9fe53daa9737b786ab3646f04d0150dc50ef4e75f59509d83667ad5adb20,
    msg.sender,
    _id
);
```

donde el numero hexadecimal largo es igual a `keccak256("Deposit(address,hash256,uint256)")`, la firma del evento.

Recursos adicionales para entender los eventos

- Documentación de Javascript
- Ejemplo de uso de los eventos
- Cómo acceder a eventos con js

Herencia

Solidity soporta multiples herencias copiando el código, incluyendo el polimorfismo.

Todas las llamadas a funciones son virtuales, lo que significa que es la función más derivada la que se llama, excepto cuando el nombre del contrato se menciona explícitamente.

Cuando un contrato hereda de múltiples contratos, un solo contrato está creado en la blockchain, y el código de todos los contratos base está copiado dentro del contrato creado.

El sistema general de herencia es muy similar al de Python, especialmente en lo que se refiere a herencias multiples.

En el siguiente ejemplo se dan más detalles.

```
pragma solidity ^0.4.0;

contract owned {
    function owned() { owner = msg.sender; }
    address owner;
}


// Usar "is" para derivar de otro contrato. Los contratos derivados
// pueden acceder a todos los miembros no privados, incluidas las
// funciones internas y variables de estado. A éstas sin embargo
// no se puede acceder externamente mediante `this`.
contract mortal is owned {
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
    }
}


// Estos contratos abstractos sólo se proporcionan para que el compilador
// sepa de la interfaz. Nótese que la función no tiene cuerpo. Si un contrato
// no implementa todas las funciones, sólo puede usarse como interfaz.
contract Config {
    function lookup(uint id) returns (address adr);
}
```

```
contract NameReg {
    function register(bytes32 name);
    function unregister();
 }


// Las herencias multiples son posibles. Nótese que "owned" también es una clase base
// de "mortal", aun así hay una sóla instancia de "owned" (igual que para las herencias virtuales en C++).
contract named is owned, mortal {
    function named(bytes32 name) {
        Config config = Config(0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970);
        NameReg(config.lookup(1)).register(name);
    }

    // Las funciones pueden ser sobreescritas por otras funciones con el mismo nombre
    // y el mismo numero/tipo de entradas. Si la función que sobreescribe tiene distintos
    // tipos de parámetros de salida, esto provocará un error.
    // Tanto las llamadas a funciones locales como las que están basadas en mensajes
    // tienen en cuenta estas sobreescrituras.
    function kill() {
        if (msg.sender == owner) {
            Config config = Config(0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970);
            NameReg(config.lookup(1)).unregister();
            // Sigue siendo posible llamar a una función específica que ha sido sobreescrita.
            mortal.kill();
        }
    }
}


// Si un constructor acepta un argumento, es necesario proporcionarlo en la cabecera
// (o de forma similar a como se hace con los modificadores, en el constructor
// del contrato derivado (ver más abajo)).
contract PriceFeed is owned, mortal, named("GoldFeed") {
   function updateInfo(uint newInfo) {
      if (msg.sender == owner) info = newInfo;
   }

   function get() constant returns(uint r) { return info; }

   uint info;
}
```

Nótese que arriba llamamos a `mortal.kill()` para "reenviar" la orden de destrucción. Hacerlo de esta forma es problemático, como se puede ver en el siguiente ejemplo.

```
pragma solidity ^0.4.0;

contract mortal is owned {
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
    }
}
```

```
contract Base1 is mortal {
    function kill() { /* hacer limpieza 1 */ mortal.kill(); }
}


contract Base2 is mortal {
    function kill() { /* hacer limpieza 2 */ mortal.kill(); }
}


contract Final is Base1, Base2 {
}
```

Una llamada a `Final.kill()` llamará a `Base2.kill` al ser la última sobreescritura, pero esta función obviará `Base1.kill`, básicamente porque ni siquiera sabe de la existencia de `Base1`. La forma de solucionar esto es usando `super`.

```
pragma solidity ^0.4.0;

contract mortal is owned {
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
    }
}


contract Base1 is mortal {
    function kill() { /* hacer limpieza 1 */ super.kill(); }
}


contract Base2 is mortal {
    function kill() { /* hacer limpieza 2 */ super.kill(); }
}


contract Final is Base2, Base1 {
}
```

If `Base1` calls a function of `super`, it does not simply call this function on one of its base contracts. Rather, it calls this function on the next base contract in the final inheritance graph, so it will call `Base2.kill()` (note that the final inheritance sequence is – starting with the most derived contract: Final, Base1, Base2, mortal, owned). The actual function that is called when using super is not known in the context of the class where it is used, although its type is known. This is similar for ordinary virtual method lookup.

Es similar a la búsqueda de métodos virtual

Si `Base1` llama a una función de `super`, no simplemente llama a esta función en uno de sus contratos base. En su lugar, llama a esta función en el siguiente contrato base en el último grafo de herencias, por lo tanto llama a `Base2.kill()` (nótese que la secuencia final de herencia es – empezando por el contrato más derivado: Final, Base1, Base2, mortal, owned). La función a la que se llama cuando se usa super no se sabe en el contexto de la clase donde se usa, aunque su tipo es conocido. Es similar a la búsqueda de métodos virtuales.

Argumentos para constructores base

Se requiere que los contratos derivados proporcionen todos los argumentos necesarios para los constructores base. Esto se puede hacer de dos maneras.

```
pragma solidity ^0.4.0;

contract Base {
    uint x;
    function Base(uint _x) { x = _x; }
}


contract Derived is Base(7) {
    function Derived(uint _y) Base(_y * _y) {
    }
}
```

Una es directamente en la lista de herencias (`is Base(7)`). La otra es en la misma línea en que un modificador se invoca como parte de la cabecera de un constructor derivado (`Base(_y * _y)`). La primera manera es más conveniente si el argumento del constructor es una constante y define el comportamiento del contrato o por lo menos lo describe. La segunda manera se tiene que usar si los argumentos del constructor de la base dependen de los argumentos del contrato derivado. Si, como en este ejemplo sencillo, ambos sitios están utilizados, el argumento estilo modificador tiene la prioridad.

Herencia múltiple y linearización

Los lenguajes que permiten herencias múltiples tienen que lidiar con varios problemas. Uno es el Problema del diamante. Solidity le sigue la pista a Python y utiliza la "Linearización C3" para forzar un orden específico en el DAG de las clases base. Esto hace que se consigua la propiedad deseada de monotonicidad pero impide algunos grafos de herencia. El orden en el que las clases base se van dando con la instrucción `is` es especialmente importante. En el siguiente código, Solidity dará el error "Linearization of inheritance graph impossible".

```
pragma solidity ^0.4.0;

contract X {}
contract A is X {}
contract C is A, X {}
```

El motivo de que se produzca este error es que `C` solicita a `X` que sobreescriba `A` (especificando `A`, `X` en este orden), pero el propio `A` solicita sobreescribir `X`, lo que presenta una contradicción que no puede resolverse.

Una regla simple para recordar es especificar las clases base en el orden desde "la más base" hasta "la más derivada".

Heredar distintos tipos de miembros con el mismo nombre

Cuando la herencia termina en un contrato con una función y un modificador con el mismo nombre, se considera esta herencia un error. Este error también se produciría en el caso en que un evento y un modificador tuvieran el mismo nombre, así como con una función y un evento con el mismo nombre. Como excepción, una variable de estado getter puede sobre escribir una función pública.

Contratos abstractos

Las funciones de un contrato pueden carecer de una implementación como pasa en el siguiente ejemplo (nótese que la cabecera de declaración de la función se termina con un `;`).

```
pragma solidity ^0.4.0;

contract Feline {
    function utterance() returns (bytes32);
}
```

Estos contratos no pueden compilarse (aunque contengan funciones implementadas junto con funciones no implementadas), pero pueden usarse como contratos base.

```
pragma solidity ^0.4.0;

contract Cat is Feline {
    function utterance() returns (bytes32) { return "miaow"; }
}
```

Si un contrato hereda de un contrato abstracto y éste no implementa todas las funciones no implementadas con sobrescritura, él mismo será un contrato abstracto.

Interfaces

Las interfaces son similares a los contratos abstractos, pero no pueden tener ninguna función implementada. Y hay más restricciones:

1. No pueden heredar otros contratos o interfaces.
2. No pueden definir contructores.
3. No pueden definir variables.
4. No pueden definir structs.
5. No pueden definir enums.

Es posible que en el futuro, algunas de estas restricciones se levanten.

Las interfaces son limitadas a lo que básicamente el contrato ABI puede representar, y la conversion entre el ABI y la interfaz debería hacerse sin perdida de información.

Las interfaces se indican por su propia palabra clave:

```
interface Token {
    function transfer(address recipient, uint amount);
}
```

Los contratos pueden heredar interfaces como lo heredarían otros contratos.

Librerías

Las librerías son similares a los contratos, pero su propósito es que se desplieguen una sola vez a una dirección especifica y su código se pueda reutilizar utilizando la característica `DELEGATECALL` (`CALLCODE` hasta Homestead) de la EVM. Lo que significa que si se llama a las funciones de una librería, su código es ejecutado en el contexto del contrato que llama, es decir, `this` apunta al contrato que llama y en especial, se puede acceder al almacenamiento del contrato que llama. Como una librería es un trozo de código fuente aislado, una librería sólo puede acceder a las

variables de estado de un contrato emisor si estas variables están específicamente proporcionadas (de lo contrario, no tendría la posibilidad de nombrarlas).

Las librería pueden considerarse como contratos base implícitos del contrato que las usa. Las librerías no son explícitamente visibles en la jerarquía de herencia, pero las llamadas a las funciones de una librería se parecen completamente a las llamadas a funciones de contratos base explícitos (`L.f()` si `L` es el nombre de la librería). Además, las funciones `internal` de las librerías son visibles en todos los contratos, como si la librería fuera un contrato base. Por supuesto las llamadas a funciones internas utilizan las normas de llamadas internas, lo que significa que todos los tipos internos pueden ser enviados y que los tipos de memoria serán enviados mediante referencia y no copiados. Para realizar esta operación en la EVM, se incluirá en el contrato emisor el código de las funciones internas de la librería y todas las funciones llamadas desde dentro usando el comando habitual `JUMP` en lugar del `DELEGATECALL`.

El siguiente ejemplo ilustra cómo usar las librerías (pero asegúrese de leer *using for* para tener un ejemplo más avanzado de cómo implementar un set):

```
pragma solidity ^0.4.11;

library Set {
  // Definimos un nuevo tipo de datos para un struct que se va a utilizar para
  // conservar sus datos en el contrato que efectúa la llamanda.
  struct Data { mapping(uint => bool) flags; }

  // Nótese que el primer parametro es del tipo "referencia de almacenamiento",
  // por lo tanto solamente su dirección de almacenamiento y no su contenido se
  // envía como parte de la llamada. Esto es una característica especial de las
  // funciones de librerías. Es idiomático llamar al primer parámetro 'self', si
  // la función puede verse como un método de este objeto.
  function insert(Data storage self, uint value)
      returns (bool)
  {
      if (self.flags[value])
          return false; // ya está
      self.flags[value] = false;
      return true;
  }

  function contains(Data storage self, uint value)
      returns (bool)
  {
      return self.flags[value];
  }
}


contract C {
    Set.Data knownValues;

    function register(uint value) {
        // Las funciones de librerías pueden llamarse sin una instancia
        // específica de la librería, ya que la "instancia" es el contrato actual.
        require(Set.insert(knownValues, value));
    }
    // En este contrato, si se quiere, también se puede acceder directamente a␣
↪knownValues.flags.
}
```

No es por supuesto obligatorio usar las librerías de esta manera - también pueden usarse sin definir tipos de datos struct. Las funciones también funcionan sin parámetros de referencia de almacenamiento, y pueden tener multiples

parámetros de referencia de almacenamiento y en cualquier posición.

Las llamadas a `Set.contains`, `Set.insert` y `Set.remove` son compiladas como llamadas (`DELEGATECALL`) en un contrato/librería externa. Si se usan librerías, hay que asegurarse de que de verdad se realiza una llamada a una función externa. `msg.sender`, `msg.value` y `this` conservarán sus valores en esta llamada, aunque hasta Homestead, por el uso de *CALLCODE*, `msg.sender` y `msg.value` cambiaban.

En el siguiente ejemplo se muestra cómo usar tipos de memoria y funciones internas en las librerías para implementar tipos a medida sin la necesidad de usar llamadas a funciones externas:

```
pragma solidity ^0.4.0;

library BigInt {
    struct bigint {
        uint[] limbs;
    }

    function fromUint(uint x) internal returns (bigint r) {
        r.limbs = new uint[](1);
        r.limbs[0] = x;
    }

    function add(bigint _a, bigint _b) internal returns (bigint r) {
        r.limbs = new uint[](max(_a.limbs.length, _b.limbs.length));
        uint carry = 0;
        for (uint i = 0; i < r.limbs.length; ++i) {
            uint a = limb(_a, i);
            uint b = limb(_b, i);
            r.limbs[i] = a + b + carry;
            if (a + b < a || (a + b == uint(-1) && carry > 0))
                carry = 1;
            else
                carry = 0;
        }
        if (carry > 0) {
            // ¡Qué mal! Tenemos que añadir un "limb"
            uint[] memory newLimbs = new uint[](r.limbs.length + 1);
            for (i = 0; i < r.limbs.length; ++i)
                newLimbs[i] = r.limbs[i];
            newLimbs[i] = carry;
            r.limbs = newLimbs;
        }
    }

    function limb(bigint _a, uint _limb) internal returns (uint) {
        return _limb < _a.limbs.length ? _a.limbs[_limb] : 0;
    }

    function max(uint a, uint b) private returns (uint) {
        return a > b ? a : b;
    }
}


contract C {
    using BigInt for BigInt.bigint;

    function f() {
        var x = BigInt.fromUint(7);
```

```
        var y = BigInt.fromUint(uint(-1));
        var z = x.add(y);
    }
}
```

Puesto que el compilador no puede saber en qué dirección será desplegada la librería, estas direcciones deben ser insertadas en el bytecode final por un *linker* (véase *Utilizar el compilador de línea de comandos* para saber cómo usar el compilador de lineas de comando para establecer vínculos). Si las direcciones no están facilitadas como argumentos al compilador, el código hex compilado contendrá marcadores de posición de la forma `__Set_____` (donde `Set` es el nombre de la librería). La dirección puede ser facilitada manualmente remplazando cada uno de estos 40 símbolos por la codificación hexadecimal de la dirección del contrato de la librería.

Las restricciones para las librerías con respecto a las restricciones para los contratos son las siguientes:

- No hay variables de estado
- No puede heredar ni ser heredadas
- No pueden recibir Ether

(Puede que estas restricciones se levanten en un futuro.)

Using For

La directiva `using A for B;` se puede usar para adjuntar funciones de librería (desde la librería `A`) a cualquier tipo (`B`). Estas funciones recibirán el objeto con el que se les llama como su primer parámetro (igual que con el parámetro `self` en Python).

El efecto que tiene esta directiva `using A for *;` es que las funciones de la librería `A` se adjunten a cualquier tipo.

En ambas situaciones, todas las funciones se adjuntan, incluso las funciones donde el tipo del primer parámetro no coincide con el tipo del objeto. El tipo se comprueba en el punto en que se llama a la función y se resuelven problemas de sobrecarga de la función.

Con el alcance actual, que por ahora está limitado a un contrato, la directiva `using A for B;` está activa. Más adelante tendrá un alcance global, lo que hará que cuando se incluya un módulo, sus tipos de datos, incluyendo las funciones de librería, estarán disponibles sin tener que añadir más código.

Volvamos a escribir el ejemplo de Set del apartado *librerías* de la siguiente manera:

```
pragma solidity ^0.4.11;

// Es el mismo código que antes pero sin los comentarios
library Set {
  struct Data { mapping(uint => bool) flags; }

  function insert(Data storage self, uint value)
      returns (bool)
  {
      if (self.flags[value])
        return false; // está
      self.flags[value] = true;
      return true;
  }

  function remove(Data storage self, uint value)
      returns (bool)
  {
      if (!self.flags[value])
```

```
            return false; // no está
        self.flags[value] = false;
        return true;
    }

    function contains(Data storage self, uint value)
        returns (bool)
    {
        return self.flags[value];
    }
}


contract C {
    using Set for Set.Data; // este es el cambio importante
    Set.Data knownValues;

    function register(uint value) {
        // Aquí, cada una de las variables con el tipo Set.Data tiene una función␣
↪miembro correspondiente.
        // La siguiente llamada es idéntica a Set.insert(knownValues, value)
        require(knownValues.insert(value));
    }
}
```

También es posible extender los tipos elementales de la siguiente manera:

```
pragma solidity ^0.4.0;

library Search {
    function indexOf(uint[] storage self, uint value) returns (uint) {
        for (uint i = 0; i < self.length; i++)
            if (self[i] == value) return i;
        return uint(-1);
    }
}


contract C {
    using Search for uint[];
    uint[] data;

    function append(uint value) {
        data.push(value);
    }

    function replace(uint _old, uint _new) {
        // Esto es lo que realiza la llamada a la función de la librería
        uint index = data.indexOf(_old);
        if (index == uint(-1))
            data.push(_new);
        else
            data[index] = _new;
    }
}
```

Nótese que cualquier llamada a una librería es en realidad una llamada a una función de la EVM. Esto significa que si se envían tipos de memoria o de valor, se va a realizar una copia, incluso de la variable `self`. La única situación en la

que no se va a realizar una copia es cuando se utilizan variables que hacen referencia al almacenamiento.

Ensamblador de Solidity

Solidity define un lenguaje ensamblador que también puede ser usado sin Solidity. Este lenguaje ensamblador también se puede usar como "ensamblador inline" dentro del código fuente de Solidity. Vamos a comenzar explicando cómo usar el ensamblador inline y sus diferencias con el ensamblador independiente, y luego especificaremos el ensamblador propiamente dicho.

PENDIENTE DE HACER: Escribir sobre de qué manera el ámbito del ensamblador inline es un poco diferente y las complicaciones que aparecen cuando, por ejemplo, se usan funciones internas o librerías. Además, escribir sobre los símbolos definidos por el compilador.

Ensamblador inline

Para tener un control más fino, especialmente para mejorar el lenguaje excribiendo librerías, es posible intercalar las declaraciones hechas en Solidity con el ensamblador inline en un lenguaje cercano al lenguaje de la máquina virtual. Debido a que el EVM (la máquina virtual de Ethereum) es un máquina de tipo pila (stack machine), suele ser difícil dirigirse a la posición correcta de la pila y proporcinar los argumentos a los opcodes en el punto correcto en la pila. El ensamblador inline de Solidity intenta facilitar esto, y otras complicaciones que ocurren cuando se escribe el ensamblador de forma manual, con las siguientes funcinalidades:

- opcodes de estilo funcional: `mul(1, add(2, 3))` en lugar de `push1 3 push1 2 add push1 1 mul`
- variables de ensamblador local: `let x := add(2, 3) let y := mload(0x40) x := add(x, y)`
- acceso a variables externas: `function f(uint x) { assembly { x := sub(x, 1) } }`
- etiquetas: `let x := 10 repeat: x := sub(x, 1) jumpi(repeat, eq(x, 0))`
- bucles: `for { let i := 0 } lt(i, x) { i := add(i, 1) } { y := mul(2, y) }`
- declaraciones de intercambio: `switch x case 0 { y := mul(x, 2) } default { y := 0 }`
- llamadas a funciones: `function f(x) -> y { switch x case 0 { y := 1 } default { y := mul(x, f(sub(x, 1))) } }`

Ahora queremos decribir el lenguaje del ensamblador inline en detalles.

Advertencia: El ensamblador inline es una forma de acceder a bajo nivel a la Máquina Virtual de Ethereum. Esto ignora varios elementos de seguridad de Solidity.

Ejemplo

El siguiente ejemplo proporciona el código de librería que permite acceder al código de otro contrato y cargarlo en una variable `bytes`. Esto no es para nada factible con "Solidity puro". La idea es que se usen librerías de ensamblador para aumentar las capacidades del lenguaje en ese sentido.

```
pragma solidity ^0.4.0;

library GetCode {
    function at(address _addr) returns (bytes o_code) {
        assembly {
```

```
            // recupera el tamaño del código - esto necesita ensamblador
            let size := extcodesize(_addr)
            // asigna (output byte array) - esto se podría hacer también sin
↪ensamblador
            // usando o_code = new bytes(size)
            o_code := mload(0x40)
            // nuevo "fin de memoria" incluyendo el relleno (padding)
            mstore(0x40, add(o_code, and(add(add(size, 0x20), 0x1f), not(0x1f))))
            // almacenar el tamaño en memoria
            mstore(o_code, size)
            // recuperar de verdad el código - esto necesita ensamblador
            extcodecopy(_addr, add(o_code, 0x20), 0, size)
        }
    }
}
```

El ensamblador inline también es útil es los casos en los que el optimizador falla en producir un código eficiente. Tenga en cuenta que es mucho más difíciil escribir el ensamblador porque el compilador no realiza controles, por lo tanto use ensamblador solamente para cosas complejas y solo si sabe lo que está haciendo.

```
pragma solidity ^0.4.0;

library VectorSum {
    // Esta función es menos eficiente porque el optimizador falla en quitar los
↪controles de límite en el acceso al array.
    function sumSolidity(uint[] _data) returns (uint o_sum) {
        for (uint i = 0; i < _data.length; ++i)
            o_sum += _data[i];
    }

