
Solidity Documentation

Version 0.8.12

Ethereum

févr. 24, 2022

1	Pour commencer	3
2	Traductions	5
3	Contenu	7
3.1	Introduction Aux Smart Contracts	7
3.2	Installation du compilateur Solidity	16
3.3	Solidity par l'exemple	24
3.4	Mise en page d'un fichier source Solidity	46
3.5	Structure d'un contrat	50
3.6	Types	53
3.7	Unités et variables disponibles dans le monde entier	85
3.8	Expressions et structures de contrôle	91
3.9	Contrats	105
3.10	Assemblage en ligne	144
3.11	Aide-mémoire	149
3.12	Utilisation du compilateur	152
3.13	Analyse de la sortie du compilateur	170
3.14	Changements apportés au Codegen basé sur Solidity IR	173
3.15	Disposition des variables d'état dans le stockage	179
3.16	Mise en page en mémoire	186
3.17	Mise en page des données d'appel	187
3.18	Nettoyer les variables	187
3.19	Source Mappings (SourceMap de Compilation)	188
3.20	L'optimiseur	189
3.21	Métadonnées du contrat	209
3.22	Spécification ABI pour les contrats	212
3.23	Solidity v0.5.0 Changements de rupture	226
3.24	Solidity v0.6.0 Changements de rupture	234
3.25	Solidity v0.7.0 Changements de rupture	237
3.26	Solidity v0.8.0 Changements de rupture	240
3.27	Format NatSpec	242
3.28	Considérations de sécurité	247
3.29	SMTChecker et vérification formelle	254
3.30	Ressources	268
3.31	Résolution du chemin d'importation	270
3.32	Yul	280

3.33	Guide de style	303
3.34	Modèles communs	324
3.35	Liste des bogues connus	331
3.36	Contribution	347
3.37	Guide de la marque Solidity	355
3.38	Influences de la langue	356

Index		357
--------------	--	------------

Solidity est un langage orienté objet et de haut niveau pour la mise en œuvre de contrats intelligents. Les contrats intelligents sont des programmes qui régissent le comportement des comptes dans l'état Ethereum.

Solidity est un [langage d'accolades](#). Il est influencé par le C++, le Python et le JavaScript, et est conçu pour cibler la machine virtuelle Ethereum (EVM). Vous pouvez trouver plus de détails sur les langages dont Solidity s'est inspiré dans la section sur les [influences linguistiques](#).

Solidity est typée statiquement, supporte l'héritage, les bibliothèques et les types complexes définis par l'utilisateur, entre autres caractéristiques.

Avec Solidity, vous pouvez créer des contrats pour des utilisations telles que le vote, le crowdfunding, les enchères à l'aveugle et les portefeuilles à signatures multiples.

Lorsque vous déployez des contrats, vous devez utiliser la dernière version publiée de Solidity. Sauf cas exceptionnel, seule la dernière version reçoit des [correctifs de sécurité](#). En outre, les changements de rupture ainsi que les nouvelles fonctionnalités sont introduites régulièrement. Nous utilisons actuellement un numéro de version 0.y.z [pour indiquer ce rythme rapide de changement](#).

Avertissement : Solidity a récemment publié la version 0.8.x qui a introduit de nombreux changements. Assurez-vous de lire [la liste complète](#).

Les idées pour améliorer Solidity ou cette documentation sont toujours les bienvenues, lisez notre [guide des contributeurs](#) pour plus de détails.

Astuce : Vous pouvez télécharger cette documentation au format PDF, HTML ou Epub en cliquant sur le menu déroulant des versions dans le coin inférieur gauche et en sélectionnant le format de téléchargement préféré.

1. Comprendre les bases des contrats intelligents

Si le concept des contrats intelligents est nouveau pour vous, nous vous recommandons de commencer par vous plonger dans la section « Introduction aux contrats intelligents ». dans la section « Introduction aux contrats intelligents », qui couvre :

- *Un exemple simple de smart contract* écrit sous Solidity.
- *Les bases de la blockchain.*
- *La Ethereum Virtual Machine.*

2. Apprenez à connaître Solidity

Une fois que vous êtes habitué aux bases, nous vous recommandons de lire les sections « *Solidity by Example* » et « Description du langage » pour comprendre les concepts fondamentaux du langage.

3. Installer le compilateur Solidity

Il existe plusieurs façons d'installer le compilateur Solidity. Il vous suffit de choisir votre option préférée et de suivre les étapes décrites sur la *installation page*.

Indication : Vous pouvez essayer des exemples de code directement dans votre navigateur grâce à la fonction [Remix IDE](#). Remix est un IDE basé sur un navigateur web qui vous permet d'écrire, de déployer et d'administrer les smart contracts Solidity, sans avoir à sans avoir besoin d'installer Solidity localement.

Avertissement : Comme les humains écrivent des logiciels, ceux-ci peuvent comporter des bugs. Vous devez suivre les meilleures pratiques établies en matière de développement de logiciels lorsque vous écrivez vos contrats intelligents. Cela inclut la révision du code, les tests, les audits et les preuves de correction. Les utilisateurs de contrats intelligents sont parfois plus confiants dans le code que ses auteurs, et les blockchains et les contrats intelligents ont leurs propres problèmes à surveiller. Avant de travailler sur le code de production, assurez-vous de lire la section *Considérations de sécurité*.

4. En savoir plus

Si vous souhaitez en savoir plus sur la création d'applications décentralisées sur Ethereum, le programme [Ethereum Developer Resources](#) peut vous aider à trouver de la documentation générale sur Ethereum, ainsi qu'une large sélection de tutoriels, d'outils et de cadres de développement.

Si vous avez des questions, vous pouvez essayer de chercher des réponses ou de les poser sur [Ethereum StackExchange](#), ou sur notre [salon Gitter](#).

CHAPITRE 2

Traductions

Des bénévoles de la communauté aident à traduire cette documentation en plusieurs langues. Leur degré d'exhaustivité et de mise à jour varie. La version anglaise est une référence.

Note : Nous avons récemment mis en place une nouvelle organisation GitHub et un nouveau flux de traduction pour aider à rationaliser les efforts de la communauté. Veuillez vous référer au [guide de traduction](#) pour obtenir des informations sur la manière de contribuer aux traductions communautaires en cours.

- Français (en cours)
- Italien (en cours)
- Japonais
- Coréen (en cours)
- Russe (rather outdated)
- Chinois simplifié (en cours)
- Espagnol
- Turc (partiel)

[Index des mots-clés](#), [Page de recherche](#)

3.1 Introduction Aux Smart Contracts

3.1.1 Un Simple Smart Contract

Commençons par un exemple de base qui définit la valeur d'une variable et l'expose à l'accès d'autres contrats. Ce n'est pas grave si vous ne comprenez pas tout de suite, nous entrerons dans les détails plus tard.

Exemple de stockage

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    uint storedData;

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

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

La première ligne vous indique que le code source est sous la licence GPL version 3.0. Les spécificateurs de licence lisibles par machine sont importants dans un contexte où la publication du code source est le défaut.

La ligne suivante spécifie que le code source est écrit pour Solidity version 0.4.16, ou une version plus récente du langage jusqu'à, mais sans inclure, la version 0.9.0. Cela permet de s'assurer que le contrat n'est pas compilable avec une nouvelle version du compilateur (en rupture), où il pourrait se comporter différemment. *Pragmas* sont des instructions courantes pour les compilateurs sur la manière de traiter le code source (par exemple, `pragma once`).

Un contrat, au sens de Solidity, est une collection de code (ses *fonctions*) et de données (son *état*) qui réside à une adresse spécifique sur la blockchain. La ligne `uint storedData;` déclare une variable d'état appelée `storedData` de type `uint` (*unsigned integer* de 256 bits). Vous pouvez l'imaginer comme un emplacement unique dans une base de données que vous pouvez interroger et modifier en appelant des fonctions du code qui gère la base de données. Dans cet exemple, le contrat définit les fonctions `set` et `get` qui peuvent être utilisées pour modifier ou récupérer la valeur de la variable.

Pour accéder à un membre (comme une variable d'état) du contrat en cours, vous n'ajoutez généralement pas le préfixe `this.`, vous y accédez directement par son nom. Contrairement à d'autres langages, l'omettre n'est pas seulement une question de style, il en résulte une façon complètement différente d'accéder au membre, mais nous y reviendrons plus tard.

Ce contrat ne fait pas grand-chose pour l'instant, à part (en raison de l'infrastructure construite par Ethereum) permettant à quiconque de stocker un nombre unique qui est accessible par n'importe qui dans le monde sans un moyen (faisable) de vous empêcher de publier ce numéro. N'importe qui pourrait appeler `set` à nouveau avec une valeur différente et écraser votre numéro, mais le numéro est toujours stocké dans l'historique de la blockchain. Plus tard, vous verrez comment vous pouvez imposer des restrictions d'accès afin que vous soyez le seul à pouvoir modifier le numéro.

Avertissement : Soyez prudent lorsque vous utilisez du texte Unicode, car des caractères d'apparence similaire (ou même identiques) peuvent avoir des points de code différents et sont donc codés dans un tableau d'octets différent.

Note : Tous les identifiants (noms de contrats, noms de fonctions et noms de variables) sont limités au jeu de caractères ASCII. Il est possible de stocker des données encodées en UTF-8 dans des variables de type chaîne.

Exemple de sous-monnaie

Le contrat suivant met en œuvre la forme la plus simple d'une crypto-monnaie. Le contrat permet uniquement à son créateur de créer de nouvelles pièces (différents schémas d'émission sont possibles). Tout le monde peut s'envoyer des pièces sans avoir besoin de s'enregistrer avec un nom d'utilisateur et un mot de passe, tout ce dont vous avez besoin est une paire de clés Ethereum.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract Coin {
    // Le mot clé "public" rend les variables
    // accessibles depuis d'autres contrats
    address public minter;
    mapping (address => uint) public balances;

    // Les événements permettent aux clients de réagir à des
    // changements de contrat que vous déclarez
    event Sent(address from, address to, uint amount);

    // Le code du constructeur n'est exécuté que lorsque le contrat
    // est créé
```

(suite sur la page suivante)

(suite de la page précédente)

```

constructor() {
    minter = msg.sender;
}

// Envoie une quantité de pièces nouvellement créées à une adresse.
// Ne peut être appelé que par le créateur du contrat
function mint(address receiver, uint amount) public {
    require(msg.sender == minter);
    balances[receiver] += amount;
}

// Les erreurs vous permettent de fournir des informations sur
// pourquoi une opération a échoué. Elles sont renvoyées
// à l'appelant de la fonction.
error InsufficientBalance(uint requested, uint available);

// Envoie un montant de pièces existantes
// de n'importe quel appelant à une adresse
function send(address receiver, uint amount) public {
    if (amount > balances[msg.sender])
        revert InsufficientBalance({
            requested: amount,
            available: balances[msg.sender]
        });

    balances[msg.sender] -= amount;
    balances[receiver] += amount;
    emit Sent(msg.sender, receiver, amount);
}
}

```

Ce contrat introduit quelques nouveaux concepts, passons-les en revue un par un.

La ligne `address public minter;` déclare une variable d'état de type *address*. Le type *address* est une valeur de 160 bits qui ne permet aucune opération arithmétique. Il convient pour stocker les adresses des contrats, ou un hachage de la moitié publique d'une paire de clés appartenant à comptes externes.

Le mot clé « `public` » génère automatiquement une fonction qui vous permet d'accéder à la valeur actuelle de la variable d'état depuis l'extérieur du contrat. Sans ce mot-clé, les autres contrats n'ont aucun moyen d'accéder à la variable. Le code de la fonction générée par le compilateur est équivalent à ce qui suit (ignorez `external` et `view` pour le moment) :

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

Vous pourriez ajouter vous-même une fonction comme celle ci-dessus, mais vous auriez une fonction et une variable d'état avec le même nom. Vous n'avez pas besoin de le faire, le compilateur s'en charge pour vous.

La ligne suivante, `mapping (adresse => uint) public balances;` crée également une variable d'état publique, mais il s'agit d'un type de données plus complexe. Le type *mapping* fait correspondre les adresses aux `:ref:`internes non signés <integers>``.

Les mappings peuvent être vus comme des *tableaux de hachage* qui sont initialisées virtuellement, de telle sorte que chaque clé possible existe dès le départ et est mise en correspondance avec une valeur dont la représentation par octet est constituée de zéros. Cependant, il n'est pas possible d'obtenir une liste de toutes les clés d'un mappage, ni une liste de toutes les valeurs. Enregistrez ce que vous avez ajouté au mappage, ou utilisez-le dans un contexte où cela n'est pas

nécessaire. Ou encore mieux, gardez une liste, ou utilisez un type de données plus approprié.

La fonction *getter* créée par le mot-clé `public` est plus complexe dans le cas d'un mapping. Elle ressemble à ce qui suit suivante :

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

Vous pouvez utiliser cette fonction pour demander le solde d'un seul compte.

La ligne event `Sent(adresse from, adresse to, uint amount);` déclare un « événement », qui est émis dans la dernière ligne de la fonction `send`. Les clients Ethereum tels que les applications web peuvent écouter ces événements émis sur la blockchain sans trop de coût. Dès que l'événement est émis, l'écouteur reçoit les arguments « from », « to » et « amount », ce qui permet de suivre les transactions.

Pour écouter cet événement, vous pouvez utiliser le code suivant Du code JavaScript, qui utilise `web3.js` pour créer l'objet du contrat `Coin`, et toute interface utilisateur appelle la fonction `balances` générée automatiquement ci-dessus :

```
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));
    }
})
```

Le *constructeur* est une fonction spéciale qui est exécutée pendant la création du contrat et ne peut pas être appelée par la suite. Dans ce cas, elle stocke de manière permanente l'adresse de la personne qui crée le contrat. La variable `msg` (avec `tx` et `block`) est une *variable globale spéciale* qui contient des propriétés qui permettent d'accéder à la blockchain. `msg.sender` est toujours l'adresse d'où provient l'appel de fonction (externe) actuel.

Les fonctions qui constituent le contrat, et que les utilisateurs et les contrats peuvent appeler sont `mint` et `send`.

La fonction `mint` envoie une quantité de pièces nouvellement créées à une autre adresse. La fonction *require* définit des conditions qui annulent toutes les modifications si elles ne sont pas respectées. Dans cet exemple, `require(msg.sender == minter);` garantit que seul le créateur du contrat peut appeler `mint`. En général, le créateur peut monnayer autant de jetons qu'il le souhaite, mais à un moment donné, cela conduira à un phénomène appelé « overflow ». Notez qu'à cause de l'option par défaut *Checked arithmetic*, la transaction s'inversera si l'expression `balances[receiver] += amount;` déborde, c'est-à-dire lorsque `balances[receiver] + amount` en arithmétique de précision arbitraire est plus grand que la valeur maximale de `uint` ($2^{256} - 1$). Ceci est également vrai pour l'instruction `balances[receiver] += amount;` dans la fonction `send`.

Les erreurs vous permettent de fournir plus d'informations à l'appelant sur pourquoi une condition ou une opération a échoué. Les erreurs sont utilisées avec l'instruction *revert statement*. L'instruction `revert` interrompt et annule sans condition inconditionnellement et annule toutes les modifications, de manière similaire à la fonction `require`, mais elle vous permet également de fournir le nom d'une erreur et des données supplémentaires qui seront fournies à l'appelant (et éventuellement à l'application frontale ou à l'explorateur de blocs) afin qu'un l'application frontale ou l'explorateur de blocs) afin de pouvoir déboguer ou réagir plus facilement à un échec.

La fonction « envoyer » peut être utilisée par n'importe qui (qui possède déjà certaines de ces pièces) pour envoyer un message à un autre utilisateur. qui possède déjà certaines de ces pièces) pour envoyer des pièces à quelqu'un d'autre. Si l'expéditeur n'a pas assez de pièces à envoyer, la condition `if` est évaluée à `true`. En conséquence, la condition `revert` fera échouer l'opération tout en fournissant à l'expéditeur les détails de l'erreur en utilisant l'erreur « `InsufficientBalance` ».

Note : Si vous utilisez ce contrat pour envoyer des pièces de monnaie à une adresse, vous ne verrez rien lorsque vous regardez cette adresse sur un explorateur de blockchain, parce que l'enregistrement que vous avez envoyé des pièces et les soldes modifiés sont uniquement stockés dans le stockage de données de ce contrat de pièces particulier. En utilisant des événements, vous pouvez créer un « explorateur de blockchain » qui suit les transactions et les soldes de votre nouvelle pièce, mais vous devez inspecter l'adresse du contrat de la pièce et non les adresses des propriétaires des pièces.

3.1.2 Les bases de la blockchain

Les blockchains en tant que concept ne sont pas trop difficiles à comprendre pour les programmeurs. La raison en est que la plupart des complications (minage, [hashing](#), [cryptographie à courbe elliptique](#), [réseaux de pair à pair](#), etc.) sont juste là pour fournir un certain ensemble de fonctionnalités et de promesses pour la plate-forme. Une fois que vous acceptez ces caractéristiques comme données, vous n'avez pas à vous soucier de la technologie sous-jacente - ou vous n'avez pas à savoir comment le système AWS d'Amazon fonctionne en interne pour pouvoir l'utiliser ?

Transactions

Une blockchain est une base de données transactionnelle partagée à l'échelle mondiale. Cela signifie que tout le monde peut lire les entrées de la base de données simplement en participant au réseau. Si vous voulez modifier quelque chose dans la base de données, vous devez créer ce qu'on appelle une transaction qui doit être acceptée par tous les autres participants. Le mot « transaction » implique que la modification que vous souhaitez effectuer (supposons que vous souhaitiez modifier deux valeurs en même temps) n'est pas effectuée du tout ou est complètement appliquée. En outre, pendant que votre transaction est appliquée à la base de données, aucune autre transaction ne peut la modifier.

À titre d'exemple, imaginez une table qui répertorie les soldes de tous les comptes dans une monnaie électronique. Si un transfert d'un compte à un autre est demandé, la nature transactionnelle de la base de données garantit que si le montant est soustrait d'un compte, il est toujours ajouté à l'autre compte. Si pour une raison quelconque, l'ajout du montant au compte cible n'est pas possible, le compte source n'est pas non plus modifié.

En outre, une transaction est toujours signée de manière cryptographique par l'expéditeur (créateur). Cela permet de protéger facilement l'accès à certaines modifications de la base de données. Dans l'exemple de la monnaie électronique, un simple contrôle permet de s'assurer que seule la personne détenant les clés du compte peut transférer de l'argent depuis celui-ci.

Blocs

L'un des principaux obstacles à surmonter est ce que l'on appelle (en termes de bitcoin) une « attaque par double dépense » : Que se passe-t-il si deux transactions existent dans le réseau qui veulent toutes deux vider un compte ? Seule une des transactions peut être valide, généralement celle qui est acceptée en premier. Le problème est que « premier » n'est pas un terme objectif dans un réseau peer-to-peer.

La réponse abstraite à cette question est que vous n'avez pas à vous en soucier. Un ordre globalement accepté des transactions sera sélectionné pour vous, résolvant ainsi le conflit. Les transactions seront regroupées dans ce qu'on appelle un « bloc ». puis elles seront exécutées et distribuées entre tous les nœuds participants. Si deux transactions se contredisent, celle qui arrive en deuxième position sera rejetée et ne fera pas partie du bloc.

Ces blocs forment une séquence linéaire dans le temps et c'est de là que vient le mot « blockchain ». Les blocs sont ajoutés à la chaîne à intervalles assez réguliers. Ethereum, c'est à peu près toutes les 17 secondes.

Dans le cadre du « mécanisme de sélection des ordres » (appelé « minage »), il peut arriver que des blocs soient révoqués de temps en temps, mais seulement à la « pointe » de la chaîne. Plus de blocs sont ajoutés au-dessus d'un bloc

particulier, moins ce bloc a de chances d'être inversé. Il se peut donc que vos transactions soient inversées et même supprimées de la blockchain, mais plus vous attendez, moins cela est probable.

Note : Les transactions ne sont pas garanties d'être incluses dans le bloc suivant ou dans un bloc futur spécifique, puisque ce n'est pas à celui qui soumet une transaction, mais aux mineurs de déterminer dans quel bloc la transaction est incluse.

Si vous souhaitez planifier les appels futurs de votre contrat, vous pouvez utiliser un outil d'automatisation de contrat intelligent ou un service oracle.

3.1.3 La machine virtuelle Ethereum

Vue d'ensemble

La machine virtuelle d'Ethereum ou EVM est l'environnement d'exécution pour les contrats intelligents dans Ethereum. Il n'est pas seulement sandboxé mais complètement isolé, ce qui signifie que le code s'exécutant dans l'EVM n'a pas accès au réseau, au système de fichiers ou à d'autres processus. Les smart contracts ont même un accès limité aux autres smart contracts.

Comptes

Il y a deux sortes de comptes dans Ethereum qui partagent le même espace d'adresse : **Les comptes externes** qui sont contrôlés par paires de clés publiques-privées (c'est-à-dire les humains) et **les comptes de contrat** qui sont contrôlés par le code stocké avec le compte.

L'adresse d'un compte externe est déterminée à partir de de la clé publique, tandis que l'adresse d'un contrat est déterminée au moment où le contrat est créé (elle est dérivée de l'adresse du créateur et du nombre de transactions envoyées depuis cette adresse, le fameux « nonce »).

Que le compte stocke ou non du code, les deux types sont traités de la même manière par l'EVM.

Chaque compte dispose d'une mémoire persistante clé-valeur qui met en correspondance des mots de 256 bits avec des mots de 256 bits, appelés **storage**.

En outre, chaque compte dispose d'un **solde** en Ether (en « Wei » pour être exact, « 1 ether » est « 10^{18} wei ») qui peut être modifié en envoyant des transactions qui incluent de l'Ether.

Transactions

Une transaction est un message qui est envoyé d'un compte à un autre compte (qui peut être le même ou vide, voir ci-dessous). Il peut contenir des données binaires (appelées « charge utile ») et de l'Ether.

Si le compte cible contient du code, ce code est exécuté et les données utiles sont fournies comme données d'entrée.

Si le compte cible n'est pas défini (la transaction n'a pas de destinataire ou que le destinataire a la valeur null), la transaction crée un **nouveau contrat**. Comme nous l'avons déjà mentionné, l'adresse de ce contrat n'est pas l'adresse zéro mais une adresse dérivée de l'émetteur et de son nombre de transactions envoyées (le « nonce »). La charge utile d'une telle transaction de création de contrat est considérée comme étant bytecode EVM et est exécutée. Les données de sortie de cette exécution sont stockées de façon permanente en tant que code du contrat. Cela signifie que pour créer un contrat, vous n'envoyez pas le code réel du contrat, mais en fait du code qui renvoie ce code lorsqu'il est exécuté.

Note : Pendant qu'un contrat est en cours de création, son code est encore vide. Pour cette raison, vous ne devriez pas faire appel au contrat en cours de construction avant que son constructeur n'ait fini de s'exécuter.

Gas

Lors de sa création, chaque transaction est chargée d'une certaine quantité de **gaz**, dont le but est de limiter la quantité de travail nécessaire pour exécuter la transaction et de payer en même temps pour cette exécution. Pendant que l'EVM exécute la transaction, le gaz est progressivement épuisé selon des règles spécifiques.

Le **prix du gaz** est une valeur fixée par le créateur de la transaction, qui doit payer « prix du gaz * gaz » à l'avance à partir du compte d'envoi. S'il reste du gaz après l'exécution, il est remboursé au créateur de la même manière.

Si le gaz est épuisé à un moment donné (c'est-à-dire qu'il serait négatif), une exception pour épuisement du gaz est déclenchée, ce qui rétablit toutes les modifications apportées à l'état dans la trame d'appel actuelle.

Stockage, mémoire et pile

La machine virtuelle d'Ethereum a trois zones où elle peut stocker des données- stockage, la mémoire et la pile, qui sont expliqués dans les paragraphes suivants.

Chaque compte dispose d'une zone de données appelée **storage**, qui est persistante entre les appels de fonction et les transactions. Le stockage est un magasin clé-valeur qui fait correspondre des mots de 256 bits à des mots de 256 bits. Il n'est pas possible d'énumérer le stockage à partir d'un contrat. Relativement coûteux à lire, et encore plus à initialiser et à modifier le stockage. En raison de ce coût, vous devez limiter ce que vous stockez dans le stockage persistant à ce dont le contrat a besoin pour fonctionner. Stockez les données telles que les calculs dérivés, la mise en cache et les agrégats en dehors du contrat. Un contrat ne peut ni lire ni écrire dans un stockage autre que le sien.

La deuxième zone de données est appelée **memory**, dont un contrat obtient une instance fraîchement effacée pour chaque appel de message. La mémoire est linéaire et peut être adressée au niveau de l'octet, mais la lecture est limitée à une largeur de 256 bits, tandis que l'écriture peuvent avoir une largeur de 8 bits ou de 256 bits. La mémoire est étendue d'un mot (256 bits), lorsqu'on accède (en lecture ou en écriture) à un mot de mémoire qui n'a pas encore été touché (c'est-à-dire à l'intérieur d'un mot). Au moment de l'expansion, le coût en gaz doit être payé. La mémoire est d'autant plus coûteuse qu'elle est grande (elle s'étend de façon quadratique).

L'EVM n'est pas une machine à registre mais une machine à pile. Tous les calculs sont effectués dans une zone de données appelée la **stack**. Sa taille maximale est de 1024 éléments et contient des mots de 256 bits. L'accès à la pile est limitée à l'extrémité supérieure de la manière suivante : Il est possible de copier l'un des 16 éléments les plus élevés au sommet de la pile ou d'échanger l'élément le plus élevé avec l'un des 16 éléments inférieurs. Il est possible de copier l'un des 16 éléments supérieurs au sommet de la pile ou d'échanger l'élément supérieur avec l'un des 16 éléments inférieurs. Toutes les autres opérations prennent les deux (ou un, ou plusieurs, selon l'opération) de la pile et poussent le résultat sur la pile. Bien sûr, il est possible de déplacer les éléments de la pile vers le stockage ou la mémoire afin d'avoir un accès plus profond à la pile, mais il n'est pas possible d'accéder à des éléments arbitraires plus profondément dans la pile sans avoir préalablement retiré le sommet de la pile.

Jeu d'instructions

Le jeu d'instructions de l'EVM est maintenu à un niveau minimal afin d'éviter les implémentations incorrectes ou incohérentes qui pourraient causer des problèmes de consensus. Toutes les instructions opèrent sur le type de données de base, les mots de 256 bits ou les tranches de mémoire (ou autres tableaux d'octets). Les opérations arithmétiques, binaires, logiques et de comparaison habituelles sont présentes. Les sauts conditionnels et inconditionnels sont possibles. En outre, les contrats peuvent accéder aux propriétés pertinentes du bloc actuel comme son numéro et son horodatage.

Pour une liste complète, veuillez consulter la *liste des opcodes* faisant partie de la documentation de l'assemblage en ligne.

Appels de messages

Les contrats peuvent appeler d'autres contrats ou envoyer de l'Ether à des comptes par le biais d'appels de messages. Les appels de messages sont similaires aux transactions, en ce sens qu'ils ont une source, une cible, des données utiles, de l'Ether, du gaz et des données de retour. En fait, chaque transaction consiste en un appel de message de niveau supérieur qui, à son tour, peut créer d'autres appels de message.

Un contrat peut décider quelle quantité de son **gaz** restant doit être envoyée avec l'appel de message interne et combien il souhaite conserver. Si une exception d'épuisement du gaz se produit dans l'appel interne (ou toute autre exception), cela sera signalé par une valeur d'erreur placée sur la pile. Dans ce cas, seul le gaz envoyé avec l'appel est consommé. Dans Solidity, le contrat d'appel provoque par défaut une exception manuelle dans de telles situations, de sorte que les exceptions « s'accumulent » dans la pile.

Comme déjà dit, le contrat appelé (qui peut être le même que l'appelant) recevra une instance de mémoire fraîchement nettoyée et aura accès à la charge utile de l'appel - qui sera fournie dans une zone séparée appelée **calldata**. Après avoir terminé son exécution, il peut retourner des données qui seront stockées à un emplacement dans la mémoire de l'appelant pré-alloué par ce dernier. Tous ces appels sont entièrement synchrones.

Les appels sont **limités** à une profondeur de 1024, ce qui signifie que pour des opérations plus complexes, les boucles doivent être préférées aux appels récursifs. En outre, seuls 63/64ème du gaz peuvent être transmis dans un appel de message, ce qui entraîne une limite de profondeur d'un peu moins de 1000 en pratique.

Delegatecall / Callcode et bibliothèques

Il existe une variante spéciale d'un appel de message, appelée **delegatecall**, qui est identique à un appel de message, à l'exception du fait que le code à l'adresse cible est exécuté dans le contexte du contrat d'appel et appelant et que les valeurs de `msg.sender` et `msg.value` ne changent pas.

Cela signifie qu'un contrat peut charger dynamiquement du code provenant d'une autre différente au moment de l'exécution. Le stockage, l'adresse actuelle et le solde font toujours référence au contrat appelant, seul le code est pris de l'adresse appelée.

Cela permet de mettre en œuvre la fonctionnalité de « bibliothèque » dans Solidity : Un code de bibliothèque réutilisable qui peut être appliqué au stockage d'un contrat, par exemple pour mettre en œuvre une structure de données complexe.

Logs

Il est possible de stocker des données dans une structure de données spécialement indexée qui s'applique jusqu'au niveau du bloc. Cette fonctionnalité appelée **logs** est utilisée par Solidity afin d'implémenter *events*. Les contrats ne peuvent pas accéder aux données des logs après leur création, mais elles peuvent être efficacement accessibles depuis l'extérieur de la blockchain. Puisqu'une partie des données du journal est stockée dans *bloom filters*, il est possible de rechercher ces données de manière efficace et cryptographique, de sorte que les pairs du réseau qui ne téléchargent pas l'ensemble de la blockchain (appelés « clients légers ») peuvent toujours trouver ces journaux.

Créer

Les contrats peuvent même créer d'autres contrats en utilisant un opcode spécial (c'est-à-dire qu'ils n'appellent pas simplement l'adresse zéro comme le ferait une transaction). La seule différence entre ces appels **create** et les appels de message normaux est que les données utiles sont exécutées et le résultat reçoit l'adresse du nouveau contrat sur la pile.

Désactivation et autodestruction

Le seul moyen de supprimer un code de la blockchain est lorsqu'un contrat à cette adresse effectue l'opération d'« autodestruction ». L'Ether restant stocké à cette adresse est envoyé à une cible désignée et ensuite le stockage et le code est retiré de l'état. En théorie, supprimer le contrat semble être une bonne idée, mais elle est potentiellement dangereuse, car si quelqu'un envoie de l'Ether à des contrats supprimés, l'Ether est perdu à jamais.

Avertissement : Même si un contrat est supprimé par « autodestruction », il fait toujours partie de l'histoire de la blockchain et probablement conservé par la plupart des nœuds Ethereum. Ainsi, utiliser « l'autodestruction » n'est pas la même chose que de supprimer des données d'un disque dur.

Note : Même si le code d'un contrat ne contient pas d'appel à `selfdestruct`, il peut quand même effectuer cette opération en utilisant `delegatecall` ou `callcode`.

Si vous voulez désactiver vos contrats, vous devriez plutôt **désactiver** ceux-ci en modifiant un état interne qui entraîne le retour en arrière de toutes les fonctions. Ceci rend impossible l'utilisation du contrat, car il retourne immédiatement de l'Ether.

Contrats précompilés

Il existe un petit ensemble d'adresses de contrat qui sont spéciales : La plage d'adresses comprise entre 1 et (y compris) 8 contient des « contrats précompilés » qui peuvent être appelés comme n'importe quel autre contrat, mais leur comportement (et leur consommation de gaz) n'est pas défini par le code EVM stocké à cette adresse (ils ne contiennent pas de code), mais est plutôt mis en œuvre dans l'environnement d'exécution EVM lui-même.

Différentes chaînes compatibles EVM peuvent utiliser un ensemble différent de contrats précompilés. Il est également possible que de nouveaux contrats précompilés soient ajoutés à la chaîne principale d'Ethereum à l'avenir, mais vous pouvez raisonnablement vous attendre à ce qu'ils soient toujours dans la gamme entre 1 et `0xffffffff` (inclus).

3.2 Installation du compilateur Solidity

3.2.1 Versionnage

Les versions de Solidity suivent le [versionnement sémantique](#) et en plus des versions, **des builds de développement nocturnes** sont également mis à disposition. Les nightly builds ne sont pas garanties et, malgré tous les efforts, elles peuvent contenir et/ou des changements non documentés. Nous recommandons d'utiliser la dernière version. Les installateurs de paquets ci-dessous utiliseront la dernière version.

3.2.2 Remix

Nous recommandons Remix pour les petits contrats et pour apprendre rapidement Solidity.

Access [Remix en ligne](#), vous n'avez pas besoin d'installer quoi que ce soit. Si vous voulez l'utiliser sans connexion à l'Internet, allez sur <https://github.com/ethereum/remix-live/tree/gh-pages> et téléchargez le fichier `.zip` comme expliqué sur cette page. Remix est également une option pratique pour tester les constructions nocturnes sans installer plusieurs versions de Solidity.

D'autres options sur cette page détaillent l'installation du compilateur Solidity en ligne de commande sur votre ordinateur. Choisissez un compilateur en ligne de commande si vous travaillez sur un contrat plus important ou si vous avez besoin de plus d'options de compilation.

3.2.3 npm / Node.js

Utilisez `npm` pour une manière pratique et portable d'installer `solcjs`, un compilateur Solidity. Le programme `solcjs` a moins de fonctionnalités que les façons d'accéder au compilateur décrites plus bas dans cette page. La documentation *Utilisation du compilateur en ligne de commande* suppose que vous utilisez le compilateur complet, `solc`. L'utilisation de `solcjs` est documentée à l'intérieur de son propre [repository](#).

Note : Le projet `solc-js` est dérivé du projet C++ `solc`. `solc` en utilisant Emscripten ce qui signifie que les deux utilisent le même code source du compilateur. `solc-js` peut être utilisé directement dans des projets JavaScript (comme Remix). Veuillez vous référer au dépôt `solc-js` pour les instructions.