    // Sabemos que solamente accedemos al array dentro de los límites, así que
↪podemos evitar los controles.
    // Se tiene que añadir 0x20 a un array porque la primera posición contiene el
↪tamaño del array.
    function sumAsm(uint[] _data) returns (uint o_sum) {
        for (uint i = 0; i < _data.length; ++i) {
            assembly {
                o_sum := mload(add(add(_data, 0x20), mul(i, 0x20)))
            }
        }
    }
}
```

Síntaxis

El ensamblador analiza comentarios, literales e identificadores de igual manera que en Solidity, así que se puede usar los comentarios habituales: `//` y `/* */`. El ensamblador inline está señalado por `assembly { ... }` y dentro de estos corchetes se pueden usar los siguentes elementos (véase las secciones más abajo para más detalles):

- literales, es decir `0x123`, `42` o `.ªbc"` (strings de hasta 32 carácteres)
- opcodes (en “estilo instruccional”), p.ej. `mload sload dup1 sstore`, véase más abajo para tener una lista
- opcodes en estilo fucional, e.g. `add(1, mlod(0))`
- etiquetas, p.ej. `name:`
- declaraciones de variable, p.ej. `let x := 7` o `let x := add(y, 3)`

- identificadores (etiquetas o variables de ensamblador local y externos si se usa como ensamblador inline), p.ej. `jump(name)`, `3 x add`
- tareas (en "estilo instruccional"), e.g. `3 =: x`
- tareas en estilo fucional, p.ej. `x := add(y, 3)`
- bloques donde las variables locales están incluidas dentro, p.ej. `{ let x := 3 { let y := add(x, 1) } }`

Opcodes

Este documento no preten de ser una descripción exhaustiva de la máquina virtual de Ethereum, pero la lista siguiente puede servir de referencia para sus opcodes.

Si un opcode recibe un argumento (siempre desde lo más alto de la pila), se ponen entre paréntesis. Note que el orden de los argumentos puede verse invertido en estilo no funcional (se explica más abajo). Opcodes marcados con un − no empuja un elemento encima de la pila, los marcados con $\star$ son especiales, y todos los demás empujan exactamente un elemento encima de la pila.

En el ejemplo `mem[a...b)`, se indica los bytes de memoria empezando en la posición `a` hasta la posición (excluida) `b` y en el ejemplo `storage[p]`, se indica los índices de almacenamiento en la posicón `p`.

Los opcodes `pushi` y `jumpdest` no se pueden usar directamente.

En la gramática, los opcodes se representan como identificadores predefinidos.

stop	-	parar ejecución, idéntico a return(0,0)
add(x, y)		x + y
sub(x, y)		x - y
mul(x, y)		x * y
div(x, y)		x / y
sdiv(x, y)		x / y, para números con signo en complemento de dos
mod(x, y)		x % y
smod(x, y)		x % y, para números con signo en complemento de dos
exp(x, y)		x elevado a y
not(x)		~x, cada bit de x está negado
lt(x, y)		1 si x < y, 0 de lo contrario
gt(x, y)		1 si x > y, 0 de lo contrario
slt(x, y)		1 si x < y, 0 de lo contrario, para números con signo en complemento de dos
sgt(x, y)		1 si x > y, 0 de lo contrario, para números con signo en complemento de dos
eq(x, y)		1 si x == y, 0 de lo contrario
iszero(x)		1 si x == 0, 0 de lo contrario
and(x, y)		bitwise and de x e y
or(x, y)		bitwise or of x and y
xor(x, y)		bitwise xor of x and y
byte(n, x)		n byte de x, donde el más significante byte es el byte 0
addmod(x, y, m)		(x + y) % m con una precisión aritmética arbitraria
mulmod(x, y, m)		(x * y) % m con una precisión aritmética arbitraria
signextend(i, x)		el signo se extiende desde el bit (i*8+7) contando desde el menos significante
keccak256(p, n)		keccak(mem[p...(p+n)))
sha3(p, n)		keccak(mem[p...(p+n)))

Tabla 6.1 – provie

jump(label)	-	saltar a la posición de label / código
jumpi(label, cond)	-	saltar a label si cond no es cero
pc		posición actual en el código
pop(x)	-	quitar el elemento empujado por x
dup1 ... dup16		copiar posición i de la pila en la posición de arriba (contando desde arriba)
swap1 ... swap16	*	intercambiar la posición la más arriba con la posición i de la pila justo debajo
mload(p)		mem[p..(p+32))
mstore(p, v)	-	mem[p..(p+32)) := v
mstore8(p, v)	-	mem[p] := v & 0xff - sólo modifica un único byte
sload(p)		storage[p]
sstore(p, v)	-	storage[p] := v
msize		tamaño de la memoria , es decir el índice más grande de la memoria que ha sido acced
gas		el gas todavía disponible para ejecución
address		dirección del contrato actual / contexto de ejecución
balance(a)		balance en wei de la dirección a
caller		llamar el remitente (excluyendo delegatecall)
callvalue		wei que se enviaron junto con la llamada actual
calldataload(p)		llamar datos empezando por la posición p (32 bytes)
calldatasize		tamaño de la llamada a datos en bytes
calldatacopy(t, f, s)	-	copiar s bytes de la llamada a datos en la posición f a la memoria en la posición t
codesize		tamaño del código de contrato actual / contexto de ejecución
codecopy(t, f, s)	-	copiar s bytes del código en la posición f a la memoria en la posición t
extcodesize(a)		tamaño del código en la dirección a
extcodecopy(a, t, f, s)	-	igual que codecopy(t, f, s) pero tomando el código de la dirección a
returndatasize		tamaño del último returndata
returndatacopy(t, f, s)	-	copiar s bytes de returndata de la posición f a la memoria en la posición t
create(v, p, s)		crear un nuevo contrato con el código mem[p..(p+s)) y mandar v wei y devolver la nue
create2(v, n, p, s)		crear un nuevo contrato con el código mem[p..(p+s)) en la dirección keccak256(<addr
call(g, a, v, in, insize, out, outsize)		llamar el contrato a la dirección a con la entrada mem[in..(in+insize)) proporcionando
callcode(g, a, v, in, insize, out, outsize)		indéntico a *call* pero usando solo el código de a y si no, quedarse en el contexto del co
delegatecall(g, a, in, insize, out, outsize)		indéntico a *callcode* pero mantener también `caller` y `callvalue`
staticcall(g, a, in, insize, out, outsize)		idéntico a *call(g, a, 0, in, insize, out, outsize)* pero no admite modificaciones de estado
return(p, s)	-	termina la ejecución, devuelve los datos de mem[p..(p+s))
revert(p, s)	-	termina la ejecución, revierte los cambios de estado, devuelve los datos de mem[p..(p+
selfdestruct(a)	-	termina la ejecución, destruye el contrato actual y manda los fondos a a
invalid	-	termina la ejecución con una instrucción no válida
log0(p, s)	-	log sin tópicos y datos mem[p..(p+s))
log1(p, s, t1)	-	log sin tópicos t1 y datos mem[p..(p+s))
log2(p, s, t1, t2)	-	log sin tópicos t1, t2 y datos mem[p..(p+s))
log3(p, s, t1, t2, t3)	-	log sin tópicos t1, t2, t3 y datos mem[p..(p+s))
log4(p, s, t1, t2, t3, t4)	-	log sin tópicos t1, t2, t3, t4 y datos mem[p..(p+s))
origin		remitente de la transacción
gasprice		precio del gas price de la transacción
blockhash(b)		hash del bloque número b - solamente para los últimos 256 bloques, exluyendo al bloq
coinbase		el beneficiario actual del minado
timestamp		timestamp en segundos del bloque actual desde epoch
number		número del bloque actual
difficulty		dificultad del bloque actual
gaslimit		límite de gas del bloque para el bloque actual

Literales

Se pueden usar constantes íntegras usando la notación decimal o hexadecimal y con eso, una instrucción apropiada `PUSHi` sereá automáticamente generada. El siguiente código suma 2 con 3, lo que resulta en 5 y luego computa el bitwise con el string "abc". Los strings están almacenados alineados a la izquierda y no pueden ser más largos que 32 bytes.

```
assembly { 2 3 add "abc" and }
```

Estilo funcional

Se puede entrar opcodes uno trás el otro de la misma manera que van aparecer en el bytecode. Por ejemplo sumar "3" al contenido de la memoria en la posición `0x80` sería:

```
3 0x80 mload add 0x80 mstore
```

Como suele ser complicado ver cuales son los argumentos actuales para algunos de los opcodes, el ensamblador inline de Solidity proporciona también una notación de "estilo fucinonal" donde el mismo código se escribiría de la siguiente manera:

```
mstore(0x80, add(mload(0x80), 3))
```

Expresiones en estilo funcional no pueden hacer uso internamente del estilo instruccional, es decir que `1 2 mstore(0x80, add)` no es ensamblador válido, debería escribirse como `mstore(0x80, add(2, 1))`. Para los opcodes que no toman argumentos, las parentésis pueden obviarse.

Nótese que el orden de los argumentos está invertido en estilo funcional con respecto al estilo instruccional. Si hace uso del estilo funcional, el primer argumento aparecerá arriba de la pila.

Acceso a variables y funciones internas

Se accede a las variables y otros identificadores de Solidity simplemente por su nombre. Para las variables de memoria, esto tiene como consecuencia que es la dirección y no el nombre que se empuja en la pila. Con las variables de almacenamiento es diferente: los valores en almacenamiento podrían no ocupar una posición completa en la pila, de tal manera que su dirección estaría compuesta por una posición y un decalage en bytes dentro de esta posición. Para recuperar la posición a la que apunta la variable `x`, se usa `x_slot` y para recuperar el decalage en bytes se usa `x_offset`.

En las asignaciones (ver abajo), hasta se pueden usar las variables locales de Solidity y asignarlas.

También se puede acceder a las funciones externas al ensamblador inline: el ensamblador empujará su etiqueta de entrada (aplicando la resolución de funciones virtuales). Las semánticas de llamada en Solidity son las siguientes:

- el que llama empuja return etiqueta, arg1, arg2, ..., argn
- la llamada devuelve ret1, ret2, ..., retm

Esta funcionalidad está todavía un poco dificultosa de usar, esensialmente porque el decalage de pila cambia durante la llamada, y por lo tanto las referencias a las variables locales estarán mal.

```
pragma solidity ^0.4.11;

contract C {
    uint b;
    function f(uint x) returns (uint r) {
```

```
        assembly {
            r := mul(x, sload(b_slot)) // ignorar el decalage, sabemos que es cero
        }
    }
}
```

Etiquetas

Otro de los problemas en el ensamblador del EVM es que `jump` y `jumpi` hacen uso de direcciones absolutas que pueden facilmente cambiar. El ensamblador inline de Solidity proporciona etiquetas para hacer el uso de saltos más cómodo. Nótese que las etiquetas son una funcionalidad de bajo nivel y que es perfectamente posible escribir un ensamblador eficicente sin etiquetas, usando solo funciones de ensamblador, bucles e instrucciones de intercambio (ver abajo). El siguiente código computa un elemento en una serie de Fibonacci.

```
{
    let n := calldataload(4)
    let a := 1
    let b := a
loop:
    jumpi(loopend, eq(n, 0))
    a add swap1
    n := sub(n, 1)
    jump(loop)
loopend:
    mstore(0, a)
    return(0, 0x20)
}
```

Tenga en cuenta que acceder automáticamente a variables de pila sólo funciona si el ensamblador conoce la altura de la pila actual. Esto falla si el inicio y el destino del salto tienen alturas de pila distintas. Aún así se pueden usar estos saltos, pero no debería entonces acceder a ninguna variables de pila (incluso variables de ensamblador)

Además, el analizador de la altura de la pila lee el código opcode por opcode (y no acorde al control de flujo), por lo tanto y según indica al ejemplo que figura abajo, el ensamblador tendría una falsa idea de la altura de la pila al llegar a la etiqueta `two`:

```
{
    let x := 8
    jump(two)
    one:
        // Aquí la altura de la pila es de 2 (porque empujamos x y 7), pero el␣
↪ensamblador cree que la altura es de 1 porque lee desde arriba abajo.
        // Acceder a la variable de pila x aquí nos llevaría a un error.
        x := 9
        jump(three)
    two:
        7 // empujar algo arriba de la pila
        jump(one)
    three:
}
```

Este problema puede resolverse manualmente ajustando la altura de la pila en lugar de que lo haga el ensamblador - puede proporcionar un delta de altura de pila que se suma a la altura de la pila justo antes de la etiqueta. Nótese que no va a tener que preocuparse por estas cosas si sólo usa bucles y funciones de nivel ensamblador.

Para ilustrar cómo esto se puede hacer en casos extremos, véase el ejemplo siguiente:

```
{
    let x := 8
    jump(two)
    0 // Este código no se puede acceder pero se ajustará correctamente la altura de
↪la pila
    one:
        x := 9 // Ahora se puede acceder correctamente a x
        jump(three)
        pop // Corrección negativa similar
    two:
        7 // Empujar algo arriba de la pila
        jump(one)
    three:
    pop // Tenemos que hacer pop con el valor empujado manualmente aqui otra vez
}
```

Declarando variables de ensamblador local

Puede usar la palabra clave `let` para declarar variables que están visibles solamente en ensamblador inline y en realidad solamente en el bloque actual `{...}`. Lo que pasa es que la instrucción `let` crea una nueva posición en la pila que está reservada para la variable. Esta posición se quitará automaticamente cuando se llegue al final del bloque. Es necesario proporcionar un valor inicial para la variable, que puede ser simplemente `0`, pero también puede ser una expresión compleja en el estilo funcional.

```
pragma solidity ^0.4.0;

contract C {
    function f(uint x) returns (uint b) {
        assembly {
            let v := add(x, 1)
            mstore(0x80, v)
            {
                let y := add(sload(v), 1)
                b := y
            } // aquí se "desasigna" y
            b := add(b, v)
        } // aquí se "desasigna" v
    }
}
```

Asignaciones

Las asignaciones son posibles para las variables de ensamblador local y para las variables de función local. Tenga cuidado que cuando usted asigna a variables que apuntan a la memoria o al almacenamiento, usted sólo cambiará lo que apunta pero no los datos.

Existen asignaciones de dos tipos: las de estilo funcional y las de estilo instruccional. Para las asignaciones de estilo funcional, (`variable := value`), se requiere proporcionarun valor dentro de una expresión de estilo funcional que resulta en exactamente un valor de pila. Para las asignaciones de estilo instruccional (`=: variable`), el valor simplemente se toma desde arriba de la pila. Para ambas maneras, la coma apunta al nombre de la variable. La asignación se ejecuta remplazando el valor de la variable en la pila por el valor nuevo.

```
assembly {
    let v := 0 // asignación de estilo funcional como parte de la declaración de
↪variable
```

```
    let g := add(v, 2)
    sload(10)
    =: v // asignación de estilo instruccional, pone el resultado de sload(10) en v
}
```

Intercambio

Se puede usar una declaración de intercambio como una versión muy básica de un "if/else". Toma el valor de una expresión y lo compara con distintas constantes. Se elige la rama correspondiente a la constante que combina. A contrario de algunos de los lenguages de programación que son propenses a errores de comportamiento, el flujo de control, después de un caso, no pasa al siguiente. Puede haber un fallback o un caso por defecto llamado `default`.

```
assembly {
    let x := 0
    switch calldataload(4)
    case 0 {
        x := calldataload(0x24)
    }
    default {
        x := calldataload(0x44)
    }
    sstore(0, div(x, 2))
}
```

Una lista de casos no necesita llaves, pero el cuerpo de un caso si.

Bucles

El ensamblador soporta un bucle simple de tipo for. Los bucles de tipo for contienen un encabezado con la inicialización, una condición y una parte post interación. La condición debe ser una expresión de estilo funcional, mientras que las otras dos partes son bloques. Si se declaran variables en la parte de inicialización, el alcance de estas variables se extenderá hasta el cuerpo (incluyendo la condición y la parte post iteración).

El ejemplo siguiente computa la suma de un área en la memoria.

```
assembly {
    let x := 0
    for { let i := 0 } lt(i, 0x100) { i := add(i, 0x20) } {
        x := add(x, mload(i))
    }
}
```

Funciones

El ensamblador permite la definición de funciones de bajo nivel. Éstas toman sus argumentos (y un return PC) desde la pila y también ponen los resultados arriba de la pila. Llamar una función se parece a la ejecución de un opcode de estilo funcional.

Las funciones pueden definirse en cualquier lugar y son visibles en el bloque en el que se han declarado. Dentro de una función, no se permite acceder a variables locales definidas fuera de esta función. No existe una declaración `return` explícita.

Si se llama una función que devuelve múltiples valores, es obligatorio asignarlos a un tuple usando `a, b := f(x)` o `let a, b := f(x)`.

El siguiente ejemplo implementa la función de potencia con cuadrados y multiplicaciones.

```
assembly {
    function power(base, exponent) -> result {
        switch exponent
        case 0 { result := 1 }
        case 1 { result := base }
        default {
            result := power(mul(base, base), div(exponent, 2))
            switch mod(exponent, 2)
                case 1 { result := mul(base, result) }
        }
    }
}
```

Cosas a evitar

Aunque el ensamblador inline puede dar la sensación de tener un aspecto de alto nivel, es en realidad de nivel extremadamente bajo. Se convierten las llamadas a funciones, los bucles y los interruptores con simples reglas de reescritura y luego, lo único que el ensamblador hace para el usuario es reorganizar los opcodes en estilo funcional, manejando etiquetas de salto, contando la altura de la pila para el acceso a variables y quitando posiciones en la pila para variables locales del ensamblador cuando se alcanza el final de su bloque. Especialmente en estos dos últimos casos, es importante saber que el ensamblador solo cuenta la altura de la pila desde arriba abajo, y no necesariamente siguiendo el flujo de control. Además, las operaciones como los intercambios solo van a intercambiar los contenidos de la pila pero no la ubicación de las variables.

Convenciones en Solidity

A cambio del ensamblador EVM, Solidity conoce tipos que son más estrechos que 256 bits, por ejemplo `uint24`. Para hacerlos más eficientes, la mayoría de las operaciones aritméticas los tratan como números de 256 bits y los bits de mayor orden sólo se limpian cuando es necesario, es decir sólo un poco antes de almacenarse en memoria o antes de realizar comparaciones. Lo que significa que si se accede dicha variable desde dentro del ensamblador inline, puede que se tenga primero que limpiar los bits de mayor orden.

Solidity maneja la memoria de una manera muy simple: hay un "cursor de memoria disponible" en la posición `0x40` de la memoria. Si se desea asignar memoria, tan solo se tiene que usar la memoria a partir de este punto y actualizar el cursor en consecuencia.

En Solidity, los elementos en arrays de memoria siempre ocupan múltiples de 32 bytes (y si, esto también es cierto para `byte[]`, pero no para `bytes` y `string`). Arrays multidimensionales son cursores de arrays de memoria. La longitud de un array dinámico se almacena en la primera posición del array y luego sólo le siguen elementos del array.

Advertencia: Arrays de memoria estáticos en tamaño no tienen un campo para la longitud, pero se añadirá pronto para permitir una mejor convertibilidad entre arrays de tamaño estático y dinñamico, pero es importante no contar con esto por ahora.

Ensamblador independiente

El lenguage ensamblador que hemos descrito más arriba como ensamblador inline también puede usarse de forma independiente. De hecho, el plan es de usarlo como un lenguage intermedio para el compilador de Solidity. De esta forma, intenta cumplir con varios objetivos:

1. Los programas escritos en el lenguage ensamblador deben de ser legibles, aunque el código sea generado por un compilador de Solidity.
2. La traducción del lenguage ensamblador al bytecode deben de contener el número de sorpresas el más reducido posible.
3. El control de flujo debe de ser fácil de detectar para ayudar a la verificación formal y a la optimización.

Para cumplir con el primero y el útlimo de los objetivos, el ensamblador proporciona constructs de alto nivel como bucles `for`, declaraciones `switch` y llamadas a funciones. Debería de ser posible de escribir programas de ensamblador que no hacen uso de declaraciones explícitas de tipo `SWAP`, `DUP`, `JUMP` y `JUMPI`, porque las dos primeras declaraciones ofuscan el flujo de datos y las dos últimas ofuscan el control de flujo. Además, hay que privilegiar declaraciones funcionales del tipo `mul(add(x, y), 7)` a las declaraciones de opcodes puras como `7 y x add mul` porque en la primera, es mucho más fácil ver qué operando se usa para qué opcode.

El segundo objetivo se cumple introduciendo una fase de desazucarización que sólo quita los constructs de más alto nivel de una forma muy regular pero permitiendo todavía la inspección el código ensamblador de bajo nivel generado. La única operación no local realizada por el ensamblador es la búsqueda de nombre de identificadores (funciones, variables, ...) definidos por el usuario, lo que se hace siguiendo reglas con un alcance muy simple y regular y con un proceso de limpieza de variables locales desde la pila.

Alcance: Un identificador que está declarado (etiqueta, variable, función, ensamblador) sólo es visible en el bloque donde ha sido declarado (incluyendo bloques anidados dentro del bloque actual). No es legal acceder variables locales cruzando los límites de la función, aunque estas variables estuvieran dentro del alcance. Ocultar no está permitido. No se puede acceder variables locales antes de que estén declaradas, pero está permitido acceder etiquetas, funciones y ensambladores sin que lo estén. Ensambladores son bloques especiales que se usan para, por ejemplo, devolver el tiempo de ejecución del código o crear contratos. Identificadores externos a un ensamblador no son visibles en un sub ensamblador.

Si el flujo de control va más allá del final de un bloque, se insertan instrucciones pop que corresponden al número de variables locales declaradas en este bloque. Cuando se referencia una variables local, el generador de código necesita saber su posición relativa actual en la pila y por lo tanto necesita hacer un seguimiento de la así llamada altura actual de la pila. Como se quitan todas las variables locales al final de un bloque, la altura de la pila antes y después de un bloque debería ser la misma. Si esto no fuera el caso, se emite un aviso.

¿Por qué usamos constructs de alto nivel como `switch`, `for` y funciones:

Usando `switch`, `for` y funciones, debería ser posible escribir códigos complejos sin usar `jump` o `jumpi` manualmente. Esto simplifica mucho el análisis del control de flujo, lo que permite hacer mejor la verificación formal y la optimización.

Además, si se permiten los saltos manuales, computar la altura de la pila se hace bastante complicado. Se necesita saber la posición de todas las variables locales de la pila, de lo contrario ni las referencias a las variables locales ni quitar automáticamente variables locales de la pila al final de un bloque funcionará correctamente. El mecanismo de desazucarado inserta operaciones correctamente en bloques inalcanzables que ajustan correctamente la altura de la pila en el caso de tener saltos que no tengan un control de flujo en marcha

Ejemplo:

Vamos a seguir un ejemplo de compilación de Solidity a ensamblador desazucarado. Consideramos el tiempo de ejecución del bytecode del siguiente programa escrito en Solidity:

```
contract C {
  function f(uint x) returns (uint y) {
```

```
    y = 1;
    for (uint i = 0; i < x; i++)
      y = 2 * y;
  }
}
```

Se va a generar el siguiente ensamblador:

```
{
  mstore(0x40, 0x60) // almacenar el "puntero de memoria libre"
  // función dispatcher
  switch div(calldataload(0), exp(2, 226))
  case 0xb3de648b {
    let (r) = f(calldataload(4))
    let ret := $allocate(0x20)
    mstore(ret, r)
    return(ret, 0x20)
  }
  default { revert(0, 0) }
  // repartidor de memoria
  function $allocate(size) -> pos {
    pos := mload(0x40)
    mstore(0x40, add(pos, size))
  }
  // la función del contrato
  function f(x) -> y {
    y := 1
    for { let i := 0 } lt(i, x) { i := add(i, 1) } {
      y := mul(2, y)
    }
  }
}
```

Después de la fase de desazucarado, se parece a lo siguiente:

```
{
  mstore(0x40, 0x60)
  {
    let $0 := div(calldataload(0), exp(2, 226))
    jumpi($case1, eq($0, 0xb3de648b))
    jump($caseDefault)
    $case1:
    {
      // la llamada de función - ponemos la etiqueta return y los argumentos encima␣
↪de la pila
      $ret1 calldataload(4) jump(f)
      // Esto es código inalcanzable. Se añaden opcodes que reproducen el efecto de␣
↪la función sobre la altura de la pila: se quitan argumentos y se introducen valores␣
↪devueltos.
      pop pop
      let r := 0
      $ret1: // el punto de retorno actual
      $ret2 0x20 jump($allocate)
      pop pop let ret := 0
      $ret2:
      mstore(ret, r)
      return(ret, 0x20)
      // aunque no sirva de nada, se inserta el salto automáticamente, ya que el␣
↪proceso de desazucarado es una operación puramente sintáctica que no analiza el␣
↪control de flujo.
```

```
      jump($endswitch)
    }
    $caseDefault:
    {
      revert(0, 0)
      jump($endswitch)
    }
    $endswitch:
  }
  jump($afterFunction)
  allocate:
  {
    // nos saltamos el código inalcanzable que introduce los argumentos de la función
    jump($start)
    let $retpos := 0 let size := 0
    $start:
    // las variables de salida están dentro del mismo alcance que los argumentos y
↪ahora se reparten
    let pos := 0
    {
      pos := mload(0x40)
      mstore(0x40, add(pos, size))
    }
    // Este código remplaza los argumentos por los valores de retorno y los saltos
↪hacía atrás.
    swap1 pop swap1 jump
    // Esto es, de nuevo, código inalcanzable que corrige la altura de la pila.
    0 0
  }
  f:
  {
    jump($start)
    let $retpos := 0 let x := 0
    $start:
    let y := 0
    {
      let i := 0
      $for_begin:
      jumpi($for_end, iszero(lt(i, x)))
      {
        y := mul(2, y)
      }
      $for_continue:
      { i := add(i, 1) }
      jump($for_begin)
      $for_end:
    } // Aquí, se inserta una instrucción pop para i
    swap1 pop swap1 jump
    0 0
  }
  $afterFunction:
  stop
}
```

El ensamblador sucede en cuatro etapas:

1. Análisis sintático (o parsing en inglés)
2. Desazucarización (quita switch, for y funciones)

3. Generación de la transmisión de opcodes
4. Generación del bytecode

Vamos a especificar las etapas uno a tres de una forma pseudo formal. Especificaciones más formales se darán luego.

Análisis sintático / Gramática

Éstas son las tareas del módulo de análisis (o del parser en inglés):

- Convertir la transmisión de byte en una transmisión de token, desechando comentarios de tipo C++ (existe un comentario especial para las referencias fuente, pero no lo vamos a explicar aquí).
- Convertir la transmisión de token en un AST según la gramática que figura más abajo.
- Registrar identificadores con el bloque en el que están definidos (anotación al nodo AST) y anotar sobre el punto a partir del cual se puede acceder a las variables.

El ensamblador lexer sigue el ensamblador definido por Solidity.

El espacio en blanco se usa para delimitar tokens y consiste en los caracteres Espacio, Tabular y Línea de alimentación. Los comentarios aceptados son comentarios típicos de JavaScript/C++ y se interpretan de la misma manera que el Espacio.

Gramática:

```
AssemblyBlock = '{' AssemblyItem* '}'
AssemblyItem =
    Identifier |
    AssemblyBlock |
    FunctionalAssemblyExpression |
    AssemblyLocalDefinition |
    FunctionalAssemblyAssignment |
    AssemblyAssignment |
    LabelDefinition |
    AssemblySwitch |
    AssemblyFunctionDefinition |
    AssemblyFor |
    'break' | 'continue' |
    SubAssembly | 'dataSize' '(' Identifier ')' |
    LinkerSymbol |
    'errorLabel' | 'bytecodeSize' |
    NumberLiteral | StringLiteral | HexLiteral
Identifier = [a-zA-Z_$] [a-zA-Z_0-9]*
FunctionalAssemblyExpression = Identifier '(' ( AssemblyItem ( ',' AssemblyItem )* )?
↪')'
AssemblyLocalDefinition = 'let' IdentifierOrList ':=' FunctionalAssemblyExpression
FunctionalAssemblyAssignment = IdentifierOrList ':=' FunctionalAssemblyExpression
IdentifierOrList = Identifier | '(' IdentifierList ')'
IdentifierList = Identifier ( ',' Identifier)*
AssemblyAssignment = '=:' Identifier
LabelDefinition = Identifier ':'
AssemblySwitch = 'switch' FunctionalAssemblyExpression AssemblyCase*
    ( 'default' AssemblyBlock )?
AssemblyCase = 'case' FunctionalAssemblyExpression AssemblyBlock
AssemblyFunctionDefinition = 'function' Identifier '(' IdentifierList? ')'
    ( '->' '(' IdentifierList ')' )? AssemblyBlock
AssemblyFor = 'for' ( AssemblyBlock | FunctionalAssemblyExpression)
    FunctionalAssemblyExpression ( AssemblyBlock | FunctionalAssemblyExpression)␣
↪AssemblyBlock
```