```
npm install -g solc
```

Note : L'exécutable en ligne de commande est nommé `solcjs`.

Les options en ligne de commande de `solcjs` ne sont pas compatibles avec `solc` et les outils (tels que `geth`) qui attendent le comportement de `solc` ne fonctionneront pas avec `solcjs`.

3.2.4 Docker

Les images Docker des constructions Solidity sont disponibles en utilisant l'image `solc` de l'organisation `ethereum`. Utilisez la balise `stable` pour la dernière version publiée, et `nightly` pour les changements potentiellement instables dans la branche de développement.

L'image Docker exécute l'exécutable du compilateur, vous pouvez donc lui passer tous les arguments du compilateur. Par exemple, la commande ci-dessous récupère la version stable de l'image `solc` (si vous ne l'avez pas déjà), et l'exécute dans un nouveau conteneur, en passant l'argument `--help`.

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

Vous pouvez également spécifier les versions de build de la version dans la balise, par exemple, pour la version 0.5.4.

```
docker run ethereum/solc:0.5.4 --help
```

Pour utiliser l'image Docker afin de compiler les fichiers Solidity sur la machine hôte, montez un dossier local pour l'entrée et la sortie, et spécifier le contrat à compiler. Par exemple.

```
docker run -v /local/path:/sources ethereum/solc:stable -o /sources/output --abi --bin /
↪sources/Contract.sol
```

Vous pouvez également utiliser l'interface JSON standard (ce qui est recommandé lorsque vous utilisez le compilateur avec des outils). Lors de l'utilisation de cette interface, il n'est pas nécessaire de monter des répertoires tant que l'entrée JSON est autonome (c'est-à-dire qu'il ne fait pas référence à des fichiers externes qui devraient être *chargés par la callback d'importation*).

```
docker run ethereum/solc:stable --standard-json < input.json > output.json
```

3.2.5 Paquets Linux

Les paquets binaires de Solidity sont disponibles à l'adresse [solidity/releases](https://github.com/ethereum/solidity/releases).

Nous avons également des PPA pour Ubuntu, vous pouvez obtenir la dernière version stable en utilisant les commandes suivantes :

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

La version nocturne peut être installée en utilisant ces commandes :

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

Nous publions également un paquet [snap](#), qui est installable dans toutes les [distros Linux supportées](#). Pour installer la dernière version stable de solc :

```
sudo snap install solc
```

Si vous voulez aider à tester la dernière version de développement de Solidity avec les changements les plus récents, veuillez utiliser ce qui suit :

```
sudo snap install solc --edge
```

Note : Le snap solc utilise un confinement strict. Il s'agit du mode le plus sûr pour les paquets snap mais il comporte des limitations, comme l'accès aux seuls fichiers de vos répertoires `/home` et `/media`. Pour plus d'informations, consultez la page [Démystifier le confinement de Snap](#).

Arch Linux dispose également de paquets, bien que limités à la dernière version de développement :

```
pacman -S solidity
```

Gentoo Linux possède un [Ethereum overlay](#) qui contient un paquet Solidity. Après la configuration de l'overlay, `solc` peut être installé dans les architectures `x86_64` par :

```
emerge dev-lang/solidity
```

3.2.6 Paquets macOS

Nous distribuons le compilateur Solidity via Homebrew comme une version construite à partir des sources. Les bouteilles préconstruites ne sont actuellement pas supportées.

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

Pour installer la plus récente version 0.4.x / 0.5.x de Solidity, vous pouvez également utiliser `brew install solidity@4` et `brew install solidity@5`, respectivement.

Si vous avez besoin d'une version spécifique de Solidity, vous pouvez installer une formule Homebrew directement depuis Github.

Voir [solidity.rb commits](#) sur Github.

Copiez le hash de commit de la version que vous voulez et vérifiez-la sur votre machine.

```
git clone https://github.com/ethereum/homebrew-ethereum.git  
cd homebrew-ethereum  
git checkout <your-hash-goes-here>
```

Installez-le en utilisant brew :

```
brew unlink solidity  
# eg. Install 0.4.8  
brew install solidity.rb
```

3.2.7 Binaires statiques

Nous maintenons un dépôt contenant des constructions statiques des versions passées et actuelles du compilateur pour toutes les plateformes supportées. plates-formes supportées à [solc-bin](#). C'est aussi l'endroit où vous pouvez trouver les nightly builds.

Le dépôt n'est pas seulement un moyen rapide et facile pour les utilisateurs finaux d'obtenir des binaires prêts à l'emploi, mais il est également conçu pour être convivial pour les outils tiers :

- Le contenu est mis en miroir sur <https://binaries.soliditylang.org>, où il peut être facilement téléchargé via HTTPS sans authentification, ni contrôle. HTTPS sans authentification, limitation de débit ou nécessité d'utiliser git.
- Le contenu est servi avec des en-têtes *Content-Type* corrects et une configuration CORS indulgente afin qu'il puisse être directement chargé par des outils s'exécutant dans le navigateur.
- Les binaires ne nécessitent pas d'installation ou de déballage (à l'exception des anciennes versions de Windows fournies avec les DLL nécessaires).

- Nous nous efforçons d'assurer un haut niveau de compatibilité ascendante. Les fichiers, une fois ajoutés, ne sont pas supprimés ou déplacés sans fournir un lien symbolique/une redirection à l'ancien emplacement. Ils ne sont jamais modifiés non plus en place et doivent toujours correspondre à la somme de contrôle d'origine. La seule exception serait les fichiers cassés ou inutilisables, susceptibles de causer plus de tort que de bien s'ils sont laissés en l'état.
- Les fichiers sont servis à la fois par HTTP et HTTPS. Tant que vous obtenez la liste des fichiers d'une manière sécurisée (via git, HTTPS, IPFS ou simplement en la mettant en cache localement) et que vous vérifiez les hachages des binaires après les avoir téléchargés, vous n'avez pas besoin d'utiliser HTTPS pour les binaires eux-mêmes.

Les mêmes binaires sont dans la plupart des cas disponibles sur la page [`Solidity release page on Github`](#). La différence est que nous ne mettons généralement pas à jour les anciennes versions sur la page Github. Cela signifie que nous ne les renommons pas si la convention de nommage change et que nous n'ajoutons pas de builds pour les plates-formes qui n'étaient pas supportées au moment de la publication. Ceci n'arrive que dans solc-bin.

Le dépôt solc-bin contient plusieurs répertoires de haut niveau, chacun représentant une seule plate-forme. Chacun contient un fichier `list.json` listant les binaires disponibles. Par exemple dans `emscripten-wasm32/list.json`, vous trouverez les informations suivantes sur la version 0.7.4 :

```
{
  "path": "solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js",
  "version": "0.7.4",
  "build": "commit.3f05b770",
  "longVersion": "0.7.4+commit.3f05b770",
  "keccak256": "0x300330ecd127756b824aa13e843cb1f43c473cb22eaf3750d5fb9c99279af8c3",
  "sha256": "0x2b55ed5fec4d9625b6c7b3ab1abd2b7fb7dd2a9c68543bf0323db2c7e2d55af2",
  "urls": [
    "bzzr://16c5f09109c793db99fe35f037c6092b061bd39260ee7a677c8a97f18c955ab1",
    "dweb:/ipfs/QmTLs5MuLEWXQkths41HiACoXDiH8zxyqBHGfDRSzVE5CS"
  ]
}
```

Cela signifie que :

- Vous pouvez trouver le binaire dans le même répertoire sous le nom de `solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js`. Notez que le fichier pourrait être un lien symbolique, et vous devrez le résoudre vous-même si vous n'utilisez pas git pour le télécharger ou si votre système de fichiers ne supporte pas les liens symboliques.
- Le binaire est également mis en miroir à <https://binaries.soliditylang.org/emscripten-wasm32/solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js>. Dans ce cas, git n'est pas nécessaire et les liens symboliques sont résolus de manière transparente, soit en fournissant une copie du fichier ou en renvoyant une redirection HTTP.
- Le fichier est également disponible sur IPFS à l'adresse `QmTLs5MuLEWXQkths41HiACoXDiH8zxyqBHGfDRSzVE5CS`.
- Le fichier pourrait à l'avenir être disponible sur Swarm à l'adresse `16c5f09109c793db99fe35f037c6092b061bd39260ee7a677c8a97f18c955ab1`.
- Vous pouvez vérifier l'intégrité du binaire en comparant son hachage keccak256 à `0x300330ecd127756b824aa13e843cb1f43c473cb22eaf3750d5fb9c99279af8c3`. Le hachage peut être calculé en ligne de commande à l'aide de l'utilitaire `keccak256sum` fourni par `sha3sum` ou de la fonction `keccak256()` de `ethereumjs-util` en JavaScript.
- Vous pouvez également vérifier l'intégrité du binaire en comparant son hachage sha256 à `0x2b55ed5fec4d9625b6c7b3ab1abd2b7fb7dd2a9c68543bf0323db2c7e2d55af2`.

Avvertissement : En raison de la forte exigence de compatibilité ascendante, le référentiel contient quelques éléments anciens mais vous devriez éviter de les utiliser lorsque vous écrivez de nouveaux outils :

- Utilisez `emscripten-wasm32/` (avec une solution de repli sur `emscripten-asmjs/`) au lieu de `bin/` si vous voulez les meilleures performances. Jusqu'à la version 0.6.1, nous ne fournissions que les binaires

asm.js. À partir de la version 0.6.2, nous sommes passés à des constructions ``WebAssembly`_` avec de bien meilleures performances. Nous avons reconstruit les anciennes versions pour wasm mais les fichiers asm.js originaux restent dans bin/. Les nouveaux fichiers ont dû être placés dans un répertoire séparé pour éviter les conflits de noms.

- Utilisez `emscripten-asmjs/` et `emscripten-wasm32/` au lieu des répertoires `bin/` et `wasm/` si vous voulez être sûr que vous téléchargez un binaire wasm ou asm.js.
- Utilisez `list.json` au lieu de `list.js` et `list.txt`. Le format de liste JSON contient toutes les informations des anciens formats et plus encore.
- Utilisez <https://binaries.soliditylang.org> au lieu de <https://solc-bin.ethereum.org>. Pour garder les choses simples, nous avons déplacé presque tout ce qui concerne le compilateur sous le nouveau domaine `soliditylang.org`, et cela s'applique aussi à `solc-bin`. Bien que le nouveau domaine soit recommandé, l'ancien domaine est toujours entièrement supporté et garanti pour pointer au même endroit.

Avertissement : Les binaires sont également disponibles à <https://ethereum.github.io/solc-bin/> mais cette page a cessé d'être mise à jour juste après la sortie de la version 0.7.2, ne recevra pas de nouvelles versions ou nightly builds pour n'importe quelle plateforme et ne sert pas la nouvelle structure de répertoire, y compris les constructions non-emscriptées.

Si vous l'utilisez, veuillez basculer vers <https://binaries.soliditylang.org>, qui est une solution de remplacement. Ceci nous permet d'apporter des changements à l'hébergement sous-jacent de manière transparente et de minimiser les perturbations. Contrairement au domaine `ethereum.github.io`, sur lequel nous n'avons aucun contrôle, ```binaries.github.io``` est un domaine sur lequel nous n'avons aucun contrôle, « `binaries.soliditylang.org` » est garanti de fonctionner et de maintenir la même structure d'URL à long terme.

3.2.8 Construire à partir de la source

Conditions préalables - Tous les systèmes d'exploitation

Les éléments suivants sont des dépendances pour toutes les versions de Solidity :

Logiciel	Notes
CMake (version 3.13+)	Générateur de fichiers de construction multiplateforme.
Boost (version 1.77+ sur Windows, 1.65+ sinon)	Librairies C++.
Git	Outil en ligne de commande pour la récupération du code source.
z3 (version 4.8+, Optionnel)	À utiliser avec le vérificateur SMT.
cvc4 (Optionnel)	À utiliser avec le vérificateur SMT.

Note : Les versions de Solidity antérieures à 0.5.10 ne parviennent pas à se lier correctement avec les versions Boost 1.70+. Une solution possible est de renommer temporairement le répertoire `<Chemin d'installation de Boost>/lib/cmake/Boost-1.70.0` avant d'exécuter la commande `cmake` pour configurer solidity.

A partir de la 0.5.10, la liaison avec Boost 1.70+ devrait fonctionner sans intervention manuelle.

Note : La configuration de construction par défaut requiert une version spécifique de Z3 (la plus récente au moment de la dernière mise à jour du code). Les changements introduits entre les versions de Z3 entraînent souvent des résultats légèrement différents (mais toujours valides). Nos tests SMT ne tiennent pas compte de ces différences et échoueront probablement avec une version différente de celle pour laquelle ils ont été écrits. Cela ne veut pas dire qu'une

compilation utilisant une version différente est défectueuse. Si vous passez l'option `-DSTRICT_Z3_VERSION=OFF` à CMake, vous pouvez construire avec n'importe quelle version qui satisfait aux exigences données dans la table ci-dessus. Si vous faites cela, cependant, n'oubliez pas de passer l'option `--no-smt` à `scripts/tests.sh` pour sauter les tests SMT.

Versions minimales du compilateur

Les compilateurs C++ suivants et leurs versions minimales peuvent construire la base de code Solidity :

- GCC, version 8+
- Clang, version 7+
- MSVC, version 2019+

Conditions préalables - macOS

Pour les builds macOS, assurez-vous que vous avez la dernière version de Xcode installée. Cela contient le compilateur Clang C++, l'Xcode IDE et d'autres outils qui sont nécessaires à la création d'applications C++ sous OS X. Si vous installez Xcode pour la première fois, ou si vous venez d'installer une nouvelle version, vous devrez accepter la licence avant de pouvoir effectuer des constructions en ligne de commande :

```
sudo xcodebuild -license accept
```

Notre script de construction OS X utilise le gestionnaire de paquets Homebrew <<https://brew.sh>> pour installer les dépendances externes. Voici comment désinstaller Homebrew, si vous voulez un jour repartir de zéro.

Conditions préalables - Windows

Vous devez installer les dépendances suivantes pour les versions Windows de Solidity :

Logiciel	Notes
Visual Studio 2019 Outils de construction	C++ compiler
Visual Studio 2019 (Optionnel)	Compilateur C++ et environnement de développement.
Boost (version 1.77+)	Librairies C++.

Si vous avez déjà un IDE et que vous avez seulement besoin du compilateur et des bibliothèques, vous pouvez installer Visual Studio 2019 Build Tools.

Visual Studio 2019 fournit à la fois l'IDE et le compilateur et les bibliothèques nécessaires. Donc, si vous n'avez pas d'IDE et que vous préférez développer Solidity, Visual Studio 2019 peut être un choix pour vous afin de tout configurer facilement.

Voici la liste des composants qui doivent être installés dans Visual Studio 2019 Build Tools ou Visual Studio 2019 :

- Fonctions de base de Visual Studio C++
- VC++ 2019 v141 toolset (x86,x64)
- SDK CRT universel Windows
- SDK Windows 8.1
- Support C++/CLI

Nous avons un script d'aide que vous pouvez utiliser pour installer toutes les dépendances externes requises :

```
scripts\install_deps.ps1
```

Ceci installera boost et cmake dans le sous-répertoire deps.

Clonez le référentiel

Pour cloner le code source, exécutez la commande suivante :

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

Si vous voulez aider à développer Solidity, vous devez forker Solidity et ajouter votre fork personnel en tant que second remote :

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

Note : Cette méthode aboutira à une construction preerelease conduisant par exemple à ce qu'un drapeau dans chaque bytecode produit par un tel compilateur. Si vous souhaitez recompiler un compilateur Solidity déjà publié, alors veuillez utiliser le tarball source sur la page de publication github :

https://github.com/ethereum/solidity/releases/download/v0.X.Y/solidity_0.X.Y.tar.gz

(et non le « code source » fourni par github).

Construction en ligne de commande

Assurez-vous d'installer les dépendances externes (voir ci-dessus) avant la construction.

Le projet Solidity utilise CMake pour configurer la construction. Vous pourriez vouloir installer `ccache` pour accélérer les constructions répétées, CMake le récupérera automatiquement. La construction de Solidity est assez similaire sur Linux, macOS et autres Unices :

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

ou encore plus facilement sur Linux et macOS, vous pouvez exécuter :

```
#note: this will install binaries solc and soltest at usr/local/bin
./scripts/build.sh
```

Avertissement : Les versions BSD devraient fonctionner, mais ne sont pas testées par l'équipe Solidity.

Et pour Windows :

```
mkdir build
cd build
cmake -G "Visual Studio 16 2019" ..
```

Si vous voulez utiliser la version de boost installée par `scripts\install_deps.ps1`, vous aurez vous devrez en plus passer `-DBoost_DIR="deps\boost\lib\cmake\Boost-*" et -DCMAKE_MSVC_RUNTIME_LIBRARY=MultiThreaded comme arguments à l'appel à cmake.`

Cela devrait entraîner la création de `solidity.sln` dans ce répertoire de construction. En double-cliquant sur ce fichier, Visual Studio devrait se lancer. Nous suggérons de construire **Release**, mais toutes les autres configurations fonctionnent.

Alternativement, vous pouvez construire pour Windows sur la ligne de commande, comme ceci :

```
cmake --build . --config Release
```

3.2.9 Options CMake

Si vous êtes intéressé par les options CMake disponibles, lancez `cmake .. -LH`.

Solveurs SMT

Solidity peut être construit avec des solveurs SMT et le fera par défaut s'ils sont trouvés dans le système. Chaque solveur peut être désactivé par une option `cmake`.

Note : Dans certains cas, cela peut également être une solution de contournement potentielle pour les échecs de construction.

Dans le dossier de construction, vous pouvez les désactiver, puisqu'ils sont activés par défaut :

```
# disables only Z3 SMT Solver.
cmake .. -DUSE_Z3=OFF

# disables only CVC4 SMT Solver.
cmake .. -DUSE_CVC4=OFF

# disables both Z3 and CVC4
cmake .. -DUSE_CVC4=OFF -DUSE_Z3=OFF
```

3.2.10 La chaîne de version en détail

La chaîne de la version de Solidity contient quatre parties :

- le numéro de version
- l'étiquette de préversion, généralement définie par `development.YYYY.MM.DD` ou `nightly.YYYY.MM.DD`.
- le commit au format `commit.GITHASH`.
- platform, qui comporte un nombre arbitraire d'éléments, contenant des détails sur la plate-forme et le compilateur.

S'il y a des modifications locales, le commit sera postfixé avec `.mod`.

Ces parties sont combinées comme requis par SemVer, où la balise pre-release Solidity est égale à la pre-release SemVer et le commit Solidity et la plateforme combinés constituent les métadonnées de construction SemVer.

Exemple de version : `» 0.4.8+commit.60cc1668.Emscripten.clang « .`

Exemple de préversion : `» 0.4.9-nightly.2017.1.17+commit.6ecb4aa3.Emscripten.clang « .`

3.2.11 Informations importantes sur les versions

Après la sortie d'une version, le niveau de version du patch est augmenté, car nous supposons que seuls les changements de niveau patch suivent. Lorsque les changements sont fusionnés, la version doit être augmentée en fonction de SemVer et de la gravité de la modification. Enfin, une version est toujours faite avec la version du nightly build actuel, mais sans le spécificateur ```prerelease```.

Exemple :

0. La version 0.4.0 est faite.
1. Le nightly build a une version 0.4.1 à partir de maintenant.

2. Des changements non cassants sont introduits → pas de changement de version.
3. Un changement de rupture est introduit → la version passe à 0.5.0.
4. La version 0.5.0 est publiée.

Ce comportement fonctionne bien avec la version *pragma*.

3.3 Solidity par l'exemple

3.3.1 Contrat de Vote

Le contrat suivant est assez complexe, mais met en valeur de nombreuses fonctionnalités de Solidity. Il met en place un Contrat de Vote. Bien sûr, les principaux problèmes du vote électronique est la façon d'attribuer des droits de vote au bon personnes et comment prévenir la manipulation. Nous n'allons pas résoudre tous les problèmes ici, mais au moins nous montrerons comment le vote délégué peut être fait pour que le décompte des voix est **automatique et complètement transparent** en même temps.

L'idée est de créer un contrat par scrutin, fournissant un nom court pour chaque option. Ensuite, le créateur du contrat qui sert de président donnera le droit de vote à chacun adresse individuellement.

Les personnes derrière les adresses peuvent alors choisir de voter eux-mêmes ou de déléguer leur vote pour une personne de confiance.

A la fin du temps de vote, `winningProposal()` renverra la proposition avec le plus grand nombre de suffrages.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
/// @title Vote avec délégation.
contract Ballot {
    // Ceci déclare un nouveau type complexe qui va
    // être utilisé pour les variables plus tard.
    // Il représentera un seul électeur.
    struct Voter {
        uint weight; // le poids est cumulé par délégation
        bool voted; // si vrai, cette personne a déjà voté
        address delegate; // personne déléguée à
        uint vote; // index de la proposition votée
    }

    // Il s'agit d'un type pour une seule proposition.
    struct Proposal {
        bytes32 name; // nom court (jusqu'à 32 octets)
        uint voteCount; // nombre de votes cumulés
    }

    address public chairperson;

    // Ceci déclare une variable d'état qui
    // stocke une structure `Voter` pour chaque adresse possible.
    mapping(address => Voter) public voters;

    // Un tableau de taille dynamique de structures `Proposal`.
    Proposal[] public proposals;
```

(suite sur la page suivante)

(suite de la page précédente)

```

/// Créez un nouveau bulletin de vote pour choisir l'un des `proposalNames`.
constructor(bytes32[] memory proposalNames) {
    chairperson = msg.sender;
    voters[chairperson].weight = 1;

    // Pour chacun des noms de proposition fournis,
    // crée un nouvel objet de proposition et l'ajoute
    // à la fin du tableau.
    for (uint i = 0; i < proposalNames.length; i++) {
        // `Proposal({...})` crée un temporaire
        // Objet de proposition et `proposals.push(...)`
        // l'ajoute à la fin de `proposals`.
        proposals.push(Proposal({
            name: proposalNames[i],
            voteCount: 0
        }));
    }
}

// Donne à `voter` le droit de voter sur ce bulletin de vote.
// Ne peut être appelé que par `chairperson`.
function giveRightToVote(address voter) external {
    // Si le premier argument de `require` est
    // `false`, l'exécution se termine et tout
    // modifications de l'état et des soldes Ether
    // sont annulés.
    // Cela consommait tout le gaz dans les anciennes versions d'EVM, mais
    // plus maintenant.
    // C'est souvent une bonne idée d'utiliser `require` pour vérifier si
    // les fonctions sont appelées correctement.
    // Comme deuxième argument, vous pouvez également fournir un
    // explication de ce qui s'est mal passé.
    require(
        msg.sender == chairperson,
        "Seul le président peut donner droit de vote."
    );
    require(
        !voters[voter].voted,
        "L'électeur a déjà voté."
    );
    require(voters[voter].weight == 0);
    voters[voter].weight = 1;
}

/// Déléguez votre vote au votant `to`.
function delegate(address to) external {
    // attribue une référence
    Voter storage sender = voters[msg.sender];

    require(!sender.voted, "Vous avez déjà voté.");
    require(to != msg.sender, "L'autodélégation est interdite.");
}

```

(suite sur la page suivante)

```

// Transférer la délégation tant que
// `to` également délégué.
// En général, de telles boucles sont très dangereuses,
// parce que s'ils tournent trop longtemps, ils pourraient
// qvoir besoin de plus de gaz que ce qui est disponible dans un bloc.
// Dans ce cas, la délégation ne sera pas exécutée,
// mais dans d'autres situations, de telles boucles pourraient
// provoquer le "blocage" complet d'un contrat.
while (voters[to].delegate != address(0)) {
    to = voters[to].delegate;

    // Nous avons trouvé une boucle dans la délégation, non autorisée.
    require(to != msg.sender, "Found loop in delegation.");
}

// Puisque `sender` est une référence, cela
// modifie `voters[msg.sender].voted`
sender.voted = true;
sender.delegate = to;
Voter storage delegate_ = voters[to];
if (delegate_.voted) {
    // Si le délégué a déjà voté,
    // ajouter directement au nombre de votes
    proposals[delegate_.vote].voteCount += sender.weight;
} else {
    // Si le délégué n'a pas encore voté,
    // ajoute à son poids.
    delegate_.weight += sender.weight;
}
}

/// Donnez votre vote (y compris les votes qui vous sont délégués)
/// à la proposition `propositions[proposition].nom`.
function vote(uint proposal) external {
    Voter storage sender = voters[msg.sender];

    require(sender.weight != 0, "N'a pas le droit de voter");
    require(!sender.voted, "Déjà voté.");

    sender.voted = true;
    sender.vote = proposal;

    // Si `proposal` est hors de la plage du tableau,
    // cela lancera automatiquement et annulera tout
    // changements.
    proposals[proposal].voteCount += sender.weight;
}

/// @dev Calcule la proposition gagnante en prenant tous
/// les votes précédents en compte.
function winningProposal() public view
    returns (uint winningProposal_)

```

(suite de la page précédente)

```

{
    uint winningVoteCount = 0;
    for (uint p = 0; p < proposals.length; p++) {
        if (proposals[p].voteCount > winningVoteCount) {
            winningVoteCount = proposals[p].voteCount;
            winningProposal_ = p;
        }
    }
}

// Appelle la fonction winProposal() pour obtenir l'index
// du gagnant contenu dans le tableau de propositions puis
// renvoie le nom du gagnant
function winnerName() external view
    returns (bytes32 winnerName_)
{
    winnerName_ = proposals[winningProposal()].name;
}
}

```

Améliorations possibles

Actuellement, de nombreuses transactions sont nécessaires pour céder les droits de voter à tous les participants. Pouvez-vous penser à une meilleure façon ?

3.3.2 Enchères à l'aveugle

Dans cette section, nous allons montrer à quel point il est facile de créer un smart contrat d'enchères sur Ethereum. Nous allons commencer par un contrat d'enchère où tout le monde peut voir les offres qui sont faites, puis nous étendrons ce contrat pour des d'enchères à aveugles où il n'est pas possible de voir l'offre réelle jusqu'à la fin de la période d'enchères.

Simple Enchères

L'idée générale d'une enchères est que chacun peut envoyer ses offres pendant une période d'enchères. Les offres doivent comprendre avec l'envoi un certain nombre Ether pour valider leur enchère. Si l'offre la plus élevée est augmentée, le précédent enchérisseur le plus élevé récupère son argent. Après la fin de la période d'enchères, le contrat doit être appelé manuellement pour que le bénéficiaire reçoive son argent - les contrats ne peuvent pas s'activer eux-mêmes.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract SimpleAuction {
    // Paramètres des enchères. Le temps est soit
    // un timestamp unix (nombres de secondes depuis le 01-01-1970)
    // ou une période en secondes.
    address payable public beneficiary;
    uint public auctionEndTime;

    // État actuel de l'enchère.

```

(suite sur la page suivante)

```

address public highestBidder;
uint public highestBid;

// Listes de tous les enrichisseurs pouvant retirer leurs enchères
mapping(address => uint) pendingReturns;

// Définit sur `true` à la fin de l'enchère, pour refuser les changements
// By default initialized to `false`.
bool ended;

// Événements qui vont être émis lors des enchères (pour votre front-end, par
↳exemple)
event HighestBidIncreased(address bidder, uint amount);
event AuctionEnded(address winner, uint amount);

// Erreurs qui décrivent les potentielles problèmes rencontrés.

// Les commentaires à triple barre oblique sont appelés commentaires natspec.
// Ils seront affichés lorsque l'utilisateur
// est invité à confirmer une transaction ou à confirmer une opération.
// lorsqu'une erreur est affichée.

/// L'enchère est terminée.
error AuctionAlreadyEnded();
/// Il existe déjà une offre supérieure ou égale.
error BidNotHighEnough(uint highestBid);
/// L'enchère n'est pas encore terminée.
error AuctionNotYetEnded();
/// La fonction auctionEnd a déjà été appelée.
error AuctionEndAlreadyCalled();

/// Créer une simple enchère avec `biddingTime`
/// en secondes avant la fin de l'enchère (3600=1H)
/// et `beneficiaryAddress` au nom de l'adresse l'auteur de l'enchère.
constructor(
    uint biddingTime,
    address payable beneficiaryAddress
) {
    beneficiary = beneficiaryAddress;
    auctionEndTime = block.timestamp + biddingTime;
}

/// Enchérir sur l'enchère avec la valeur envoyée
/// avec cette transaction.
/// La valeur ne sera remboursée que si le
/// l'enchère n'est pas gagnée.
function bid() external payable {
    // Aucun argument n'est nécessaire, toutes
    // informations fait déjà partie de
    // la transaction. Le mot clé "payable"
    // est requis pour que la fonction
    // puisse recevoir Ether.

```

(suite de la page précédente)

```

// Renvoie (revert) l'appel si la période
// de l'enchère est terminée.
if (block.timestamp > auctionEndTime)
    revert AuctionAlreadyEnded();

// Si l'enchère n'est pas plus élevée, le
// remboursement est envoyé
// ("revert" annulera tous les changements incluant
// l'argent reçu, qui sera automatiquement renvoyer au propriétaire).
if (msg.value <= highestBid)
    revert BidNotHighEnough(highestBid);

if (highestBid != 0) {
    // Renvoyer l'argent en utilisant simplement
    // "mostbidder.send(highestBid)" est un risque de sécurité
    // car il ça pourrait exécuter un contrat non fiable.
    // Il est toujours plus sûr de laisser les destinataires
    // retirer leur argent eux-mêmes.
    pendingReturns[highestBidder] += highestBid;
}
highestBidder = msg.sender;
highestBid = msg.value;
emit HighestBidIncreased(msg.sender, msg.value);
}

/// Retirer une enchère qui a été surenchérie.
function withdraw() external returns (bool) {
    uint amount = pendingReturns[msg.sender];
    if (amount > 0) {
        // Il est important de remettre à zéro l'enchère du destinataire
        // car il peut rappeler cette fonction et récupérer un seconde fois son
↪enchère
        // puis une troisième, quatrième fois...
        pendingReturns[msg.sender] = 0;

        // msg.sender n'est pas de type `address payable` mais il le doit
        // du type adresse payable (pour dire à solidity qu'il peut envoyer de l
↪'argent dessus
        // grâce à `send()`)
        // La conversion `address` -> `address payable` peut se faire grâce
        // à `payable(msg.sender)`
        if (!payable(msg.sender)).send(amount)) {
            // Si la tx ne s'exécute pas:
            // Pas besoin de renvoyer une erreur ici, remettez juste l'argent à l
↪'encherriseur,
            // il pourra revenir plus tard
            pendingReturns[msg.sender] = amount;
            return false;
        }
    }
}
return true;

```

(suite sur la page suivante)

```

}

/// Terminez l'enchère et envoyez l'offre la plus élevée
/// au bénéficiaire.
function auctionEnd() external {
    // C'est un bon guide pour structurer les fonctions qui interagissent
    // avec d'autres contrats (c'est-à-dire qu'ils appellent des fonctions ou envoient
    ↪ de l'Ether)
    // en trois phases :
    // 1. conditions de vérification
    // 2. effectuer des actions
    // 3. interaction avec d'autres contrats
    // Si ces phases sont mélangées, d'autre contrat pourrait
    // modifier l'état ou
    // prendre des actions (paiement d'éther) à effectuer plusieurs fois.
    // Si les fonctions appelées en interne incluent l'interaction avec des
    // contrats, ils doivent également être considérés comme une interaction avec
    // des contrats externes.

    // 1. Conditions
    if (block.timestamp < auctionEndTime)
        revert AuctionNotYetEnded();
    if (ended)
        revert AuctionEndAlreadyCalled();

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

    // 3. Interactions
    beneficiary.transfer(highestBid);
}
}

```

Blind Auction

Nous allons maintenant étendre ce contract à une enchère à l'aveugle. L'avantage d'une enchère à l'aveugle c'est qu'il n'y a pas de pression temporelle vers la fin de la période d'enchère. La création d'une enchère à l'aveugle sur une plateforme transparente peut sembler contradictoire, mais la cryptographie vient à la rescousse.

Pendant la **période d'enchère**, un enchérisseur n'envoie pas réellement son offre, mais seulement une version hachée de celle-ci. Étant donné qu'il est actuellement considéré comme pratiquement impossible de trouver deux valeurs (suffisamment longues) dont les hash sont égales, l'enchérisseur s'engage à faire son offre par ce biais. À la fin de la période d'enchères, les enchérisseurs doivent révéler leurs offres : Ils envoient leurs valeurs non cryptées et le contrat vérifie que le hash est le même que celui fournie pendant la période d'enchères.

Un autre défi est de savoir comment rendre l'enchère **liante et aveugle** en même temps. La seule façon d'empêcher l'enchérisseur de ne pas envoyer l'argent après avoir remporté l'enchère après avoir remporté l'enchère est de l'obliger à l'envoyer en même temps que l'offre. Puisque les transferts de valeur ne peuvent pas être censurée dans Ethereum, tout le monde peut voir leur valeur.

Le contrat suivant résout ce problème en acceptant toute valeur qui est supérieure à l'offre la plus élevée. Puisque cela ne peut bien sûr être vérifié que pendant la phase de révélation, certaines offres peuvent être **invalides**, et c'est voulu (il

y a même un drapeau explicite pour placer des offres invalides avec des transferts de grande valeur) : Les enchérisseurs peuvent confondre la concurrence en plaçant plusieurs offres non valides, hautes ou basses.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract BlindAuction {
    struct Bid {
        bytes32 blindedBid;
        uint deposit;
    }

    address payable public beneficiary;
    uint public biddingEnd;
    uint public revealEnd;
    bool public ended;

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

    address public highestBidder;
    uint public highestBid;

    // Permettre le retrait des offres précédentes
    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);

    // Erreurs qui décrivent des échecs.

    /// La fonction a été appelée trop tôt.
    /// Essayez à nouveau à `time`.
    error TooEarly(uint time);
    /// La fonction a été appelée trop tard.
    /// Elle ne peut pas être appelée après `time`.
    error TooLate(uint time);
    /// La fonction auctionEnd a déjà été appelée.
    error AuctionEndAlreadyCalled();

    // Les modificateurs sont un moyen pratique de valider les entrées de
    // fonctions. `onlyBefore` est appliqué à `bid` ci-dessous :
    // Le nouveau corps de la fonction est le corps du modificateur où
    // `_` est remplacé par l'ancien corps de la fonction.
    modifier onlyBefore(uint time) {
        if (block.timestamp >= time) revert TooLate(time);
        _;
    }
    modifier onlyAfter(uint time) {
        if (block.timestamp <= time) revert TooEarly(time);
        _;
    }

    constructor(
        uint biddingTime,
        uint revealTime,
```

(suite sur la page suivante)

```

    address payable beneficiaryAddress
  ) {
    beneficiary = beneficiaryAddress;
    biddingEnd = block.timestamp + biddingTime;
    revealEnd = biddingEnd + revealTime;
  }

  /// Placez une enchère aveugle avec `blindedBid` =
  /// keccak256(abi.encodePacked(value, fake, secret)).
  /// L'éther envoyé n'est remboursé que si l'offre est
  /// correctement révélée lors de la phase de révélation. L'offre est valide si
  /// l'éther envoyé avec l'offre est au moins égal à "value" et que
  /// "fake" n'est pas vrai. Mettre "fake" à true et ne pas envoyer
  /// le montant exact sont des moyens de cacher la véritable enchère mais
  /// tout en effectuant le dépôt requis. Une même adresse peut
  /// placer plusieurs enchères.
  function bid(bytes32 blindedBid)
    external
    payable
    onlyBefore(biddingEnd)
  {
    bids[msg.sender].push(Bid({
      blindedBid: blindedBid,
      deposit: msg.value
    }));
  }

  /// Révélez vos enchères aveugles. Vous obtiendrez un remboursement pour toutes
  /// les offres invalides correctement masquées et pour toutes les offres sauf pour
  /// l'enchère la plus élevée.
  function reveal(
    uint[] calldata values,
    bool[] calldata fakes,
    bytes32[] calldata secrets
  )
    external
    onlyAfter(biddingEnd)
    onlyBefore(revealEnd)
  {
    uint length = bids[msg.sender].length;
    require(values.length == length);
    require(fakes.length == length);
    require(secrets.length == length);

    uint refund;
    for (uint i = 0; i < length; i++) {
      Bid storage bidToCheck = bids[msg.sender][i];
      (uint value, bool fake, bytes32 secret) =
        (values[i], fakes[i], secrets[i]);
      if (bidToCheck.blindedBid != keccak256(abi.encodePacked(value, fake,
↵secret))) {
        // L'enchère n'a pas été réellement révélée.

```

(suite sur la page suivante)

(suite de la page précédente)

```

        // Ne pas rembourser le dépôt.
        continue;
    }
    refund += bidToCheck.deposit;
    if (!fake && bidToCheck.deposit >= value) {
        if (placeBid(msg.sender, value))
            refund -= value;
    }
    // Rendre impossible pour l'expéditeur de réclamer à nouveau
    // le même dépôt.
    bidToCheck.blindedBid = bytes32(0);
}
payable(msg.sender).transfer(refund);
}

/// Retirer une offre qui a été surenchérie.
function withdraw() external {
    uint amount = pendingReturns[msg.sender];
    if (amount > 0) {
        // Il est important de mettre cette valeur à zéro car le destinataire
        // peut appeler cette fonction à nouveau dans le cadre de l'appel de
↪réception
        // avant que `transfer` ne revienne (voir la remarque ci-dessus à propos des
        // conditions -> effets -> interaction).
        pendingReturns[msg.sender] = 0;

        payable(msg.sender).transfer(amount);
    }
}

/// Mettre fin à l'enchère et envoyer l'offre la plus élevée
/// au bénéficiaire.
function auctionEnd()
    external
    onlyAfter(revealEnd)
{
    if (ended) revert AuctionEndAlreadyCalled();
    emit AuctionEnded(highestBidder, highestBid);
    ended = true;
    beneficiary.transfer(highestBid);
}

// Il s'agit d'une fonction "interne", ce qui signifie qu'elle
// ne peut être appelée qu'à partir du contrat lui-même (ou à partir de
// contrats dérivés).
function placeBid(address bidder, uint value) internal
    returns (bool success)
{
    if (value <= highestBid) {
        return false;
    }
    if (highestBidder != address(0)) {

```

(suite sur la page suivante)

```

        // Refund the previously highest bidder.
        pendingReturns[highestBidder] += highestBid;
    }
    highestBid = value;
    highestBidder = bidder;
    return true;
}
}

```

3.3.3 Achat à distance sécurisé

L'achat de biens à distance nécessite actuellement plusieurs parties qui doivent se faire confiance. La configuration la plus simple implique un vendeur et un acheteur. L'acheteur souhaite recevoir un article du vendeur et le vendeur souhaite obtenir de l'argent (ou un équivalent) en retour. La partie problématique est l'expédition ici : il n'y a aucun moyen de déterminer pour sûr que l'article est arrivé à l'acheteur.

Il existe plusieurs façons de résoudre ce problème, mais toutes échouent d'une manière ou d'une autre. Dans l'exemple suivant, les deux parties doivent mettre deux fois la valeur de l'article dans le contrat en tant qu'entiercement. Dès que cela s'est produit, l'argent restera enfermé à l'intérieur le contrat jusqu'à ce que l'acheteur confirme qu'il a bien reçu l'objet. Après ça, l'acheteur reçoit la valeur (la moitié de son acompte) et le vendeur reçoit trois fois la valeur (leur dépôt plus la valeur). L'idée derrière c'est que les deux parties ont une incitation à résoudre la situation ou autrement leur argent est verrouillé pour toujours.