```
SubAssembly = 'assembly' Identifier AssemblyBlock
LinkerSymbol = 'linkerSymbol' '(' StringLiteral ')'
NumberLiteral = HexNumber | DecimalNumber
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+
```

Desazucarización

Una transformación AST quita los for, switch y funciones constructs. El resultados aún es pasible de ser analizado desde el punto de vista sintáctico por el mismo analizador, pero no usará algunos de los constructs. Si se añaden saltos a los que sólo se salta y desde los que luego no se continúa, se añade información sobre el contenido de la pila, a no ser que no se acceda a ninguna variable local de alcance externo o que la altura de la pila sea la misma que para la instrucción anterior.

Pseudo código:

```
desugar item: AST -> AST =
match item {
AssemblyFunctionDefinition('function' name '(' arg1, ..., argn ')' '->' ( '(' ret1, ..
↪., retm ')' body) ->
  <name>:
  {
    jump($<name>_start)
    let $retPC := 0 let argn := 0 ... let arg1 := 0
    $<name>_start:
    let ret1 := 0 ... let retm := 0
    { desugar(body) }
    swap and pop items so that only ret1, ... retm, $retPC are left on the stack
    jump
    0 (1 + n times) to compensate removal of arg1, ..., argn and $retPC
  }
AssemblyFor('for' { init } condition post body) ->
  {
    init // no puede ser su propio bloque porque queremos que el alcance de la␣
↪variable se extienda hasta el cuerpo
    // encuentra I de tal manera que no haya etiquetas $forI_*
    $forI_begin:
    jumpi($forI_end, iszero(condition))
    { body }
    $forI_continue:
    { post }
    jump($forI_begin)
    $forI_end:
  }
'break' ->
  {
    // encuentra el alcance que encierra más cercano con la etiqueta $forI_end
    pop all local variables that are defined at the current point
    but not at $forI_end
    jump($forI_end)
    0 (as many as variables were removed above)
  }
'continue' ->
  {
```

```
    // encuentra el alcance envolvente más cercano con la etiqueta $forI_continue
    pop all local variables that are defined at the current point
    but not at $forI_continue
    jump($forI_continue)
    0 (as many as variables were removed above)
  }
AssemblySwitch(switch condition cases ( default: defaultBlock )? ) ->
  {
    // // encuentra I de tal manera que no haya etiqueta o variable $switchI*
    let $switchI_value := condition
    for each of cases match {
      case val: -> jumpi($switchI_caseJ, eq($switchI_value, val))
    }
    if default block present: ->
      { defaultBlock jump($switchI_end) }
    for each of cases match {
      case val: { body } -> $switchI_caseJ: { body jump($switchI_end) }
    }
    $switchI_end:
  }
FunctionalAssemblyExpression( identifier(arg1, arg2, ..., argn) ) ->
  {
    if identifier is function <name> with n args and m ret values ->
      {
        // encuentra I de tal manera que no exista $funcallI_*
        $funcallI_return argn  ... arg2 arg1 jump(<name>)
        pop (n + 1 times)
        if the current context is `let (id1, ..., idm) := f(...)` ->
          let id1 := 0 ... let idm := 0
          $funcallI_return:
        else ->
          0 (m times)
          $funcallI_return:
          turn the functional expression that leads to the function call
          into a statement stream
      }
    else -> desugar(children of node)
  }
default node ->
  desugar(children of node)
}
```

Generación de la transmisión de opcodes

Durante la generación de la transmisión de opcodes, hacemos un seguimiento de la altura de la pila en un contador, de tal manera que se pueda acceder a la variables de la pila por su nombre. Se modifica la altura de la pila con cada opcode que modifica la pila y con cada etiqueta que se anota con un ajuste de la pila. Cada vez que se introduce una nueva variable local, se registra junto con la altua actual de la pila. Si se accede a una variable (bien para copiar su valor, bien para asignar algo), se selecciona la instrucción DUP o SWAP, dependiendo de la diferencia entre la altura actual de la pila y la altura de la pila en el momento en que se introdujo esta variable.

Pseudo código:

```
codegen item: AST -> opcode_stream =
match item {
AssemblyBlock({ items }) ->
```

```
  join(codegen(item) for item in items)
  if last generated opcode has continuing control flow:
    POP for all local variables registered at the block (including variables
    introduced by labels)
    warn if the stack height at this point is not the same as at the start of the␣
↪block
Identifier(id) ->
  lookup id in the syntactic stack of blocks
  match type of id
    Local Variable ->
      DUPi where i = 1 + stack_height - stack_height_of_identifier(id)
    Label ->
      // se tendrá que resolver esta referencia durante la generación del bytecode
      PUSH<bytecode position of label>
    SubAssembly ->
      PUSH<bytecode position of subassembly data>
FunctionalAssemblyExpression(id ( arguments ) ) ->
  join(codegen(arg) for arg in arguments.reversed())
  id (which has to be an opcode, might be a function name later)
AssemblyLocalDefinition(let (id1, ..., idn) := expr) ->
  register identifiers id1, ..., idn as locals in current block at current stack␣
↪height
  codegen(expr) - assert that expr returns n items to the stack
FunctionalAssemblyAssignment((id1, ..., idn) := expr) ->
  lookup id1, ..., idn in the syntactic stack of blocks, assert that they are␣
↪variables
  codegen(expr)
  for j = n, ..., i:
  SWAPi where i = 1 + stack_height - stack_height_of_identifier(idj)
  POP
AssemblyAssignment(=: id) ->
  look up id in the syntactic stack of blocks, assert that it is a variable
  SWAPi where i = 1 + stack_height - stack_height_of_identifier(id)
  POP
LabelDefinition(name:) ->
  JUMPDEST
NumberLiteral(num) ->
  PUSH<num interpreted as decimal and right-aligned>
HexLiteral(lit) ->
  PUSH32<lit interpreted as hex and left-aligned>
StringLiteral(lit) ->
  PUSH32<lit utf-8 encoded and left-aligned>
SubAssembly(assembly <name> block) ->
  append codegen(block) at the end of the code
dataSize(<name>) ->
  assert that <name> is a subassembly ->
  PUSH32<size of code generated from subassembly <name>>
linkerSymbol(<lit>) ->
  PUSH32<zeros> and append position to linker table
}
```

Diverso

Layout de las variables de estado en almacenamiento

Variables de tamaño estáticas (todo menos mapping y tipos arrays de tamaño variable) se disponen contiguamente en almacenamiento empezando de la posición `0`. Cuando múltiples elementos necesitan menos de 32 bytes, son empaquetados en un slot de almacenamiento cuando es posible de acuerdo a las reglas:

- El primer elemento en un slot de almacenamiento es almacenado alineado en lower-order.
- Tipos elementales sólo usan la cantidad de bytes que se necesita para almacenarlos.
- Si un tipo elemental no cabe en la parte restante de un slot de almacenamiento, es movido al próximo slot de almacenamiento.
- Structs y datos de array siempre comienzan en un nuevo slot y ocupan slots enteros (pero elementos dentro de un struct o array son empacados estrechamente de acuerdo a estas reglas).

Advertencia: Cuando se usan elementos que son más pequeños que 32 bytes, el uso del gas del contrato puede ser más alto. Esto es porque la EVM opera en 32 bytes a la vez. Por lo tanto, si el elemento es más pequeño que eso, la EVM usa más operaciones para reducir el tamaño del elemento de 32 bytes al tamaño deseado.

Sólo es beneficioso reducir el tamaño de los argumentos si estás tratando con valores de almacenamiento porque el compilador empacará múltiples elementos en un slot de almacenamiento, y entonces, combina múltiples lecturas y escrituras en una sóla operación. Cuando se trata con argumentos de función o valores de memoria, no hay beneficio inherente porque el compilador no empaca estos valores.

Finalmente, para permitir a la EVM optimizar esto, asegúrate de ordenar las variables de almacenamiento y miembros `struct` para que puedan ser empacados estrechamente. Por ejemplo, declarando tus variables de almacenamiento en el orden de `uint128, uint128, uint256` en vez de `uint128, uint256, uint128`, ya que el primero utilizará sólo dos slots de almacenamiento y éste último tres.

Los elementos de structs y arrays son almacenados después de ellos mismos, como si fueran dados explícitamente.

Dado a su tamaño impredecible, los tipos de array dinámicos y de mapping usan computación hash Keccak-256 para encontrar la posición de inicio del valor o del dato del array. Estas posiciones de inicio son siempre slots completos.

El mapping o el array dinámico en sí ocupa un slot (sin llenar) en alguna posición `p` de acuerdo a la regla de arriba (o por aplicar esta regla de mappings a mappings o arrays de arrays). Para un array dinámico, este slot guarda el número de elementos en el array (los byte arrays y cadenas son excepciones aquí, mirar abajo). Para un mapping, el slot no es utilizado (pero es necesario para que dos mappings iguales seguidos usen diferentes distribuciones de hash). Datos de array son ubicados en `kecakk256(p)` y el valor correspondiente a una clave de mapping `k` es ubicada en `kecakk256(k . p)` donde `.` es una concatenación. Si el valor de nuevo es un tipo no elemental, las posiciones son encontradas agregando un offset de `keccak256(k . p)`.

`bytes` y `string` almacenan sus datos en el mismo slot junto con su longitud si es corta. En particular: si los datos son al menos `31` bytes de largo, es almacenado en bytes de orden mayor (alineados a la izquierda) y el byte de orden menor almacena `length * 2`. Si es más largo, el slot principal almacena `length * 2 + 1` y los datos son almacenados como siempre en `keccak256(slot)`.

Entonces para el siguiente sinppet de contrato:

```
contract C {
  struct s { uint a; uint b; }
  uint x;
  mapping(uint => mapping(uint => s)) data;
}
```

La posición de `data[4][9].b` está en `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1))) + 1`.

Layout en memoria

Solidity reserva tres slots de 256-bits:

- 0 - 64: espacio de scratch para métodos de hash
- 64 - 96: tamaño de memoria actualmente asignada (también conocida como free memory pointer)

El espacio de scratch puede ser usado entre declaraciones (ej. dento de inline assembly).

Solidity siempre emplaza los nuevos objetos en el puntero de memoria libre y la memoria nunca es liberada (esto puede cambiar en el futuro).

Advertencia: Hay algunas operaciones en Solidity que necesitan un área temporal de memoria mas grande que 64 bytes y por lo tanto no caben en el espacio scratch. Estas operaciones serán emplazadas donde apunta la memoria libre, pero dado su corta vida, el puntero no es actualizado. La memoria puede o no ser puesta a cero. Por esto, uno no debiera esperar que la memoria libre sea puesta a cero.

Layout de Call Data

Cuando un contrato de Solidity es desplegado y cuando es llamado desde una cuenta, los datos de entrada se asume que están en el formato de la especificación ABI. La especificación ABI requiere que los argumentos sean ajustados a múltiplos de 32 bytes. Las llamadas de función internas usan otra convención.

Internas - Limpiando variables

Cuando un valor es más corto que 256-bit, en algunos casos los bits restantes tienen que ser limpiados. El compilador de Solidity está diseñado para limpiar estos bits restantes antes de cualquier operación que pueda ser afectada adversamente por la potencial basura en los bits restantes. Por ejemplo, antes de escribir un valor en la memoria, los bits restantes tienen que ser limpiados porque los contenidos de la memoria pueden ser usados para computar hashes o ser enviados como datos en una llamada. De manera similar, antes de almacenar un valor en el almacenamiento, los bits restantes tienen que ser limpiados porque si no, el valor ilegible puede ser observado.

Por otro lado, no limpiamos los bits si la operación siguiente no es afectada. Por ejemplo, ya que cualquier valor no-cero es considerado `true` por una instrucción `JUMPI`, no limpiamos los valores booleanos antes de que sean utilizados como condición para `JUMPI`.

Además de este principio de diseño, el compilador de Solidity limpia los datos de entrada cuando está cargado en el stack.

Diferentes tipos tienen diferentes reglas para limpiar valores inválidos:

Tipo	Valores válidos	Val. inv. significan
enum de n miembros	0 hasta n - 1	excepción
bool	0 o 1	1
enteros con signo	palabra extendida por signo	por ahora envuelve silenciosamente; en el futuro esto arrojará excepciones
enteros sin signos	altos bits a cero	por ahora envuelve silenciosamente; en el futuro esto arrojará excepciones

Internos - El optimizador

El optimizador de solidity funciona con ensamblador, así que puede y es usado con otros lenguajes. Divide la secuencia de las instrucciones en bloques básicos de JUMPs y JUMPDESTs. Dentro de estos bloques, las instrucciones son analizadas y cada modificación al stack, a la memoria o al almacenamiento son guardadas como una expresión que consiste en una instrucción y una lista de argumentos que son esencialmente punteros a otras expresiones. La idea principal es encontrar expresiones que sean siempre iguales (en cada entrada) y combinarlas a una clase de expresión. El optimizador primero intenta encontrar una nueva expresión en una lista de expresiones conocidas. Si esto no funciona, la expresión es simplificada de acuerdo a reglas como `constante + constante = suma_de_constantes` o `X * 1 = X`. Ya que esto es hecho recursivamente, también podemos aplicar la última regla si el segundo factor es una expresión más compleja donde sabemos que siempre evaluará a uno. Modificaciones al almacenamiento y ubicaciones de memoria tienen que borrar el conocimiento de almacenamiento y ubicaciones de memoria que no son conocidas como diferentes: Si primero escribimos a ubicación *x* y luego a ubicación *y* y ambas son variables de entrada, la segunda puede sobrescribir la primera, entonces no sabemos realmente lo que es almacenado en *x* después de escribir a *y*. Por otro lado, si una simplificación de la expresión *x* - *y* evalúa a una constante distinta de cero, sabemos que podemos mantener nuestro conocimiento de lo que es almacenado en x.

Al final de este proceso, sabemos qué expresiones tienen que estar al final del stack y tienen una lista de modificaciones a la memoria y almacenamiento. Esta información es almacenada junto con los bloques básicos y es usada para unirlas. Además, información sobre el stack, almacenamiento y configuración de memoria es enviada al (los) próximo(s) bloque(s). Si sabemos los objetivos de cada una de las instrucciones JUMP y JUMPI, podemos construir un gráfico de flujo completo del programa. Si hay un sólo objetivo que no conocemos (esto puede pasar ya que en principio, los objetivos de jumps pueden ser computados de las entradas), tenemos que borrar toda información sobre los estados de entrada de un bloque ya que puede ser el objetivo del JUMP desconocido. Si se encuentra un JUMPI cuya condición evalúa a una constante, es transformado en un jump incondicional.

Como en el último paso, el código en cada bloque es completamente regenerado. Un gráfico de dependencias es creado de la expresión en el stack al final del bloque y cada operación que no es parte de este gráfico es esencialmente olvidada. Ahora se genera código que aplica las modificaciones a la memoria y al almacenamiento en el orden en que fueron hechas en el código original (olvidando modificaciones que fueron encontradas innecesarias) y finalmente, genera todos los valores que son requeridos para estar en el stack en el lugar correcto.

Estos pasos son aplicados a cada bloque básico y el nuevo código generado se usa como reemplazo si es más pequeño. Si un bloque básico es dividido en un JUMPI y durante el análisis la condición evalúa a una constante, el JUMPI es reemplazado dependiendo del valor de la constante, y por lo tanto código como

```
var x = 7;
data[7] = 9;
if (data[x] != x + 2)
  return 2;
else
  return 1;
```

es simplificado a código que también puede ser compilado de

```
data[7] = 9;
return 1;
```

aunque las instrucciones contenían un jump en el inicio.

Mappings de fuente

Como parte del AST (Abstract syntax tree) de salida, el compilador provee el rango del código fuente que es representado por el nodo respecto al AST. Esto puede ser usado para varios propósitos desde herramientas de análisis estático que reportan errores basados en el AST y herramientas de debugging que demarcan variables locales y sus usos.

Además, el compilador también puede generar un mapping del bytecode al rango en el código fuente que generó la instrucción. Esto es importante para herramientas de análisis estático que operan a nivel bytecode y para mostrar la posición actual en el código fuente dentro del debugger o para manejar los breakpoints.

Ambos tipos de mappings de fuente usan identificadores enteros para referirse a archivos fuente. Estos son índices de arrays regulares en una lista de archivos fuente habitualmente llamados `"sourcelist"`, que es parte del combined-json y el output del compilador json / npm.

Los mappings de fuente dentro del AST usan la siguiente notación:

`s:l:f`

Donde `s` es el byte-offset de el inicio del rango en el archivo fuente, `l` es el largo del rango de la fuente en bytes y `f` es el índice de fuente mencionado arriba.

La codificación en el mapping de fuente para el bytecode es más complicada: Es una lista de `s:l:f:j` separada por `;`. Cada una de estos elementos corresponde a una instrucción, i.e. no puedes usar el byte-offset si no que tienes que usar la instrucción offset (las instrucciones push son mas largas que un sólo byte). Los campos `s`, `l` y `f` son como detallamos arriba y `j` puede ser `i`, `o` o `-` y significa si una instrucción jump va en la función, devuelve desde una función o si es un jump regular como parte de un (ej) bucle.

A fin de comprimir estos mappings de fuente especialmente para bytecode, las siguientes reglas son usadas:

- Si un campo está vacío, el valor del elemento precedente es usado.
- Si falta un `:`, todos los campos siguientes son considerados vacíos.

Esto significa que los siguientes mappings de fuente representan la misma información:

`1:2:1;1:9:1;2:1:2;2:1:2;2:1:2`

`1:2:1;:9;2::2;;`

Metadata del contrato

El compilador de Solidity genera automáticamente un archivo JSON, el metadata del contrato, que contiene información sobre el contrato actual. Se puede usar para consultar la versión del compilador, las fuentes usadas, el ABI y documentación NatSpec a fin de interactuar con más seguridad con el contrato y verificar su código fuente.

El compilador agrega un hash Swarm del archivo metadata al final del bytecode (para detalles, mirar abajo) de cada contrato, para que se pueda recuperar el archivo en una manera autentificada sin tener que usar un proveedor de datos centrales.

Sin embargo, se tiene que publicar el archivo metadata a Swarm (o otro servicio) para que otros puedan verlo. El archivo puede ser producido usando `solc --metadata` y el archivo será llamado `NombreContrato_meta.json`. Contendrá referencias Swarm al código fuente, así que tienes que subir todos los archivos de código fuente y el archivo metadata.

El archivo metadata tiene el formato siguiente. El ejemplo de abajo es presentado de manera legible por humanos. Los metadatos formateados correctamente deben usar comillas correctamente, reducir el espacio en blanco a un mínimo y ordenarse de manera diferente. Los comentarios obviamente tampoco son permitidos y son usados aquí sólo por razones explicativos.

```
{
  // Requerido: La versión del formato de metadata
  version: "1",
  // Requerido: lenguaje de código fuente, settea una "sub-versión"
  // de la especificación
  language: "Solidity",
  // Requerido: Detalles del compilador, los contenidos son específicos
  // al lenguaje
```

```
  compiler: {
    // Requerido para Solidity: Version del compilador
    version: "0.4.6+commit.2dabbdf0.Emscripten.clang",
    // Opcional: Hash del compilador binario que produjo este resultado
    keccak256: "0x123..."
  },
  // Requerido: Compilación de archivos fuente/unidades fuente, las claves son
  // nombres de archivos
  sources:
  {
    "myFile.sol": {
      // Requerido: keccak256 hash del archivo fuente
      "keccak256": "0x123...",
      // Requerido (al menos que "content" sea usado, ver abajo): URL(s) ordenadas
      // al archivo fuente, el protocolo es menos arbitrario, pero se recomienda una
      // URL de Swarm
      "urls": [ "bzzr://56ab..." ]
    },
    "mortal": {
      // Requerido: hash keccak256 del archivo fuente
      "keccak256": "0x234...",
      // Requerido (al menos que "url" sea usado): contenidos literales del archivo␣
↪fuente
      "content": "contract mortal is owned { function kill() { if (msg.sender ==␣
↪owner) selfdestruct(owner); } }"
    }
  },
  // Requerido: Configuración del compilador
  settings:
  {
    // Requerido para Solidity: Lista ordenada de remappeos
    remappings: [ ":g/dir" ],
    // Opcional: Configuración del optimizador (por defecto falso)
    optimizer: {
      enabled: true,
      runs: 500
    },
    // Requerido para Solidity: Archivo y nombre del contrato o librería para
    // la librería para el cual es creado este archivo metadata.
    compilationTarget: {
      "myFile.sol": "MyContract"
    },
    // Requerido para Solidity: Direcciones de las librerías usadas
    libraries: {
      "MyLib": "0x123123..."
    }
  },
  // Requerido: Información generada sobre el contrato.
  output:
  {
    // Requerido: definición ABI del contrato
    abi: [ ... ],
    // Requerido: documentación de usuario del contrato de NatSpec
    userdoc: [ ... ],
    // Requerido: documentación de desarrollador del contrato de NatSpec
    devdoc: [ ... ],
  }
}
```

Nota: Nótese que la definición ABI de arriba no tiene orden fijo. Puede cambiar en distintas versiones del compilador.

Nota: Ya que el bytecode del contrato resultante contiene el hash del metadata, cualquier cambio a la metadata resultará en un cambio en el bytecode. Además, ya que la metadata incluye un hash de todos los fuentes usados, un simple espacio en blanco en cualquiera de los archivos de fuente resultará en un metadata diferente, y posteriormente en bytecode diferente.

Codificación del hash de metadata en el bytecode

Ya que en el futuro puede que soportemos otras maneras de consultar el archivo metadata, el mapping `{"bzzr0": <Swarm hash>}` es guardado codificado en [CBOR](https://tools.ietf.org/html/rfc7049). Ya que el principio de esa codificación no es fácil de encontrar, la longitud se añade en una codificación two-byte big-endian. La versión actual del compilador de Solidity entonces agrega lo siguiente al final del bytecode desplegado:

```
0xa1 0x65 'b' 'z' 'z' 'r' '0' 0x58 0x20 <32 bytes swarm hash> 0x00 0x29
```

Para recuperar estos datos, el final del bytecode desplegado puede ser revisado para ver si coincide con ese patrón y usar el hash de Swarm para recuperar el archivo.

Uso para generar automáticamente la interfaz y NatSpec

El metadata es usado de la siguiente forma: un componente que quiere interactuar con un contrato (ej. Mist) obtiene el código del contrato, a partir de eso el hash de Swarm de un archivo que luego es recuperado. Ese archivo es un JSON con la estructura como la de arriba.

El componente puede luego usar el ABI para generar automáticamente una rudimentaria interfaz de usuario para el contrato.

Además, Mist puede usar el userdoc para mostrar un mensaje de confirmación al usuario cuando interactúe con el contrato.

Uso para la verificación de código fuente

A fin de verificar la compilación, el código fuente pueden ser recuperado de Swarm desde el enlace en el archivo metadata. La versión correcta del compilador (que se comprueba que sea parte de los compiladores "oficiales") es invocada en esa entrada con la configuración específicada. El resultado bytecode es comparado a los datos de la transacción de creación o a datos del opcode CREATE. Esto verifica automáticamente la metadata ya que su hash es parte del bytecode. Datos en exceso corresponden a los datos de entrada del constructor, que deben ser decodificados de acuerdo a la interfaz y presentados al usuario.

Trucos y consejos

- Usar `delete` en arrays para borrar sus elementos.
- Usar tipos mas cortos para elementos struct y ordenarlos para que los elementos mas cortos estén agrupados. Esto puede disminuir los costes de gas ya que múltiples operaciones SSTORE puden ser combinadas en una sóla (SSTORE cuesta 5000 o 20000 gas, así que esto es lo que se optimiza). ¡Usa el estimador de precio de gas (con optimizador activado) para probar!

- Hacer las variables de estado públicas - el compilador creará *getters* automáticamente.
- Si revisa las condiciones de entrada o de estado muchas veces en el inicio de las funciones, intenta usar *Modificadores de funciones*.
- Si tu contrato tiene una función llamada `send` pero quieres usar la función interna de envío, usa `address(contractVariable).send(amount)`.
- Inicia structs de almacenamiento con una sola asignación: `x = MyStruct({a: 1, b: 2});`

Cheatsheet

Orden de preferencia de operadores

El siguiente es el orden de precedencia para operadores, listado en orden de evaluación.

Precedencia	Descripción	Operador
1	Postfix incremento y decremento	`++`, `--`
	llamada de tipo función	`<func>(<args...>)`
	subíndice de array	`<array>[<index>]`
	Acceso de miembro	`<object>.<member>`
	Paréntesis	`(<statement>)`
2	Prefijo incremento y decremento	`++`, `--`
	Más y menos unarios	`+`, `-`
	Operaciones unarias	`delete`
	NOT lógico	`!`
	NOT a nivel de bits	`~`
3	Exponenciación	`**`
4	Multiplicación, división and módulo	`*`, `/`, `%`
5	Adición and sustracción	`+`, `-`
6	Desplazamiento de bits	`<<`, `>>`
7	AND a nivel de bits	`&`
8	XOR a nivel de bits	`^`
9	OR a nivel de bits	`|`
10	Operadores de desigualdad	`<`, `>`, `<=`, `>=`
11	Operadores de igualdad	`==`, `!=`
12	AND lógico	`&&`
13	OR lógico	`||`
14	Operador ternario	`<conditional> ? <if-true> : <if-false>`
15	Operador de asignación	`=`, `|=`, `^=`, `&=`, `<<=`, `>>=`, `+=`, `-=`, `*=`, `/=`, `%=`
16	Operador de coma	`,`

Variables globales

- `block.blockhash(uint blockNumber) returns (bytes32)`: hash del bloque dado - sólo funciona para los últimos 256 bloques
- `block.coinbase` (`address`): address del minero del bloque actual
- `block.difficulty` (`uint`): dificultad del bloque actual
- `block.gaslimit` (`uint`): gaslimit del bloque actual
- `block.number` (`uint`): número del bloque actual
- `block.timestamp` (`uint`): timestamp del bloque actual

- `msg.data` (`bytes`): calldata completa
- `msg.gas` (`uint`): gas restante
- `msg.sender` (`address`): sender del mensaje (llamada actual)
- `msg.value` (`uint`): número de wei enviados con el mensaje
- `now` (`uint`): timestamp del bloque actual (alias para `block.timestamp`)
- `tx.gasprice` (`uint`): precio de gas de la transacción
- `tx.origin` (`address`): sender de la transacción (cadena de llamada completa)
- `assert(bool condition)`: abortar ejecución y deshacer cambios de estado si la condición es `false` (uso para error interno)
- `require(bool condition)`: abortar ejecución y deshacer cambios de estado si la condición es `false` (uso para entradas erróneas)
- `revert()`: abortar ejecución y deshacer cambios de estado
- `keccak256(...) returns (bytes32)`: computar el hash Ethereum-SHA-3 (Keccak-256) de los argumentos (empacados)
- `sha3(...) returns (bytes32)`: un alias a *keccak256()*
- `sha256(...) returns (bytes32)`: computar el hash SHA-256 de los argumentos (empacados)
- `ripemd160(...) returns (bytes20)`: computar el hash RIPEMD-160 de los argumentos (empacados)
- `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`: recuperar address asociada con la llave pública desde la firma de la curva elíptica, devuelve cero en caso de error
- `addmod(uint x, uint y, uint k) returns (uint)`: computa `(x + y) % k` donde la suma es hecha con precisión arbitraria y no envuelve `2**256`
- `mulmod(uint x, uint y, uint k) returns (uint)`: computa `(x * y) % k` donde la multiplicación es hecha con precisión arbitraria y no envuelve `2**256`
- `this` (tipo del contrato actual): el contrato actual, explícitamente convertible a `address`
- `super`: el contrato un nivel más alto en la jerarquía de herencia
- `selfdestruct(address recipient)`: destruir el contrato actual, enviando sus fondos al address dada
- `<address>.balance` (`uint256`): saldo de *Address* en Wei
- `<address>.send(uint256 amount) returns (bool)`: enviar monto dado de Wei a *Address*, devuelve `false` en caso de error
- `<address>.transfer(uint256 amount)`: enviar monto dado de Wei a *Address*, arroja excepción si falla

Especificadores de visibilidad de función

```
function myFunction() <visibility specifier> returns (bool) {
    return true;
}
```

- `public`: visible externa e internamente (crea función getter para almacenamiento/variables de estado)

- `private`: sólo visible en el contrato actual
- `external`: sólo visible desde el exterior (sólo para funciones) - e.j. sólo puede ser llamado mediante mensajes (via `this.func`)
- `internal`: sólo visible internamente

Modificadores

- `constant` para variables de estado: no permite asignaciones (excepto inicialización), no ocupa un slot de almacenamiento.
- `constant` para funciones: no permite modificación de estado - no está forzado aún.
- `anonymous` para eventos: no guarda la firma del evento como topic.
- `indexed` para parámetros de eventos: guarda el parámetro como topic.
- `payable` para funciones: les permite recibir ether junto a una llamada.

Keywords reservadas

Estas palabras están reservadas en Solidity. Pueden incorporarse a la sintaxis en el futuro:

`abstract`, `after`, `case`, `catch`, `default`, `final`, `in`, `inline`, `interface`, `let`, `match`, `null`, `of`, `pure`, `relocatable`, `static`, `switch`, `try`, `type`, `typeof`, `view`.

Gramática del lenguaje

```
SourceUnit = (PragmaDirective | ImportDirective | ContractDefinition)*

// Pragma actually parses anything up to the trailing ';' to be fully forward-
↪compatible.
PragmaDirective = 'pragma' Identifier ([^;]+) ';'

ImportDirective = 'import' StringLiteral ('as' Identifier)? ';'
        | 'import' ('*' | Identifier) ('as' Identifier)? 'from' StringLiteral ';'
        | 'import' '{' Identifier ('as' Identifier)? ( ',' Identifier ('as'␣
↪Identifier)? )* '}' 'from' StringLiteral ';'

ContractDefinition = ( 'contract' | 'library' | 'interface' ) Identifier
                     ( 'is' InheritanceSpecifier (',' InheritanceSpecifier )* )?
                     '{' ContractPart* '}'

ContractPart = StateVariableDeclaration | UsingForDeclaration
             | StructDefinition | ModifierDefinition | FunctionDefinition |␣
↪EventDefinition | EnumDefinition

InheritanceSpecifier = UserDefinedTypeName ( '(' Expression ( ',' Expression )* ')' )?

StateVariableDeclaration = TypeName ( 'public' | 'internal' | 'private' )? Identifier␣
↪('=' Expression)? ';'
UsingForDeclaration = 'using' Identifier 'for' ('*' | TypeName) ';'
StructDefinition = 'struct' Identifier '{'
                     ( VariableDeclaration ';' (VariableDeclaration ';')* )? '}'
ModifierDefinition = 'modifier' Identifier ParameterList? Block
```

```
FunctionDefinition = 'function' Identifier? ParameterList
                     ( FunctionCall | Identifier | 'constant' | 'payable' | 'external
↪' | 'public' | 'internal' | 'private' )*
                     ( 'returns' ParameterList )? ( ';' | Block )
EventDefinition = 'event' Identifier IndexedParameterList 'anonymous'? ';'

EnumValue = Identifier
EnumDefinition = 'enum' Identifier '{' EnumValue? (',' EnumValue)* '}'

IndexedParameterList = '(' ( TypeName 'indexed'? Identifier? (',' TypeName 'indexed'?␣
↪Identifier?)* )? ')'
ParameterList =        '(' ( TypeName            Identifier? (',' TypeName          ␣
↪Identifier?)* )? ')'
TypeNameList =         '(' ( TypeName (',' TypeName )* )? ')'

// semantic restriction: mappings and structs (recursively) containing mappings
// are not allowed in argument lists
VariableDeclaration = TypeName StorageLocation? Identifier

TypeName = ElementaryTypeName
         | UserDefinedTypeName
         | Mapping
         | ArrayTypeName
         | FunctionTypeName

UserDefinedTypeName = Identifier ( '.' Identifier )*

Mapping = 'mapping' '(' ElementaryTypeName '=>' TypeName ')'
ArrayTypeName = TypeName '[' Expression? ']'
FunctionTypeName = 'function' TypeNameList ( 'internal' | 'external' | 'constant' |
↪'payable' )*
                   ( 'returns' TypeNameList )?
StorageLocation = 'memory' | 'storage'

Block = '{' Statement* '}'
Statement = IfStatement | WhileStatement | ForStatement | Block |␣
↪InlineAssemblyStatement |
            ( DoWhileStatement | PlaceholderStatement | Continue | Break | Return |
              Throw | SimpleStatement ) ';'

ExpressionStatement = Expression
IfStatement = 'if' '(' Expression ')' Statement ( 'else' Statement )?
WhileStatement = 'while' '(' Expression ')' Statement
PlaceholderStatement = '_'
SimpleStatement = VariableDefinition | ExpressionStatement
ForStatement = 'for' '(' (SimpleStatement)? ';' (Expression)? ';'␣
↪(ExpressionStatement)? ')' Statement
InlineAssemblyStatement = 'assembly' StringLiteral? InlineAssemblyBlock
DoWhileStatement = 'do' Statement 'while' '(' Expression ')'
Continue = 'continue'
Break = 'break'
Return = 'return' Expression?
Throw = 'throw'
VariableDefinition = ('var' IdentifierList | VariableDeclaration) ( '=' Expression )?
IdentifierList = '(' ( Identifier? ',' )* Identifier? ')'

// Precedence by order (see github.com/ethereum/solidity/pull/732)
Expression =
```

```
  ( Expression ('++' | '--') | FunctionCall | IndexAccess | MemberAccess | '('␣
↪Expression ')' )
  | ('!' | '~' | 'delete' | '++' | '--' | '+' | '-') Expression
  | Expression '**' Expression
  | Expression ('*' | '/' | '%') Expression
  | Expression ('+' | '-') Expression
  | Expression ('<<' | '>>') Expression
  | Expression '&' Expression
  | Expression '^' Expression
  | Expression '|' Expression
  | Expression ('<' | '>' | '<=' | '>=') Expression
  | Expression ('==' | '!=') Expression
  | Expression '&&' Expression
  | Expression '||' Expression
  | Expression '?' Expression ':' Expression
  | Expression ('=' | '|=' | '^=' | '&=' | '<<=' | '>>=' | '+=' | '-=' | '*=' | '/='␣
↪| '%=') Expression
  | Expression? (',' Expression)
  | PrimaryExpression

PrimaryExpression = Identifier
                  | BooleanLiteral
                  | NumberLiteral
                  | HexLiteral
                  | StringLiteral
                  | ElementaryTypeNameExpression

ExpressionList = Expression ( ',' Expression )*
NameValueList = Identifier ':' Expression ( ',' Identifier ':' Expression )*

FunctionCall = ( PrimaryExpression | NewExpression | TypeName ) ( ( '.' Identifier )␣
↪| ( '[' Expression ']' ) )* '(' FunctionCallArguments ')'
FunctionCallArguments = '{' NameValueList? '}'
                      | ExpressionList?

NewExpression = 'new' TypeName
MemberAccess = Expression '.' Identifier
IndexAccess = Expression '[' Expression? ']'

BooleanLiteral = 'true' | 'false'
NumberLiteral = ( HexNumber | DecimalNumber ) (' ' NumberUnit)?
NumberUnit = 'wei' | 'szabo' | 'finney' | 'ether'
           | 'seconds' | 'minutes' | 'hours' | 'days' | 'weeks' | 'years'
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
Identifier = [a-zA-Z_$] [a-zA-Z_$0-9]*

HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+

ElementaryTypeNameExpression = ElementaryTypeName

ElementaryTypeName = 'address' | 'bool' | 'string' | 'var'
                   | Int | Uint | Byte | Fixed | Ufixed

Int = 'int' | 'int8' | 'int16' | 'int24' | 'int32' | 'int40' | 'int48' | 'int56' |
↪'int64' | 'int72' | 'int80' | 'int88' | 'int96' | 'int104' | 'int112' | 'int120' |
↪'int128' | 'int136' | 'int144' | 'int152' | 'int160' | 'int168' | 'int176' | 'int184
↪' | 'int192' | 'int200' | 'int208' | 'int216' | 'int224' | 'int232' | 'int240' |
↪'int248' | 'int256'
```

```
Uint = 'uint' | 'uint8' | 'uint16' | 'uint24' | 'uint32' | 'uint40' | 'uint48' |
↪'uint56' | 'uint64' | 'uint72' | 'uint80' | 'uint88' | 'uint96' | 'uint104' |
↪'uint112' | 'uint120' | 'uint128' | 'uint136' | 'uint144' | 'uint152' | 'uint160' |
↪'uint168' | 'uint176' | 'uint184' | 'uint192' | 'uint200' | 'uint208' | 'uint216' |
↪'uint224' | 'uint232' | 'uint240' | 'uint248' | 'uint256'

Byte = 'byte' | 'bytes' | 'bytes1' | 'bytes2' | 'bytes3' | 'bytes4' | 'bytes5' |
↪'bytes6' | 'bytes7' | 'bytes8' | 'bytes9' | 'bytes10' | 'bytes11' | 'bytes12' |
↪'bytes13' | 'bytes14' | 'bytes15' | 'bytes16' | 'bytes17' | 'bytes18' | 'bytes19' |
↪'bytes20' | 'bytes21' | 'bytes22' | 'bytes23' | 'bytes24' | 'bytes25' | 'bytes26' |
↪'bytes27' | 'bytes28' | 'bytes29' | 'bytes30' | 'bytes31' | 'bytes32'

Fixed = 'fixed' | 'fixed0x8' | 'fixed0x16' | 'fixed0x24' | 'fixed0x32' | 'fixed0x40'
↪| 'fixed0x48' | 'fixed0x56' | 'fixed0x64' | 'fixed0x72' | 'fixed0x80' | 'fixed0x88'
↪| 'fixed0x96' | 'fixed0x104' | 'fixed0x112' | 'fixed0x120' | 'fixed0x128' |
↪'fixed0x136' | 'fixed0x144' | 'fixed0x152' | 'fixed0x160' | 'fixed0x168' |
↪'fixed0x176' | 'fixed0x184' | 'fixed0x192' | 'fixed0x200' | 'fixed0x208' |
↪'fixed0x216' | 'fixed0x224' | 'fixed0x232' | 'fixed0x240' | 'fixed0x248' |
↪'fixed0x256' | 'fixed8x8' | 'fixed8x16' | 'fixed8x24' | 'fixed8x32' | 'fixed8x40' |
↪'fixed8x48' | 'fixed8x56' | 'fixed8x64' | 'fixed8x72' | 'fixed8x80' | 'fixed8x88' |
↪'fixed8x96' | 'fixed8x104' | 'fixed8x112' | 'fixed8x120' | 'fixed8x128' |
↪'fixed8x136' | 'fixed8x144' | 'fixed8x152' | 'fixed8x160' | 'fixed8x168' |
↪'fixed8x176' | 'fixed8x184' | 'fixed8x192' | 'fixed8x200' | 'fixed8x208' |
↪'fixed8x216' | 'fixed8x224' | 'fixed8x232' | 'fixed8x240' | 'fixed8x248' |
↪'fixed16x8' | 'fixed16x16' | 'fixed16x24' | 'fixed16x32' | 'fixed16x40' |
↪'fixed16x48' | 'fixed16x56' | 'fixed16x64' | 'fixed16x72' | 'fixed16x80' |
↪'fixed16x88' | 'fixed16x96' | 'fixed16x104' | 'fixed16x112' | 'fixed16x120' |
↪'fixed16x128' | 'fixed16x136' | 'fixed16x144' | 'fixed16x152' | 'fixed16x160' |
↪'fixed16x168' | 'fixed16x176' | 'fixed16x184' | 'fixed16x192' | 'fixed16x200' |
↪'fixed16x208' | 'fixed16x216' | 'fixed16x224' | 'fixed16x232' | 'fixed16x240' |
↪'fixed24x8' | 'fixed24x16' | 'fixed24x24' | 'fixed24x32' | 'fixed24x40' |
↪'fixed24x48' | 'fixed24x56' | 'fixed24x64' | 'fixed24x72' | 'fixed24x80' |
↪'fixed24x88' | 'fixed24x96' | 'fixed24x104' | 'fixed24x112' | 'fixed24x120' |
↪'fixed24x128' | 'fixed24x136' | 'fixed24x144' | 'fixed24x152' | 'fixed24x160' |
↪'fixed24x168' | 'fixed24x176' | 'fixed24x184' | 'fixed24x192' | 'fixed24x200' |
↪'fixed24x208' | 'fixed24x216' | 'fixed24x224' | 'fixed24x232' | 'fixed32x8' |
↪'fixed32x16' | 'fixed32x24' | 'fixed32x32' | 'fixed32x40' | 'fixed32x48' |
↪'fixed32x56' | 'fixed32x64' | 'fixed32x72' | 'fixed32x80' | 'fixed32x88' |
↪'fixed32x96' | 'fixed32x104' | 'fixed32x112' | 'fixed32x120' | 'fixed32x128' |
↪'fixed32x136' | 'fixed32x144' | 'fixed32x152' | 'fixed32x160' | 'fixed32x168' |
↪'fixed32x176' | 'fixed32x184' | 'fixed32x192' | 'fixed32x200' | 'fixed32x208' |
↪'fixed32x216' | 'fixed32x224' | 'fixed40x8' | 'fixed40x16' | 'fixed40x24' |
↪'fixed40x32' | 'fixed40x40' | 'fixed40x48' | 'fixed40x56' | 'fixed40x64' |
↪'fixed40x72' | 'fixed40x80' | 'fixed40x88' | 'fixed40x96' | 'fixed40x104' |
↪'fixed40x112' | 'fixed40x120' | 'fixed40x128' | 'fixed40x136' | 'fixed40x144' |
↪'fixed40x152' | 'fixed40x160' | 'fixed40x168' | 'fixed40x176' | 'fixed40x184' |
↪'fixed40x192' | 'fixed40x200' | 'fixed40x208' | 'fixed40x216' | 'fixed48x8' |
↪'fixed48x16' | 'fixed48x24' | 'fixed48x32' | 'fixed48x40' | 'fixed48x48' |
↪'fixed48x56' | 'fixed48x64' | 'fixed48x72' | 'fixed48x80' | 'fixed48x88' |
↪'fixed48x96' | 'fixed48x104' | 'fixed48x112' | 'fixed48x120' | 'fixed48x128' |
↪'fixed48x136' | 'fixed48x144' | 'fixed48x152' | 'fixed48x160' | 'fixed48x168' |
↪'fixed48x176' | 'fixed48x184' | 'fixed48x192' | 'fixed48x200' | 'fixed48x208' |
↪'fixed56x8' | 'fixed56x16' | 'fixed56x24' | 'fixed56x32' | 'fixed56x40' |
↪'fixed56x48' | 'fixed56x56' | 'fixed56x64' | 'fixed56x72' | 'fixed56x80' |
↪'fixed56x88' | 'fixed56x96' | 'fixed56x104' | 'fixed56x112' | 'fixed56x120' |
↪'fixed56x128' | 'fixed56x136' | 'fixed56x144' | 'fixed56x152' | 'fixed56x160' |
↪'fixed56x168' | 'fixed56x176' | 'fixed56x184' | 'fixed56x192' | 'fixed56x200' |
↪'fixed64x8' | 'fixed64x16' | 'fixed64x24' | 'fixed64x32' | 'fixed64x40' |
↪'fixed64x48' | 'fixed64x56' | 'fixed64x64' | 'fixed64x72' | 'fixed64x80' |
↪'fixed64x88' | 'fixed64x96' | 'fixed64x104' | 'fixed64x112' | 'fixed64x120' |
↪'fixed64x128' | 'fixed64x136' | 'fixed64x144' | 'fixed64x152' | 'fixed64x160' |
↪'fixed64x168' | 'fixed64x176' | 'fixed64x184' | 'fixed64x192' | 'fixed72x8' |
↪'fixed72x16' | 'fixed72x24' | 'fixed72x32' | 'fixed72x40' | 'fixed72x48' |
↪'fixed72x56' | 'fixed72x64' | 'fixed72x72' | 'fixed72x80' | 'fixed72x88' |
```

```
Ufixed = 'ufixed' | 'ufixed0x8' | 'ufixed0x16' | 'ufixed0x24' | 'ufixed0x32' |
↪'ufixed0x40' | 'ufixed0x48' | 'ufixed0x56' | 'ufixed0x64' | 'ufixed0x72' |
↪'ufixed0x80' | 'ufixed0x88' | 'ufixed0x96' | 'ufixed0x104' | 'ufixed0x112' |
↪'ufixed0x120' | 'ufixed0x128' | 'ufixed0x136' | 'ufixed0x144' | 'ufixed0x152' |
↪'ufixed0x160' | 'ufixed0x168' | 'ufixed0x176' | 'ufixed0x184' | 'ufixed0x192' |
↪'ufixed0x200' | 'ufixed0x208' | 'ufixed0x216' | 'ufixed0x224' | 'ufixed0x232' |
↪'ufixed0x240' | 'ufixed0x248' | 'ufixed0x256' | 'ufixed8x8' | 'ufixed8x16' |
↪'ufixed8x24' | 'ufixed8x32' | 'ufixed8x40' | 'ufixed8x48' | 'ufixed8x56' |
↪'ufixed8x64' | 'ufixed8x72' | 'ufixed8x80' | 'ufixed8x88' | 'ufixed8x96' |
↪'ufixed8x104' | 'ufixed8x112' | 'ufixed8x120' | 'ufixed8x128' | 'ufixed8x136' |
↪'ufixed8x144' | 'ufixed8x152' | 'ufixed8x160' | 'ufixed8x168' | 'ufixed8x176' |
↪'ufixed8x184' | 'ufixed8x192' | 'ufixed8x200' | 'ufixed8x208' | 'ufixed8x216' |
↪'ufixed8x224' | 'ufixed8x232' | 'ufixed8x240' | 'ufixed8x248' | 'ufixed16x8' |
↪'ufixed16x16' | 'ufixed16x24' | 'ufixed16x32' | 'ufixed16x40' | 'ufixed16x48' |
↪'ufixed16x56' | 'ufixed16x64' | 'ufixed16x72' | 'ufixed16x80' | 'ufixed16x88' |
↪'ufixed16x96' | 'ufixed16x104' | 'ufixed16x112' | 'ufixed16x120' | 'ufixed16x128' |
↪'ufixed16x136' | 'ufixed16x144' | 'ufixed16x152' | 'ufixed16x160' | 'ufixed16x168'␣
↪| 'ufixed16x176' | 'ufixed16x184' | 'ufixed16x192' | 'ufixed16x200' | 'ufixed16x208
↪' | 'ufixed16x216' | 'ufixed16x224' | 'ufixed16x232' | 'ufixed16x240' | 'ufixed24x8
↪' | 'ufixed24x16' | 'ufixed24x24' | 'ufixed24x32' | 'ufixed24x40' | 'ufixed24x48' |
↪'ufixed24x56' | 'ufixed24x64' | 'ufixed24x72' | 'ufixed24x80' | 'ufixed24x88' |
↪'ufixed24x96' | 'ufixed24x104' | 'ufixed24x112' | 'ufixed24x120' | 'ufixed24x128' |
↪'ufixed24x136' | 'ufixed24x144' | 'ufixed24x152' | 'ufixed24x160' | 'ufixed24x168'␣
↪| 'ufixed24x176' | 'ufixed24x184' | 'ufixed24x192' | 'ufixed24x200' | 'ufixed24x208
↪' | 'ufixed24x216' | 'ufixed24x224' | 'ufixed24x232' | 'ufixed32x8' | 'ufixed32x16'␣
↪| 'ufixed32x24' | 'ufixed32x32' | 'ufixed32x40' | 'ufixed32x48' | 'ufixed32x56' |
↪'ufixed32x64' | 'ufixed32x72' | 'ufixed32x80' | 'ufixed32x88' | 'ufixed32x96' |
↪'ufixed32x104' | 'ufixed32x112' | 'ufixed32x120' | 'ufixed32x128' | 'ufixed32x136'␣
↪| 'ufixed32x144' | 'ufixed32x152' | 'ufixed32x160' | 'ufixed32x168' | 'ufixed32x176
↪' | 'ufixed32x184' | 'ufixed32x192' | 'ufixed32x200' | 'ufixed32x208' |
↪'ufixed32x216' | 'ufixed32x224' | 'ufixed40x8' | 'ufixed40x16' | 'ufixed40x24' |
↪'ufixed40x32' | 'ufixed40x40' | 'ufixed40x48' | 'ufixed40x56' | 'ufixed40x64' |
↪'ufixed40x72' | 'ufixed40x80' | 'ufixed40x88' | 'ufixed40x96' | 'ufixed40x104' |
↪'ufixed40x112' | 'ufixed40x120' | 'ufixed40x128' | 'ufixed40x136' | 'ufixed40x144'␣
↪| 'ufixed40x152' | 'ufixed40x160' | 'ufixed40x168' | 'ufixed40x176' | 'ufixed40x184
↪' | 'ufixed40x192' | 'ufixed40x200' | 'ufixed40x208' | 'ufixed40x216' | 'ufixed48x8
↪' | 'ufixed48x16' | 'ufixed48x24' | 'ufixed48x32' | 'ufixed48x40' | 'ufixed48x48' |
↪'ufixed48x56' | 'ufixed48x64' | 'ufixed48x72' | 'ufixed48x80' | 'ufixed48x88' |
↪'ufixed48x96' | 'ufixed48x104' | 'ufixed48x112' | 'ufixed48x120' | 'ufixed48x128' |
↪'ufixed48x136' | 'ufixed48x144' | 'ufixed48x152' | 'ufixed48x160' | 'ufixed48x168'␣
↪| 'ufixed48x176' | 'ufixed48x184' | 'ufixed48x192' | 'ufixed48x200' | 'ufixed48x208
↪' | 'ufixed56x8' | 'ufixed56x16' | 'ufixed56x24' | 'ufixed56x32' | 'ufixed56x40' |
↪'ufixed56x48' | 'ufixed56x56' | 'ufixed56x64' | 'ufixed56x72' | 'ufixed56x80' |
↪'ufixed56x88' | 'ufixed56x96' | 'ufixed56x104' | 'ufixed56x112' | 'ufixed56x120' |
↪'ufixed56x128' | 'ufixed56x136' | 'ufixed56x144' | 'ufixed56x152' | 'ufixed56x160'␣
↪| 'ufixed56x168' | 'ufixed56x176' | 'ufixed56x184' | 'ufixed56x192' | 'ufixed56x200
↪' | 'ufixed64x8' | 'ufixed64x16' | 'ufixed64x24' | 'ufixed64x32' | 'ufixed64x40' |
↪'ufixed64x48' | 'ufixed64x56' | 'ufixed64x64' | 'ufixed64x72' | 'ufixed64x80' |
↪'ufixed64x88' | 'ufixed64x96' | 'ufixed64x104' | 'ufixed64x112' | 'ufixed64x120' |
↪'ufixed64x128' | 'ufixed64x136' | 'ufixed64x144' | 'ufixed64x152' | 'ufixed64x160'␣
↪| 'ufixed64x168' | 'ufixed64x176' | 'ufixed64x184' | 'ufixed64x192' | 'ufixed72x8'␣
↪| 'ufixed72x16' | 'ufixed72x24' | 'ufixed72x32' | 'ufixed72x40' | 'ufixed72x48' |
↪'ufixed72x56' | 'ufixed72x64' | 'ufixed72x72' | 'ufixed72x80' | 'ufixed72x88' |
↪'ufixed72x96' | 'ufixed72x104' | 'ufixed72x112' | 'ufixed72x120' | 'ufixed72x128' |
↪'ufixed72x136' | 'ufixed72x144' | 'ufixed72x152' | 'ufixed72x160' | 'ufixed72x168'␣
↪| 'ufixed72x176' | 'ufixed72x184' | 'ufixed80x8' | 'ufixed80x16' | 'ufixed80x24' |
↪'ufixed80x32' | 'ufixed80x40' | 'ufixed80x48' | 'ufixed80x56' | 'ufixed80x64' |
↪'ufixed80x72' | 'ufixed80x80' | 'ufixed80x88' | 'ufixed80x96' | 'ufixed80x104' |
↪'ufixed80x112' | 'ufixed80x120' | 'ufixed80x128' | 'ufixed80x136' | 'ufixed80x144'␣
↪| 'ufixed80x152' | 'ufixed80x160' | 'ufixed80x168' | 'ufixed80x176' | 'ufixed88x8'␣
↪| 'ufixed88x16' | 'ufixed88x24' | 'ufixed88x32' | 'ufixed88x40' | 'ufixed88x48' |
↪'ufixed88x56' | 'ufixed88x64' | 'ufixed88x72' | 'ufixed88x80' | 'ufixed88x88' |
↪'ufixed88x96' | 'ufixed88x104' | 'ufixed88x112' | 'ufixed88x120' | 'ufixed88x128' |
↪'ufixed88x136' | 'ufixed88x144' | 'ufixed88x152' | 'ufixed88x160' | 'ufixed88x168'␣
```