Bien entendu, ce contrat ne résout pas le problème, mais donne un aperçu de la manière dont vous pouvez utiliser des constructions de type machine d'état dans un contrat.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract Purchase {
    uint public value;
    address payable public seller;
    address payable public buyer;

    enum State { Created, Locked, Release, Inactive }
    // La variable d'état a une valeur par défaut du premier membre, `State.created`
    State public state;

    modifier condition(bool condition_) {
        require(condition_);
        -;
    }

    /// Seul l'acheteur peut appeler cette fonction.
    error OnlyBuyer();
    /// Seul le vendeur peut appeler cette fonction.
    error OnlySeller();
    /// La fonction ne peut pas être appelée à l'état actuel.
    error InvalidState();
    /// La valeur fournie doit être paire.
    error ValueNotEven();

    modifier onlyBuyer() {

```

(suite sur la page suivante)

(suite de la page précédente)

```

    if (msg.sender != buyer)
        revert OnlyBuyer();
    -;
}

modifier onlySeller() {
    if (msg.sender != seller)
        revert OnlySeller();
    -;
}

modifier inState(State state_) {
    if (state != state_)
        revert InvalidState();
    -;
}

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

// Assurez-vous que `msg.value` est un nombre pair.
// La division sera tronquée si c'est un nombre impair.
// Vérifie par multiplication qu'il ne s'agit pas d'un nombre impair.
constructor() payable {
    seller = payable(msg.sender);
    value = msg.value / 2;
    if ((2 * value) != msg.value)
        revert ValueNotEven();
}

/// Abandonnez l'achat et récupérez l'éther.
/// Ne peut être appelé que par le vendeur avant
/// le contrat est verrouillé.
function abort()
    external
    onlySeller
    inState(State.Created)
{
    emit Aborted();
    state = State.Inactive;
    // Nous utilisons directement le transfert ici. Il est
    // anti-réentrance, car c'est le
    // dernier appel dans cette fonction et nous
    // a déjà changé l'état.
    seller.transfer(address(this).balance);
}

/// Confirmez l'achat en tant qu'acheteur.
/// La transaction doit inclure `2 * value` ether.
/// L'éther sera verrouillé jusqu'à confirmationReceived

```

(suite sur la page suivante)

```
/// soit appelé.
function confirmPurchase()
    external
    inState(State.Created)
    condition(msg.value == (2 * value))
    payable
{
    emit PurchaseConfirmed();
    buyer = payable(msg.sender);
    state = State.Locked;
}

/// Confirmez que vous (l'acheteur) avez reçu l'article.
/// Cela libérera l'éther verrouillé.
function confirmReceived()
    external
    onlyBuyer
    inState(State.Locked)
{
    emit ItemReceived();
    // Il est important de changer d'abord l'état car
    // sinon, les contrats appelés en utilisant `send` ci-dessous
    // peut rappeler ici.
    state = State.Release;

    buyer.transfer(value);
}

/// Cette fonction rembourse le vendeur, c'est-à-dire
/// rembourse les fonds bloqués du vendeur.
function refundSeller()
    external
    onlySeller
    inState(State.Release)
{
    emit SellerRefunded();
    // Il est important de changer d'abord l'état car
    // sinon, les contrats appelés en utilisant `send` ci-dessous
    // peut rappeler ici.
    state = State.Inactive;

    seller.transfer(3 * value);
}
}
```

3.3.4 Canal de micropaiement

Dans cette section, nous allons apprendre à construire un exemple d'implémentation d'un canal de paiement. Il utilisera des signatures cryptographiques pour faire des transferts répétés d'Ether entre les mêmes parties sécurisés, instantanés et sans frais de transaction. Pour l'exemple, nous devons comprendre comment signer et vérifier les signatures, et configurer le canal de paiement.

Création et vérification de signatures

Imaginez qu'Alice veuille envoyer de l'éther à Bob, c'est-à-dire Alice est l'expéditeur et Bob est le destinataire.

Alice n'a besoin que d'envoyer des messages signés cryptographiquement off-chain (exemple : par e-mail) à Bob et c'est similaire à la rédaction de chèques.

Alice et Bob utilisent des signatures pour autoriser les transactions, ce qui est possible avec les Smart Contract d'Ethereum. Alice construira un simple Smart Contract qui lui permettra de transmettre Ether, mais au lieu d'appeler elle-même une fonction pour initier un paiement, elle laissera Bob le faire, qui paiera donc les frais de transaction.

Le contrat fonctionnera comme ça :

1. Alice déploie le contrat `ReceiverPays`, avec suffisamment d'Ether pour couvrir les paiements qui seront effectués.
2. Alice autorise un paiement en signant un message avec sa clé privée.
3. **Alice envoie le message signé cryptographiquement à Bob. Le message n'a pas besoin d'être gardé secret** (expliqué plus loin), et le mécanisme pour l'envoyer n'a pas d'importance.
4. **Bob réclame son paiement en présentant le message signé au smart contract, celui-ci vérifie le** l'authenticité du message, puis débloque les fonds.

Création de la signature :

Alice n'a pas besoin d'interagir avec le réseau Ethereum pour signer la transaction, le processus est complètement hors ligne. Dans ce tutoriel, nous allons signer des messages dans le navigateur en utilisant `web3.js` et `MetaMask`, avec la méthode décrite dans l'[EIP-712](#), car il offre un certain nombre d'autres avantages en matière de sécurité.

```

/// Hasher en premier, va nous faciliter les choses
const hash = web3.utils.sha3("message to sign");
web3.eth.personal.sign(hash, web3.eth.defaultAccount, function () { console.log("Signed
↵"); });

```

Note : Le `web3.eth.personal.sign` ajoute la longueur du message aux données signées. Puisque nous hachons d'abord, le message fera toujours exactement 32 octets, et donc cette longueur le préfixe est toujours le même.

Quoi signer ?

Pour qu'un contrat exécute des paiements, le message signé doit inclure :

1. L'adresse du destinataire.
2. Le montant à transférer.
3. Protection contre les attaques par replay (replay attacks in English).

*Une attaque par rejeu se produit lorsqu'un message signé est réutilisé pour réclamer une seconde fois l'autorisation de la même action (exemple : renvoyer le même montant d'Eth). Pour éviter ces attaques nous utilisons la même technique que dans les transactions Ethereum elles-mêmes, le fameux *nonce*, qui est le nombre de transactions envoyées par un compte. Le Smart Contract vérifie si le *nonce* est utilisé plusieurs fois.

Un autre type d'attaque par rejeu peut se produire lorsque le propriétaire déploie un Smart Contract `ReceiverPays`, fait quelques paiements, puis détruit le contrat. Plus tard, ils décident de déployer à nouveau le Smart Contract `RecipientPays`, mais le nouveau contrat ne connaît pas les nonces utilisés dans le précédent déploiement, donc les attaquants peuvent à nouveau utiliser les anciens messages.

Alice peut se protéger contre cette attaque en incluant le l'adresse du contrat dans le message, et seuls les messages contenant l'adresse du contrat seront acceptés. Tu peux trouver un exemple de ceci dans les deux premières lignes de la fonction `claimPayment()` du contrat complet à la fin de cette section.

Packing arguments

Maintenant que nous avons identifié les informations à inclure dans le message signé, nous sommes prêts à construire le message, à le hacher et à le signer. Par question de simplicité, nous concaténons les données. Le `ethereumjs-abi` fournit une fonction appelée `soliditySHA3` qui imite le comportement de la fonction `keccak256` de Solidity en appliquant aux arguments encodés la fonction `abi.encodePacked`. Voici une fonction JavaScript qui crée la bonne signature pour l'exemple `ReceiverPays` :

```
// Le "recipient" est l'adresse qui doit être payée.
// Le "amount" est en wei et spécifie la quantité d'éther à envoyer.
// "nonce" peut être n'importe quel nombre unique pour empêcher les attaques par rejeu
// "contractAddress" est utilisé pour empêcher les attaques de relecture de contrats.
↳croisés
function signPayment(recipient, amount, nonce, contractAddress, callback) {
  var hash = "0x" + abi.soliditySHA3(
    ["address", "uint256", "uint256", "address"],
    [recipient, amount, nonce, contractAddress]
  ).toString("hex");

  web3.eth.personal.sign(hash, web3.eth.defaultAccount, callback);
}
```

Récupération du signataire du message dans Solidity

En général, les signatures ECDSA se composent de deux paramètres, *r* et *s*. Les signatures dans Ethereum incluent un troisième paramètre appelé *v*, que vous pouvez utiliser pour vérifier quel clé privée du compte a été utilisée pour signer le message, et l'expéditeur de la transaction. Solidity fournit une fonction `ecrecover` qui accepte un message avec les paramètres *r*, *s* et *v* et renvoie l'adresse qui a été utilisée pour signer le message.

Extraction des paramètres de signature

Les signatures produites par web3.js sont la concaténation de `r`, `s` et `v`, la première étape consiste donc à diviser ces paramètres à part. Vous pouvez le faire côté client, mais le faire à l'intérieur le Smart Contract signifie que vous n'avez besoin d'envoyer qu'un seule paramètre signature plutôt que trois. Séparer un Array d'octets en ses parties constituantes est un gâchis, nous utilisons donc *inline assembly* pour faire le travail dans la fonction `splitSignature` (la troisième fonction dans le contrat complet à la fin de cette section).

Haché le message

Le Smart Contract doit savoir exactement quels paramètres ont été signés, et donc il doit recréer le message à partir des paramètres et l'utiliser pour la vérification de la signature. Les fonctions `prefixed` et `recoverSigner` le font dans la fonction `claimPayment`.

Le contrat complet

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract ReceiverPays {
    address owner = msg.sender;

    mapping(uint256 => bool) usedNonces;

    constructor() payable {}

    function claimPayment(uint256 amount, uint256 nonce, bytes memory signature)
↳external {
        require(!usedNonces[nonce]);
        usedNonces[nonce] = true;

        // ceci recrée le message qui a été signé sur le client
        bytes32 message = prefixed(keccak256(abi.encodePacked(msg.sender, amount, nonce,
↳this)));

        require(recoverSigner(message, signature) == owner);

        payable(msg.sender).transfer(amount);
    }

    /// détruit le contrat et récupère les fonds restants.
    function shutdown() external {
        require(msg.sender == owner);
        selfdestruct(payable(msg.sender));
    }

    /// La method de signature.
    function splitSignature(bytes memory sig)
        internal
        pure
        returns (uint8 v, bytes32 r, bytes32 s)
    {
```

(suite sur la page suivante)

```

require(sig.length == 65);

assembly {
    // 32 premiers octets, après le préfixe de longueur.
    r := mload(add(sig, 32))
    // 32 octets suivant.
    s := mload(add(sig, 64))
    // Derniers octets (premier octet des 32 octets suivants).
    v := byte(0, mload(add(sig, 96)))
}

return (v, r, s);
}

function recoverSigner(bytes32 message, bytes memory sig)
    internal
    pure
    returns (address)
{
    (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

    return ecrecover(message, v, r, s);
}

/// construit un hachage préfixé pour imiter le comportement de eth_sign.
function prefixed(bytes32 hash) internal pure returns (bytes32) {
    return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
}
}

```

Écrire un canal de paiement simplifié

Alice construit maintenant une implémentation simple mais complète d'un paiement canalisé. Les canaux de paiement utilisent des signatures cryptographiques pour effectuer transferts répétés d'Ether en toute sécurité, instantanément et sans frais de transaction.

Qu'est-ce qu'un canal de paiement ?

Les canaux de paiement permettent aux participants d'effectuer des transferts répétés d'Ether sans utiliser de transactions. Cela signifie que vous pouvez éviter les retards et les frais liés aux transactions. Nous allons explorer un simple canal de paiement unidirectionnel entre deux parties (Alice et Bob). Cela implique trois étapes :

1. Alice finance un contrat intelligent avec Ether. Cela « ouvre » le canal de paiement.
2. Alice signe des messages qui précisent combien de cet Ether est dû au destinataire. Cette étape est répétée pour chaque paiement.
3. Bob « ferme » le canal de paiement, retire sa part de l'Ether et renvoie le reste à l'expéditeur.

Note : Seules les étapes 1 et 3 nécessitent des transactions Ethereum, l'étape 2 signifie que l'expéditeur transmet un message signé cryptographiquement au destinataire via des méthodes off-chain (exemple : par e-mail). Cela signifie que seules deux transactions sont nécessaires pour prendre en charge n'importe quel nombre de transferts.

Bob est assuré de recevoir ses fonds car le Smart Contract garde l'Ether et honore un message signé valide. Le Smart Contract impose également un délai d'attente, donc Alice est garantie de récupérer éventuellement ses fonds même si le destinataire refuse de fermer le canal. C'est l'initiateur du paiement qui décide combien de temps il gardera le canal ouvert. Pour une transaction de courte durée, comme payer un cybercafé pour chaque minute d'accès au réseau, le paiement sera maintenu ouvert pendant une durée limitée. En revanche, pour un paiement récurrent, comme le paiement d'un salaire à un employé, le canal de paiement peuvent rester ouverts pendant plusieurs mois ou années.

Ouverture du canal de paiement

Pour ouvrir le canal de paiement, Alice déploie le Smart Contract, attachant l'Ether à garder et en précisant le destinataire prévu et une durée maximale d'existence du canal. C'est la fonction `SimplePaymentChannel` dans le contrat, à la fin de cette section.

Effectuer des paiements

Alice effectue des paiements en envoyant des messages signés à Bob. Cette étape est effectuée entièrement en dehors du réseau Ethereum. Les messages sont signés cryptographiquement par l'expéditeur, puis transmis directement au destinataire.

Chaque message comprend les informations suivantes :

- L'adresse du Smart Contract, utilisée pour empêcher les attaques de relecture de contrats croisés.
- Le montant total d'Ether qui est dû au destinataire jusqu'à présent.

Un canal de paiement n'est fermé qu'une seule fois, à la fin d'une série de virements. Pour cette raison, seul un des messages envoyés est racheté. C'est pourquoi chaque message spécifie un montant total cumulé d'Ether dû, plutôt que le montant du micropaiement individuel. Le destinataire choisira naturellement de racheter le message le plus récent car c'est celui avec le total le plus élevé. Le nonce par message n'est plus nécessaire, car le Smart Contract n'honore qu'un seul message. L'adresse du contrat intelligent est toujours utilisée pour empêcher qu'un message destiné à un canal de paiement ne soit utilisé pour un autre canal.

Voici le code JavaScript modifié pour signer cryptographiquement un message de la section précédente :

```
function constructPaymentMessage(contractAddress, amount) {
  return abi.soliditySHA3(
    ["address", "uint256"],
    [contractAddress, amount]
  );
}

function signMessage(message, callback) {
  web3.eth.personal.sign(
    "0x" + message.toString("hex"),
    web3.eth.defaultAccount,
    callback
  );
}

// contractAddress est utilisé pour empêcher les attaques de relecture de contrats
// ↪croisés.
// Le montant, en wei, spécifie la quantité d'Ether à envoyer.

function signPayment(contractAddress, amount, callback) {
  var message = constructPaymentMessage(contractAddress, amount);
```

(suite sur la page suivante)

```

    signMessage(message, callback);
}

```

Fermeture du canal de paiement

Lorsque Bob est prêt à recevoir ses fonds, il est temps de fermer le canal de paiement en appelant une fonction `close` sur le Smart Contrat. La fermeture du canal paie au destinataire l'éther qui lui est dû et détruit le contrat, renvoyant tout Ether restant à Alice. À fermer le canal, Bob doit fournir un message signé par Alice.

Le Smart Contrat doit vérifier que le message contient une signature valide de l'expéditeur. Le processus pour effectuer cette vérification est le même que celui utilisé par le destinataire. Les fonctions `isValidSignature` et `recoverSigner` (Solidity) fonctionnent exactement comme leur les fonctions JavaScript dans la section précédente, cette dernière fonction étant empruntée au contrat `ReceiverPays`.

Seul le destinataire du canal de paiement peut appeler la fonction `close`, qui transmet naturellement le message de paiement le plus récent parce que ce message porte le total dû le plus élevé. Si l'expéditeur était autorisé à appeler cette fonction, ils pourraient fournir un message avec un montant inférieur et tromper le destinataire sur ce qui lui est dû.

La fonction vérifie que le message signé correspond aux paramètres donnés. Si tout se vérifie, le destinataire reçoit sa part de l'Ether, et l'expéditeur reçoit le reste via un `selfdestruction`. Vous pouvez voir la fonction `close` dans le contrat complet.

Expiration du canal

Bob peut fermer le canal de paiement à tout moment, mais s'il ne le fait pas, Alice a besoin d'un moyen de récupérer ses fonds bloqués. Un délai d'*expiration* a été défini au moment du déploiement du contrat. Une fois ce délai atteint, Alice peut appeler `claimTimeout` pour récupérer ses fonds. Vous pouvez voir la fonction `claimTimeout` dans le contrat complet.

Après l'appel de cette fonction, Bob ne peut plus recevoir d'Ether, il est donc important que Bob ferme le canal avant que l'expiration ne soit atteinte.

Le contrat complet

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract SimplePaymentChannel {
    address payable public sender; // The account sending payments.
    address payable public recipient; // The account receiving the payments.
    uint256 public expiration; // Timeout in case the recipient never closes.

    constructor (address payable recipientAddress, uint256 duration)
        payable
    {
        sender = payable(msg.sender);
        recipient = recipientAddress;
        expiration = block.timestamp + duration;
    }

    /// le destinataire peut fermer le canal à tout moment en présentant un

```

(suite sur la page suivante)

(suite de la page précédente)

```

/// montant signé de l'expéditeur. le destinataire recevra ce montant,
/// et le reste reviendra à l'expéditeur
function close(uint256 amount, bytes memory signature) external {
    require(msg.sender == recipient);
    require(isValidSignature(amount, signature));

    recipient.transfer(amount);
    selfdestruct(sender);
}

/// l'expéditeur peut prolonger l'expiration à tout moment
function extend(uint256 newExpiration) external {
    require(msg.sender == sender);
    require(newExpiration > expiration);

    expiration = newExpiration;
}

/// si le timeout est atteint sans que le destinataire ferme le canal,
/// puis l'Ether est renvoyé à l'expéditeur.
function claimTimeout() external {
    require(block.timestamp >= expiration);
    selfdestruct(sender);
}

function isValidSignature(uint256 amount, bytes memory signature)
    internal
    view
    returns (bool)
{
    bytes32 message = prefixed(keccak256(abi.encodePacked(this, amount)));

    // vérifie que la signature provient de l'expéditeur du paiement
    return recoverSigner(message, signature) == sender;
}

/// Toutes les fonctions ci-dessous sont extraites du chapitre
/// chapitre 'Création et vérification de signatures'.

function splitSignature(bytes memory sig)
    internal
    pure
    returns (uint8 v, bytes32 r, bytes32 s)
{
    require(sig.length == 65);

    assembly {
        // first 32 bytes, after the length prefix
        r := mload(add(sig, 32))
        // second 32 bytes
        s := mload(add(sig, 64))
        // final byte (first byte of the next 32 bytes)
    }
}

```

(suite sur la page suivante)

```

        v := byte(0, mload(add(sig, 96)))
    }

    return (v, r, s);
}

function recoverSigner(bytes32 message, bytes memory sig)
    internal
    pure
    returns (address)
{
    (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

    return ecrecover(message, v, r, s);
}

/// construit un hachage préfixé pour imiter le comportement de eth_sign.
function prefixed(bytes32 hash) internal pure returns (bytes32) {
    return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
}
}

```

Note : La fonction `splitSignature` n'utilise pas toutes les sécurités nécessaires pour un Smart Contrat sécurisé. Une véritable implémentation devrait utiliser une bibliothèque plus rigoureusement testée, comme la [version d'openzeppelin](#) de ce code.

Vérification des paiements

Contrairement à la section précédente, les messages d'un canal de paiement ne sont pas racheté tout de suite. Le destinataire garde une trace du dernier message et l'échange lorsqu'il est temps de fermer le canal de paiement. Cela signifie que c'est critique que le destinataire effectue sa propre vérification de chaque message. Sinon, il n'y a aucune garantie que le destinataire pourra être payé à la fin.

Le destinataire doit vérifier chaque message en utilisant le processus suivant :

1. Vérifiez que l'adresse du contrat dans le message correspond au canal de paiement.
2. Vérifiez que le nouveau total correspond au montant attendu.
3. Vérifiez que le nouveau total ne dépasse pas le montant d'Ether bloqué.
4. Vérifiez que la signature est valide et provient de l'expéditeur du canal de paiement.

Nous utiliserons la librairie `ethereumjs-util` pour écrire cette vérification. L'étape finale peut être effectuée de plusieurs façons, et nous utilisons JavaScript. Le code suivant emprunte la fonction `constructPaymentMessage` au **code JavaScript** de signature ci-dessus :

```

// cela imite le comportement de préfixation de la méthode eth_sign JSON-RPC.
function prefixed(hash) {
    return ethereumjs.ABI.soliditySHA3(
        ["string", "bytes32"],
        ["\x19Ethereum Signed Message:\n32", hash]
    );
}

```

(suite sur la page suivante)

(suite de la page précédente)

```

}

function recoverSigner(message, signature) {
    var split = ethereumjs.Util.fromRpcSig(signature);
    var publicKey = ethereumjs.Util.ecrecover(message, split.v, split.r, split.s);
    var signer = ethereumjs.Util.pubToAddress(publicKey).toString("hex");
    return signer;
}

function isValidSignature(contractAddress, amount, signature, expectedSigner) {
    var message = prefixed(constructPaymentMessage(contractAddress, amount));
    var signer = recoverSigner(message, signature);
    return signer.toLowerCase() ==
        ethereumjs.Util.stripHexPrefix(expectedSigner).toLowerCase();
}

```

3.3.5 Contrats modulaires (Librairie)

Une approche modulaire de la construction de vos contrats vous aide à réduire la complexité et améliorer la lisibilité ce qui aidera à identifier les bugs et les vulnérabilités pendant le développement et la relecture de code. Si vous spécifiez et contrôlez le comportement de chaque module, les interactions que vous devrez prendre en compte sont uniquement celles entre les spécifications du module et non toutes les autres parties mobiles du contrat. Dans l'exemple ci-dessous, le contrat utilise la méthode `move` des `Balances` (*library*) pour vérifier que les soldes envoyés entre les adresses correspondent à ce que vous attendez. Ainsi, la *library* `Balances` fournit un composant isolé des contrats qui suit correctement les soldes des comptes. Il est facile de vérifier que la *library* `Balances` ne produise jamais de soldes négatifs ou de débordements grâce au terme `require()`. De ce faites, la somme de tous les soldes est un invariant sur la durée de vie du contrat.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

library Balances {
    function move(mapping(address => uint256) storage balances, address from, address to,
    ↪ uint amount) internal {
        require(balances[from] >= amount);
        require(balances[to] + amount >= balances[to]);
        balances[from] -= amount;
        balances[to] += amount;
    }
}

contract Token {
    mapping(address => uint256) balances;
    using Balances for *;
    mapping(address => mapping (address => uint256)) allowed;

    event Transfer(address from, address to, uint amount);
    event Approval(address owner, address spender, uint amount);

    function transfer(address to, uint amount) external returns (bool success) {
        balances.move(msg.sender, to, amount);
    }
}

```

(suite sur la page suivante)

```

    emit Transfer(msg.sender, to, amount);
    return true;
}

function transferFrom(address from, address to, uint amount) external returns (bool_
↳success) {
    require(allowed[from][msg.sender] >= amount);
    allowed[from][msg.sender] -= amount;
    balances.move(from, to, amount);
    emit Transfer(from, to, amount);
    return true;
}

function approve(address spender, uint tokens) external returns (bool success) {
    require(allowed[msg.sender][spender] == 0, "");
    allowed[msg.sender][spender] = tokens;
    emit Approval(msg.sender, spender, tokens);
    return true;
}

function balanceOf(address tokenOwner) external view returns (uint balance) {
    return balances[tokenOwner];
}
}

```

3.4 Mise en page d'un fichier source Solidity

Les fichiers sources peuvent contenir un nombre arbitraire de *définitions des contrats*, directives d'importation, *directives pragmatiques* et *struct*, *enum*, *function*, *error* et *constant variable* définitions.

3.4.1 Identificateur de licence SPDX

La confiance dans les contrats intelligents peut être mieux établie si leur code source est disponible. Puisque la mise à disposition du code source touche toujours à des problèmes juridiques en ce qui concerne le droit d'auteur, le compilateur Solidity encourage l'utilisation d'identifiants de licence *SPDX lisibles par machine*. Chaque fichier source doit commencer par un commentaire indiquant sa licence :

```
// SPDX-License-Identifier: MIT
```

Le compilateur ne valide pas que la licence fait partie de la *liste autorisée par SPDX*, mais il inclut la chaîne fournie dans les *métadonnées du code source*.

Si vous ne voulez pas spécifier une licence ou si le code source n'est pas open-source, veuillez utiliser la valeur spéciale UNLICENSED.

Le fait de fournir ce commentaire ne vous libère bien sûr pas des autres obligations liées à la licence, comme l'obligation de mentionner un en-tête de licence spécifique dans chaque fichier source ou le détenteur du droit d'auteur original.

Le commentaire est reconnu par le compilateur à n'importe quel endroit du fichier, mais il est recommandé de le placer en haut du fichier.

Plus d'informations sur la façon d'utiliser les identifiants de licence SPDX peuvent être trouvées sur le site web de [SPDX](#).

3.4.2 Pragma

Le mot-clé `pragma` est utilisé pour activer certaines fonctionnalités du compilateur ou des vérifications. Une directive `pragma` est toujours locale à un fichier source. vous devez ajouter la directive `pragma` à tous vos fichiers si vous voulez l'activer dans l'ensemble de votre projet. Si vous *import* un autre fichier, la directive `pragma` de ce fichier ne s'applique pas automatiquement au fichier d'importation.

Pragma de version

Les fichiers sources peuvent (et doivent) être annotés avec un `pragma` de version pour rejeter la compilation avec de futures versions du compilateur qui pourraient introduire des changements incompatibles. Nous essayons de limiter ces changements au strict minimum et de les introduire de manière à ce que les changements sémantiques nécessitent aussi dans la syntaxe, mais cela n'est pas toujours possible. Pour cette raison, il est toujours une bonne idée de lire le journal des modifications, au moins pour les versions qui contiennent des des changements de rupture. Ces versions ont toujours des versions de la forme `0.x.0` ou `x.0.0`.