```
InlineAssemblyBlock = '{' AssemblyItem* '}'

AssemblyItem = Identifier | FunctionalAssemblyExpression | InlineAssemblyBlock |␣
↪AssemblyLocalBinding | AssemblyAssignment | AssemblyLabel | NumberLiteral |␣
↪StringLiteral | HexLiteral
AssemblyLocalBinding = 'let' Identifier ':=' FunctionalAssemblyExpression
AssemblyAssignment = ( Identifier ':=' FunctionalAssemblyExpression ) | ( '=:'␣
↪Identifier )
AssemblyLabel = Identifier ':'
FunctionalAssemblyExpression = Identifier '(' AssemblyItem? ( ',' AssemblyItem )* ')'
```

Consideraciones de seguridad

Aunque en general es bastante fácil hacer software que funcione como esperamos, es mucho más difícil de chequear que nadie lo pueda usar de alguna forma que **no** fue anticipada.

En solidity, esto es aún más importante ya que se pueden usar los contratos inteligentes para mover tokens, o posiblemente, cosas aún más valiosas. Además, cada ejecución de un contrato inteligente es pública, y a ello se suma que el código fuente muchas veces está disponible.

Por supuesto, siempre se tiene que considerar lo que está en juego: Puedes comparar un contrato inteligente con un servicio web que está abierto al público (y por lo tanto, también a gente malintencionada) y quizá de código abierto. Si sólo se guarda la lista de compras en ese servicio web, puede que no tengas que tener mucho cuidado, pero si accedes a tu cuenta bancaria usando ese servicio, deberías tener más cuidado.

Esta sección nombrará algunos errores comunes y recomendaciones de seguridad generales pero no puede, por supuesto, ser una lista completa. Además, recordar que incluso si tu contrato inteligente está libre de errores, el compilador o la plataforma puede que los tenga. Una lista de errores de seguridad públicamente conocidos del compilador puede encontrarse en: *lista de errores conocidos*, la lista también es legible por máquina. Nótese que hay una recompensa por encontrar errores (bug-bounty) que cubre el generador de código del compilador de Solidity.

Como siempre ocurre con la documentación de código abierto, por favor, ayúdanos a extender esta sección, ¡especialmente con algunos ejemplos!

Errores Comunes

Información privada y aleatoriedad

Todo lo que usas en un contrato inteligente es públicamente visible, incluso variables locales y variables de estado marcadas como `private`.

Es bastante complejo usar números aleatorios en contratos inteligentes si no quieres que los mineros puedan hacer trampa.

Reentrada

Cualquier interacción desde un contrato (A) con otro contrato (B) y cualquier transferencia de Ether, le da el control a ese contrato (B). Esto hace posible que B vuelva a llamar a A antes de terminar la interacción. Para dar un ejemplo, el siguiente código contiene un error (esto es sólo un snippet y no un contrato completo):

```
pragma solidity ^0.4.0;

// ESTE CONTRATO CONTIENE UN ERROR - NO USAR
contract Fund {
    /// Mapping de distribución de ether del contrato.
    mapping(address => uint) shares;
    /// Retira tu parte.
    function withdraw() {
        if (msg.sender.send(shares[msg.sender]))
            shares[msg.sender] = 0;
    }
}
```

El problema aquí no es tan grave por el límite de gas de la función `send`, pero aun así, expone una debilidad: una transferencia de Ether siempre incluye ejecución de código, así que el receptor puede ser un contrato que vuelve a llamar a `withdraw`. Esto le permitiría obtener múltiples devoluciones, y por lo tanto, vaciar el Ether del contrato.

Para evitar reentradas, puedes usar el orden Comprobaciones-Efectos-Interacciones como detallamos aquí:

```
pragma solidity ^0.4.11;

contract Fund {
    /// Mapping de distribución de ether del contrato.
    mapping(address => uint) shares;
    /// Retira tu parte.
    function withdraw() {
        var share = shares[msg.sender];
        shares[msg.sender] = 0;
        msg.sender.transfer(share);
    }
}
```

Nótese que las reentradas no sólo son un riesgo al transferir Ether, sino de cualquier ejecución de una función de otro contrato. Además, también tienes que considerar situaciones de multi-contrato. Un contrato ejecutado podría modificar el estado de otro contrato del cual dependes.

Límite de gas y bucles

Los bucles que no tienen un número fijo de iteraciones, por ejemplo, bucles que dependen de valores de almacenamiento, tienen que usarse con cuidado: Dado el límite de gas del bloque, las transacciones sólo pueden consumir una cierta cantidad de gas. Ya sea explícitamente, o por una operación normal, el número de iteraciones en un bucle puede crecer más allá del límite de gas, lo que puede causar que el contrato se detenga por completo en un cierto punto. Esto no se aplica a funciones `constant` que sólo se llaman para leer información de la blockchain. Pero aun así, estas funciones pueden ser llamadas por otros contratos como parte de operaciones on-chain y detenerlos a ellos. Por favor, sé explícito con estos casos en la documentación de tus contratos.

Envío y recibo de Ether

- Ni los contratos ni las "cuentas externas", son actualmente capaces de prevenir que alguien les envíe Ether. Los contratos pueden reaccionar y rechazar una transferencia normal, pero hay maneras de mover Ether sin provocar la ejecución de código. Una manera es simplemente "minando" a la cuenta del contrato y la otra es usando `selfdestruct(x)`.
- Si un contrato recibe Ether (sin que se llame a ninguna función), se ejecuta la función fallback. Si no tiene una función fallback, el Ether será rechazado (lanzando una excepción). Durante la ejecución de la función fallback,

el contrato sólo puede depender del "estipendio de gas" (2300 gas) que tiene disponible en ese momento. Este estipendio no es suficiente para acceder al almacenamiento de ninguna forma. Para asegurarte de que tu contrato pueda recibir Ether de ese modo, verifica los requerimientos de gas de la función fallback (por ejemplo, en la sección de "details" de Remix).

- Hay una manera de enviar más gas al contrato receptor usando `addr.call.value(x)()`. Esto es esencialmente lo mismo que `addr.transfer(x)`, solo que envía todo el gas restante y permite la posibilidad al receptor de realizar acciones más caras (y sólo devuelve un código de error y no propaga automáticamente el error). Dentro de estas acciones se incluyen llamar de nuevo al contrato emisor u otros cambios de estado que no fueron previstos. Permite más flexibilidad para usuarios honestos pero también para los usuarios maliciosos.
- Si quieres enviar Ether usando `address.transfer`, hay ciertos detalles que hay que saber:

  1. Si el receptor es un contrato, se ejecutará la función fallback, lo cual puede llamar de vuelta al contrato que envía Ether. 2. El envío de Ether puede fallar debido a la profundidad de la llamada subiendo por encima 1024. Puesto que el que llama tiene el control total de la profundidad de la llamada, pueden forzar la transferencia para que falle; tened en consideración esta posibilidad o utilizad siempre `send` y aseguraos siempre de revisar el valor de retorno. O mejor aún, escribid el contrato siguiendo un patrón de modo que sea el receptor el que tenga que sacar el Ether. 3. El envío de Ether también puede fallar porque la ejecución del contrato receptor necesita más gas que la cantidad asignada (OOG, por sus siglas en inglés "Out of Gas"). Esto ocurre porque explícitamente se usó `require`, `assert`, `revert`, `throw`, o simplemente porque la operación es demasiado cara. Si usas `transfer` o `send` comprobando el valor de retorno, esto puede hacer que el receptor bloquee el progreso en el contrato emisor.Pero volviendo a insistir, aquí lo mejor es usar un *patrón de "retirada" en vez de un "orden de envío"*.

Profundidad de la pila de llamadas (Callstack)

Las llamadas externas a funciones pueden fallar en cualquier momento al exceder la profundidad máxima de la pila de llamadas de 1024. En tales situaciones, Solidity lanza una excepción. Los usuarios maliciosos podrían forzar la profundidad de la pila a un valor alto antes de interactuar con el contrato.

Ten en cuenta que `.send()` **no** lanza una excepción si la pila está vacía, sino que retorna `false` en ese caso. Las funciones de bajo nivel como `.call()`, `.callcode()` `.delegatecall()` se comportan de la misma manera.

tx.origin

No uses nunca tx.origin para dar autorización. Digamos que tienes un contrato de cartera como este:

```
pragma solidity ^0.4.11;

// ESTE CONTRATO CONTIENE UN ERROR - NO USAR
contract TxUserWallet {
    address owner;

    function TxUserWallet() {
        owner = msg.sender;
    }

    function transferTo(address dest, uint amount) {
        require(tx.origin == owner);
        dest.transfer(amount);
    }
}
```

Ahora alguien te engaña para que le envíes Ether a esta cartera maliciosa:

```
pragma solidity ^0.4.0;

contract TxAttackWallet {
    address owner;

    function TxAttackWallet() {
        owner = msg.sender;
    }

    function() {
        TxUserWallet(msg.sender).transferTo(owner, msg.sender.balance);
    }
}
```

Si tu cartera hubiera comprobado `msg.sender` para darle autorización, recibiría la cuenta de la cartera atacante, en vez de la cartera del 'owner'. Pero al chequear `tx.origin`, recibe la cuenta original que envió la transacción, que es la cuenta owner. La cartera atacante inmediatamente vacía todos tus fondos.

Detalles Menores

- En `for (var i = 0; i < arrayName.length; i++) { ... }`, el tipo de `i` será `uint8`, porque este es el tipo más pequeño que es requerido para guardar el valor `0`. Si el vector (array) tiene más de 255 elementos, el bucle no se terminará.
- La palabra reservada `constant` para funciones no es actualmente forzada por el compilador. Tampoco está forzada por la EVM, de modo que una función de un contrato que "pretenda" ser constante, todava podría hacer cambios al estado.
- Los tipos que no utilizan totalmente los 32 bytes pueden contener basura en los bits más significativos. Esto es especialmente importante si se accede a `msg.data` ya que supone un riesgo de maleabilidad: Puedes crear transacciones que llaman una función `f(uint8 x)` con un argumento de tipo byte de `0xff000001` y `0x00000001`. Ambos se pasarán al contrato y ambos se verán como el número `1`. Pero `msg.data` es diferente, así que si se usa `keccak256(msg.data)` para algo, tendrás resultados diferentes.

Recomendaciones

Restringir la cantidad de Ether

Restringir la cantidad de Ether (u otros tokens) que puedan ser almacenados en un contrato inteligente. Si el código fuente, el compilador o la plataforma tiene un error, estos fondos podrían perderse. Si quieres limitar la pérdida, limita la cantidad de Ether.

Pequeño y modular

Mantén tus contratos pequeños y fáciles de entender. Separa las funcionalidades no relacionadas en otros contratos o en librerías. Se pueden aplicar las recomendaciones estándar de calidad de código: Limitar la cantidad de variables locales, limitar la longitud de las funciones, etc. Documenta tus funciones para que otros puedan ver cuál era la intención del código y para ver si hace algo diferente de lo que pretendía.

Usa el orden Comprobaciones-Efectos-Interacciones

La mayoría de las funciones primero ejecutan algunos chequeos (¿quién ha llamado a la función?, ¿los argumentos están en el rango?, ¿mandaron suficiente Ether? ¿La cuenta tiene tokens?, etc.). Estos chequeos deben de hacerse primero.

Como segundo paso, si se pasaron todos los chequeos, se deben ejecutar los efectos a las variables de estado del contrato actual. La interacción con otros contratos debe hacerse como último paso en cualquier función.

Al principio, algunos contratos retrasaban la ejecución de los efectos y esperaban a que una función externa devolviera un estado sin errores. Esto es un error grave ya que se puede hacer un reingreso, como explicamos arriba.

También hay que tener en cuenta que las llamadas a contratos conocidos pueden a su vez causar llamadas a otros contratos no conocidos, así que siempre es mejor aplicar este orden.

Incluir un modo a prueba de fallos

Aunque hacer que tu sistema sea completamente descentralizado eliminará cualquier intermediario, puede que sea una buena idea, especialmente para nuevo código, incluir un sistema a prueba de fallos:

Puedes agregar una función a tu contrato que realize algunas comprobaciones internas como "¿se ha filtrado Ether?", "¿es igual la suma de los tokens al balance de la cuenta?" o cosas similares. Recordad que no se puede usar mucho gas para eso, así que ayuda mediante computaciones off-chain podrían ser necesarias.

Si los chequeos fallan, el contrato automáticamente cambia a modo a prueba de fallos, donde, por ejemplo, se desactivan muchas funciones, da el control a una entidad tercera de confianza o se convierte en un contrato "devuélveme mi dinero".

Verificación formal

Usando verificación formal, es posible realizar pruebas matemáticas automatizadas de que el código sigue una cierta especificación formal. La especificación aún es formal (como el código fuente), pero normalmente mucho más simple. Hay un prototipo en Solidity que realiza verificación formal y pronto se documentará mejor.

Ten en cuenta que la verificación formal en sí misma sólo puede ayudarte a entender la diferencia entre lo que hiciste (la especificación) y cómo lo hiciste (la implementación real). Aún necesitas chequear si la especificación es lo que querías y que no hayas olvidado efectos inesperados de ello.

Uso del compilador

Utilizar el compilador de línea de comandos

Uno de los objetivos de compilación del repositorio de Solidity es `solc`, el compilador de línea de comandos de solidity.

Utilizando `solc --help` proporciona una explicación de todas las opciones. El compilador puede producir varias salidas, desde binarios simples y ensamblador sobre un árbol de sintaxis abstracto (árbol de análisis) hasta estimaciones de uso de gas. Si sólo deseas compilar un único archivo, lo ejecutas como `solc --bin sourceFile.sol` y se imprimirá el binario. Antes de implementar tu contrato, activa el optimizador mientras compilas usando `solc --optimize --bin sourceFile.sol`. Si deseas obtener algunas de las variantes de salida más avanzadas de `solc`, probablemente sea mejor decirle que salga todo por ficheros separados usando‘‘solc -o outputDirectory –bin –ast –asm sourceFile.sol‘‘.

El compilador de línea de comandos leerá automáticamente los archivos importados del sistema de archivos, aunque es posible proporcionar un redireccionamiento de ruta utilizando `prefix=path` de la siguiente manera:

```
solc github.com/ethereum/dapp-bin/
↪=/usr/local/lib/dapp-bin/ =/usr/local/lib/fallback file.sol
```

Esencialmente esto instruye al compilador a buscar cualquier cosa que empiece con `github.com/ethereum/dapp-bin/` bajo `/usr/local/lib/dapp-bin` y si no localiza el fichero ahí, mirará en `/usr/local/lib/fallback` (el prefijo vacío siempre coincide). `solc` no leerá ficheros del sistema de ficheros que se encuentren fuera de los objetivos de reasignación y fuera de los directorios donde se especifica explícitamente la fuente de ficheros donde residen, con lo cual cosas como `import /etc/passwd";` sólo funcionan si le añades `=/` como una reasignacion.

Si hay coincidencias múltiples debido a reasignaciones, se selecciona el prefijo común más largo.

Por razones de seguridad, el compilador tiene restricciones sobre a qué directorios puede acceder. Las rutas de acceso (y sus subdirectorios) de los archivos de origen especificados en la línea de comandos y las rutas definidas por las reasignaciones se permiten para las instrucciones de importación, pero todo lo demás se rechaza. Se pueden permitir rutas adicionales (y sus subdirectorios) a través de la opción `--allow-paths /sample/path,/another/sample/path`.

Si tus contratos usan *bibliotecas*, notarás que el bytecode contiene subcadenas del tipo `__LibraryName_____`. Puedes utilizar `solc` como un enlazador, lo que significa que insertará las direcciones de la biblioteca en esos puntos:

Agregue `--libraries "Math:0x12345678901234567890 Heap:0xabcdef0123456"` a su comando para proporcionar una dirección para cada biblioteca o almacenar la cadena en un archivo (una biblioteca por línea) y ejecutar ‘‘ solc‘‘ usando ‘‘ –libraries fileName‘‘.

Si `solc` se llama con la opción `--link`, todos los archivos de entrada se interpretan como binarios desvinculados (codificados en hexadecimal) en el `__LibraryName_____`-format dado anteriormente y están enlazados in situ (si la entrada se lee desde stdin, se escribe en stdout). Todas las opciones excepto `--libraries` son ignoradas (incluyendo `-o`) en este caso.

Si `solc` se llama con la opción `--standard-json`, esperará una entrada JSON (como se explica a continuación) en la entrada estándar, y devolverá una salida JSON a la salida estándar.

Compilador de entrada y salida JSON Descripción

Estos formatos JSON son utilizados por la API del compilador y están disponibles a través de `solc`. Estos están sujetos a cambios, algunos campos son opcionales (como se ha señalado), pero está dirigido a hacer sólo cambios compatibles hacia atrás.

La API del compilador espera una entrada con formato JSON y genera el resultado de la compilación en una salida con formato JSON.

Por supuesto, los comentarios no se permiten y se utilizan aquí sólo con fines explicativos.

Descripción de entrada

```
{
    // Requerido: Lenguaje del código fuente, tal como "Solidity", "serpent", "lll",
↪"assembly", etc.
    language: "Solidity",
    // Requerido
    sources:
    {
    // Las claves aquí son los nombres "globales" de los ficheros fuente,
    // las importaciones pueden utilizar otros ficheros mediante remappings (vér más␣
↪abajo).
```

```
    "myFile.sol":
    {
      // Opcional: keccak256 hash del fichero fuente
      // Se utiliza para verificar el contenido recuperado si se importa a través de␣
↪URLs.
      "keccak256": "0x123...",
      // Requerido (a menos que se use "contenido", ver abajo): URL (s) al fichero␣
↪fuente.
      // URL(s) deben ser importadas en este orden y el resultado debe ser verificado␣
↪contra el fichero
      // keccak256 hash (si está disponible). Si el hash no coincide con ninguno de␣
↪los
      // URL(s) resultado en el éxito, un error debe ser elevado.
      "urls":
      [
        "bzzr://56ab...",
        "ipfs://Qma...",
        "file:///tmp/path/to/file.sol"
      ]
    },
    "mortal":
    {
      // Opcional: keccak256 hash del fichero fuente
      "keccak256": "0x234...",
      // Requerido (a menos que se use "urls"): contenido literal del fichero fuente
      "content": "contract mortal is owned { function kill() { if (msg.sender ==␣
↪owner) selfdestruct(owner); } }"
    }
    },
    // Opcional
    settings:
    {
    // Opcional: Lista ordenada de remappings
    remappings: [ ":g/dir" ],
    // Opcional: Ajustes de optimización (activación de valores predeterminados a␣
↪false)
    optimizador: {
      enabled: true,
      runs: 500
    },
    // Configuración de metadatos (opcional)
    metadata: {
      // Usar sólo contenido literal y no URLs (falso por defecto)
      useLiteralContent: true
    },
    // Direcciones de las bibliotecas. Si no todas las bibliotecas se dan aquí, puede␣
↪resultar con objetos no vinculados cuyos datos de salida son diferentes.
    libraries: {
      // La clave superior es el nombre del fichero fuente donde se utiliza la␣
↪biblioteca.
      // Si se utiliza remappings, este fichero fuente debe coincidir con la ruta␣
↪global después de que se hayan aplicado los remappings.
      // Si esta clave es una cadena vacía, se refiere a un nivel global.

      "myFile.sol": {
        "MyLib": "0x123123..."
      }
    }
```

```
    // Para seleccionar las salidas deseadas se puede utilizar lo siguiente.
    // Si este campo se omite, el compilador se carga y comprueba el tipo, pero no␣
↪genera ninguna salida aparte de errores.
    // La clave de primer nivel es el nombre del fichero y la segunda es el nombre␣
↪del contrato, donde el nombre vacío del contrato se refiere al fichero mismo,
    // mientras que la estrella se refiere a todos los contratos.
    //
    // Las clases de mensajes disponibles son las siguientes:
    //   abi - ABI
    //   ast - AST de todos los ficheros fuente
    //   legacyAST - legado AST de todos los ficheros fuente
    //   devdoc - Documentación para desarrolladores (natspec)
    //   userdoc - Documentación de usuario (natspec)
    //   metadata - Metadatos
    //   ir - Nuevo formato de ensamblador antes del desazucarado
    //   evm.assembly - Nuevo formato de ensamblador después del desazucarado
    //   evm.legacyAssembly - Formato de ensamblador antiguo en JSON
    //   evm.bytecode.object - Objeto bytecode
    //   evm.bytecode.opcodes - Lista de Opcodes
    //   evm.bytecode.sourceMap - Asignación de fuentes (útil para depuración)
    //   evm.bytecode.linkReferences - Referencias de enlace (si es objeto no␣
↪enlazado)
    //   evm.deployedBytecode* - Desplegado bytecode (tiene las mismas opciones que␣
↪evm.bytecode)
    //   evm.methodIdentifiers - La lista de funciones de hashes
    //   evm.gasEstimates - Funcion de estimación de gas
    //   ewasm.wast - eWASM S-formato de expresiones (no compatible por el momento)
    //   ewasm.wasm - eWASM formato binario (no compatible por el momento)
    //
    // Ten en cuenta que el uso de `evm`, `evm.bytecode`, `ewasm`, etc. seleccionara␣
↪cada
    // parte objetiva de esa salida.
    //
    outputSelection: {
      // Habilita los metadatos y las salidas de bytecode de cada contrato.
      "*": {
        "*": [ "metadata", "evm.bytecode" ]
      },
      // Habilitar la salida abi y opcodes de MyContract definida en el fichero def.
      "def": {
        "MyContract": [ "abi", "evm.opcodes" ]
      },
      // Habilita la salida del mapa de fuentes de cada contrato individual.
      "*": {
        "*": [ "evm.sourceMap" ]
      },
      // Habilita la salida AST heredada de cada archivo.
      "*": {
        "": [ "legacyAST" ]
      }
    }
  }
}
```

Descripción de los mensajes de salida

```
{
    // Opcional: no está presente si no se han encontrado errores/avisos
    errors: [
    {
      // Opcional: Ubicación dentro del fichero fuente.
      sourceLocation: {
        file: "sourceFile.sol",
        start: 0,
        end: 100
      },
      // Obligatorio: Tipo de error, como "TypeError", "InternalCompilerError",
↪"Exception", etc
      type: "TypeError",
      // Obligatorio: Componente donde se originó el error, como "general", "ewasm",␣
↪etc.
      component: "general",
      // Obligatorio ("error" o "warning")
      severity: "error",
      // Obligatorio
      message: "Invalid keyword"
      // Opcional: el mensaje formateado con la ubicación de origen
      formattedMessage: "sourceFile.sol:100: Invalid keyword"
    }
    ],
    // Contiene las salidas a nivel de fichero. Puede ser limitado/filtrado por los␣
↪ajustes de outputSelection.
    sources: {
    "sourceFile.sol": {
      // Identificador (utilizado en los mapas fuente)
      id: 1,
      // El objeto AST
      ast: {},
      // El objeto legado AST
      legacyAST: {}
    }
    },
    // Contiene las salidas contract-level. Puede ser limitado/filtrado por los␣
↪ajustes de outputSelection.
    contracts: {
    "sourceFile.sol": {
      // Si el idioma utilizado no tiene nombres de contrato, este campo debe ser␣
↪igual a una cadena vacía.
      "ContractName": {
        // El ABI del contrato de Ethereum. Si está vacío, se representa como una␣
↪matriz vacía.
        // Ver https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI
        abi: [],
        // Ver la documentación de salida de metadatos (cadena JSON seriada)
        metadata: "{...}",
        // Documentación de usuario (natspec)
        userdoc: {},
        // Documentación para desarrolladores (natspec)
        devdoc: {},
        // Representación intermedia (cadena)
        ir: "",
        // EVM-salidas relacionadas
```

```
        evm: {
          // Ensamblador (cadena)
          assembly: "",
          // Ensamblador antiguo (objeto)
          legacyAssembly: {},
          // Bytecode y detalles relacionados.
          bytecode: {
            // El bytecode como una cadena hexadecimal.
            object: "00fe",
            // Lista de Opcodes (cadena)
            opcodes: "",
            // El mapeo de fuentes como una cadena. Ve la definición del mapeo de␣
↪fuentes.
            sourceMap: "",
            // Si se da, este es un objeto no ligado.
            linkReferences: {
              "libraryFile.sol": {
                // Traslados de bytes en el bytecode. El enlace sustituye a los 20␣
↪bytes que se encuentran allí.
                "Library1": [
                  { start: 0, length: 20 },
                  { start: 200, length: 20 }
                ]
              }
            }
          },
          // La misma disposición que la anterior.
          deployedBytecode: { },
          // La lista de hashes de función
          methodIdentifiers: {
            "delegate(address)": "5c19a95c"
          },
          // Estimación de gas de la función
          gasEstimates: {
            creation: {
              codeDepositCost: "420000",
              executionCost: "infinite",
              totalCost: "infinite"
            },
            external: {
              "delegate(address)": "25000"
            },
            internal: {
              "heavyLifting()": "infinite"
            }
          }
        },
        // eWASM resultados relacionados
        ewasm: {
          // S-formato de expressiones
          wast: "",
          // Formato Binario (cadena hexagonal)
          wasm: ""
        }
      }
    }
  }
}
```

Especificación de Application Binary Interface

Diseño básico

La Application Binary Interface (Interfaz Binaria de Aplicación) o ABI es el modo estándar de interactuar con contratos en el ecosistema Ethereum, tanto desde fuera de la blockchain como en interacciones contrato-contrato. Los datos se codifican siguiendo su tipo acorde a esta especificación.

Asumimos que la Application Binary Interface (ABI) está fuertemente tipada, es conocida en tiempo de compilación y es estática. No se van a proveer mecanismos de introspección. Además, afirmamos que los contratos tendrán las definiciones de la interfaz de cada contrato que vayan a llamar en tiempo de compilación.

Esta especificación no abarca los contratos cuya interfaz sea dinámica o conocida exclusivamente en tiempo de ejecución. Estos casos, de volverse importantes, podrían manejarse adecuadamente como servicios construidos dentro del ecosistema Ethereum.

Función Selector

Los primeros cuatro bytes de los datos de una llamada a una función especifican la función a llamar. Se trata de los primeros (los más a la izquierda, los más extremos por orden) cuatro bytes del hash Keccak (SHA-3) de la firma de la función. La firma se define como la expresión canónica del prototipo básico como puede ser el nombre de la función con la lista de parámetros entre paréntesis. Los tipos de parámetros se separan por comas, no por espacios.

Codificación de argumentos

A partir del quinto byte, prosiguen los argumentos codificados. Esta codificación es también usada en otros sitios, por ejemplo, en los valores de retorno y también en los argumentos de eventos, sin los cuatro bytes especificando la función.

Tipos

Los tipos elementales existentes son:

- *uint<M>*: enteros sin signo de *M* bits, *0 < M <= 256*, *M % 8 == 0*. Ejemplos: *uint32*, *uint8*, *uint256*.
- *int<M>*: enteros con signo complemento a dos de *M* bits, *0 < M <= 256*, *M % 8 == 0*.
- *address*: equivalente a *uint160*, exceptuando la interpretación asumida y la tipología del lenguaje.
- *uint*, *int*: sinónimos de *uint256*, *int256* respectivamente (no para ser usados para computar la función selector).
- *bool*: equivalente a *uint8* restringido a los valores 0 y 1.
- *fixed<M>x<N>*: número decimal con signo y formato decimal fijo de *M* bits, *0 < M <= 256*, *M % 8 ==0*, y *0 < N <= 80*, que denota el valor *v* como *v / (10 ** N)*.
- *ufixed<M>x<N>*: variante sin signo de *fixed<M>x<N>*.
- *fixed*, *ufixed*: sinónimos de *fixed128x19*, *ufixed128x19* respectivamente (no para ser usados para computar la función selector).
- *bytes<M>*: tipo binario de *M* bytes, *0 < M <= 32*.
- *function*: equivalente a *bytes24*: un address, seguido de la función selector

El siguiente array, de tipo fijo, existente es:

- *<type>[M]*: un array de longitud fija del tipo de longitud fija dada.

Los siguientes tipos de tamaño no fijo existentes son:

- *bytes*: secuencia de bytes de tamaño dinámico.
- *string*: string unicode de tamaño dinámico codificado como UTF-8.
- *<type>[]*: array de longitud variable del tipo de longitud fija dada.

Los distintos tipos se pueden combinar en structs anónimos cerrando un número finito no negativo de ellos entre paréntesis, separados por comas:

- *(T1,T2,...,Tn)*: struct anónimo (tupla ordenada) consistente de los tipos *T1*, ..., *Tn*, *n >= 0*

Es posible formar structs de structs, arrays de structs, etc.

Especificación formal de la codificación

Vamos a especificar formalmente la codificación, de tal forma que tendrá las siguientes propiedades, que son especialmente útiles si los argumentos son arrays anidados:

Propiedades:

1. El número de lecturas necesarias para acceder a un valor es, como mucho, equivalente a la máxima profundidad del array. Por ejemplo, cuatro lecturas se requieren para obtener *a_i[k][l][r]*. En una versión previa de la ABI, el número de lecturas escalaba linealmente con el número total de parámetros dinámicos en el peor caso.
2. Los datos de una variable o elemento de un array no se intercalan con otros datos y son recolocables. Por ejemplo, sólo usan "addresses" relativos.

Distinguimos entre tipos estáticos y dinámicos. Los estáticos se codifican insitu y los dinámicos se codifican en una posición asignada separadamente después del bloque actual.

Definición: Los siguientes tipos se llaman "dinámicos": * *bytes* * *string* * *T[]* para cada *T* * *T[k]* para cualquier dinámico *T* y todo *k > 0*

Todo el resto de tipos son "estáticos".

Definición: *len(a)* es el número de bytes en un string binario *a*. El tipo de *len(a)* se presume como *uint256*.

Definimos *enc*, la codificación actual, como un mapping de valores de tipos de la ABI a string binarios como *len(enc(X))* depende del valor de *X* si y solo si el tipo de *X* es dinámico.

Definición: Para cada valor de ABI *X*, definimos recursivamente *enc(X)*, dependiendo del tipo de *X* siendo

- *(T1,...,Tk)* para *k >= 0* y cualquier tipo *T1*, ..., *Tk*

  enc(X) = head(X(1)) ... head(X(k-1)) tail(X(0)) ... tail(X(k-1))

  donde *X(i)* es el *ith* componente del valor, y *head* y *tail* son definidos por *Ti* siendo un tipo estático como

  head(X(i)) = enc(X(i)) y *tail(X(i)) = ""* (el string vacío)

  y como

  head(X(i)) = enc(len(head(X(0)) ... head(X(k-1)) tail(X(0)) ... tail(X(i-1)))) tail(X(i)) = enc(X(i))

  en otros casos como si, por ejemplo, *Ti* es un tipo dinámico.

  Hay que tener en cuenta que en el caso dinámico, *head(X(i))* está bien definido ya que las longitudes de las partes de head sólo dependen de los tipos y no de los valores. Su valor es el offset del principio de *tail(X(i))* relativo al comienzo de *enc(X)*.

- *T[k]* para cada *T* y *k*:

  enc(X) = enc((X[0], ..., X[k-1]))

  como ejemplo, es codificado como si fuera un struct anónimo de *k* elementos del mismo tipo.

- *T[]* donde *X* tiene *k* elementos (*k* se presume que es del tipo *uint256*):

  enc(X) = enc(k) enc([X[1], ..., X[k]])

  Otro ejemplo codificado como si fuera un array estático de tamaño *k*, prefijado con el número de elementos.

- *bytes*, de longitud *k* (que se presume que es del tipo *uint256*):

  enc(X) = enc(k) pad_right(X). Por ejemplo, el número de bytes es codificado como un *uint256* seguido del valor actual de *X* como una secuencia de bytes, seguido por el número mínimo de bytes-cero como que *len(enc(X))* es un múltiplo de 32.

- *string*:

  enc(X) = enc(enc_utf8(X)), en este caso *X* se codifica como utf-8 y su valor se interpreta como de tipo *bytes* y codificado posteriormente. Hay que tener en cuenta que la longitud usada en la subsecuente codificación es el número de bytes del string codificado como utf-8, no su número de caracteres.

- *uint<M>*: *enc(X)* es el mayor extremo de la codificación de *X*, rellenado en el lado de orden mayor (izquierda) con bytes cero de tal forma que la longitud acabe siendo de 32 bytes.

- *address*: como en el caso de *uint160*

- *int<M>*: *enc(X)* es el complemento a dos de mayor extremo en la codificación de *X*, rellenado en el lado de mayor orden (izquierda) con *0xff* para *X* negativo y bytes cero para *X* positivo de tal forma que la longitud final sea un múltiplo de 32 bytes.

- *bool*: como en el caso de *uint8*, donde *1* se usa para *true* y *0* para *false*

- *fixed<M>x<N>*: *enc(X)* es *enc(X * 10**N)* donde *X * 10**N* se interpreta como un *int256*.

- *fixed*: como en el caso de *fixed128x19*

- *ufixed<M>x<N>*: *enc(X)* es *enc(X * 10**N)* donde *X * 10**N* se interpreta como un *uint256*.

- *ufixed*: como en el caso de *ufixed128x19*

- *bytes<M>*: *enc(X)* es la secuencia de bytes en *X* rellenado con bytes cero hasta una longitud de 32.

Resaltar que para cada *X*, *len(enc(X))* es un múltiplo de 32.

Función Selector y codificación de argumentos

Siempre, una llamada a la función *f* con parámetros *a_1, ..., a_n* se codifican como

function_selector(f) enc((a_1, ..., a_n))

y los valores de retorno *v_1, ..., v_k* de *f* son codificados como

enc((v_1, ..., v_k))

p.ej.: los valores se combinan en struct anónimos y codificados.

Ejemplos

Para el siguiente contrato:

```
contract Foo {
  function bar(bytes3[2] xy) {}
  function baz(uint32 x, bool y) returns (bool r) { r = x > 32 || y; }
  function sam(bytes name, bool z, uint[] data) {}
}
```

Para nuestro ejemplo *Foo*, si queremos llamar a *baz* pasando como parámetros *69* y *true*, emplearíamos 68 bytes en total, que se podrían dividir en las siguientes partes:

- *0xcdcd77c0*: el ID del método. Se deriva como los 4 primeros bytes del hash Keccak en ASCII de la firma *baz(uint32,bool)*.
- *0x0000000000000000000000000000000000000000000000000000000000000045*: el primer parámetro, un uint32 de valor *69* rellenado hasta 32 bytes
- *0x0000000000000000000000000000000000000000000000000000000000000001*: el segundo parámetro - boolean *true*, rellenado hasta 32 bytes

En total:

```
0xcdcd77c000000000000000000000000000000000000000000000000000000000000000450000000000000000000000000000000
```

Devuelve un *bool* simple. Si, por ejemplo, devolviese *false*, su salida sería un array de byte sencillo *0x0000000000000000000000000000000000000000000000000000000000000000*, un único bool.

Si quisiéramos llamar a *bar* con el argumento *["abc", "def"]*, pasaríamos 68 bytes en total, divido en:

- *0xfce353f6*: el ID del método. Este se deriva de la firma *bar(bytes3[2])*.
- *0x6162630000000000000000000000000000000000000000000000000000000000*: La primera parte del primer parámetro, un valor *bytes3 "abc"* (alineado a la izquierda).
- *0x6465660000000000000000000000000000000000000000000000000000000000*: La segunda parte del primer parámetro, un valor *bytes3 "def"* (alineado a la izquierda).

En total:

```
0xfce353f661626300000000000000000000000000000000000000000000000000000000006465660000000000000000000000
```

Si quisiéramos llamar a *sam* con los argumentos *"dave"*, *true* y *[1,2,3]*, pasaríamos 292 bytes en total, dividido en: - *0xa5643bf2*: el ID del método. Este se deriva de la firma *sam(bytes,bool,uint256[])*. Aquí *uint* se reemplaza por su representación canónica *uint256*. - *0x0000000000000000000000000000000000000000000000000000000000000060*: La localización de la parte de datos del primer parámetro (tipo dinámico), medido en bytes desde el principio del bloque de argumentos. En este caso, *0x60*. - *0x0000000000000000000000000000000000000000000000000000000000000001*: el segundo parámetro: boolean verdadero. - *0x00000000000000000000000000000000000000000000000000000000000000a0*: La localización de la parte de datos del tercer parámetro (tipo dinámico), medido en bytes. En este caso, *0xa0*. - *0x0000000000000000000000000000000000000000000000000000000000000004*: La parte de datos del primer argumento, comienza con la longitud del array de bytes en elementos, en este caso, 4. - *0x6461766500000000000000000000000000000000000000000000000000000000*: Los contenidos del primer argumento: el UTF-8 (equivalente a ASCII en este caso) codificación de *"dave"*, rellenado hasta 32 bytes por la derecha. - *0x0000000000000000000000000000000000000000000000000000000000000003*: La parte de datos del tercer argumento, comenzando con la longitud del array en elementos, en este caso, 3. - *0x0000000000000000000000000000000000000000000000000000000000000001*: la primera entrada del tercer parámetro. - *0x0000000000000000000000000000000000000000000000000000000000000002*: la segunda entrada del tercer parámetro. - *0x0000000000000000000000000000000000000000000000000000000000000003*: la tercera entrada del tercer parámetro.

En total:

```
0xa5643bf20000000000000000000000000000000000000000000000000000000000000060000000000000000000000000000
```

Uso de tipos dinámicos

Una llamada a una función con la firma *f(uint,uint32[],bytes10,bytes)* con valores *(0x123, [0x456, 0x789], “1234567890”, “Hello, world!”)* se codifica de la siguiente manera:

Obtenemos los primeros cuatro bytes de *sha3(“f(uint256,uint32[],bytes10,bytes)”)*, p.ej.: *0x8be65246*. Entonces codificamos las cabeceras de los cuatro argumentos. Para los tipos estáticos *uint256* y *bytes10*, estos son los valores que queremos pasar directamente, mientras que para los tipos dinámicos *uint32[]* y *bytes*, usamos el offset en bytes hasta el inicio de su área de datos, contando desde el comienzo de la codificación del valor (p.ej.: sin contar los primeros cuatro bytes que contienen el hash de la firma de la función). Estos son:

- *0x0000000000000000000000000000000000000000000000000000000000000123* (*0x123* rellenado hasta 32 bytes)
- *0x0000000000000000000000000000000000000000000000000000000000000080* (offset del inicio de la parte de datos del segundo parámetro, 4*32 bytes, exactamente el tamaño de la parte de la cabecera)
- *0x3132333435363738393000000000000000000000000000000000000000000000* (*“1234567890”* rellenado hasta 32 bytes por la derecha)
- *0x00000000000000000000000000000000000000000000000000000000000000e0* (offset del comienzo de la parte de datos del cuarto parámetro = offset del inicio de la parte de datos del primer parámetro dinámico + tamaño de la parte de datos del primer parámetro dinámico = 4*32 + 3*32 (ver abajo))

Después de esto, la parte de datos del primer argumento dinámico, *[0x456, 0x789]* sigue así:

- *0x0000000000000000000000000000000000000000000000000000000000000002* (número de elementos del array, 2)
- *0x0000000000000000000000000000000000000000000000000000000000000456* (primer elemento)
- *0x0000000000000000000000000000000000000000000000000000000000000789* (segundo elemento)

Finalmente, codificamos la parte de datos del segundo argumento dinámico, *“Hello, world!”*:

- *0x000000000000000000000000000000000000000000000000000000000000000d* (número de elementos (bytes en este caso): 13)
- *0x48656c6c6f2c20776f726c642100000000000000000000000000000000000000* (*“Hello, world!”* rellenado hasta 32 bytes por la derecha)

Todo junto, la codificación es (nueva línea después de la función selector y cada 32-bytes por claridad):

```
0x8be65246
  0000000000000000000000000000000000000000000000000000000000000123
  0000000000000000000000000000000000000000000000000000000000000080
  3132333435363738393000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000000000e0
  0000000000000000000000000000000000000000000000000000000000000002
  0000000000000000000000000000000000000000000000000000000000000456
  0000000000000000000000000000000000000000000000000000000000000789
  000000000000000000000000000000000000000000000000000000000000000d
  48656c6c6f2c20776f726c642100000000000000000000000000000000000000
```

Eventos

Los eventos son una abstracción del protocolo de monitorización de eventos de Ethereum. Las entradas de log proveen la dirección del contrato, una cadena de máximo cuatro tópicos y algún dato binario de longitud arbitraria. Los eventos apalancan la función ABI existente para poder interpretarla (junto con una especificación de interfaz) como una estructura apropiada.

Dado un nombre de evento y una serie de parámetros de evento, los separamos en dos sub-series: los que están indexados y los que no. Los indexados, cuyo número podría llegar hasta tres, se usan junto al hash Keccack de la firma del evento para formar los tópicos de la entrada de log. Los no indexados forman el array de bytes del evento.

En efecto, una entrada de log que usa esta ABI se define como:

- *address*: la dirección del contrato (intrínsecamente provista por Ethereum);
- *topics[0]*: *keccak(EVENT_NAME+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")")* (*canonical_type_of* es una función que simplemente devuelve el tipo canónico del argumento dado, p.ej.: para *uint indexed foo*, devolvería *uint256*). Si el evento se declara como *anonymous* no se genera *topics[0]*;
- *topics[n]*: *EVENT_INDEXED_ARGS[n - 1]* (*EVENT_INDEXED_ARGS* es la serie de *EVENT_ARGS* que están indexados);
- *data*: *abi_serialise(EVENT_NON_INDEXED_ARGS)* (*EVENT_NON_INDEXED_ARGS* es la serie de *EVENT_ARGS* que no están indexados, *abi_serialise* es la función de serialización ABI usada para devolver una serie de valores tipificados desde una función, como se detalla abajo).

JSON

El formato JSON para la interfaz de un contrato viene dada por un array de descripciones de función y/o evento. Una descripción de función es un objeto JSON con los siguientes campos:

- *type*: *"function"*, *"constructor"*, o *"fallback"* (el *función sin nombre "default"*);
- *name*: nombre de la función;
- *inputs*: array de objetos, cada uno contiene: * *name*: nombre del parámetro; * *type*: tipo canónico del parámetro.
- *outputs*: un array de objetos similar a *inputs*, puede omitirse si la función no devuelve nada;
- *constant*: *true* si la función es *especificada para no modificar el estado de la blockchain*);
- *payable*: *true* si la función acepta ether, por defecto a *false*.

type se puede omitir, dejándolo por defecto a *"function"*.

La función Constructor y fallback nunca tienen *name* o *outputs*. Fallback tampoco tiene *inputs*.

Enviar una cantidad de ether no-nula a una función no payable lanzará excepción. No lo hagas.

Una descripción de evento es un objeto JSON con prácticamente los mismos campos:

- *type*: siempre *"event"*
- *name*: nombre del evento;
- *inputs*: array de objetos, cada uno contiene: * *name*: nombre del parámetro; * *type*: tipo canónico del parámetro. * *indexed*: *true* si el campo es parte de los tópicos del log, *false* si es parte del segmento de datos del log.
- *anonymous*: *true* si el evento se declaró *anonymous*.

Por ejemplo,

```
contract Test {
  function Test(){ b = 0x1234567890123456789012345678901234567890; }
  event Event(uint indexed a, bytes32 b)
  event Event2(uint indexed a, bytes32 b)
  function foo(uint a) { Event(a, b); }
  bytes32 b;
}
```

resultaría en el JSON:

```
[{
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event2"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event2"
}, {
"type":"function",
"inputs": [{"name":"a","type":"uint256"}],
"name":"foo",
"outputs": []
}]
```

Guía de estilo

Introducción

Esta guía pretende proporcionar convenciones de codificación para escribir código con Solidity. Esta guía debe ser entendida como un documento en evolución que cambiará con el tiempo según aparecen nuevas convenciones útiles y antiguas convenciones se vuelven obsoletas.

Muchos proyectos implementarán sus propias guías de estilo. En el caso de conflictos, las guías de estilo específicas del proyecto tendrán prioridad.

La estructura y muchas de las recomendaciones de esta guía de estilo fueron tomadas de Python: guía de estilo pep8.

El objetivo de esta guía *no* es ser la forma correcta o la mejor manera de escribir código con Solidity. El objetivo de esta guía es la *consistencia*. Una cita de python pep8 capta bien este concepto.

> Una guía de estilo es sobre consistencia. La consistencia con esta guía de estilo es importante. La consistencia dentro de un proyecto es más importante. La consistencia dentro de un módulo o función es lo más importante. Pero sobre todo: saber cuándo ser inconsistente - a veces la guía de estilo simplemente no se aplica. En caso de duda, use su mejor juicio. Mire otros ejemplos y decida qué parece mejor. ¡Y no dude en preguntar!

Diseño del código

Sangría

Utilice 4 espacios por nivel de sangría.

Tabulador o espacios

Los espacios son el método de indentación preferido.

Se deben evitar la mezcla del tabulador y los espacios.

Líneas en blanco

Envuelva las declaraciones de nivel superior en el código de Solidity con dos líneas en blanco.

Sí:

```
contract A {
    ...
}


contract B {
    ...
}


contract C {
    ...
}
```

No:

```
contract A {
    ...
}
contract B {
    ...
}

contract C {
    ...
}
```

Dentro de un contrato, rodee las declaraciones de una función con una sola línea en blanco.

Las líneas en blanco se pueden omitir entre grupos de una frase relacionada (tales como las funciones stub en un contrato abstracto)

Sí:

```
contract A {
    function spam();
    function ham();
}
```

```
contract B is A {
    function spam() {
        ...
    }

    function ham() {
        ...
    }
}
```

No:

```
contract A {
    function spam() {
        ...
    }
    function ham() {
        ...
    }
}
```

Codificación de archivos de origen

Se prefiere la codificación del texto en UTF-8 o ASCII.

Importación

Las declaraciones de importación siempre deben colocarse en la parte superior del archivo.

Sí:

```
import "owned";


contract A {
    ...
}


contract B is owned {
    ...
}
```

No:

```
contract A {
    ...
}


import "owned";


contract B is owned {
```

```
    ...
}
```

Orden de funciones

La ordenación ayuda a que los lectores puedan identificar las funciones que pueden invocar y encontrar las definiciones de constructor y de retorno más fácilmente.

Las funciones deben agruparse de acuerdo con su visibilidad y ser ordenadas de acuerdo a:

- constructor
- función fallback (Si existe)
- external
- public
- internal
- private

Dentro de un grupo, coloque las funciones `constant` al final.

Sí:

```
contract A {
    function A() {
        ...
    }

    function() {
        ...
    }

    // Funciones external
    // ...

    // Funciones external que son constantes
    // ...

    // Funciones public
    // ...

    // Funciones internal
    // ...

    // Funciones private
    // ...
}
```

No:

```
contract A {

    // Funciones external
    // ...