Le `pragma` de version est utilisé comme suit : `pragma solidity ^0.5.2;`

Un fichier source avec la ligne ci-dessus ne compile pas avec un compilateur antérieur à la version 0.5.2, et il ne fonctionne pas non plus avec un compilateur à partir de la version 0.6.0 (cette deuxième condition est ajoutée en utilisant `^`). Parce que il n'y aura pas de changements de rupture jusqu'à la version `0.6.0`, vous pouvez être sûr que votre code compile comme vous l'aviez prévu. La version exacte du compilateur n'est pas fixée, de sorte que les versions de correction de bogues sont toujours possibles.

Il est possible de spécifier des règles plus complexes pour la version du compilateur, celles-ci suivent la même syntaxe que celle utilisée par `npm`.

Note : L'utilisation du `pragma version` *ne change pas* la version du compilateur. Il ne permet pas non plus d'activer ou de désactiver des fonctionnalités du compilateur. Il indique simplement au compilateur de vérifier si sa version correspond à celle requise par le `pragma`. Si elle ne correspond pas, le compilateur émet une erreur.

Pragma du codeur ABI

En utilisant `pragma abicoder v1` ou `pragma abicoder v2`, vous pouvez choisir entre les deux implémentations du codeur et du décodeur ABI.

Le nouveau codeur ABI (v2) est capable de coder et de décoder tableaux et structs. Il peut produire un code moins optimal et n'a pas été testé autant que l'ancien codeur, mais est considéré comme non expérimental à partir de Solidity 0.6.0. Vous devez toujours explicitement l'activer en utilisant `pragma abicoder v2;`. Puisqu'il sera activé par défaut à partir de Solidity 0.8.0, il existe une option pour sélectionner l'ancien codeur en utilisant `pragma abicoder v1;`.

L'ensemble des types supportés par le nouveau codeur est un sur-ensemble strict de ceux supportés par l'ancien. Les contrats qui l'utilisent peuvent interagir avec ceux qui ne l'utilisent pas sans limitations. L'inverse n'est possible que dans la mesure où le contrat `non-abicoder v2` n'essaie pas de faire des appels qui nécessiteraient de décoder des types uniquement supportés par le nouvel encodeur. Le compilateur peut détecter cela et émettra une erreur. Il suffit d'activer « `abicoder v2` » pour votre contrat pour que l'erreur disparaisse.

Note : Ce `pragma` s'applique à tout le code défini dans le fichier où il est activé, quel que soit l'endroit où ce code se retrouve finalement. Cela signifie qu'un contrat dont le fichier source est sélectionné pour être compilé avec le codeur

ABI v1 peut toujours contenir du code qui utilise le nouveau codeur en l'héritant d'un autre contrat. Ceci est autorisé si les nouveaux types sont uniquement utilisés en interne et non dans les signatures de fonctions externes.

Note : Jusqu'à Solidity 0.7.4, il était possible de sélectionner le codeur ABI v2 en utilisant `pragma experimental ABIEncoderV2`, mais il n'était pas possible de sélectionner explicitement le codeur v1 parce qu'il était par défaut.

Pragma expérimental

Le deuxième pragma est le pragma expérimental. Il peut être utilisé pour activer des fonctionnalités du compilateur ou du langage qui ne sont pas encore activées par défaut. Les pragmes expérimentaux suivants sont actuellement supportés :

ABIEncoderV2

Parce que le codeur ABI v2 n'est plus considéré comme expérimental, il peut être sélectionné via `pragma abicoder v2` (voir ci-dessus) depuis Solidity 0.7.4.

SMTChecker

Ce composant doit être activé lorsque le compilateur Solidity est construit, et n'est donc pas disponible dans tous les binaires Solidity. Les *instructions de construction* expliquent comment activer cette option. Elle est activée pour les versions PPA d'Ubuntu dans la plupart des versions, mais pas pour les images Docker, les binaires Windows ou les binaires Linux construits de manière statique. Elle peut être activée pour solc-js via l'option `smtCallback` si vous avez un solveur SMT installé localement et que vous exécutez solc-js via node (et non via le navigateur).

Si vous utilisez `pragma experimental SMTChecker;`, alors vous obtenez des *avertissements de sécurité* supplémentaires qui sont obtenus en interrogeant un solveur SMT. Ce composant ne prend pas encore en charge toutes les fonctionnalités du langage Solidity et produit probablement de nombreux avertissements. S'il signale des fonctionnalités non supportées, l'analyse n'est peut-être pas entièrement solide.

3.4.3 Importation d'autres fichiers sources

Syntaxe et sémantique

Solidity prend en charge des déclarations d'importation pour aider à modulariser votre code. Ils sont similaires à celles disponibles en JavaScript (à partir de ES6). Cependant, Solidity ne supporte pas le concept de l'*exportation par défaut*.

Au niveau global, vous pouvez utiliser des déclarations d'importation de la forme suivante :

```
import "filename";
```

La partie `filename` est appelée un « chemin d'importation ». Cette déclaration importe tous les symboles globaux de « nom de fichier » (et les symboles qui y sont importés) dans la portée globale actuelle (différent de ES6 mais compatible avec Solidity). L'utilisation de cette forme n'est pas recommandée, car elle pollue l'espace de noms de manière imprévisible. Si vous ajoutez de nouveaux éléments de haut niveau à l'intérieur de « filename », ils apparaissent automatiquement dans tous les fichiers qui importent de la sorte depuis « nom de fichier ». Il est préférable d'importer des symboles spécifiques de manière explicite.

L'exemple suivant crée un nouveau symbole global `symbolName` dont les membres sont tous les symboles globaux de « filename ». les symboles globaux de « nom_de_fichier » :

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

ce qui a pour conséquence que tous les symboles globaux sont disponibles dans le format `symbolName.symbol`.

Une variante de cette syntaxe qui ne fait pas partie de ES6, mais qui peut être utile, est la suivante :

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

qui est équivalent à `import * as symbolName from "filename";`.

S'il y a une collision de noms, vous pouvez renommer les symboles pendant l'importation. Par exemple, le code ci-dessous crée de nouveaux symboles globaux `alias` et `symbol2` qui font référence à `symbol1` et `symbole2` à l'intérieur de « filename », respectivement.

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

Importation de chemins

Afin de pouvoir supporter des constructions reproductibles sur toutes les plateformes, le compilateur Solidity doit faire abstraction des détails du système de fichiers dans lequel les fichiers sources sont stockés. Pour cette raison, les chemins d'importation ne se réfèrent pas directement aux fichiers dans le système de fichiers hôte. Au lieu de cela, le compilateur maintient une base de données interne (*système de fichiers virtuel* ou *VFS* en abrégé) dans laquelle chaque unité source se voit attribuer un *nom d'unité source* unique qui est un identifiant opaque et non structuré. Le chemin d'importation spécifié dans une instruction d'importation est traduit en un nom d'unité source et utilisé pour trouver l'unité source correspondante dans cette base de données.

En utilisant l'API *Standard JSON*, il est possible de fournir directement les noms et le contenu de tous les fichiers sources comme une partie de l'entrée du compilateur. Dans ce cas, les noms des unités sources sont vraiment arbitraires. Si, par contre, vous voulez que le compilateur trouve et charge automatiquement le code source dans le VFS, vos noms d'unité source doivent être structurés de manière à rendre possible un *import callback* de les localiser. Lorsque vous utilisez le compilateur en ligne de commande, le callback d'importation par défaut ne supporte que le chargement du code source depuis le système de fichiers de l'hôte, ce qui signifie que les noms de vos unités sources doivent être des chemins. Certains environnements fournissent des callbacks personnalisés qui sont plus polyvalents. Par exemple l'IDE *Remix* en fournit une qui vous permet d'importer des fichiers à partir d'URL HTTP, IPFS et Swarm ou de vous référer directement à des paquets dans le registre NPM.

Pour une description complète du système de fichiers virtuel et de la logique de résolution de chemin utilisée par le compilateur, voir *Résolution de chemin*.

3.4.4 Commentaires

Les commentaires d'une seule ligne (`//`) et les commentaires de plusieurs lignes (`/* . . . */`) sont possibles.

```
// Il s'agit d'un commentaire d'une seule ligne.

/*
Ceci est un
commentaire de plusieurs lignes.
*/
```

Note : Un commentaire d'une seule ligne est terminé par n'importe quel terminateur de ligne unicode (LF, VF, FF, CR, NEL, LS ou PS) en codage UTF-8. Le terminateur fait toujours partie du code source après le commentaire, donc

s'il ne s'agit pas d'un symbole ASCII (il s'agit de NEL, LS et PS), cela entraînera une erreur d'analyse syntaxique.

En outre, il existe un autre type de commentaire appelé commentaire NatSpec, qui est détaillé dans le guide de style. Ils sont écrits avec une triple barre oblique (///) ou un double astérisque (/** ... */). Ils doivent être utilisés directement au-dessus des déclarations de fonctions ou des instructions.

3.5 Structure d'un contrat

Les contrats dans Solidity sont similaires aux classes dans les langages orientés objet. Chaque contrat peut contenir des déclarations de *Variables d'état*, *Fonctions*, *Modificateurs de fonction*, *Événements*, *Erreurs*, structure-structure-types et *Types d'Enum*. De plus, les contrats peuvent hériter d'autres contrats.

Il existe également des types spéciaux de contrats appelés *libraries* et *interfaces*.

La section sur les contrats contient plus de détails que cette section, qui sert à donner un aperçu rapide.

3.5.1 Variables d'état

Les variables d'état sont des variables dont les valeurs sont stockées de manière permanente dans le contrat stockage.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract SimpleStorage {
    uint storedData; // Variable d'état
    // ...
}
```

Voir la section *Types* pour les types de variables d'état valides et la section *Visibilité et Getters* pour les choix possibles en matière de visibilité.

3.5.2 Fonctions

Les fonctions sont les unités exécutables du code. Les fonctions sont généralement définies à l'intérieur d'un contrat, mais elles peuvent aussi être définies en dehors des contrats.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

contract SimpleAuction {
    function bid() public payable { // Fonction
        // ...
    }
}

// Fonction d'aide définie en dehors d'un contrat
function helper(uint x) pure returns (uint) {
    return x * 2;
}
```

Appels de fonction peut se produire en interne ou en externe et avoir différents niveaux de *visibilité* vers d'autres contrats. *Les fonctions* acceptent les *paramètres et variables de retour* pour passer des paramètres et des valeurs entre elles.

3.5.3 Modificateurs de fonction

Les modificateurs de fonctions peuvent être utilisés pour modifier la sémantique des fonctions de manière déclarative (voir *Modificateurs de fonction* dans la section sur les contrats).

La surcharge, c'est-à-dire le fait d'avoir le même nom de modificateur avec différents paramètres, n'est pas possible.

Comme les fonctions, les modificateurs peuvent être *overridden*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract Purchase {
    address public seller;

    modifier onlySeller() { // Modificateur
        require(
            msg.sender == seller,
            "Seul le vendeur peut l'appeler."
        );
        _;
    }

    function abort() public view onlySeller { // Utilisation des modificateurs
        // ...
    }
}
```

3.5.4 Événements

Les événements sont des interfaces pratiques avec les fonctions de journalisation de l'EVM.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.21 <0.9.0;

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

    function bid() public payable {
        // ...
        emit HighestBidIncreased(msg.sender, msg.value); // Événement déclencheur
    }
}
```

Voir *Événements* dans la section contrats pour des informations sur la façon dont les événements sont déclarés et peuvent être utilisés à l'intérieur d'une application.

3.5.5 Erreurs

Les erreurs vous permettent de définir des noms et des données descriptives pour les situations d'échec. Les erreurs peuvent être utilisées dans *revert statements*. Par rapport aux descriptions de chaînes de caractères, les erreurs sont beaucoup moins coûteuses et vous permettent d'encoder des données supplémentaires. Vous pouvez utiliser NatSpec pour décrire l'erreur à l'utilisateur.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

/// Pas assez de fonds pour le transfert. Demandé `requested`,
/// mais seulement `available` disponible.
error NotEnoughFunds(uint requested, uint available);

contract Token {
    mapping(address => uint) balances;
    function transfer(address to, uint amount) public {
        uint balance = balances[msg.sender];
        if (balance < amount)
            revert NotEnoughFunds(amount, balance);
        balances[msg.sender] -= amount;
        balances[to] += amount;
        // ...
    }
}
```

Voir *Les erreurs et la déclaration de retour en arrière* dans la section sur les contrats pour plus d'informations.

3.5.6 Types de structures

Les structures sont des types personnalisés qui peuvent regrouper plusieurs variables (voir *Structs* dans la section sur les types).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

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

3.5.7 Types d'Enum

Les Enums peuvent être utilisées pour créer des types personnalisés avec un ensemble fini de « valeurs constantes » (voir *Enums* dans la section sur les types).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

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

3.6 Types

Solidity est un langage statiquement typé, ce qui signifie que le type de chaque variable (état et locale) doit être spécifié. Solidity fournit plusieurs types élémentaires qui peuvent être combinés pour former des types complexes.

De plus, les types peuvent interagir entre eux dans des expressions contenant des opérateurs. Pour une référence rapide des différents opérateurs, voir *Ordre de Préséance des Opérateurs*.

Le concept de valeurs « indéfinies » ou « nulles » n'existe pas dans Solidity, mais les variables nouvellement déclarées ont toujours une *valeur par défaut* dépendant de son type. Pour gérer toute valeur inattendue, vous devez utiliser la fonction *revert* pour annuler toute la transaction, ou retourner un tuple avec une seconde valeur `bool` indiquant le succès.

3.6.1 Value Types

The following types are also called value types because variables of these types will always be passed by value, i.e. they are always copied when they are used as function arguments or in assignments.

Booleans

`bool` : The possible values are constants `true` and `false`.

Operators :

- `!` (logical negation)
- `&&` (logical conjunction, « and »)
- `||` (logical disjunction, « or »)
- `==` (equality)
- `!=` (inequality)

The operators `||` and `&&` apply the common short-circuiting rules. This means that in the expression `f(x) || g(y)`, if `f(x)` evaluates to `true`, `g(y)` will not be evaluated even if it may have side-effects.

Integers

`int` / `uint` : Signed and unsigned integers of various sizes. Keywords `uint8` to `uint256` in steps of 8 (unsigned of 8 up to 256 bits) and `int8` to `int256`. `uint` and `int` are aliases for `uint256` and `int256`, respectively.

Operators :

- Comparisons : `<=`, `<`, `==`, `!=`, `>=`, `>` (evaluate to `bool`)
- Bit operators : `&`, `|`, `^` (bitwise exclusive or), `~` (bitwise negation)
- Shift operators : `<<` (left shift), `>>` (right shift)
- Arithmetic operators : `+`, `-`, unary `-` (only for signed integers), `*`, `/`, `%` (modulo), `**` (exponentiation)

For an integer type `X`, you can use `type(X).min` and `type(X).max` to access the minimum and maximum value representable by the type.

Avertissement : Integers in Solidity are restricted to a certain range. For example, with `uint32`, this is `0` up to $2^{32} - 1$. There are two modes in which arithmetic is performed on these types : The « wrapping » or « unchecked » mode and the « checked » mode. By default, arithmetic is always « checked », which mean that if the result of an operation falls outside the value range of the type, the call is reverted through a *failing assertion*. You can switch to « unchecked » mode using `unchecked { ... }`. More details can be found in the section about *unchecked*.

Comparisons

The value of a comparison is the one obtained by comparing the integer value.

Bit operations

Bit operations are performed on the two's complement representation of the number. This means that, for example `~int256(0) == int256(-1)`.

Shifts

The result of a shift operation has the type of the left operand, truncating the result to match the type. The right operand must be of unsigned type, trying to shift by a signed type will produce a compilation error.

Shifts can be « simulated » using multiplication by powers of two in the following way. Note that the truncation to the type of the left operand is always performed at the end, but not mentioned explicitly.

- `x << y` is equivalent to the mathematical expression $x * 2^{y}$.
- `x >> y` is equivalent to the mathematical expression $x / 2^{y}$, rounded towards negative infinity.

Avertissement : Before version `0.5.0` a right shift `x >> y` for negative `x` was equivalent to the mathematical expression $x / 2^{y}$ rounded towards zero, i.e., right shifts used rounding up (towards zero) instead of rounding down (towards negative infinity).

Note : Overflow checks are never performed for shift operations as they are done for arithmetic operations. Instead, the result is always truncated.

Addition, Subtraction and Multiplication

Addition, subtraction and multiplication have the usual semantics, with two different modes in regard to over- and underflow :

By default, all arithmetic is checked for under- or overflow, but this can be disabled using the *unchecked block*, resulting in wrapping arithmetic. More details can be found in that section.

The expression `-x` is equivalent to $(T(0) - x)$ where T is the type of x . It can only be applied to signed types. The value of `-x` can be positive if x is negative. There is another caveat also resulting from two's complement representation :

If you have `int x = type(int).min;`, then `-x` does not fit the positive range. This means that `unchecked { assert(-x == x); }` works, and the expression `-x` when used in checked mode will result in a failing assertion.

Division

Since the type of the result of an operation is always the type of one of the operands, division on integers always results in an integer. In Solidity, division rounds towards zero. This means that `int256(-5) / int256(2) == int256(-2)`.

Note that in contrast, division on *literals* results in fractional values of arbitrary precision.

Note : Division by zero causes a *Panic error*. This check can **not** be disabled through `unchecked { ... }`.

Note : The expression `type(int).min / (-1)` is the only case where division causes an overflow. In checked arithmetic mode, this will cause a failing assertion, while in wrapping mode, the value will be `type(int).min`.

Modulo

The modulo operation `a % n` yields the remainder r after the division of the operand a by the operand n , where $q = \text{int}(a / n)$ and $r = a - (n * q)$. This means that modulo results in the same sign as its left operand (or zero) and `a % n == -(-a % n)` holds for negative a :

- `int256(5) % int256(2) == int256(1)`
- `int256(5) % int256(-2) == int256(1)`
- `int256(-5) % int256(2) == int256(-1)`
- `int256(-5) % int256(-2) == int256(-1)`

Note : Modulo with zero causes a *Panic error*. This check can **not** be disabled through `unchecked { ... }`.

Exponentiation

Exponentiation is only available for unsigned types in the exponent. The resulting type of an exponentiation is always equal to the type of the base. Please take care that it is large enough to hold the result and prepare for potential assertion failures or wrapping behaviour.

Note : In checked mode, exponentiation only uses the comparatively cheap `exp` opcode for small bases. For the cases of `x**3`, the expression `x*x*x` might be cheaper. In any case, gas cost tests and the use of the optimizer are advisable.

Note : Note that `0**0` is defined by the EVM as 1.

Fixed Point Numbers

Avertissement : Fixed point numbers are not fully supported by Solidity yet. They can be declared, but cannot be assigned to or from.

`fixed` / `ufixed` : Signed and unsigned fixed point number of various sizes. Keywords `ufixedMxN` and `fixedMxN`, where `M` represents the number of bits taken by the type and `N` represents how many decimal points are available. `M` must be divisible by 8 and goes from 8 to 256 bits. `N` must be between 0 and 80, inclusive. `ufixed` and `fixed` are aliases for `ufixed128x18` and `fixed128x18`, respectively.

Operators :

- Comparisons : `<=`, `<`, `==`, `!=`, `>=`, `>` (evaluate to `bool`)
- Arithmetic operators : `+`, `-`, unary `-`, `*`, `/`, `%` (modulo)

Note : The main difference between floating point (`float` and `double` in many languages, more precisely IEEE 754 numbers) and fixed point numbers is that the number of bits used for the integer and the fractional part (the part after the decimal dot) is flexible in the former, while it is strictly defined in the latter. Generally, in floating point almost the entire space is used to represent the number, while only a small number of bits define where the decimal point is.

Address

The address type comes in two flavours, which are largely identical :

- `address` : Holds a 20 byte value (size of an Ethereum address).
- `address payable` : Same as `address`, but with the additional members `transfer` and `send`.

The idea behind this distinction is that `address payable` is an address you can send Ether to, while a plain `address` cannot be sent Ether.

Type conversions :

Implicit conversions from `address payable` to `address` are allowed, whereas conversions from `address` to `address payable` must be explicit via `payable(<address>)`.

Explicit conversions to and from `address` are allowed for `uint160`, integer literals, `bytes20` and contract types.

Only expressions of type `address` and contract-type can be converted to the type `address payable` via the explicit conversion `payable(. . .)`. For contract-type, this conversion is only allowed if the contract can receive Ether, i.e., the contract either has a *receive* or a payable fallback function. Note that `payable(0)` is valid and is an exception to this rule.

Note : If you need a variable of type `address` and plan to send Ether to it, then declare its type as `address payable` to make this requirement visible. Also, try to make this distinction or conversion as early as possible.

Operators :

- `<=`, `<`, `==`, `!=`, `>=` and `>`

Avertissement : If you convert a type that uses a larger byte size to an address, for example `bytes32`, then the address is truncated. To reduce conversion ambiguity version 0.4.24 and higher of the compiler force you make the truncation explicit in the conversion. Take for example the 32-byte value `0x111122223333444455556666777788889999AAAABBBBCCCCDDDEEEFFFFFFCC`.

You can use `address(uint160(bytes20(b)))`, which results in `0x111122223333444455556666777788889999Aaa`, or you can use `address(uint160(uint256(b)))`, which results in `0x777788889999AaAbBbbCccddDdeeeEfffCcCc`.

Note : The distinction between `address` and `address payable` was introduced with version 0.5.0. Also starting from that version, contracts do not derive from the `address` type, but can still be explicitly converted to `address` or to `address payable`, if they have a `receive` or `payable` fallback function.

Members of Addresses

For a quick reference of all members of address, see *Membres des types d'adresses*.

— `balance` and `transfer`

It is possible to query the balance of an address using the property `balance` and to send Ether (in units of wei) to a payable address using the `transfer` function :

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

The `transfer` function fails if the balance of the current contract is not large enough or if the Ether transfer is rejected by the receiving account. The `transfer` function reverts on failure.

Note : If `x` is a contract address, its code (more specifically : its *Fonction de réception d'Ether*, if present, or otherwise its *Fonction de repli*, if present) will be executed together with the `transfer` call (this is a feature of the EVM and cannot be prevented). If that execution runs out of gas or fails in any way, the Ether transfer will be reverted and the current contract will stop with an exception.

— `send`

`Send` is the low-level counterpart of `transfer`. If the execution fails, the current contract will not stop with an exception, but `send` will return `false`.

Avertissement : There are some dangers in using `send` : The transfer fails if the call stack depth is at 1024 (this can always be forced by the caller) and it also fails if the recipient runs out of gas. So in order to make safe Ether transfers, always check the return value of `send`, use `transfer` or even better : use a pattern where the recipient withdraws the money.

— `call`, `delegatecall` and `staticcall`

In order to interface with contracts that do not adhere to the ABI, or to get more direct control over the encoding, the functions `call`, `delegatecall` and `staticcall` are provided. They all take a single `bytes memory` parameter and return the success condition (as a `bool`) and the returned data (`bytes memory`). The functions `abi.encode`, `abi.encodePacked`, `abi.encodeWithSelector` and `abi.encodeWithSignature` can be used to encode structured data.

Example :

```
bytes memory payload = abi.encodeWithSignature("register(string)", "MyName");
(bool success, bytes memory returnData) = address(nameReg).call(payload);
require(success);
```

Avertissement : All these functions are low-level functions and should be used with care. Specifically, any unknown contract might be malicious and if you call it, you hand over control to that contract which could in turn call back into your contract, so be prepared for changes to your state variables when the call returns. The regular way to interact with other contracts is to call a function on a contract object (`x.f()`).

Note : Previous versions of Solidity allowed these functions to receive arbitrary arguments and would also handle a first argument of type `bytes4` differently. These edge cases were removed in version 0.5.0.

It is possible to adjust the supplied gas with the gas modifier :

```
address(nameReg).call{gas: 1000000}(abi.encodeWithSignature("register(string)", "MyName
↪"));
```

Similarly, the supplied Ether value can be controlled too :

```
address(nameReg).call{value: 1 ether}(abi.encodeWithSignature("register(string)", "MyName
↪"));
```

Lastly, these modifiers can be combined. Their order does not matter :