    // Funciones private
    // ...
```

```
    // Funciones public
    // ...

    function A() {
        ...
    }

    function() {
        ...
    }

    // Funciones internal
    // ...
}
```

Espacios en blanco en expresiones

Evite los espacios en blanco superfluos en las siguientes situaciones:

Inmediatamente entre paréntesis, llaves o corchetes, con la excepción de declaraciones de una función en una sola línea.

Sí:

```
spam(ham[1], Coin({name: "ham"}));
```

No:

```
spam( ham[ 1 ], Coin( { name: "ham" } ) );
```

Excepción:

```
function singleLine() { spam(); }
```

Inmediatamente antes de una coma, punto y coma:

Sí:

```
function spam(uint i, Coin coin);
```

No:

```
function spam(uint i , Coin coin) ;
```

Más de un espacio alrededor de una asignación u otro operador para alinearlo con otro:

Sí:

```
x = 1;
y = 2;
long_variable = 3;
```

No:

```
x             = 1;
y             = 2;
long_variable = 3;
```

No incluya un espacio en blanco en la función fallback:

Sí:

```
function() {
    ...
}
```

No:

```
function () {
    ...
}
```

Estructuras de control

Las llaves que denotan el cuerpo de un contrato, biblioteca, funciones y estructuras deberán:

- Abrir en la misma línea que la declaración
- Cerrar en su propia línea en el mismo nivel de sangría que la declaración.
- La llave de apertura debe ser procedida por un solo espacio.

Sí:

```
contract Coin {
    struct Bank {
        address owner;
        uint balance;
    }
}
```

No:

```
contract Coin
{
    struct Bank {
        address owner;
        uint balance;
    }
}
```

Las mismas recomendaciones se aplican a las estructuras de control `if`, `else`, `while` y `for`.

Además, debería existir un único espacio entre las estructuras de control `if`, `while`, y `for`, y el bloque entre paréntesis que representa el condicional, así como un único espacio entre el bloque del paréntesis condicional y la llave de apertura.

Sí:

```
if (...) {
    ...
}

for (...) {
    ...
}
```

No:

```
if (...)
{
    ...
}

while(...){
}

for (...) {
    ...;}
```

Para las estructuras de control cuyo cuerpo sólo contiene declaraciones únicas, se pueden omitir los corchetes *si* la declaración cabe en una sola línea.

Sí:

```
if (x < 10)
    x += 1;
```

No:

```
if (x < 10)
    someArray.push(Coin({
        name: 'spam',
        value: 42
    }));
```

Para los bloques `if` que contienen una condición `else` o `else if`, el `else` debe estar en la misma línea que el corchete de cierre del `if`. Esto es una excepción en comparación con las reglas de otras estructuras de tipo bloque.

Sí:

```
if (x < 3) {
    x += 1;
} else if (x > 7) {
    x -= 1;
} else {
    x = 5;
}


if (x < 3)
    x += 1;
else
    x -= 1;
```

No:

```
if (x < 3) {
    x += 1;
}
else {
    x -= 1;
}
```

Declaración de funciones

Para declaraciones de función cortas, se recomienda dejar el corchete de apertura del cuerpo de la función en la misma línea que la declaración de la función.

El corchete de cierre debe estar al mismo nivel de sangría que la declaración de la función.

El corchete de apertura debe estar precedido por un solo espacio.

Sí:

```
function increment(uint x) returns (uint) {
    return x + 1;
}

function increment(uint x) public onlyowner returns (uint) {
    return x + 1;
}
```

No:

```
function increment(uint x) returns (uint)
{
    return x + 1;
}

function increment(uint x) returns (uint){
    return x + 1;
}

function increment(uint x) returns (uint) {
    return x + 1;
    }

function increment(uint x) returns (uint) {
    return x + 1;}
```

Se debe especificar la visibilidad de los modificadores para una función antes de cualquier modificador personalizado.

Sí:

```
function kill() public onlyowner {
    selfdestruct(owner);
}
```

No:

```
function kill() onlyowner public {
    selfdestruct(owner);
}
```

Para las declaraciones de función largas, se recomienda dejar a cada argumento su propia línea al mismo nivel de sangría que el cuerpo de la función. El paréntesis de cierre y el corchete de apertura deben de estar en su propia línea también y con el mismo nivel de sangría que la declaración de la función.

Sí:

```
function thisFunctionHasLotsOfArguments(
    address a,
    address b,
```

```
    address c,
    address d,
    address e,
    address f
) {
    doSomething();
}
```

No:

```
function thisFunctionHasLotsOfArguments(address a, address b, address c,
    address d, address e, address f) {
    doSomething();
}

function thisFunctionHasLotsOfArguments(address a,
                                        address b,
                                        address c,
                                        address d,
                                        address e,
                                        address f) {
    doSomething();
}

function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f) {
    doSomething();
}
```

Si una declaración de función larga tiene modificadores, cada uno de ellos debe de estar en su propia línea.

Sí:

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyowner
    priced
    returns (address)
{
    doSomething();
}

function thisFunctionNameIsReallyLong(
    address x,
    address y,
    address z,
)
    public
    onlyowner
    priced
    returns (address)
{
    doSomething();
```

```
}
```

No:

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
                                      public
                                      onlyowner
                                      priced
                                      returns (address) {
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
    public onlyowner priced returns (address)
{
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyowner
    priced
    returns (address) {
    doSomething();
}
```

Para las funciones de tipo constructor en contratos heredados que requieren argumentos, si la declaración de la función es larga o difícil de leer, se recomienda poner cada constructor base en su propia línea de la misma manera que con los modificadores.

Sí:

```
contract A is B, C, D {
    function A(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4)
    {
        // do something with param5
    }
}
```

No:

```
contract A is B, C, D {
    function A(uint param1, uint param2, uint param3, uint param4, uint param5)
    B(param1)
    C(param2, param3)
    D(param4)
    {
        // do something with param5
    }
}

contract A is B, C, D {
    function A(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
```

```
        D(param4) {
        // do something with param5
    }
}
```

Cuando se declara funciones cortas con una sola declaración, está permitido hacerlo en una solo línea.

Permisible:

```
function shortFunction() { doSomething(); }
```

Esta guía sobre la declaración de funciones está pensada para mejorar la legibilidad. Sin embargo, los autores deberían utilizar su mejor juicio, ya que esta guía tampoco intenta cubrir todas las posibles permutaciones para las declaraciones de función.

Mapeo

Pendiente de hacer

Declaración de variables

La declaración de variables tipo array no deben incluir un espacio entre el tipo y el corchete.

Sí:

```
uint[] x;
```

No:

```
uint [] x;
```

Otras recommendaciones

- Los strings deben de citarse con comillas dobles en lugar de comillas simples.

Sí:

```
str = "foo";
str = "Hamlet says, 'To be or not to be...'";
```

No:

```
str = 'bar';
str = '"Be yourself; everyone else is already taken." -Oscar Wilde';
```

- Se envuelve los operadores con un solo espacio de cada lado.

Sí:

```
x = 3;
x = 100 / 10;
x += 3 + 4;
x |= y && z;
```

No:

```
x=3;
x = 100/10;
x += 3+4;
x |= y&&z;
```

- Para los operadores con una prioridad mayor que otros, se pueden omitir los espacios de cada lado del operador para marcar la precedencia. Esto se hace para mejorar la legibilidad de declaraciones complejas. Se debe usar siempre el mismo número de espacios a cada lado de un operador.

Sí:

```
x = 2**3 + 5;
x = 2*y + 3*z;
x = (a+b) * (a-b);
```

No:

```
x = 2** 3 + 5;
x = y+z;
x +=1;
```

Convención sobre nombres

Las convenciones sobre nombres son extremadamente útiles siempre y cuando se usen de forma ámplia. El uso de diferentes convenciones puede transmitir *meta* información significativa a la que de otro modo no tendríamos acceso de manera inmediata.

Las recomendaciones de nombres que se dan aquí están pensadas para mejorar la legibilidad, y por lo tanto no se deben considerar como reglas. Son más bien una guía para intentar transmitir la mayor información posible a través del nombre de las cosas.

Finalmente, la consistencia dentro de un bloque de código siempre debe prevalecer sobre cualquier convención destacada en este documento.

Estilos para poner nombres

Para evitar confusiones, se usarán los siguiente nombres para referirse a diferentes estilos para poner nombres.

- `b` (letra minúscula única)
- `B` (letra mayúscula única)
- `minuscula`
- `minuscula_con_guiones_bajos`
- `MAYUSCULA`
- `MAYUSCULA_CON_GUIONES_BAJOS`
- `PalabrasConLaInicialEnMayuscula` (también llamado CapWords)
- `mezclaEntreMinusculaYMayuscula` (¡distinto a PalabrasConLaInicialEnMayuscula por el uso de una minúscula en la letra inicial!)
- `Palabras_Con_La_Inicial_En_Mayuscula_Y_Guiones_Bajos`

Nota: Cuando se usan abreviaciones en CapWords, usar mayúsculas para todas las letras de la abreviación. Es decir que HTTPServerError es mejor que HttpServerError

Nombres a evitar

- `l` - Letra minúscula ele
- `O` - Letra mayúscula o
- `I` - Letra mayúscula i

No usar jamás ninguna de estas letras únicas para nombrar una variable. Estas letras generalmente no se diferencian de los dígitos uno y cero.

Contratos y librerías de nombres

Contratos y librerías deben de nombrarse usando el estilo CapWord (PalabrasConLaInicialEnMayuscula).

Eventos

Los eventos deben de nombrarse usando el estilo CapWord (PalabrasConLaInicialEnMayuscula).

Nombres para funciones

Las funciones deben de nombrarse usando el estilo mezclaEntreMinusculaYMayuscula.

Argumentos de funciones

Cuando se escriben funciones de librerías que operan sobre un struct personalizado, el struct debe ser el primer argumento y debe nombrarse siempre `self`.

Variables locales y de estado

Usar el estilo mezclaEntreMinusculaYMayuscula.

Constantes

Las constantes deben de nombrarse con todas las letras mayúsculas y guiones bajos para separar las palabras (p.ej. `MAX_BLOCKS`).

Modificadores

Usar el estilo mezclaEntreMinusculaYMayuscula.

Evitar Colisiones

- `guion_bajo_unico_con_cola_`

Se recomienda usar esta convención cuando el nombre deseado colisiona con un nombre inherente al sistema o reservado.

Recommendaciones generales

Pendiente de hacer

Patrones comunes

Retirada desde contratos

El método recomendado de envío de fondos después de una acción, es usando el patrón de retirada (withdrawal pattern). Aunque el método más intuitivo para enviar Ether tras una acción es llamar directamente a `send`, esto no es recomendable ya que introduce un potencial riesgo de seguridad. Puedes leer más sobre esto en la página *Consideraciones de seguridad*.

Este es un ejemplo del patrón de retirada en un contrato donde el objetivo es enviar la mayor cantidad de Ether al contrato a fin de convertirse en el más "adinerado", inspirado por King of the Ether.

En el siguiente contrato, si dejas de ser el más adinerado, recibes los fondos de la persona que te destronó.

```
pragma solidity ^0.4.11;

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    mapping (address => uint) pendingWithdrawals;

    function WithdrawalContract() payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() payable returns (bool) {
        if (msg.value > mostSent) {
            pendingWithdrawals[richest] += msg.value;
            richest = msg.sender;
            mostSent = msg.value;
            return true;
        } else {
            return false;
        }
    }

    function withdraw() {
        uint amount = pendingWithdrawals[msg.sender];
        // Acuérdate de poner a cero la cantidad a reembolsar antes
        // de enviarlo para evitar re-entrancy attacks
        pendingWithdrawals[msg.sender] = 0;
        msg.sender.transfer(amount);
```

```
    }
}
```

Esto en lugar del patrón más intuitivo de envío:

```
pragma solidity ^0.4.11;

contract SendContract {
    address public richest;
    uint public mostSent;

    function SendContract() payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() payable returns (bool) {
        if (msg.value > mostSent) {
            // Esta línea puede causar problemas (explicado abajo).
            richest.transfer(msg.value);
            richest = msg.sender;
            mostSent = msg.value;
            return true;
        } else {
            return false;
        }
    }
}
```

Nótese que, en este ejemplo, un atacante puede bloquear el contrato en un estado inútil haciendo que `richest` sea la dirección de un contrato que tiene una función fallback que falla (ej. usando `revert()` o simplemente consumiendo más de 2300 de gas). De esa forma, cuando se llama a `transfer` para enviar fondos al contrato "envenenado", fallará y también fallará la función `becomeRichest`, bloqueando el contrario para siempre.

Por el contrario, si usas el patrón "withdrawl" del primer ejemplo, el atacante sólo puede causar que su propio withdrawl falle y no el resto del contrato.

Restringiendo el acceso

Restringiendo el acceso (Restricting access) es un patrón común para contratos. Nótese que nunca se puede evitar que un humano o un ordenador lean el contenido de una transacción o el estado de un contrato. Lo puedes hacer un poco más difícil de leer usando criptografía, pero si tu contrato debe leer los datos, todos podrán leerlo.

Puedes restringir el acceso de lectura al estado de tu contrato por **otros contratos**. Esto ocurre por defecto salvo que declares tus variables como `public`.

Además, puedes restringir quién puede hacer modificaciones al estado de tu contrato o quién puede llamar a las funciones. De eso se trata esta sección.

El uso de **modificadores de funciones** hace estas restricciones altamente visibles.

```
pragma solidity ^0.4.11;

contract AccessRestriction {
    // Estas serán asignadas en la fase de
    // compilación, donde `msg.sender` es
    // la cuenta que crea este contrato.
```

```
    address public owner = msg.sender;
    uint public creationTime = now;

    // Los modificadores pueden usarse para
    // cambiar el cuerpo de una función.
    // Si se usa este modificador, agregará
    // un chequeo que sólo pasa si la
    // función se llama desde una cierta
    // dirección.
    modifier onlyBy(address _account)
    {
        require(msg.sender == _account);
        // ¡No olvides el "_;"!
        // Esto se reemplazará por el cuerpo
        // de la función cuando se use
        // el modificador
        _;
    }

    /// Hacer que `_newOwner` sea el nuevo owner de
    /// este contrato.
    function changeOwner(address _newOwner)
        onlyBy(owner)
    {
        owner = _newOwner;
    }

    modifier onlyAfter(uint _time) {
        require(now >= _time);
        _;
    }

    /// Borra la información del dueño.
    /// Sólo puede llamarse 6 semanas
    /// después de que el contrato haya sido
    /// creado.
    function disown()
        onlyBy(owner)
        onlyAfter(creationTime + 6 weeks)
    {
        delete owner;
    }

    // Este modificador requiere del pago de
    // una comisión asociada a la llamada
    // de una función.
    // Si el llamador envió demasiado, será
    // reembolsado, pero sólo después del cuerpo
    // de la función.
    // Esto era peligroso antes de la versión
    // 0.4.0 de Solidity, donde era posible
    // saltarse la parte después de `_;`.
    modifier costs(uint _amount) {
        require(msg.value >= _amount);
        _;
        if (msg.value > _amount)
            msg.sender.send(msg.value - _amount);
    }
```

```
    function forceOwnerChange(address _newOwner)
        costs(200 ether)
    {
        owner = _newOwner;
        // sólo una condición de ejemplo
        if (uint(owner) & 0 == 1)
            // Esto no se hacía antes de Solidity
            // 0.4.0
            return;
        // reembolsar los fees excesivos
    }
}
```

Una manera más especializada de la forma en la que se puede reestringir el acceso a la llamada de funciones se verá en el próximo ejemplo.

Máquina de estados

Los contratos a menudo actúan como una máquina de estados, lo que significa que tienen ciertas **etapas** en donde se comportan de manera diferente o en donde distintas funciones pueden ser llamadas. Una llamada de función a menudo termina una etapa y pasa el contrato a la siguiente etapa (especialmente si el contrato modela la **interacción**). También es común que algunas etapas se alcancen automáticamente en cierto punto en el **tiempo**.

Un ejemplo de esto es el contrato de subastas a ciegas que comienza en la etapa "aceptando pujas a ciegas", luego pasa a "revelando pujas" que es finalizado por "determinar resultado de la subasta".

Los modificadores de funciones se pueden usar en esta situación para modelar los estados y evitar el uso incorrecto del contrato.

Ejemplo

En el siguiente ejemplo, el modificador `atStage` asegura que la función sólo pueda ser llamada en una cierta etapa.

El modificador `timeTransitions` gestiona las transiciones de etapas de forma automática en función del tiempo. Debe ser usado en todas las funciones.

Nota: El orden de los modificadores importa. Si atStage se combina con timesTransitions, asegúrate de que puedas mencionarlo después de éste, para que la nueva etapa sea tomada en cuenta.

Finalmente, el modificador `transitionNext` puede ser usado para ir automáticamente a la próxima etapa cuando la función termine.

Nota: El modificador puede ser omitido. Esto sólo se aplica a Solidity antes de la versión 0.4.0: Puesto que los modificadores se aplican simplemente reemplazando código y no realizando una llamada a una función, el código del modificador transitionNext se puede omitir si la propia función usa return. Si es lo que quieres hacer, asegúrate de llamar manualmente a nextStage desde esas funciones. A partir de la versión 0.4.0, el código de los modificadores se ejecutará incluso en el caso de que la función ejecute explícitamente un return.

```
pragma solidity ^0.4.11;

contract StateMachine {
```

```
    enum Stages {
        AcceptingBlindedBids,
        RevealBids,
        AnotherStage,
        AreWeDoneYet,
        Finished
    }

    // Ésta es la etapa actual.
    Stages public stage = Stages.AcceptingBlindedBids;

    uint public creationTime = now;

    modifier atStage(Stages _stage) {
        require(stage == _stage);
        _;
    }

    function nextStage() internal {
        stage = Stages(uint(stage) + 1);
    }

    // Hace transiciones temporizadas. Asegúrate de
    // mencionar este modificador primero, si no,
    // no se tendrá en cuenta la nueva etapa.
    modifier timedTransitions() {
        if (stage == Stages.AcceptingBlindedBids &&
                    now >= creationTime + 10 days)
            nextStage();
        if (stage == Stages.RevealBids &&
                now >= creationTime + 12 days)
            nextStage();
        // La transición del resto de etapas se produce por transacciones
        _;
    }

    // ¡El orden de los modificadores importa aquí!
    function bid()
        payable
        timedTransitions
        atStage(Stages.AcceptingBlindedBids)
    {
        // No implementaremos esto aquí
    }

    function reveal()
        timedTransitions
        atStage(Stages.RevealBids)
    {
    }

    // Este modificador pasa a la próxima etapa
    // una vez terminada la función.
    modifier transitionNext()
    {
        _;
        nextStage();
    }
```

```
    function g()
        timedTransitions
        atStage(Stages.AnotherStage)
        transitionNext
    {
    }

    function h()
        timedTransitions
        atStage(Stages.AreWeDoneYet)
        transitionNext
    {
    }

    function i()
        timedTransitions
        atStage(Stages.Finished)
    {
    }
}
```

Lista de errores conocidos

Abajo encontrarás una lista en formato JSON de algunos de los errores de seguridad en el compilador Solidity. El archivo en sí está alojado en el repositorio de Github. La lista comienza con la versión 0.3.0, y sólo los errores conocidos antes de esa versíon no están listados.

Hay otro archivo llamado bugs_by_version.json, que puede usarse para revisar qué errores afectan a una versión específica del compilador.

Herramientas de verificación de código fuente y otras herramientas para contratos deben consultar esta lista con los siguientes criterios:

- Es relativamente sospechoso que un contrato se haya compilado con una versión nightly y no con una release. Esta lista no mantiene un registro de las versiones nightly.
- También es sospechoso cuando un contrato fue compilado con una versión que no era la más reciente en el momento que el contrato fue creado. Para contratos creados de otros contratos, tienes que seguir la cadena de creación de vuelta a una transacción y revisar la fecha de esa transacción.
- Es muy sospechoso que un contrato haya sido compilado con un compilador que contiene un error conocido, y que se cree cuando una versión del compilador con una corrección ya haya sido publicada.

El archivo JSON de errores conocidos es una lista de objetos, uno por cada error, con la siguiente información:

nombre (name) Nombre único asignado al error

resumen (summary) Pequeña descripción del error

descripción (description) Descripción detallada del error

enlace (link) URL de un sitio web con más información, opcional

introducido (introduced) La primera versión del compilador que contiene ese error, opcional

corregido (fixed) La primera versión del compilador que ya no contiene el error

publicado (publish) La fecha en la cual el error se hizo públicamente conocido, opcional

gravedad (severity) Gravedad del error: baja, media, alta. Considera la facilidad de ser detectado en tests de contratos, probabilidad de ocurrir y potencial daño.

condiciones (conditions) Condiciones que tienen que cumplirse para iniciar el error. Actualmente es un objeto que puede contener un valor booleano `optimizer`, que significa que el optimizador tiene que ser activado para reproducir el error. Si no se da ninguna condición, se da por hecho que el error está presente.

```
[
    {
        "name": "ConstantOptimizerSubtraction",
        "summary": "In some situations, the optimizer replaces certain numbers in the␣
↪code with routines that compute different numbers.",
        "description": "The optimizer tries to represent any number in the bytecode␣
↪by routines that compute them with less gas. For some special numbers, an incorrect␣
↪routine is generated. This could allow an attacker to e.g. trick victims about a␣
↪specific amount of ether, or function calls to call different functions (or none at␣
↪all).",
        "link": "https://blog.ethereum.org/2017/05/03/solidity-optimizer-bug/",
        "fixed": "0.4.11",
        "severity": "low",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "name": "IdentityPrecompileReturnIgnored",
        "summary": "Failure of the identity precompile was ignored.",
        "description": "Calls to the identity contract, which is used for copying␣
↪memory, ignored its return value. On the public chain, calls to the identity␣
↪precompile can be made in a way that they never fail, but this might be different␣
↪on private chains.",
        "severity": "low",
        "fixed": "0.4.7"
    },
    {
        "name": "OptimizerStateKnowledgeNotResetForJumpdest",
        "summary": "The optimizer did not properly reset its internal state at jump␣
↪destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages.␣
↪At jump destinations, multiple code paths join and thus it has to compute a common␣
↪state from the incoming edges. Computing this common state was simplified to just␣
↪use the empty state, but this implementation was not done properly. This bug can␣
↪cause data corruption.",
        "severity": "medium",
        "introduced": "0.4.5",
        "fixed": "0.4.6",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "name": "HighOrderByteCleanStorage",
        "summary": "For short types, the high order bytes were not cleaned properly␣
↪and could overwrite existing data.",
        "description": "Types shorter than 32 bytes are packed together into the same␣
↪32 byte storage slot, but storage writes always write 32 bytes. For some types, the␣
↪higher order bytes were not cleaned properly, which made it sometimes possible to␣
↪overwrite a variable in storage when writing to another one.",
        "link": "https://blog.ethereum.org/2016/11/01/security-alert-solidity-
↪variables-can-overwritten-storage/",
```

```
        "severity": "high",
        "introduced": "0.1.6",
        "fixed": "0.4.4"
    },
    {
        "name": "OptimizerStaleKnowledgeAboutSHA3",
        "summary": "The optimizer did not properly reset its knowledge about SHA3␣
↪operations resulting in some hashes (also used for storage variable positions) not␣
↪being calculated correctly.",
        "description": "The optimizer performs symbolic execution in order to save re-
↪evaluating expressions whose value is already known. This knowledge was not␣
↪properly reset across control flow paths and thus the optimizer sometimes thought␣
↪that the result of a SHA3 operation is already present on the stack. This could␣
↪result in data corruption by accessing the wrong storage slot.",
        "severity": "medium",
        "fixed": "0.4.3",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "name": "LibrariesNotCallableFromPayableFunctions",
        "summary": "Library functions threw an exception when called from a call that␣
↪received Ether.",
        "description": "Library functions are protected against sending them Ether␣
↪through a call. Since the DELEGATECALL opcode forwards the information about how␣
↪much Ether was sent with a call, the library function incorrectly assumed that␣
↪Ether was sent to the library and threw an exception.",
        "severity": "low",
        "introduced": "0.4.0",
        "fixed": "0.4.2"
    },
    {
        "name": "SendFailsForZeroEther",
        "summary": "The send function did not provide enough gas to the recipient if␣
↪no Ether was sent with it.",
        "description": "The recipient of an Ether transfer automatically receives a␣
↪certain amount of gas from the EVM to handle the transfer. In the case of a zero-
↪transfer, this gas is not provided which causes the recipient to throw an exception.
↪",
        "severity": "low",
        "fixed": "0.4.0"
    },
    {
        "name": "DynamicAllocationInfiniteLoop",
        "summary": "Dynamic allocation of an empty memory array caused an infinite␣
↪loop and thus an exception.",
        "description": "Memory arrays can be created provided a length. If this␣
↪length is zero, code was generated that did not terminate and thus consumed all gas.
↪",
        "severity": "low",
        "fixed": "0.3.6"
    },
    {
        "name": "OptimizerClearStateOnCodePathJoin",
        "summary": "The optimizer did not properly reset its internal state at jump␣
↪destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages.␣
↪At jump destinations, multiple code paths join and thus it has to compute a common␣
↪state from the incoming edges. Computing this common state was not done correctly.␣
↪This bug can cause data corruption, but it is probably quite hard to use for
↪targeted attacks.",
```

```
        "severity": "low",
        "fixed": "0.3.6",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "name": "CleanBytesHigherOrderBits",
        "summary": "The higher order bits of short bytesNN types were not cleaned␣
↪before comparison.",
        "description": "Two variables of type bytesNN were considered different if␣
↪their higher order bits, which are not part of the actual value, were different. An␣
↪attacker might use this to reach seemingly unreachable code paths by providing␣
↪incorrectly formatted input data.",
        "severity": "medium/high",
        "fixed": "0.3.3"
    },
    {
        "name": "ArrayAccessCleanHigherOrderBits",
        "summary": "Access to array elements for arrays of types with less than 32␣
↪bytes did not correctly clean the higher order bits, causing corruption in other␣
↪array elements.",
        "description": "Multiple elements of an array of values that are shorter than␣
↪17 bytes are packed into the same storage slot. Writing to a single element of such␣
↪an array did not properly clean the higher order bytes and thus could lead to data␣
↪corruption.",
        "severity": "medium/high",
        "fixed": "0.3.1"
    },
    {
        "name": "AncientCompiler",
        "summary": "This compiler version is ancient and might contain several␣
↪undocumented or undiscovered bugs.",
        "description": "The list of bugs is only kept for compiler versions starting␣
↪from 0.3.0, so older versions might contain undocumented bugs.",
        "severity": "high",
        "fixed": "0.3.0"
    }
]
```

Contribuir

¡La ayuda siempre es bienvenida!

Para comenzar, puedes intentar *construir a partir del código* para familiarizarte con los componentes de Solidity y el proceso de build. También, puede ser útil especializarse en escribir smart contrats en Solidity.

En particular, necesitamos ayuda en las siguientes áreas:

- Mejorando la documentación
- Respondiendo a las preguntas de otros usarios en StackExchange y el Gitter de Solidity
- Corrigiendo y respondiendo a los issues del GitHub de Solidity, especialmente esos taggeados como up-for-grabs que están destinados a contribuidores externos como temas introductorios.

Cómo reportar un Issue

Para reportar un issue, por favor, usa el Issue tracker de GitHub. Cuando reportes un issue, por favor, menciona los siguientes detalles:

- Qué versión de Solidity estás usando
- Cuál es el código fuente (si es aplicable)
- En qué plataforma lo estás ejecutando
- Cómo reproducir el resultado
- Cuál fue el resultado del issue
- Cuál es el resultado esperado

Reducir el código fuente que causó el issue a lo mínimo es siempre muy útil y a veces incluso aclara un malentendido.

Flujo de trabajo para Pull Requests

A fin de contribuir, haz un fork de la rama `develop` y haz tus cambios ahí. Tus mensajes de commit deben detallar *por qué* hiciste el cambio además de *lo que* hiciste (al menos que sea un pequeño cambio).

Si necesitas hacer un pull de `develop` después de haber hecho tu fork (por ejemplo, para resolver potenciales conflictos de merge), evita utilizar `git merge`. Usa en su lugar `git rebase` para tu rama.

Adicionalmente, si estás escribiendo una nueva funcionalidad, por favor, asegúrate de hacer tests unitarios Boost y ponerlos en `test/`.

Sin embargo, si estás haciendo cambios más grandes, consulta primero con el canal Gitter.

Finalmente, siempre asegúrate de respetar los estándares de código de este proyecto. También, aunque hacemos testing CI, testea tu código y asegúrate que puedas hacer un build localmente antes de enviar un pull request.

¡Gracias por tu ayuda!

Ejecutando los tests de compilador

Solidity incluye diferentes tipos de tests. Están incluídos en la aplicación llamada `soltest`. Algunos de ellos requieren el cliente `cpp-ethereum` en modo test.

Para ejecutar `cpp-ethereum` en modo test: `eth --test -d /tmp/testeth`.

Para lanzar los tests: `soltest -- --ipcpath /tmp/testeth/geth.ipc`.

Para ejecutar un subconjunto de los tests, se pueden usar filtros: `soltest -t TestSuite/TestName -- --ipcpath /tmp/testeth/geth.ipc`, donde `TestName` puede ser un comodín `*`.

Alternativamente, hay un script de testing en `scripts/test.sh` que ejecuta todos los tests.

Preguntas frequentes

Esta lista fue originalmente compilada por fivedogit.

Preguntas Básicas

Ejemplos de contratos

Hay algunos ejemplos de contratos por fivedogit y debe haber un test contract para cada funcionalidad de Solidity.

Crear y publicar el contrato mas simple posible

Un contrato bastante simple es el greeter

¿Es posible hacer algo en un bloque específico? (ej. publicar un contrato o ejecutar una transaction)

Las transacciones no están garantizadas a ejecutarse en el próximo bloque o en cualquier bloque futuro, ya que depende de los mineros de incluir transacciones y no del remitente de la transacción. Esto se aplica a llamadas de funciones/transacciones y trasacciones de creación de contratos.

Si quieres programar llamadas de contrato a futuro, puedes usar el despertador (alarm clock).

¿Qué es la "payload" de la transacción?

Esto es sólamente el data bytecode enviado junto con la solicitud.

¿Hay un decompilador disponible?

No hay un decompilador en Solidity. Esto es en principio posible hasta un punto, pero por ejemplo los nombres de variables serán perdidas y un gran esfuerzo será necesario para replicar el código de fuente original.

Bytecode puede ser decompilado a opcoses, un servicio que es provisto por varios exploradores de blockchain.

Los contratos en la blockchain deben tener su código fuente original publicado si serán utilizados por terceros.

Crear un contrato que puede ser detenido y devolver los fondos

Primero, una advertencia: Detener contratos suena como una buena idea, porque "limpiar" siempre es bueno, pero como se ve arriba, no se limpia realmente. Además, si algo de Ether es enviado a contratos eliminados, el Ether será perdido para siempre.

Si quieres desactivar tus contratos, es mejor "inhabilitarlos" cambiando algunos estados internos que hace que todas las funciones arrojen excepciones. Esto hará que sea imposible de usar el contrato y todo ether enviado será devuelto automáticamente.

Ahora para responder la pregunta: Dento de un constructor, `msg.sender` es el creador. Guárdalo. Luego `selfdestruct(creator);` para matar y devolver los fondos.

ejemplo

Nótese que si importas `import "mortal"` arriba del contrato y declaras `contract AlgunContrato is mortal { ...` y compilas con un compilador que ya lo tiene (que incluye *Remix <https://remix.ethereum.org/>*), luego `kill()` es ejecutado por ti. Una vez que un contrato es "mortal", se puede `contractname.kill.sendTransaction({from:eth.coinbase})`, igual que en los ejemplos.

Guardar Ether en un contrato

El truco es de crear un contrato con `{from:someaddress, value: web3.toWei(3,."ether")...}`

Ver endowment_retriever.sol.

Usar una función no-constante (req `sendTransation`) para incrementar una variable en un contrato

Ver value_incrementer.sol.

Obtener que un contrato te devuelva los fondos (sin usar `selfdestruct(...)`).

Este ejemplo demuestra como envíar fondos de un contrato a una address.

Ver endowment_retriever.

¿Puedes devolver un array o un `string` desde una llamada de función solidity?

Si. Ver array_receiver_and_returner.sol.

Lo que sí es problemático es devolver cualquier data de tamaño variable (ej. un array de tamaño variable como `uint[]`) desde una función **llamada desde Solidity**. Esto es una limitación de la EVM y será resuelto en la próxima versión del protocolo.

Devolver data de tamaño variable está bien cuando es parte de una transaction o llamada externa.

¿Cómo representas `double/float`` en Solidity?

Esto no es aún posible.

Es posible de iniciar un array in-line (ej. `string[] myarray = [."a", "b"];`)

Si. Sin embargo debiera notarse que esto sólo funciona con arrays de tamaño estático. Puedes incluso crear un array en memoria en línea en la declaración de devolución. Cool, ¿no?

Example:

```
contract C {
    function f() returns (uint8[5]) {
        string[4] memory adaArr = ["Esto", "es", "un", "array"];
        return ([1, 2, 3, 4, 5]);
    }
}
```

Son de confianza los timestamps (`now`, `block.timestamp`)

Esto depende por lo que te refieres con "de confianza". En general, son entregados por los mineros y por lo tanto son vulnerables.

Al menos que haya un problema grave en la blockchain o en tu ordenador, puedes hacer las siguientes suposiciones:

Publicas una transacción en un tiempo X, esta transacción contiene el mismo código que llama `now` y es incluída en un bloque cuyo timestamp es Y y este bloque es incluído en la cadena canónica (publicado) en un tiempo Z.

El valor de `now` será idéntico a Y y X <= Y <= Z.

Nunca usa `now` o `block.hash` como una fuente aleatoria, a menos que sepas lo que estás haciendo.

¿Puede una función de contrato devolver un `struct`?

Si, pero sólo en llamadas de funciones `internal`.

Si devuelvo un `enum`, Sólo me dan valores enteros en web3.js. ¿Cómo obtengo los valores nombrados?

Enums no son soportados por la ABI, sólo son soportados por Solidity. Tienes que hacer el mapping tu mismo por ahora, aunque puede que proporcionemos ayuda mas adelante.

¿Cuál es el significado de `function() { ... }` dentro de los contratos Solidity? ¿Cómo es posible que una función no tenga nombre?

Esta función es llamada "callback function" y es llamada cuando alguien sólo envía Ether al contrato sin proveer data o si alguien se equivocó e intentó llamar una función que no existe.

El funcionamiento por defecto (si no hay función fallback explícita) en estas situaciones es arrojar una excepción.

Si el contrato debiera recibir Ether con transferencias simples, debes implementar una función callback como

```
function() payable { }
```

Otro uso de la función callback es por ejemplo registrar que tu contrato recibió ether usando un evento.

Attention: Si implementas la función fallback, cuida que use lo menos gas posible, porque `send()` sólo suministrará una cantidad limitada.

¿Es posible pasar argumentos a la función fallback?

La función fallback no puede tomar parámetros.

Bajo ciertas circunstancias, puedes enviar data. Si cuidad que ninguna de las otras funciones es llamada, puedes acceder a la data usando `msg.data`.

¿Pueden las variables de estado ser iniciadas in-line?

Si, esto es posible para todos los tipos (incluso para structs). Sin embargo, para arrays debe notarse que se le deben declarar como arrays de memoria estática.

Ejemplos:

```
contract C {
    struct S {
        uint a;
        uint b;
    }

    S public x = S(1, 2);
    string name = "Ada";
    string[4] memory adaArr = ["Esto", "es", "un", "array"];
}
```

```
contract D {
    C c = new C();
}
```

¿Cómo funcionan los structs?

Ver struct_and_for_loop_tester.sol.

¿Cómo funcionan los for loop?

Muy similarmente a Javascript. Aunque esto es un punto al cual debe hacerse atención:

Si usas `for (var i = 0; i < a.length; i ++) { a[i] = i; }`, entonces el tipo de `i` será inferido sólo de `0`, o sea, un tipo `uint8`. Esto significa que si `a` tiene más de `255` elementos, tu loop no terminará ya que `i` sólo contendrá valores hasta `255`.

Mejor usar `for (uint i = 0; i < a.length...`

Ver struct_and_for_loop_tester.sol.

¿Qué set de caracteres usa Solidity?

Solidity es agnostico de set de caracteres con respecto a strings en el código fuente, aunque UTF-8 es recomendado. Los identificadores (variables, funciones, ...) Solo pueden usar ASCII.

¿Cuáles son algunos ejemplos de manipulación de strings básicos (`substring`, `indexOf`„ `charAt`, etc)?

Hay algunas funciones de utilidad de string en stringUtils.sol que serán extendidas en el futuro. Además, Arachnid ha escrito solidity-stringutils.

Por ahora si quieres modificar un string, (incluso cuando sólo quieres saber su largo), debes siempre convertirlo en un `bytes` primero:

```
contract C {
    string s;

    function append(byte c) {
        bytes(s).push(c);
    }

    function set(uint i, byte c) {
        bytes(s)[i] = c;
    }
}
```

¿Puedo concatenar dos strings?

Tienes que hacerlo manualmente por ahora.

Por qué la función de bajo nivel `.call()` es menos favorable que instanciando un contrato con ua vaiable (`ContractBb;`) y ejecutando sus funcioens (`b.doSomething();`)?

TODO: Si usar reales funciones, el compilador le dirá si los tipos de los argumentos no concuerdan, si la función no existe o no es visible y hará el empaquetamiento de los argumentos por tí.

Ver ping.sol y pong.sol.

¿El gas inutilizado es automaticamente devuelto?

Si y es immediato, ej. hecho como parte de la transacción.

Cuando se devuelve un valor de tipo `unint`, ¿es posible devolver un *undefined*‘ o un valor `null`?

Esto no es posible, porque todos los tipos usan el rango de valores totales.

Tienes la opción de `arrojar` un error, que también revirtirá la transacción completa, que puede que sea una buena idea si obtuviste una situación inesperada.

Si no quieres deolver un error, puedes devolver un par:

```
contract C {
    uint[] counters;

    function getCounter(uint index)
        returns (uint counter, bool error) {
            if (index >= counters.length)
                return (0, true);
            else
                return (counters[index], false);
    }

    function checkCounter(uint index) {
        var (counter, error) = getCounter(index);
        if (error) {
            ...
        } else {
            ...
        }
    }
}
```

¿Los comentarios son incluídos en los contratos publicados y incrementan el gas

No. Todo lo que no sea utilizado para la ejecución es eliminado durante la compilación. Esto incluye, entre otras cosas, comentarios, nombres de variable y nombres de tipos.

¿Qué pasa si envías ether junto con una llamada de función a un contrato?

Se agrega al balance total del contrato, igual que cuando mandas ether creando un contrato. Sólo puedes envíar ether junto con una función que tiene modificador `payable`, si no, una excepción es levantada.

¿Es posible obtener una respuesta tx para una transacción ejecutada contrato-a-contrato?

No, una llamada de función de un contrato a otro no crea su propia transacción, tienes que mirar en la transacción general. Eso también es la razón por la que varios exploradores de bloques no muestran Ether envíado entre contratos correctamente.

¿Cuál es la palabra clave `memory` y qué hace?

La Máquina Virtual Ethereum tiene tres áreas donde puede guardar cosas.

La primera es "storage", donde todas las variables de estado del contrato existen. Cada contrato tiene su propio storage y es persistente entre llamdas de función y bastate caro usarlo.

La segunda es "memory", esto es usado para guardar valores temporales. Es borrado entre llamdas de función (externas) y es más barato usar.

La tercera es en el stack, que es usado para guardar pequeñas variables locales. Es casi gratis para usar, pero sólo puede guardar una cantidad limitada de valores.

Para casi todos los tipos, no puedes especificar donde deben ser gurdadas, porque son copiadas cada vez que se usan.

Los tipos donde la storage-location es importante son structs y arrays. Si, por ejemplo, pasaras estas variables en llamadas de función, su data no es copiada si puede quedar en memoria o quedar en storage. Esto significa que puedes modificar su contenido en la función llamada y estas modificaciones aún serán visibles al llamador.

Hay valores por defecto para el storage location dependiendo de que tipo de variable le concierne:

- variables de estado siempre están en storage
- argumentos de función siempre están en memoria
- variable locales siempre referencian storage

Example:

```
contract C {
    uint[] data1;
    uint[] data2;

    function appendOne() {
        append(data1);
    }

    function appendTwo() {
        append(data2);
    }

    function append(uint[] storage d) {
        d.push(1);
    }
}
```

La función `append` puede funcionar en ambos `data1` y `data2` y sus modificaciones serán guardadas permenentemente. Si quitas la palabra clave `storage`, por defecto se usa `memory` para argumentos de función. Esto tiene como efecto que en el punto donde `append(data1)` o `append(data2)` son llamadas, una copia independiente de la variable de estado es creada en memoria y `append` opera en esta copia (que no soporta `.push` - pero eso es otro tema). Las modificaciones a esta copia independiente no se llevan a `data1` o `data2`.

Un error típico es declarar una variable local y presumir que será creada en memoria, aunque será creada en storage:

```
/// ESTE CONTRATO CONTIENE UN ERROR
contract C {
    uint someVariable;
    uint[] data;

    function f() {
        uint[] x;
        x.push(2);
        data = x;
    }
}
```

El tipo de la variable local `x` es `unint[] storage`, pero ya que storage no es asignada dinámicamente, tiene que ser asignada desde una variable de estado antes de que pueda ser usada. Entonces nada de espacio en storage será asignado para `x`, pero en cambio funciona sólo como un alias para variables pre existentes en storage.

Lo que pasará es que el compilador interpreta `x` como un pointer storage y lo apuntará al slot de storage `0` por defecto. Esto significa que `unaVariable` (que reside en slot `0` de storage) es modificada por `x.push(2)`.

La manera correcta de hacerlo es la siguiente:

```
contract C {
    uint someVariable;
    uint[] data;

    function f() {
        uint[] x = data;
        x.push(2);
    }
}
```

¿Cuál es la diferencia entre `bytes` y `byte[]`?

`bytes` es en general mas eficiente: Cuando se usa como argumentos a funciones (ej en CALLDATA) o en memoria, cada elemento de un `byte[]` es acolchado a 32 bytes que derrocha 31 bytes por elemento.

¿Es posible envíar un valor mientras se llama una función que está overloaded?

Es una funcionalidad que falta conocida. https://www.pivotaltracker.com/story/show/92020468 como parte de https://www.pivotaltracker.com/n/projects/1189488

La mejor solución actualmente es introducir un caso de uso especial para gas y valor y simplemente re-checkear si están presentes en el punto de resolución de overload.

Preguntas Avanzadas

¿Cómo se obtiene un número al azar en un contrato? (Emplementar un contrato de apuestas autónomo)

Obtener números al azar es a menudo una parte crucial de un proyecto crypto y la mayoría de los errores vienen de malos generadores de números random.

Si no quieres que sea seguro, puedes contruir algo similar a la moneda flipper si no, mejor usar otro contrato que entrega azar, como el *RANDAO <https://github.com/randao/randao>*`_

Obtener un valor devuelto de una función no constante desde otro contrato

El punto clave es que el contrato que llama necesita saber sobre la función que intenta llamar.

Ver ping.sol y pong.sol.

Hacer que un contrato haga algo apenas es minado por primera vez

Usar el contructor. Cualquier cosa dentro de él será ejecutado cuando el contrato sea minado.

Ver replicator.sol.

¿Cómo se crea un array de dos dimenciones?

Ver 2D_array.sol.

Nótese que llenando un cuadrado de 10x10 de `uint8` + creación de contrato tomó mas de `800,000` gas en el momento de escribir este contrato. 17x17 tomó `2,000,000` gas. Con un límite a 3.14 millones, bueno, hay poco espacio para esto.

Nótese que simplemente "creando" un array es gratis, los costos son en rellenarlo.

Nota2: Optimizando acceso storage puede bajar los costes de gas muchísimo, porque 32 valores `uint8` pueden ser guardados en un slot simple. El problema es que estas optimizaciones actualmente no funcionan en bucles y también tienen un problema con revisión de límites. Aunque, puedes obtener mejores resultados en el futuro.

¿Qué hace `p.recipient.call.value(p.amount)(p.data)`?

Cada llamada de función externa en Solidity se puede modificar de dos maneras:

1. Puedes agregar Ether junto con la llamada
2. Puede limitar la cantidad de gas disponible para la llamada

Esto se hace "llamando a una función de la función":

`f.gas(2).value(20)()` llama a la función modificada `f` y por lo tanto envía 20 Wei y limitando el gas a 2 (así que esta llamada de función probablemente se quedará sin gas y devolvera tus 20 Wei).

En el ejemplo anterior, la función de nivel bajo `call` se utiliza para invocar otro contrato con `p.data` como carga útil y `p.amount` Wei se envía con esa llamada.

¿Qué pasa con el mapeo de una `struct` al copiar sobre una `struct`?

Es una pregunta muy interesante. Supongamos que tenemos un campo en un contrato configurado como tal:

```
struct user {
    mapping(string => address) usedContracts;
}

function somefunction {
   user user1;
   user1.usedContracts["Hello"] = "World";
   user user2 = user1;
}
```

En este caso, se ignora el mapeo de la estructura que se está copiando en la userList, ya que no existe una "lista de teclas mapeadas". Por lo tanto, no es posible averiguar qué valores deben copiarse.

¿Cómo inicializo un contrato con sólo una cantidad específica de wei?

Actualmente el enfoque es un poco feo, pero hay poco que se pueda hacer para mejorarlo. En el caso de un `contract A` llamando a una nueva instancia del `contract B`, los paréntesis tienen que ser usados alrededor de `new B` porque `B.value` se referiría a un miembro de `B` llamado `value`. Necesitaras asegurarte de que tienes ambos contratos conscientes de la presencia del uno al otro y que el `contract B` tenga un constructor `payable`. En este ejemplo:

```
contract B {
    function B() payable {}
}


contract A {
    address child;

    function test() {
        child = (new B).value(10)(); //construct a new B with 10 wei
    }
}
```

¿Puede una función de contrato aceptar un array bidimensional?

Esto no se ha implementado todavía para las llamadas externas y los arrays dinámicos - sólo se puede utilizar un nivel de arrays dinámicos.

¿Cuál es la relación entre `bytes32` y `cadena`? ¿Por qué es que `bytes32 somevar = "stringliteral";` funciona y qué significa el valor hexadecimal de 32 bytes guardado?

El tipo `bytes32` puede contener 32 (raw) bytes. En la asignación `bytes32 samevar = "stringliteral";` , la cadena literal se interpreta en su forma de byte crudo y si lo inspeccionas `somevar` y ves un valor hexadecimal de 32 bytes, esto es sólo `"stringliteral"` en hex.

El tipo `bytes` es similar, sólo que puede cambiar su longitud.

Finalmente, `string` es básicamente idéntica a "bytes", sólo que se asume que para mantener la codificación UTF-8 de una cadena real. Desde que `string` almacena el codificación UTF-8 es bastante caro calcular el número de caracteres en la cadena (la codificación de algunos caracteres requiere más que un solo byte). Debido a eso, `string s; s.length` no es todavía soportado y ni siquiera indexar el acceso `s[2]`. Pero si quieres acceder a la codificación de bytes de bajo nivel de la cadena, puede utilizar `bytes(s).length` and `bytes(s)[2]` lo que resultará en el número de bytes en la codificación UTF-8 de la cadena (no el número de caracteres) y el segundo byte (no carácter) del UTF-8 codificado respectivamente.

¿Puede un contrato pasar un array (tamaño estático) o una cadena o `bytes` (tamaño dinámico) a otro contrato?

Seguro. Ten cuidado de que si cruzas la memoria / límite de almacenamiento, se crearán copias independientes:

```
contract C {
    uint[20] x;
```

```
    function f() {
        g(x);
        h(x);
    }

    function g(uint[20] y) {
        y[2] = 3;
    }

    function h(uint[20] storage y) {
        y[3] = 4;
    }
}
```

La llamada a `g (x)` no tendrá un efecto en `x` porque necesita para crear una copia independiente del valor de almacenamiento en memoria (la ubicación de almacenamiento predeterminada es la memoria). Por otra parte, `h (x)` modifica con éxito `x` porque sólo pasas una referencia y no una copia.

A veces, cuando intento cambiar la longitud de un array con ej: `arrayname.length = 7;` me sale un error de compilado `Value must be an lvalue`. ¿Por qué?

Puedes cambiar el tamaño de un array dinámico en almacenamiento (es decir, un array declarado en el nivel de contrato) con `arrayname.length = <some new length>;`. Si consigues el error de "lvalue", probablemente estés haciendo una de dos cosas mal.

1. Es posible que estés tratando de redimensionar un array en "memoria", o
2. Podrías estar intentando redimensionar un array no dinámico.

```
int8[] memory memArr;         // Case 1
memArr.length++;              // illegal
int8[5] storageArr;           // Case 2
somearray.length++;           // legal
int8[5] storage storageArr2; // Explicit case 2
somearray2.length++;          // legal
```

Nota importante: En Solidity, las dimensiones del array se declaran al revés de la forma en que estés acostumbrado a declararlas en C o Java, pero se acceden como en C o Java.

Por ejemplo, `int8[][5] somearray;` son 5 arrays dinámicos `int8`.

La razón de esto es que `T[5]` es siempre un array de 5 `T`, no importa si `T` en sí mismo es un array o no (esto no es el caso en C o Java).

¿Es posible devolver un array de cadenas (`string[]`) desde una función de Solidity?

Todavía no, ya que esto requiere dos niveles de arrays dinámicos (`string` es un array dinámico en sí).

Si se emite una llamada para un array, es posible recuperar todo el array? ¿O tienes que escribir una función de ayuda para eso?

La función getter automática, para una variable de estado público del tipo array sólo devuelve elementos individuales. Si quieres devolver el array completo, tienes que escribir manualmente una función para hacerlo.

¿Qué podría haber sucedido si una cuenta tiene valor(es) de almacenamiento pero no tiene código? Ejemplo: http://test.ether.camp/account/5f740b3a43fbb99724ce93a879805f4dc89178b5

Lo último que hace un constructor es devolver el código del contrato. Los gastos de gas para esto dependen de la longitud del código y puede ser que el gas suministrado no es suficiente. Esta situación es la única donde una excepción "out of gas" no revierte los cambios al estado, es decir, en este caso la inicialización de las variables de estado.

https://github.com/ethereum/wiki/wiki/Subtleties

Después de la subejecución de una operación CREATE exitosa, si la operación devuelve x, 5 * len(x) gas se resta del gas restante antes de que se crea el contrato. Si el gas restante es menos de 5 * len(x), entonces no se resta ningún gas, el código del contrato creado se convierte en la cadena vacía, pero esto no se trata como una condición excepcional - no se producen reversiones.

¿Qué hace la siguiente extraña comprobación en el contrato de Custom Token?

```
require((balanceOf[_to] + _value) >= balanceOf[_to]);
```

Los números enteros en Solidity (y la mayoría de los otros lenguajes de programación relacionados con máquinas) están restringidos a un rango determinado. Para `uint256`, esto es `0` hasta `2**256 - 1`. Si el resultado de alguna operación en esos números no cabe dentro de este rango, este es truncado. Estos truncamientos pueden tener serias consecuencias, asi que codifica como el de arriba es necesario para evitar ciertos ataques.

Más Preguntas?

Si tienes más preguntas o tu pregunta no se ha contestada aquí, por favor contacta con nosotros en gitter or file an issue.

Índice