```
address(nameReg).call{gas: 1000000, value: 1 ether}(abi.encodeWithSignature(
↪"register(string)", "MyName"));
```

In a similar way, the function `delegatecall` can be used : the difference is that only the code of the given address is used, all other aspects (storage, balance, ...) are taken from the current contract. The purpose of `delegatecall` is to use library code which is stored in another contract. The user has to ensure that the layout of storage in both contracts is suitable for `delegatecall` to be used.

Note : Prior to homestead, only a limited variant called `callcode` was available that did not provide access to the original `msg.sender` and `msg.value` values. This function was removed in version 0.5.0.

Since byzantium `staticcall` can be used as well. This is basically the same as `call`, but will revert if the called function modifies the state in any way.

All three functions `call`, `delegatecall` and `staticcall` are very low-level functions and should only be used as a *last resort* as they break the type-safety of Solidity.

The gas option is available on all three methods, while the value option is only available on `call`.

Note : It is best to avoid relying on hardcoded gas values in your smart contract code, regardless of whether state is read from or written to, as this can have many pitfalls. Also, access to gas might change in the future.

Note : All contracts can be converted to `address` type, so it is possible to query the balance of the current contract using `address(this).balance`.

Contract Types

Every *contract* defines its own type. You can implicitly convert contracts to contracts they inherit from. Contracts can be explicitly converted to and from the `address` type.

Explicit conversion to and from the `address payable` type is only possible if the contract type has a `receive` or `payable` fallback function. The conversion is still performed using `address(x)`. If the contract type does not have a `receive` or `payable` fallback function, the conversion to `address payable` can be done using `payable(address(x))`. You can find more information in the section about the *address type*.

Note : Before version 0.5.0, contracts directly derived from the `address` type and there was no distinction between `address` and `address payable`.

If you declare a local variable of contract type (`MyContract c`), you can call functions on that contract. Take care to assign it from somewhere that is the same contract type.

You can also instantiate contracts (which means they are newly created). You can find more details in the “*Contracts via new*” section.

The data representation of a contract is identical to that of the `address` type and this type is also used in the *ABI*.

Contracts do not support any operators.

The members of contract types are the external functions of the contract including any state variables marked as `public`.

For a contract `C` you can use `type(C)` to access *type information* about the contract.

Fixed-size byte arrays

The value types `bytes1`, `bytes2`, `bytes3`, ..., `bytes32` hold a sequence of bytes from one to up to 32.

Operators :

- Comparisons : `<=`, `<`, `==`, `!=`, `>=`, `>` (evaluate to `bool`)
- Bit operators : `&`, `|`, `^` (bitwise exclusive or), `~` (bitwise negation)
- Shift operators : `<<` (left shift), `>>` (right shift)
- Index access : If `x` is of type `bytesI`, then `x[k]` for $0 \leq k < I$ returns the `k` th byte (read-only).

The shifting operator works with unsigned integer type as right operand (but returns the type of the left operand), which denotes the number of bits to shift by. Shifting by a signed type will produce a compilation error.

Members :

- `.length` yields the fixed length of the byte array (read-only).

Note : The type `bytes1[]` is an array of bytes, but due to padding rules, it wastes 31 bytes of space for each element (except in storage). It is better to use the `bytes` type instead.

Note : Prior to version 0.8.0, `byte` used to be an alias for `bytes1`.

Dynamically-sized byte array

bytes : Dynamically-sized byte array, see *Arrays*. Not a value-type !

string : Dynamically-sized UTF-8-encoded string, see *Arrays*. Not a value-type !

Address Literals

Hexadecimal literals that pass the address checksum test, for example `0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF` are of address type. Hexadecimal literals that are between 39 and 41 digits long and do not pass the checksum test produce an error. You can prepend (for integer types) or append (for bytesNN types) zeros to remove the error.

Note : The mixed-case address checksum format is defined in [EIP-55](#).

Rational and Integer Literals

Integer literals are formed from a sequence of digits in the range 0-9. They are interpreted as decimals. For example, 69 means sixty nine. Octal literals do not exist in Solidity and leading zeros are invalid.

Decimal fractional literals are formed by a `.` with at least one number on one side. Examples include `1.`, `.1` and `1.3`.

Scientific notation in the form of `2e10` is also supported, where the mantissa can be fractional but the exponent has to be an integer. The literal `MeE` is equivalent to `M * 10**E`. Examples include `2e10`, `-2e10`, `2e-10`, `2.5e1`.

Underscores can be used to separate the digits of a numeric literal to aid readability. For example, decimal `123_000`, hexadecimal `0x2eff_abde`, scientific decimal notation `1_2e345_678` are all valid. Underscores are only allowed between two digits and only one consecutive underscore is allowed. There is no additional semantic meaning added to a number literal containing underscores, the underscores are ignored.

Number literal expressions retain arbitrary precision until they are converted to a non-literal type (i.e. by using them together with a non-literal expression or by explicit conversion). This means that computations do not overflow and divisions do not truncate in number literal expressions.

For example, $(2^{**800} + 1) - 2^{**800}$ results in the constant 1 (of type `uint8`) although intermediate results would not even fit the machine word size. Furthermore, $.5 * 8$ results in the integer 4 (although non-integers were used in between).

Any operator that can be applied to integers can also be applied to number literal expressions as long as the operands are integers. If any of the two is fractional, bit operations are disallowed and exponentiation is disallowed if the exponent is fractional (because that might result in a non-rational number).

Shifts and exponentiation with literal numbers as left (or base) operand and integer types as the right (exponent) operand are always performed in the `uint256` (for non-negative literals) or `int256` (for a negative literals) type, regardless of the type of the right (exponent) operand.

Avertissement : Division on integer literals used to truncate in Solidity prior to version 0.4.0, but it now converts into a rational number, i.e. $5 / 2$ is not equal to 2, but to 2.5 .

Note : Solidity has a number literal type for each rational number. Integer literals and rational number literals belong to number literal types. Moreover, all number literal expressions (i.e. the expressions that contain only number literals and operators) belong to number literal types. So the number literal expressions $1 + 2$ and $2 + 1$ both belong to the same number literal type for the rational number three.

Note : Number literal expressions are converted into a non-literal type as soon as they are used with non-literal expressions. Disregarding types, the value of the expression assigned to `b` below evaluates to an integer. Because `a` is of type `uint128`, the expression `2.5 + a` has to have a proper type, though. Since there is no common type for the type of `2.5` and `uint128`, the Solidity compiler does not accept this code.

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

String Literals and Types

String literals are written with either double or single-quotes ("foo" or 'bar'), and they can also be split into multiple consecutive parts ("foo" "bar" is equivalent to "foobar") which can be helpful when dealing with long strings. They do not imply trailing zeroes as in C; "foo" represents three bytes, not four. As with integer literals, their type can vary, but they are implicitly convertible to `bytes1`, ..., `bytes32`, if they fit, to `bytes` and to `string`.

For example, with `bytes32 samevar = "stringliteral"` the string literal is interpreted in its raw byte form when assigned to a `bytes32` type.

String literals can only contain printable ASCII characters, which means the characters between and including 0x20 .. 0x7E.

Additionally, string literals also support the following escape characters :

- `<newline>` (escapes an actual newline)
- `\\` (backslash)
- `\'` (single quote)
- `\"` (double quote)
- `\n` (newline)
- `\r` (carriage return)
- `\t` (tab)
- `\xNN` (hex escape, see below)
- `\uNNNN` (unicode escape, see below)

`\xNN` takes a hex value and inserts the appropriate byte, while `\uNNNN` takes a Unicode codepoint and inserts an UTF-8 sequence.

Note : Until version 0.8.0 there were three additional escape sequences : `\b`, `\f` and `\v`. They are commonly available in other languages but rarely needed in practice. If you do need them, they can still be inserted via hexadecimal escapes, i.e. `\x08`, `\x0c` and `\x0b`, respectively, just as any other ASCII character.

The string in the following example has a length of ten bytes. It starts with a newline byte, followed by a double quote, a single quote, a backslash character and then (without separator) the character sequence `abcdef`.

```
"\n\"'\N\Nabc\N
def"
```

Any Unicode line terminator which is not a newline (i.e. LF, VF, FF, CR, NEL, LS, PS) is considered to terminate the string literal. Newline only terminates the string literal if it is not preceded by a `\`.

Unicode Literals

While regular string literals can only contain ASCII, Unicode literals – prefixed with the keyword `unicode` – can contain any valid UTF-8 sequence. They also support the very same escape sequences as regular string literals.

```
string memory a = unicode"Hello ";
```

Hexadecimal Literals

Hexadecimal literals are prefixed with the keyword `hex` and are enclosed in double or single-quotes (`hex"001122FF"`, `hex'0011_22_FF'`). Their content must be hexadecimal digits which can optionally use a single underscore as separator between byte boundaries. The value of the literal will be the binary representation of the hexadecimal sequence.

Multiple hexadecimal literals separated by whitespace are concatenated into a single literal : `hex"00112233" hex"44556677"` is equivalent to `hex"0011223344556677"`

Hexadecimal literals behave like *string literals* and have the same convertibility restrictions.

Enums

Enums are one way to create a user-defined type in Solidity. They are explicitly convertible to and from all integer types but implicit conversion is not allowed. The explicit conversion from integer checks at runtime that the value lies inside the range of the enum and causes a *Panic error* otherwise. Enums require at least one member, and its default value when declared is the first member. Enums cannot have more than 256 members.

The data representation is the same as for enums in C : The options are represented by subsequent unsigned integer values starting from 0.

Using `type(NameOfEnum).min` and `type(NameOfEnum).max` you can get the smallest and respectively largest value of the given enum.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

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

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

    // Since enum types are not part of the ABI, the signature of "getChoice"
    // will automatically be changed to "getChoice() returns (uint8)"
    // for all matters external to Solidity.
    function getChoice() public view returns (ActionChoices) {
        return choice;
    }

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

(suite sur la page suivante)

(suite de la page précédente)

```

function getLargestValue() public pure returns (ActionChoices) {
    return type(ActionChoices).max;
}

function getSmallestValue() public pure returns (ActionChoices) {
    return type(ActionChoices).min;
}
}

```

Note : Enums can also be declared on the file level, outside of contract or library definitions.

User Defined Value Types

A user defined value type allows creating a zero cost abstraction over an elementary value type. This is similar to an alias, but with stricter type requirements.

A user defined value type is defined using `type C is V`, where C is the name of the newly introduced type and V has to be a built-in value type (the « underlying type »). The function `C.wrap` is used to convert from the underlying type to the custom type. Similarly, the function `C.unwrap` is used to convert from the custom type to the underlying type.

The type C does not have any operators or bound member functions. In particular, even the operator `==` is not defined. Explicit and implicit conversions to and from other types are disallowed.

The data-representation of values of such types are inherited from the underlying type and the underlying type is also used in the ABI.

The following example illustrates a custom type `UFixed256x18` representing a decimal fixed point type with 18 decimals and a minimal library to do arithmetic operations on the type.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

// Represent a 18 decimal, 256 bit wide fixed point type using a user defined value type.
type UFixed256x18 is uint256;

/// A minimal library to do fixed point operations on UFixed256x18.
library FixedMath {
    uint constant multiplier = 10**18;

    /// Adds two UFixed256x18 numbers. Reverts on overflow, relying on checked
    /// arithmetic on uint256.
    function add(UFixed256x18 a, UFixed256x18 b) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(UFixed256x18.unwrap(a) + UFixed256x18.unwrap(b));
    }

    /// Multiplies UFixed256x18 and uint256. Reverts on overflow, relying on checked
    /// arithmetic on uint256.
    function mul(UFixed256x18 a, uint256 b) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(UFixed256x18.unwrap(a) * b);
    }

    /// Take the floor of a UFixed256x18 number.
    /// @return the largest integer that does not exceed `a`.
}

```

(suite sur la page suivante)

```

function floor(UFixed256x18 a) internal pure returns (uint256) {
    return UFixed256x18.unwrap(a) / multiplier;
}
/// Turns a uint256 into a UFixed256x18 of the same value.
/// Reverts if the integer is too large.
function toUFixed256x18(uint256 a) internal pure returns (UFixed256x18) {
    return UFixed256x18.wrap(a * multiplier);
}
}

```

Notice how `UFixed256x18.wrap` and `FixedMath.toUFixed256x18` have the same signature but perform two very different operations : The `UFixed256x18.wrap` function returns a `UFixed256x18` that has the same data representation as the input, whereas `toUFixed256x18` returns a `UFixed256x18` that has the same numerical value.

Function Types

Function types are the types of functions. Variables of function type can be assigned from functions and function parameters of function type can be used to pass functions to and return functions from function calls. Function types come in two flavours - *internal* and *external* functions :

Internal functions can only be called inside the current contract (more specifically, inside the current code unit, which also includes internal library functions and inherited functions) because they cannot be executed outside of the context of the current contract. Calling an internal function is realized by jumping to its entry label, just like when calling a function of the current contract internally.

External functions consist of an address and a function signature and they can be passed via and returned from external function calls.

Function types are notated as follows :

```

function (<parameter types>) {internal|external} [pure|view|payable] [returns (<return_
↪types>)]

```

In contrast to the parameter types, the return types cannot be empty - if the function type should not return anything, the whole `returns (<return types>)` part has to be omitted.

By default, function types are internal, so the `internal` keyword can be omitted. Note that this only applies to function types. Visibility has to be specified explicitly for functions defined in contracts, they do not have a default.

Conversions :

A function type A is implicitly convertible to a function type B if and only if their parameter types are identical, their return types are identical, their internal/external property is identical and the state mutability of A is more restrictive than the state mutability of B. In particular :

- pure functions can be converted to view and non-payable functions
- view functions can be converted to non-payable functions
- payable functions can be converted to non-payable functions

No other conversions between function types are possible.

The rule about payable and non-payable might be a little confusing, but in essence, if a function is payable, this means that it also accepts a payment of zero Ether, so it also is non-payable. On the other hand, a non-payable function will reject Ether sent to it, so non-payable functions cannot be converted to payable functions.

If a function type variable is not initialised, calling it results in a *Panic error*. The same happens if you call a function after using `delete` on it.

If external function types are used outside of the context of Solidity, they are treated as the `function` type, which encodes the address followed by the function identifier together in a single `bytes24` type.

Note that public functions of the current contract can be used both as an internal and as an external function. To use `f` as an internal function, just use `f`, if you want to use its external form, use `this.f`.

A function of an internal type can be assigned to a variable of an internal function type regardless of where it is defined. This includes private, internal and public functions of both contracts and libraries as well as free functions. External function types, on the other hand, are only compatible with public and external contract functions. Libraries are excluded because they require a `delegatecall` and use *a different ABI convention for their selectors*. Functions declared in interfaces do not have definitions so pointing at them does not make sense either.

Members :

External (or public) functions have the following members :

- `.address` returns the address of the contract of the function.
- `.selector` returns the *ABI function selector*

Note : External (or public) functions used to have the additional members `.gas(uint)` and `.value(uint)`. These were deprecated in Solidity 0.6.2 and removed in Solidity 0.7.0. Instead use `{gas: ...}` and `{value: ...}` to specify the amount of gas or the amount of wei sent to a function, respectively. See *External Function Calls* for more information.

Example that shows how to use the members :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.4 <0.9.0;

contract Example {
    function f() public payable returns (bytes4) {
        assert(this.f.address == address(this));
        return this.f.selector;
    }

    function g() public {
        this.f{gas: 10, value: 800}();
    }
}
```

Example that shows how to use internal function types :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

library ArrayUtils {
    // internal functions can be used in internal library functions because
    // they will be part of the same code context
    function map(uint[] memory self, function (uint) pure returns (uint) f)
        internal
        pure
        returns (uint[] memory r)
    {
        r = new uint[](self.length);
        for (uint i = 0; i < self.length; i++) {
            r[i] = f(self[i]);
        }
    }
}
```

(suite sur la page suivante)

```

    }
  }

  function reduce(
    uint[] memory self,
    function (uint, uint) pure returns (uint) f
  )
  internal
  pure
  returns (uint r)
  {
    r = self[0];
    for (uint i = 1; i < self.length; i++) {
      r = f(r, self[i]);
    }
  }

  function range(uint length) internal pure 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) public pure returns (uint) {
    return ArrayUtils.range(l).map(square).reduce(sum);
  }

  function square(uint x) internal pure returns (uint) {
    return x * x;
  }

  function sum(uint x, uint y) internal pure returns (uint) {
    return x + y;
  }
}

```

Another example that uses external function types :

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract Oracle {
  struct Request {
    bytes data;
    function(uint) external callback;
  }
}

```

(suite sur la page suivante)

(suite de la page précédente)

```

    }

    Request[] private requests;
    event NewRequest(uint);

    function query(bytes memory data, function(uint) external callback) public {
        requests.push(Request(data, callback));
        emit NewRequest(requests.length - 1);
    }

    function reply(uint requestID, uint response) public {
        // Here goes the check that the reply comes from a trusted source
        requests[requestID].callback(response);
    }
}

contract OracleUser {
    Oracle constant private ORACLE_CONST =
↳Oracle(address(0x00000000219ab540356cBB839Cbe05303d7705Fa)); // known contract
    uint private exchangeRate;

    function buySomething() public {
        ORACLE_CONST.query("USD", this.oracleResponse);
    }

    function oracleResponse(uint response) public {
        require(
            msg.sender == address(ORACLE_CONST),
            "Only oracle can call this."
        );
        exchangeRate = response;
    }
}

```

Note : Lambda or inline functions are planned but not yet supported.

3.6.2 Reference Types

Values of reference type can be modified through multiple different names. Contrast this with value types where you get an independent copy whenever a variable of value type is used. Because of that, reference types have to be handled more carefully than value types. Currently, reference types comprise structs, arrays and mappings. If you use a reference type, you always have to explicitly provide the data area where the type is stored : `memory` (whose lifetime is limited to an external function call), `storage` (the location where the state variables are stored, where the lifetime is limited to the lifetime of a contract) or `calldata` (special data location that contains the function arguments).

An assignment or type conversion that changes the data location will always incur an automatic copy operation, while assignments inside the same data location only copy in some cases for storage types.

Data location

Every reference type has an additional annotation, the « data location », about where it is stored. There are three data locations : `memory`, `storage` and `calldata`. `Calldata` is a non-modifiable, non-persistent area where function arguments are stored, and behaves mostly like `memory`.

Note : If you can, try to use `calldata` as data location because it will avoid copies and also makes sure that the data cannot be modified. Arrays and structs with `calldata` data location can also be returned from functions, but it is not possible to allocate such types.

Note : Prior to version 0.6.9 data location for reference-type arguments was limited to `calldata` in external functions, `memory` in public functions and either `memory` or `storage` in internal and private ones. Now `memory` and `calldata` are allowed in all functions regardless of their visibility.

Note : Prior to version 0.5.0 the data location could be omitted, and would default to different locations depending on the kind of variable, function type, etc., but all complex types must now give an explicit data location.

Data location and assignment behaviour

Data locations are not only relevant for persistency of data, but also for the semantics of assignments :

- Assignments between `storage` and `memory` (or from `calldata`) always create an independent copy.
- Assignments from `memory` to `memory` only create references. This means that changes to one memory variable are also visible in all other memory variables that refer to the same data.
- Assignments from `storage` to a **local** storage variable also only assign a reference.
- All other assignments to `storage` always copy. Examples for this case are assignments to state variables or to members of local variables of storage struct type, even if the local variable itself is just a reference.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    // The data location of x is storage.
    // This is the only place where the
    // data location can be omitted.
    uint[] x;

    // The data location of memoryArray is memory.
    function f(uint[] memory memoryArray) public {
        x = memoryArray; // works, copies the whole array to storage
        uint[] storage y = x; // works, assigns a pointer, data location of y is storage
        y[7]; // fine, returns the 8th element
        y.pop(); // fine, modifies x through y
        delete x; // fine, clears the array, also modifies y
        // The following does not work; it would need to create a new temporary /
        // unnamed array in storage, but storage is "statically" allocated:
        // y = memoryArray;
        // This does not work either, since it would "reset" the pointer, but there
        // is no sensible location it could point to.
```

(suite sur la page suivante)

(suite de la page précédente)

```

    // delete y;
    g(x); // calls g, handing over a reference to x
    h(x); // calls h and creates an independent, temporary copy in memory
}

function g(uint[] storage) internal pure {}
function h(uint[] memory) public pure {}
}

```

Arrays

Arrays can have a compile-time fixed size, or they can have a dynamic size.

The type of an array of fixed size k and element type T is written as $T[k]$, and an array of dynamic size as $T[]$.

For example, an array of 5 dynamic arrays of `uint` is written as `uint[][5]`. The notation is reversed compared to some other languages. In Solidity, `X[3]` is always an array containing three elements of type `X`, even if `X` is itself an array. This is not the case in other languages such as C.

Indices are zero-based, and access is in the opposite direction of the declaration.

For example, if you have a variable `uint[][5] memory x`, you access the seventh `uint` in the third dynamic array using `x[2][6]`, and to access the third dynamic array, use `x[2]`. Again, if you have an array `T[5] a` for a type `T` that can also be an array, then `a[2]` always has type `T`.

Array elements can be of any type, including mapping or struct. The general restrictions for types apply, in that mappings can only be stored in the `storage` data location and publicly-visible functions need parameters that are *ABI types*.

It is possible to mark state variable arrays `public` and have Solidity create a *getter*. The numeric index becomes a required parameter for the getter.

Accessing an array past its end causes a failing assertion. Methods `.push()` and `.push(value)` can be used to append a new element at the end of the array, where `.push()` appends a zero-initialized element and returns a reference to it.

bytes and string as Arrays

Variables of type `bytes` and `string` are special arrays. The `bytes` type is similar to `bytes1[]`, but it is packed tightly in calldata and memory. `string` is equal to `bytes` but does not allow length or index access.

Solidity does not have string manipulation functions, but there are third-party string libraries. You can also compare two strings by their keccak256-hash using `keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2))` and concatenate two strings using `bytes.concat(bytes(s1), bytes(s2))`.

You should use `bytes` over `bytes1[]` because it is cheaper, since using `bytes1[]` in memory adds 31 padding bytes between the elements. Note that in `storage`, the padding is absent due to tight packing, see *bytes and string*. As a general rule, use `bytes` for arbitrary-length raw byte data and `string` for arbitrary-length string (UTF-8) data. If you can limit the length to a certain number of bytes, always use one of the value types `bytes1` to `bytes32` because they are much cheaper.

Note : If you want to access the byte-representation of a string `s`, use `bytes(s).length / bytes(s)[7] = 'x'`; . Keep in mind that you are accessing the low-level bytes of the UTF-8 representation, and not the individual characters.

bytes.concat function

You can concatenate a variable number of bytes or bytes1 ... bytes32 using bytes.concat. The function returns a single bytes memory array that contains the contents of the arguments without padding. If you want to use string parameters or other types, you need to convert them to bytes or bytes1.../bytes32 first.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract C {
    bytes s = "Storage";
    function f(bytes calldata c, string memory m, bytes16 b) public view {
        bytes memory a = bytes.concat(s, c, c[:2], "Literal", bytes(m), b);
        assert((s.length + c.length + 2 + 7 + bytes(m).length + 16) == a.length);
    }
}
```

If you call bytes.concat without arguments it will return an empty bytes array.

Allocating Memory Arrays

Memory arrays with dynamic length can be created using the new operator. As opposed to storage arrays, it is **not** possible to resize memory arrays (e.g. the .push member functions are not available). You either have to calculate the required size in advance or create a new memory array and copy every element.

As all variables in Solidity, the elements of newly allocated arrays are always initialized with the *default value*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f(uint len) public pure {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        assert(a.length == 7);
        assert(b.length == len);
        a[6] = 8;
    }
}
```

Array Literals

An array literal is a comma-separated list of one or more expressions, enclosed in square brackets ([...]). For example [1, a, f(3)]. The type of the array literal is determined as follows :

It is always a statically-sized memory array whose length is the number of expressions.

The base type of the array is the type of the first expression on the list such that all other expressions can be implicitly converted to it. It is a type error if this is not possible.

It is not enough that there is a type all the elements can be converted to. One of the elements has to be of that type.

In the example below, the type of [1, 2, 3] is uint8[3] memory, because the type of each of these constants is uint8. If you want the result to be a uint[3] memory type, you need to convert the first element to uint.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

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

The array literal `[1, -1]` is invalid because the type of the first expression is `uint8` while the type of the second is `int8` and they cannot be implicitly converted to each other. To make it work, you can use `[int8(1), -1]`, for example.

Since fixed-size memory arrays of different type cannot be converted into each other (even if the base types can), you always have to specify a common base type explicitly if you want to use two-dimensional array literals :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f() public pure returns (uint24[2][4] memory) {
        uint24[2][4] memory x = [[uint24(0x1), 1], [0xffffffff, 2], [uint24(0xff), 3],
        ←[uint24(0xffff), 4]];
        // The following does not work, because some of the inner arrays are not of the
        ←right type.
        // uint[2][4] memory x = [[0x1, 1], [0xffffffff, 2], [0xff, 3], [0xffff, 4]];
        return x;
    }
}
```

Fixed size memory arrays cannot be assigned to dynamically-sized memory arrays, i.e. the following is not possible :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

// This will not compile.
contract C {
    function f() public {
        // The next line creates a type error because uint[3] memory
        // cannot be converted to uint[] memory.
        uint[] memory x = [uint(1), 3, 4];
    }
}
```

It is planned to remove this restriction in the future, but it creates some complications because of how arrays are passed in the ABI.

If you want to initialize dynamically-sized arrays, you have to assign the individual elements :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;
```

(suite sur la page suivante)

```

contract C {
    function f() public pure {
        uint[] memory x = new uint[](3);
        x[0] = 1;
        x[1] = 3;
        x[2] = 4;
    }
}

```

Array Members

length : Arrays have a `length` member that contains their number of elements. The length of memory arrays is fixed (but dynamic, i.e. it can depend on runtime parameters) once they are created.

push() : Dynamic storage arrays and bytes (not string) have a member function called `push()` that you can use to append a zero-initialised element at the end of the array. It returns a reference to the element, so that it can be used like `x.push().t = 2` or `x.push() = b`.

push(x) : Dynamic storage arrays and bytes (not string) have a member function called `push(x)` that you can use to append a given element at the end of the array. The function returns nothing.

pop : Dynamic storage arrays and bytes (not string) have a member function called `pop` that you can use to remove an element from the end of the array. This also implicitly calls *delete* on the removed element.

Note : Increasing the length of a storage array by calling `push()` has constant gas costs because storage is zero-initialised, while decreasing the length by calling `pop()` has a cost that depends on the « size » of the element being removed. If that element is an array, it can be very costly, because it includes explicitly clearing the removed elements similar to calling *delete* on them.

Note : To use arrays of arrays in external (instead of public) functions, you need to activate ABI coder v2.

Note : In EVM versions before Byzantium, it was not possible to access dynamic arrays return from function calls. If you call functions that return dynamic arrays, make sure to use an EVM that is set to Byzantium mode.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract ArrayContract {
    uint[2**20] m_aLotOfIntegers;
    // Note that the following is not a pair of dynamic arrays but a
    // dynamic array of pairs (i.e. of fixed size arrays of length two).
    // Because of that, T[] is always a dynamic array of T, even if T
    // itself is an array.
    // Data location for all state variables is storage.
    bool[2][] m_pairsOfFlags;

    // newPairs is stored in memory - the only possibility
    // for public contract function arguments

```

(suite sur la page suivante)

(suite de la page précédente)

```

function setAllFlagPairs(bool[2][] memory newPairs) public {
    // assignment to a storage array performs a copy of ``newPairs`` and
    // replaces the complete array ``m_pairsOfFlags``.
    m_pairsOfFlags = newPairs;
}

struct StructType {
    uint[] contents;
    uint moreInfo;
}
StructType s;

function f(uint[] memory c) public {
    // stores a reference to ``s`` in ``g``
    StructType storage g = s;
    // also changes ``s.moreInfo``.
    g.moreInfo = 2;
    // assigns a copy because ``g.contents``
    // is not a local variable, but a member of
    // a local variable.
    g.contents = c;
}

function setFlagPair(uint index, bool flagA, bool flagB) public {
    // access to a non-existing index will throw an exception
    m_pairsOfFlags[index][0] = flagA;
    m_pairsOfFlags[index][1] = flagB;
}

function changeFlagArraySize(uint newSize) public {
    // using push and pop is the only way to change the
    // length of an array
    if (newSize < m_pairsOfFlags.length) {
        while (m_pairsOfFlags.length > newSize)
            m_pairsOfFlags.pop();
    } else if (newSize > m_pairsOfFlags.length) {
        while (m_pairsOfFlags.length < newSize)
            m_pairsOfFlags.push();
    }
}

function clear() public {
    // these clear the arrays completely
    delete m_pairsOfFlags;
    delete m_aLotOfIntegers;
    // identical effect here
    m_pairsOfFlags = new bool[2][](0);
}

bytes m_byteData;

function byteArrayS(bytes memory data) public {

```

(suite sur la page suivante)

(suite de la page précédente)

```

// byte arrays ("bytes") are different as they are stored without padding,
// but can be treated identical to "uint8[]"
m_byteData = data;
for (uint i = 0; i < 7; i++)
    m_byteData.push();
m_byteData[3] = 0x08;
delete m_byteData[2];
}

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

function createMemoryArray(uint size) public pure returns (bytes memory) {
    // Dynamic memory arrays are created using `new`:
    uint[2][] memory arrayOfPairs = new uint[2][](size);

    // Inline arrays are always statically-sized and if you only
    // use literals, you have to provide at least one type.
    arrayOfPairs[0] = [uint(1), 2];

    // Create a dynamic byte array:
    bytes memory b = new bytes(200);
    for (uint i = 0; i < b.length; i++)
        b[i] = bytes1(uint8(i));
    return b;
}
}

```

Array Slices

Array slices are a view on a contiguous portion of an array. They are written as `x[start:end]`, where `start` and `end` are expressions resulting in a `uint256` type (or implicitly convertible to it). The first element of the slice is `x[start]` and the last element is `x[end - 1]`.

If `start` is greater than `end` or if `end` is greater than the length of the array, an exception is thrown.

Both `start` and `end` are optional : `start` defaults to `0` and `end` defaults to the length of the array.

Array slices do not have any members. They are implicitly convertible to arrays of their underlying type and support index access. Index access is not absolute in the underlying array, but relative to the start of the slice.

Array slices do not have a type name which means no variable can have an array slices as type, they only exist in intermediate expressions.

Note : As of now, array slices are only implemented for calldata arrays.

Array slices are useful to ABI-decode secondary data passed in function parameters :

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.5 <0.9.0;

```

(suite sur la page suivante)

(suite de la page précédente)

```

contract Proxy {
    /// @dev Address of the client contract managed by proxy i.e., this contract
    address client;

    constructor(address _client) {
        client = _client;
    }

    /// Forward call to "setOwner(address)" that is implemented by client
    /// after doing basic validation on the address argument.
    function forward(bytes calldata _payload) external {
        bytes4 sig = bytes4(_payload[:4]);
        // Due to truncating behaviour, bytes4(_payload) performs identically.
        // bytes4 sig = bytes4(_payload);
        if (sig == bytes4(keccak256("setOwner(address)"))) {
            address owner = abi.decode(_payload[4:], (address));
            require(owner != address(0), "Address of owner cannot be zero.");
        }
        (bool status,) = client.delegatecall(_payload);
        require(status, "Forwarded call failed.");
    }
}

```

Structs

Solidity provides a way to define new types in the form of structs, which is shown in the following example :

```

/// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

/// Defines a new type with two fields.
/// Declaring a struct outside of a contract allows
/// it to be shared by multiple contracts.
/// Here, this is not really needed.
struct Funder {
    address addr;
    uint amount;
}

contract CrowdFunding {
    /// Structs can also be defined inside contracts, which makes them
    /// visible only there and in derived contracts.
    struct Campaign {
        address payable beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping (uint => Funder) funders;
    }

    uint numCampaigns;
}

```

(suite sur la page suivante)

```

mapping (uint => Campaign) campaigns;

function newCampaign(address payable beneficiary, uint goal) public returns (uint,
↪campaignID) {
    campaignID = numCampaigns++; // campaignID is return variable
    // We cannot use "campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0)"
    // because the right hand side creates a memory-struct "Campaign" that contains
↪a mapping.
    Campaign storage c = campaigns[campaignID];
    c.beneficiary = beneficiary;
    c.fundingGoal = goal;
}

function contribute(uint campaignID) public payable {
    Campaign storage c = campaigns[campaignID];
    // Creates a new temporary memory struct, initialised with the given values
    // and copies it over to storage.
    // Note that you can also use Funder(msg.sender, msg.value) to initialise.
    c.funders[c.numFunders++] = Funder({addr: msg.sender, amount: msg.value});
    c.amount += msg.value;
}

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

```

The contract does not provide the full functionality of a crowdfunding contract, but it contains the basic concepts necessary to understand structs. Struct types can be used inside mappings and arrays and they can themselves contain mappings and arrays.

It is not possible for a struct to contain a member of its own type, although the struct itself can be the value type of a mapping member or it can contain a dynamically-sized array of its type. This restriction is necessary, as the size of the struct has to be finite.

Note how in all the functions, a struct type is assigned to a local variable with data location `storage`. This does not copy the struct but only stores a reference so that assignments to members of the local variable actually write to the state.

Of course, you can also directly access the members of the struct without assigning it to a local variable, as in `campaigns[campaignID].amount = 0`.

Note : Until Solidity 0.7.0, memory-structs containing members of storage-only types (e.g. mappings) were allowed and assignments like `campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0)` in the example above would work and just silently skip those members.

3.6.3 Type Mapping

Les types de mappage utilisent la syntaxe `mapping(_KeyType => _ValueType)` et des variables de type mapping sont déclarés en utilisant la syntaxe `mapping(_KeyType => _ValueType) _VariableName`. Le `_KeyType` peut être n'importe quel type de valeur intégré, `bytes`, `string`, ou tout type de contrat ou d'énumération. Autre défini par l'utilisateur ou les types complexes, tels que les mappages, les structures ou les types de tableau ne sont pas autorisés. `_ValueType` peut être n'importe quel type, y compris les mappages, les tableaux et les structures.

Vous pouvez considérer les mappages comme des [tables de hachage](#), qui sont virtuellement initialisées telle que chaque clé possible existe et est mappée à une valeur dont byte-representation n'est que des zéros, la *default value* d'un type. La similitude s'arrête là, les données clés ne sont pas stockées dans un mappage, seul son hachage `keccak256` est utilisé pour rechercher la valeur.

Pour cette raison, les mappages n'ont pas de longueur ou de concept de clé ou valeur définie et ne peut donc pas être effacée sans informations supplémentaires concernant les clés attribuées (voir [Effacement des mappages](#)).

Les mappages ne peuvent avoir qu'un emplacement de données : le `storage` et donc sont autorisés que pour les variables d'état (State), en tant que types de référence de stockage (storage) dans les fonctions ou comme paramètres pour les fonctions de la bibliothèque. Ils ne peuvent pas être utilisés comme paramètres ou paramètres de retour (return) des fonctions contractuelles qui sont publiquement visibles. Ces restrictions s'appliquent également aux tableaux et structures contenant des mappages.

Vous pouvez marquer les variables d'état de type mappage comme `public` et Solidity crée un *getter* pour vous. Le `_KeyType` devient un paramètre pour le getter. Si `_ValueType` est un type valeur ou une structure, le getter renvoie `_ValueType`. Si `_ValueType` est un tableau ou un mappage, le getter a un paramètre pour chaque `_KeyType`, récursivement.

Dans l'exemple ci-dessous, le contrat `MappingExample` définit un `balances` public mappage, avec le type de clé une adresse, et un type de valeur un `uint`, map une adresse Ethereum à une valeur entière non signée. Comme `uint` est un type valeur, le getter renvoie une valeur qui correspond au type, que vous pouvez voir dans le `MappingUser` contrat qui renvoie la valeur à l'adresse spécifiée.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

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

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

contract MappingUser {
    function f() public returns (uint) {
        MappingExample m = new MappingExample();
        m.update(100);
        return m.balances(address(this));
    }
}
```

L'exemple ci-dessous est une version simplifiée d'un [Jeton ERC20](#). `_allowances` est un exemple de type de mappage à l'intérieur d'un autre type de mappage. L'exemple ci-dessous utilise `_allowances` pour enregistrer le montant que quelqu'un d'autre est autorisé à retirer de votre compte.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract MappingExample {

    mapping (address => uint256) private _balances;
    mapping (address => mapping (address => uint256)) private _allowances;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    function allowance(address owner, address spender) public view returns (uint256) {
        return _allowances[owner][spender];
    }

    function transferFrom(address sender, address recipient, uint256 amount) public
↳ returns (bool) {
        require(_allowances[sender][msg.sender] >= amount, "ERC20: Allowance not high
↳ enough.");
        _allowances[sender][msg.sender] -= amount;
        _transfer(sender, recipient, amount);
        return true;
    }

    function approve(address spender, uint256 amount) public returns (bool) {
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
    }

    function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");
        require(_balances[sender] >= amount, "ERC20: Not enough funds.");

        _balances[sender] -= amount;
        _balances[recipient] += amount;
        emit Transfer(sender, recipient, amount);
    }
}
```

Mapping itérables

Vous ne pouvez pas itérer les mappages, c'est-à-dire que vous ne pouvez pas énumérer leurs clés. Il est cependant possible d'implémenter une structure de données par dessus d'eux et itérer dessus. Par exemple, le code ci-dessous implémente un bibliothèque `IterableMapping` que le contrat `User` ajoute également des données, et la fonction `sum` effectue une itération pour additionner toutes les valeurs.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.8 <0.9.0;

struct IndexValue { uint keyIndex; uint value; }
struct KeyFlag { uint key; bool deleted; }

struct itmap {
    mapping(uint => IndexValue) data;
    KeyFlag[] keys;
    uint size;
}

library IterableMapping {
    function insert(itmap storage self, uint key, uint value) internal returns (bool
    ←replaced) {
        uint keyIndex = self.data[key].keyIndex;
        self.data[key].value = value;
        if (keyIndex > 0)
            return true;
        else {
            keyIndex = self.keys.length;
            self.keys.push();
            self.data[key].keyIndex = keyIndex + 1;
            self.keys[keyIndex].key = key;
            self.size++;
            return false;
        }
    }

    function remove(itmap storage self, uint key) internal returns (bool success) {
        uint keyIndex = self.data[key].keyIndex;
        if (keyIndex == 0)
            return false;
        delete self.data[key];
        self.keys[keyIndex - 1].deleted = true;
        self.size --;
    }

    function contains(itmap storage self, uint key) internal view returns (bool) {
        return self.data[key].keyIndex > 0;
    }

    function iterate_start(itmap storage self) internal view returns (uint keyIndex) {
        return iterate_next(self, type(uint).max);
    }
}
```

(suite sur la page suivante)

```
function iterate_valid(itmap storage self, uint keyIndex) internal view returns (
↪bool) {
    return keyIndex < self.keys.length;
}

function iterate_next(itmap storage self, uint keyIndex) internal view returns (uint,
↪r_keyIndex) {
    keyIndex++;
    while (keyIndex < self.keys.length && self.keys[keyIndex].deleted)
        keyIndex++;
    return keyIndex;
}

function iterate_get(itmap storage self, uint keyIndex) internal view returns (uint,
↪key, uint value) {
    key = self.keys[keyIndex].key;
    value = self.data[key].value;
}
}

// Comme l'utiliser
contract User {
    // Juste un struct contenant nos données
    itmap data;
    // Appliquez les fonctions de la bibliothèque au type de données.
    using IterableMapping for itmap;

    // Ajouter quelque chose
    function insert(uint k, uint v) public returns (uint size) {
        // Appel IterableMapping.insert(data, k, v)
        data.insert(k, v);
        // Nous pouvons toujours accéder aux membres de la struct,
        // mais nous devons faire attention de ne pas jouer avec eux.
        return data.size;
    }

    // Calcule la somme de toutes les données stockées.
    function sum() public view returns (uint s) {
        for (
            uint i = data.iterate_start();
            data.iterate_valid(i);
            i = data.iterate_next(i)
        ) {
            (, uint value) = data.iterate_get(i);
            s += value;
        }
    }
}
```

3.6.4 Les opérateurs (arithmétique)

Les opérateurs arithmétiques et binaires peuvent être appliqués même si les deux opérandes n'ont pas le même type. Par exemple, vous pouvez calculer $y = x + z$, où x est un `uint8` et z a le type `int32`. Dans ces cas, le mécanisme suivant sera utilisé pour déterminer le type dans lequel l'opération est calculée (c'est important en cas de débordement) et le type du résultat de l'opérateur :

1. Si le type de l'opérande droit peut être implicitement converti en type de l'opérande gauche utilisez le type de l'opérande de gauche,
2. Si le type de l'opérande gauche peut être implicitement converti en type de l'opérande droite utilisez le type de l'opérande de droite,
3. Sinon, l'opération n'est pas autorisée.

Dans le cas où l'un des opérandes est un *literal number* il est d'abord converti en son « type mobile », qui est le plus petit type pouvant contenir la valeur (les types non signés de même largeur de bit sont considérés comme « plus petits » que les types signés). Si les deux sont des nombres littéraux, l'opération est calculée avec une précision arbitraire.

Le type de résultat de l'opérateur est le même que le type dans lequel l'opération est effectuée, sauf pour les opérateurs de comparaison où le résultat est toujours `bool`.

Les opérateurs `**` (exponentiation), `<<` and `>>` utilisent le type du opérande de gauche pour l'opération et le résultat.

Opérateurs composés et d'incrément/décément

Si `a` est une LValue (c'est-à-dire une variable ou quelque chose qui peut être assignée), les opérateurs suivants sont disponibles comme raccourcis :

`a += e` est équivalent à `a = a + e`. Les opérations `--`, `*=`, `/=`, `%=`, `|=`, `&=`, `^=`, `<<=` and `>>=` sont définis en conséquence. `a++` and `a--` est équivalent à `a += 1 / a -= 1` mais l'expression elle-même a toujours la valeur précédente de `a`. En revanche, `--a` et `++a` ont le même effet sur `a` mais retourne la valeur après le changement.

`delete`

`delete a` affecte la valeur initiale du type à `a`. C'est à dire. pour les entiers c'est équivalent à `a = 0`, mais il peut aussi être utilisé sur des tableaux, où il assigne une dynamique tableau de longueur zéro ou un tableau statique de même longueur avec tous les éléments mis à leur valeur initiale. `delete a[x]` supprime l'élément à l'index `x` du tableau et laisse tous les autres éléments et la longueur du tableau intacts. Cela signifie surtout qu'il laisse une lacune dans le tableau. Si vous envisagez de supprimer des éléments, un *mapping* est probablement un meilleur choix.

Pour les structures, il attribue une structure avec tous les membres réinitialisés. Autrement dit, la valeur de `a` après `delete a` est la même que si `a` était déclaré sans affectation, avec la mise en garde suivante :

`delete` n'a aucun effet sur les mapping (car les clés des mappages peuvent être arbitraires et sont généralement inconnus). Donc, si vous supprimez une structure, elle réinitialisera tous les membres qui ne sont pas des mapping et se récursent également dans les membres à moins qu'il ne s'agisse de mapping. Cependant, les clés individuelles et ce à quoi elles correspondent peuvent être supprimées : si `a` est un mapping, alors `delete a[x]` supprimera la valeur stockée à `x`.

Il est important de noter que `delete a` se comporte vraiment comme un affectation à `a`, c'est-à-dire qu'il stocke un nouvel objet dans `a`. Cette distinction est visible lorsque `a` est une variable de référence : ne réinitialisera que a lui-même, pas le valeur à laquelle il se référait précédemment.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract DeleteExample {
```

(suite sur la page suivante)

```

uint data;
uint[] dataArray;

function f() public {
    uint x = data;
    delete x; // définit x sur 0, n'affecte pas les données
    delete data; // définit les données sur 0, n'affecte pas x
    uint[] storage y = dataArray;
    delete dataArray; // cela définit dataArray.length à zéro, mais comme uint[] est
↳ un objet complexe, aussi
    // il est affecté qui est un alias de l'objet de stockage
    // Par contre : "delete y" n'est pas valide, car les affectations aux variables
↳ locales
    // les objets de stockage de référence ne peuvent être créés qu'à partir d'objets
↳ de stockage existants.
    assert(y.length == 0);
}
}

```

3.6.5 Conversions entre types élémentaires

Conversions implicites

Une conversion de type implicite est automatiquement appliquée par le compilateur dans certains cas lors des affectations, lors du passage d'arguments aux fonctions et lors de l'application d'opérateurs. En général, une conversion implicite entre les types de valeur est possible si elle est sémantique et qu'aucune information n'est perdue.

Par exemple, `uint8` est convertible en `uint16` et `int128` en `int256`, mais `int8` n'est pas convertible en `uint256`, car `uint256` ne peut pas contenir de valeurs telles que `-1`.

Si un opérateur est appliqué à différents types, le compilateur essaie implicitement convertir l'un des opérandes dans le type de l'autre (il en va de même pour les affectations). Cela signifie que les opérations sont toujours effectuées dans le type de l'un des opérandes.

Pour plus de détails sur les conversions implicites possibles, veuillez consulter les sections sur les types eux-mêmes.

Dans l'exemple ci-dessous, `y` et `z`, les opérandes de l'addition, n'ont pas le même type, mais `uint8` peut être implicitement converti en `uint16` et non l'inverse. À cause de ça, `y` est converti dans le type de `z` avant que l'addition ne soit effectuée dans le type `uint16`. Le type résultant de l'expression `y + z` est `uint16`. Parce qu'il est assigné à une variable de type `uint32` une autre conversion implicite est effectuée après l'addition.

```

uint8 y;
uint16 z;
uint32 x = y + z;

```

Conversions explicites

Si le compilateur n'autorise pas la conversion implicite mais que vous êtes sûr qu'une conversion fonctionnera, une conversion de type explicite est parfois possible. Ceci peut entraîner un comportement inattendu et vous permet de contourner certaines mesures de sécurité fonctionnalités du compilateur, assurez-vous donc de tester que le résultat est ce que vous voulez et attendez !

Prenons l'exemple suivant qui convertit un `int` négatif en un `uint` :

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

A la fin de cet extrait de code, `x` aura la valeur `0xffff...fd` (64 hex caractères), qui est -3 dans la représentation en complément à deux de 256 bits (Ce qui donnera une erreur).

Si un entier est explicitement converti en un type plus petit, les bits d'ordre supérieur sont coupés :

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b sera maintenant égale à 0x5678
```

Si un entier (integer) est explicitement converti en un type plus grand, il est rempli à gauche (c'est-à-dire à l'extrémité d'ordre supérieur). Le résultat de la conversion sera égal à l'entier d'origine :

```
uint16 a = 0x1234;
uint32 b = uint32(a); // b sera maintenant égale à 0x00001234
assert(a == b);
```

Les types d'octets de taille fixe se comportent différemment lors des conversions. Ils peuvent être considérés comme séquences d'octets individuels et la conversion en un type plus petit coupera la séquence :

```
bytes2 a = 0x1234;
bytes1 b = bytes1(a); // b sera égale à 0x12
```

Si un type d'octets de taille fixe est explicitement converti en un type plus grand, il est rempli sur la droite. L'accès à l'octet à un index fixe se traduira par la même valeur avant et après la conversion (si l'indice est toujours dans la plage) :

```
bytes2 a = 0x1234;
bytes4 b = bytes4(a); // b sera égale à 0x12340000
assert(a[0] == b[0]);
assert(a[1] == b[1]);
```

Étant donné que les entiers et les tableaux d'octets de taille fixe se comportent différemment lors de la troncature ou du padding, les conversions explicites entre entiers et tableaux d'octets de taille fixe ne sont autorisées, si les deux ont la même taille. Si vous voulez convertir entre des nombres entiers et des tableaux d'octets de taille fixe de taille différente, vous devez utiliser des conversions intermédiaires qui font la troncature et le padding souhaités Règles explicites :

```
bytes2 a = 0x1234;
uint32 b = uint16(a); // b sera égale à 0x00001234
uint32 c = uint32(bytes4(a)); // c sera égale à 0x12340000
uint8 d = uint8(uint16(a)); // d sera égale à 0x34
uint8 e = uint8(bytes1(a)); // e sera égale à 0x12
```

Les tableaux `bytes` et les tranches de `calldata bytes` peuvent être convertis explicitement en types d'octets fixes (`bytes1.../bytes32`). Si le tableau est plus long que le type d'octets fixes cible, une troncature à la fin se produira. Si le tableau est plus court que le type cible, il sera complété par des zéros à la fin.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.5;

contract C {
    bytes s = "abcdefgh";
    function f(bytes calldata c, bytes memory m) public view returns (bytes16, bytes3) {
        require(c.length == 16, "");
        bytes16 b = bytes16(m); // si la longueur de m est supérieure à 16, la
↳troncature se produira
        b = bytes16(s); // rembourré à droite, donc le résultat est "abcdefgh\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"
↳\0\0\0\0"
        bytes3 b1 = bytes3(s); // tronqué, b1 est égal à "abc"
        b = bytes16(c[:8]); // également rempli de zéros
        return (b, b1);
    }
}
```

3.6.6 Conversions entre littéraux et types élémentaires

Types entiers (Integer)

Les littéraux décimaux et hexadécimaux peuvent être implicitement convertis en n'importe quel type entier suffisamment grand pour le représenter sans troncature :

```
uint8 a = 12; // Pas d'erreurs
uint32 b = 1234; // Pas d'erreurs
uint16 c = 0x123456; // Échec, car il faudrait tronquer à 0x3456
```

Note : Avant la version 0.8.0, tous les littéraux décimaux ou hexadécimaux pouvaient être explicitement converti en un type entier. Depuis la version 0.8.0, ces conversions explicites sont aussi strictes qu'implicites conversions, c'est-à-dire qu'elles ne sont autorisées que si le littéral correspond à la plage résultante.

Tableaux d'octets de taille fixe

Les littéraux décimaux ne peuvent pas être implicitement convertis en tableaux d'octets de taille fixe. Hexadécimal les littéraux numériques peuvent être, mais seulement si le nombre de chiffres hexadécimaux correspond exactement à la taille des octets taper. Exceptionnellement, les littéraux décimaux et hexadécimaux qui ont une valeur de zéro peuvent être converti en n'importe quel type d'octets de taille fixe :

```
bytes2 a = 54321; // Interdit
bytes2 b = 0x12; // Interdit
bytes2 c = 0x123; // Interdit
bytes2 d = 0x1234; // OK
bytes2 e = 0x0012; // OK
bytes4 f = 0; // OK
bytes4 g = 0x0; // OK
```

Les littéraux de chaîne et les littéraux de chaîne hexadécimaux peuvent être implicitement convertis en tableaux d'octets de taille fixe, si leur nombre de caractères correspond à la taille du type d'octets :

```

bytes2 a = hex"1234"; // OK
bytes2 b = "xy"; // OK
bytes2 c = hex"12"; // Interdit
bytes2 d = hex"123"; // Interdit
bytes2 e = "x"; // Interdit
bytes2 f = "xyz"; // Interdit

```

Addresses

Comme décrit dans *Address Literals*, les littéraux hexadécimaux de la taille correcte qui passent la somme de contrôle test sont de type `address`. Aucun autre littéral ne peut être implicitement converti en type `address`.

Les conversions explicites de `bytes20` ou de n'importe quel type d'entier en `address payable` résultent en `address payable`.

Une `address` `a` peut être convertie en `address payable` via `payable(a)`.

3.7 Unités et variables disponibles dans le monde entier

3.7.1 Unités d'éther

Un nombre littéral peut prendre un suffixe de `wei`, `gwei` ou `ether` pour spécifier une sous-dénomination d'Ether, où les nombres d'Ether sans postfixe sont supposés être Wei.

```

assert(1 wei == 1);
assert(1 gwei == 1e9);
assert(1 ether == 1e18);

```

Le seul effet du suffixe de sous-dénomination est une multiplication par une puissance de dix.

Note : Les dénominations `finney` et `szabo` ont été supprimées dans la version 0.7.0.

3.7.2 Unités de temps

Les suffixes comme `seconds`, `minutes`, `hours`, `days` et `weeks`, après des nombres littéraux, peuvent être utilisés pour spécifier des unités de temps où les secondes sont l'unité de base et les unités sont considérées naïvement de la manière suivante :

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

Faites attention si vous effectuez des calculs de calendrier en utilisant ces unités, car chaque année n'est pas égale à 365 jours et chaque jour n'a pas 24 heures à cause des *secondes intercalaires*. En raison du fait que les secondes intercalaires ne peuvent pas être prédites, un calendrier exact doit être mis à jour par une bibliothèque doit être mise à jour par un oracle externe.

Note : Le suffixe `years` a été supprimé dans la version 0.5.0 pour les raisons ci-dessus.

Ces suffixes ne peuvent pas être appliqués aux variables. Par exemple, si vous voulez interpréter un paramètre de fonction en jours, vous pouvez le faire de la manière suivante :

```
function f(uint start, uint daysAfter) public {
    if (block.timestamp >= start + daysAfter * 1 days) {
        // ...
    }
}
```

3.7.3 Variables et fonctions spéciales

Certaines variables et fonctions spéciales existent toujours dans l'espace de nom global, et sont principalement utilisées pour fournir des informations sur la blockchain, ou sont des fonctions utilitaires d'usage général.

Propriétés des blocs et des transactions

- `blockhash(uint blockNumber)` retourne (`bytes32`) : hachage du bloc donné si `blocknumber` est l'un des 256 blocs les plus récents ; sinon retourne zéro.
- `block.basefee (uint)` : la redevance de base du bloc actuel (EIP-3198 et EIP-1559)
- `block.chainid (uint)` : identifiant de la chaîne actuelle
- `block.coinbase (address payable)` : adresse du mineur du bloc actuel
- `block.difficulty (uint)` : difficulté actuelle du bloc
- `block.gaslimit (uint)` : limite de gaz du bloc actuel
- `block.number (uint)` : numéro du bloc actuel
- `block.timestamp (uint)` : horodatage du bloc actuel en secondes depuis l'époque unix
- `gasleft()` returns (`uint256`) : gaz résiduel
- `msg.data (bytes calldata)` : données d'appel complètes
- `msg.sender (address)` : expéditeur du message (appel en cours)
- `msg.sig (bytes4)` : les quatre premiers octets des données d'appel (c'est-à-dire l'identifiant de la fonction)
- `msg.value (uint)` : nombre de wei envoyés avec le message
- `tx.gasprice (uint)` : prix du gaz de la transaction
- `tx.origin (address)` : expéditeur de la transaction (chaîne d'appel complète)

Note : Les valeurs de tous les membres de `msg`, y compris `msg.sender` et `msg.value` peuvent changer à chaque appel de fonction **externe**. Cela inclut les appels aux fonctions de la bibliothèque.

Note : Lorsque les contrats sont évalués hors chaîne plutôt que dans le contexte d'une transaction comprise dans un bloc, vous ne devez pas supposer que `block.*` et `tx.*` font référence à des valeurs d'un bloc ou d'une transaction spécifique. Ces valeurs sont fournies par l'implémentation EVM qui exécute le contrat et peuvent être arbitraires.

Note : Ne comptez pas sur `block.timestamp` ou `blockhash` comme source d'aléatoire, à moins que vous ne sachiez ce que vous faites.

L'horodatage et le hachage du bloc peuvent tous deux être influencés par les mineurs dans une certaine mesure. De mauvais acteurs dans la communauté minière peuvent par exemple exécuter une fonction de paiement de casino sur un hash choisi et réessayer un autre hash s'ils n'ont pas reçu d'argent.

L'horodatage du bloc actuel doit être strictement plus grand que l'horodatage du dernier bloc, mais la seule garantie est qu'il se situera quelque part entre les horodatages de deux blocs consécutifs dans la chaîne canonique.

Note : Les hachages des blocs ne sont pas disponibles pour tous les blocs pour des raisons d'évolutivité. Vous ne pouvez accéder qu'aux hachages des 256 blocs les plus récents. autres valeurs seront nulles.

Note : La fonction `blockhash` était auparavant connue sous le nom de `block.blockhash`, qui a été dépréciée dans la version 0.4.22 et supprimée dans la version 0.5.0.

Note : La fonction `gasleft` était auparavant connue sous le nom de `msg.gas`, qui a été dépréciée dans la version 0.4.21 et supprimée dans la version 0.5.0.

Note : Dans la version 0.7.0, l'alias `now` (pour `block.timestamp`) a été supprimé.

Fonctions de codage et de décodage de l'ABI

- `abi.decode(bytes memory encodedData, (...))` retourne `(...)` : ABI-décodes les données données, tandis que les types sont donnés entre parenthèses comme deuxième argument. Exemple : `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...)` returns `(bytes memory)` : ABI-encode les arguments donnés
- `abi.encodePacked(...)` returns `(bytes memory)` : Effectue l'*encodage emballé* des arguments donnés. Notez que l'encodage emballé peut être ambigu !
- `abi.encodeWithSelector(bytes4 selector, ...)` retourne `(bytes memory)` : ABI-encode les arguments donnés en commençant par le deuxième et ajoute en préambule le sélecteur de quatre octets donné.
- `abi.encodeWithSignature(string memory signature, ...)` retourne `(bytes memory)` : Équivalent à `abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)`
- `abi.encodeCall(function functionPointer, (...))` retourne `(bytes memory)` : ABI-encode un appel à `functionPointer` avec les arguments trouvés dans le tuple. Effectue un contrôle de type complet, en s'assurant que les types correspondent à la signature de la fonction. Le résultat est égal à `abi.encodeWithSelector(functionPointer.selector, (...))`

Note : Ces fonctions d'encodage peuvent être utilisées pour créer des données pour les appels de fonctions externes sans réellement appeler une fonction externe. De plus, `keccak256(abi.encodePacked(a, b))` est un moyen de calculer le hachage de données structurées (attention, il est possible de créer une « collision de hachage » en utilisant différents types de paramètres de fonction).

Reportez-vous à la documentation sur le *ABI* et le *codage étroitement emballé* pour plus de détails sur le codage.

Membres des octets

- `bytes.concat(...)` retourne `(bytes memory)` : *Concatène un nombre variable d'octets et les arguments bytes1, ..., bytes32 dans un tableau d'octets.*

Traitement des erreurs

Consultez la section dédiée à *assert et require* pour plus de détails sur la gestion des erreurs et quand utiliser telle ou telle fonction.

assert(bool condition) provoque une erreur de panique et donc un changement d'état si la condition n'est pas remplie - à utiliser pour les erreurs internes.

require(bool condition) revient en arrière si la condition n'est pas remplie - à utiliser pour les erreurs dans les entrées ou les composants externes.

require(bool condition, string memory message) fait marche arrière si la condition n'est pas remplie - à utiliser pour les erreurs dans les entrées ou les composants externes. Fournit également un message d'erreur.

revert() interrompt l'exécution et renverse les changements d'état

revert(string memory reason) interrompt l'exécution et annule les changements d'état, en fournissant une chaîne explicative.

Fonctions mathématiques et cryptographiques

addmod(uint x, uint y, uint k) retourne (uint) calcule $(x + y) \% k$ où l'addition est effectuée avec une précision arbitraire et ne s'arrête pas à 2^{256} . Affirme que $k \neq 0$ à partir de la version 0.5.0.

mulmod(uint x, uint y, uint k) retourne (uint) calcule $(x * y) \% k$ où la multiplication est effectuée avec une précision arbitraire et ne s'arrête pas à 2^{256} . Affirme que $k \neq 0$ à partir de la version 0.5.0.

keccak256(octets mémoire) retourne (octets32) calcule le hachage Keccak-256 de l'entrée

Note : Il y avait auparavant un alias pour keccak256 appelé sha3, qui a été supprimé dans la version 0.5.0.

sha256(bytes memory) retourne (bytes32) calcule le hachage SHA-256 de l'entrée

ripemd160(bytes memory) retourne (bytes20) calcule le hachage RIPEMD-160 de l'entrée

ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) retourne (address) récupère l'adresse associée à la clé publique de la signature à courbe elliptique ou renvoie zéro en cas d'erreur. Les paramètres de la fonction correspondent aux valeurs ECDSA de la signature :

- r = premiers 32 octets de la signature
- s = deuxième 32 octets de la signature
- v = dernier 1 octet de la signature

ecrecover retourne une adresse, et non une adresse payable. Voir *address payable* pour la conversion, au cas où vous auriez besoin de transférer des fonds à l'adresse récupérée.

Pour plus de détails, lisez *exemple usage*.

Avertissement : Si vous utilisez ecrecover, soyez conscient qu'une signature valide peut être transformée en une signature valide différente sans avoir besoin de connaître la clé privée correspondante. Dans le hard fork de Homestead, ce problème a été corrigé pour les signatures `_transaction_` (voir *EIP-2*), mais la fonction ecrecover est restée inchangée.

Ce n'est généralement pas un problème, à moins que vous n'exigiez que les signatures soient uniques ou que vous les utilisiez pour identifier des éléments. OpenZeppelin a une *ECDSA helper library* que vous pouvez utiliser comme un wrapper pour ecrecover sans ce problème.

Note : Lorsque vous exécutez les fonctions sha256, ripemd160 ou ecrecover sur une *blockchain privée*, vous pouvez rencontrer des problèmes d'épuisement. Cela est dû au fait que ces fonctions sont implémentées en tant que « contrats

précompilés » et n'existent réellement qu'après avoir reçu le premier message (bien que leur code de contrat soit codé en dur). Les messages destinés à des contrats inexistantes sont plus coûteux et l'exécution peut donc se heurter à une erreur Out-of-Gas. Une solution à ce problème consiste à envoyer d'abord du Wei (1 par exemple) à chacun des contrats avant de les utiliser dans vos contrats réels. Ce n'est pas un problème sur le réseau principal ou le réseau de test.

Membres des types d'adresses

- `<address>.balance (uint256)` solde de l'adresse dans Wei
- `<address>.code (bytes memory)` code à l'adresse (peut être vide)
- `<address>.codehash (bytes32)` le codehash de l'adresse *Address*.
- `<address payable>.transfer(uint256 amount)` envoie une quantité donnée de Wei à *adress*, revient en arrière en cas d'échec, envoie 2300 de gaz, non réglable
- `<address payable>.send(uint256 amount) returns (bool)` envoie un montant donné de Wei à *Address*, renvoie `false` en cas d'échec, envoie 2300 de gaz, non réglable
- `<address>.call(bytes memory) returns (bool, bytes memory)` émet un CALL de bas niveau avec la charge utile donnée, renvoie la condition de succès et les données de retour, transmet tous les gaz disponibles, ajustable
- `<address>.delegatecall(bytes memory) returns (bool, bytes memory)` émet un DELEGATECALL de bas niveau avec la charge utile donnée, renvoie la condition de succès et les données de retour, transmet tous les gaz disponibles, réglable
- `<address>.staticcall(bytes memory) returns (bool, bytes memory)` émet un STATICCALL de bas niveau avec la charge utile donnée, renvoie la condition de succès et les données de retour, transmet tous les gaz disponibles, réglable

Pour plus d'informations, consultez la section sur `address`.

Avertissement : Vous devez éviter d'utiliser `.call()` chaque fois que possible lors de l'exécution d'une autre fonction de contrat car elle contourne la vérification de type le contrôle d'existence de la fonction et l'emballage des arguments.

Avertissement : Il y a quelques dangers à utiliser `send` : Le transfert échoue si la profondeur de la pile d'appel est à 1024 (ceci peut toujours être forcé par l'appelant) et il échoue également si le destinataire tombe en panne sèche. Donc, afin de faire des transferts d'Ether sûrs, vérifiez toujours la valeur de retour de `send`, utilisez `transfer` ou encore mieux : Utilisez un modèle où le destinataire retire l'argent.

Avertissement : En raison du fait que l'EVM considère qu'un appel à un contrat inexistant réussit toujours, Solidity inclut une vérification supplémentaire en utilisant l'opcode `extcodesize` lors des appels externes. Cela garantit que le contrat qui est sur le point d'être appelé existe réellement (il contient du code) soit une exception est levée.

Les appels de bas niveau qui opèrent sur des adresses plutôt que sur des instances de contrat (c'est-à-dire `.call()`, `.delegatecall()`, `.staticcall()`, `.send()` et `.transfer()`) **n'incluent pas** cette vérification, ce qui les rend moins coûteux en termes de gaz mais aussi moins sûrs.

Note : Avant la version 0.5.0, Solidity permettait d'accéder aux membres `adresse` par une instance de contrat, par exemple `this.balance`. Ceci est maintenant interdit et une conversion explicite en adresse doit être faite :

`address(this).balance`.

Note : Si l'on accède à des variables d'état via un appel de délégué de bas niveau, la disposition de stockage des deux contrats doit s'aligner pour que le contrat appelé puisse accéder correctement aux variables de stockage du contrat appelant par leur nom. Ce n'est évidemment pas le cas si les pointeurs de stockage sont passés comme arguments de fonction, comme dans le cas des bibliothèques de haut niveau.

Note : Avant la version 0.5.0, `.call`, `.delegatecall` et `.staticcall` retournaient uniquement la condition de réussite et non les données de retour.

Note : Avant la version 0.5.0, il existait un membre appelé `callcode` `` avec une sémantique similaire mais légèrement différente de celle de `` `deallcode`, sémantique similaire mais légèrement différente de celle de `delegatecall`.

Concernant les contrats

this (le type du contrat actuel) le contrat actuel, explicitement convertible en *Address*.

selfdestruct(address payable recipient) Détruit le contrat actuel, en envoyant ses fonds à l'adresse *Address* donnée et mettre fin à l'exécution. Notez que `selfdestruct` a quelques particularités héritées de l'EVM :

- la fonction de réception du contrat récepteur n'est pas exécutée.
- le contrat n'est réellement détruit qu'à la fin de la transaction et les `revert` peuvent « annuler » la destruction.

En outre, toutes les fonctions du contrat en cours sont appelables directement, y compris la fonction en cours.

Note : Avant la version 0.5.0, il existait une fonction appelée `suicide` ayant la même sémantique que la fonction `selfdestruct`.

Informations sur le type de produit

L'expression `type(X)` peut être utilisée pour récupérer des informations sur le type `X`. Actuellement, la prise en charge de cette fonctionnalité est limitée (`X` peut être soit un contrat ou un type entier) mais elle pourrait être étendue dans le futur.

Les propriétés suivantes sont disponibles pour un type de contrat `C` :

type(C).name Le nom du contrat.

type(C).creationCode Tableau d'octets en mémoire qui contient le bytecode de création du contrat. Ceci peut être utilisé dans l'assemblage en ligne pour construire des routines de création personnalisées, notamment en utilisant l'opcode `create2`. Cette propriété n'est **pas** accessible dans le contrat lui-même ou dans un contrat dérivé. Elle provoque l'inclusion du bytecode dans le bytecode du site d'appel et donc les références circulaires de ce genre ne sont pas possibles.

type(C).runtimeCode Tableau d'octets en mémoire qui contient le bytecode d'exécution du contrat. Il s'agit du code qui est généralement déployé par le constructeur de `C`. Si `C` a un constructeur qui utilise l'assemblage en ligne, cela peut être différent du bytecode réellement déployé. Notez également que les bibliothèques modifient leur code d'exécution au moment du déploiement pour se prémunir contre les appels réguliers. Les mêmes restrictions que pour `.creationCode` s'appliquent à cette propriété.

En plus des propriétés ci-dessus, les propriétés suivantes sont disponibles pour une interface de type I :

type(I).interfaceId : Une valeur `bytes4` contenant le [EIP-165](#) de l'interface I donnée. Cet identificateur est défini comme étant le XOR de tous les sélecteurs de fonctions définis dans l'interface elle-même - à l'exclusion de toutes les fonctions héritées.

Les propriétés suivantes sont disponibles pour un type entier T :

type(T).min La plus petite valeur représentable par le type T.

type(T).max La plus grande valeur représentable par le type T.

3.8 Expressions et structures de contrôle

3.8.1 Structures de contrôle

La plupart des structures de contrôle connues des langages à accolades sont disponibles dans Solidity :

Il y a : `» if «`, `» else «`, `» while «`, `» do «`, `» for «`, `» break «`, `» continue «`, `» return «`, avec la sémantique la sémantique habituelle connue en C ou en JavaScript.

Solidity prend également en charge la gestion des exceptions sous la forme de déclarations `» try » et » catch «`, mais seulement pour *les appels de fonctions externes* et pour les appels de création de contrat. Les erreurs peuvent être créées en utilisant l'instruction `revert`.

Les parenthèses ne peuvent *pas* être omises pour les conditionnels, mais les accolades peuvent être omises autour des corps d'énoncés simples.

Notez qu'il n'y a pas de conversion de type de non-booléen à booléen comme en C et JavaScript. booléens comme c'est le cas en C et en JavaScript, donc `« if (1) { ... } »` n'est *pas* valide Solidité.

3.8.2 Appels de fonction

Appels de fonctions internes

Les fonctions du contrat en cours peuvent être appelées directement (« en interne »), également de manière récursive, comme on le voit dans cet exemple absurde :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

// Ceci signalera un avertissement
contract C {
    function g(uint a) public pure returns (uint ret) { return a + f(); }
    function f() internal pure returns (uint ret) { return g(7) + f(); }
}
```

Ces appels de fonction sont traduits en simples sauts à l'intérieur de l'EVM. Cela a pour l'effet que la mémoire courante n'est pas effacée, c'est-à-dire que le passage des références de mémoire aux fonctions appelées en interne est très efficace. Seules les fonctions de la même instance de contrat peuvent être appelées en interne.

Vous devez néanmoins éviter toute récursion excessive, car chaque appel de fonction interne utilise au moins un emplacement de pile et il n'y a que 1024 emplacements disponibles.

External Function Calls

Les fonctions peuvent également être appelées en utilisant la notation `» this.g(8);` et » c.g(2);` , où c est une instance de contrat et g est une fonction appartenant à c. L'appel de la fonction g de l'une ou l'autre façon a pour conséquence qu'elle est appelée « en externe », en utilisant appel de message et non directement via des sauts. Veuillez noter que les appels de fonction sur this ne peuvent pas être utilisés dans le constructeur, car le contrat réel n'a pas encore été créé.`

Les fonctions des autres contrats doivent être appelées en externe. Pour un appel externe, tous les arguments de la fonction doivent être copiés en mémoire.

Note : Un appel de fonction d'un contrat à un autre ne crée pas sa propre transaction, il s'agit d'un appel de message faisant partie de la transaction globale.

Lorsque vous appelez des fonctions d'autres contrats, vous pouvez préciser la quantité de Wei ou de gaz envoyée avec l'appel avec les options spéciales `{valeur : 10, gaz : 10000}`. Notez qu'il est déconseillé de spécifier des valeurs de gaz explicitement, puisque les coûts de gaz des opcodes peuvent changer dans le futur. Tout Wei que vous envoyez au contrat est ajouté au solde total de ce contrat :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

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

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

Vous devez utiliser le modificateur `payable` avec la fonction `info` parce que sinon, l'option `value` ne serait pas disponible.

Avertissement : Attention, `feed.info{value : 10, gaz : 800}` ne définit que localement la valeur et la quantité de gaz envoyée avec l'appel de la fonction, et que les parenthèses à la fin effectuent l'appel réel. Donc `feed.info{value : 10, gaz : 800}` n'appelle pas la fonction et les paramètres « valeur » et `gaz` sont perdus, mais seulement `feed.info{value : 10, gaz : 800}()` effectue l'appel de fonction.

En raison du fait que l'EVM considère qu'un appel vers un contrat inexistant toujours réussir, Solidity utilise l'opcode `extcodesize` pour vérifier que le contrat qui est sur le point d'être appelé existe réellement (il contient du code) et provoque une exception si ce n'est pas le cas. Cette vérification est ignorée si les données de retour seront décodées après l'appel et donc le décodeur ABI va attraper le cas d'un contrat inexistant.

Notez que cette vérification n'est pas effectuée dans le cas de *appels de bas niveau* qui opèrent sur des adresses plutôt que sur des instances de contrat.

Note : Soyez prudent lorsque vous utilisez des appels de haut niveau à *contrats précompilés*, car le compilateur les considère comme inexistantes selon la logique ci-dessus, même s'ils exécutent du code et peuvent retourner des données.

Les appels de fonction provoquent également des exceptions si le contrat appelé lui-même lève une exception ou tombe en panne.

Avertissement : Toute interaction avec un autre contrat impose un danger potentiel, surtout si le code source du contrat n'est pas connu à l'avance. Le contrat en cours transmet le contrôle au contrat appelé et celui-ci peut potentiellement faire à peu près n'importe quoi. Même si le contrat appelé hérite d'un contrat parent connu, le contrat hérité est seulement tenu d'avoir une interface correcte. Le site L'implémentation du contrat, cependant, peut être complètement arbitraire et donc., constituer un danger. En outre, il faut se préparer à l'éventualité qu'il fasse appel à d'autres contrats de votre système ou même de revenir au contrat appelant avant que le premier appel ne revienne. Cela signifie que le contrat appelé peut modifier les variables d'état du contrat appelant via ses fonctions. Écrivez vos fonctions de manière à ce que, par exemple, les appels aux fonctions externes se produisent après toute modification des variables d'état dans votre contrat afin que votre contrat ne soit pas vulnérable à un exploit de réentraînement.

Note : Avant Solidity 0.6.2, la manière recommandée de spécifier la valeur et le gaz était de utiliser « f.value(x).gas(g)``. Cette méthode a été dépréciée dans Solidity 0.6.2 et n'est plus possible depuis Solidity 0.7.0.

Appels nominatifs et paramètres de fonctions anonymes

Les arguments d'un appel de fonction peuvent être donnés par leur nom, dans n'importe quel ordre, s'ils sont entourés de { } comme on peut le voir dans l'exemple suivant. La liste d'arguments doit coïncider par son nom avec la liste des paramètres de la déclaration de la fonction, mais peut être dans un ordre arbitraire.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract C {
    mapping(uint => uint) data;

    function f() public {
        set({value: 2, key: 3});
    }

    function set(uint key, uint value) public {
        data[key] = value;
    }
}
```

Noms des paramètres de la fonction omise

Les noms des paramètres non utilisés (en particulier les paramètres de retour) peuvent être omis. Ces paramètres seront toujours présents sur la pile, mais ils seront inaccessibles.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract C {
```

(suite sur la page suivante)

```
// nom omis pour le paramètre
function func(uint k, uint) public pure returns(uint) {
    return k;
}
}
```

3.8.3 Créer des contrats via new (nouveau)

Un contrat peut créer d'autres contrats en utilisant le mot-clé `new`. Le code complet du contrat en cours de création doit être connu lorsque le contrat créateur est compilé afin que les dépendances récursives de création ne soient pas possibles.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

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

contract C {
    D d = new D(4); // will be executed as part of C's constructor

    function created(uint arg) public {
        D newD = new D(arg);
        newD.x();
    }

    function createAndEndowD(uint arg, uint amount) public payable {
        // Send ether along with the creation
        D newD = new D{value: amount}(arg);
        newD.x();
    }
}
```

Comme on le voit dans l'exemple, il est possible d'envoyer de l'Ether en créant une instance de D en utilisant l'option `value`, mais il n'est pas possible de limiter la quantité d'éther. Si la création échoue (à cause d'un dépassement de pile, d'un équilibre insuffisant ou d'autres problèmes), une exception est levée.

Créations de contrats salés / create2

Lors de la création d'un contrat, l'adresse du contrat est calculée à partir de l'adresse du contrat créateur et d'un compteur qui est augmenté à chaque création de chaque création de contrat.

Si vous spécifiez l'option `salt` (une valeur `bytes32`), alors la création de contrat utilisera un un mécanisme différent pour trouver l'adresse du nouveau contrat :

Elle calculera l'adresse à partir de l'adresse du contrat en cours de création, la valeur du sel donnée, le bytecode (de création) du contrat créé et les arguments du constructeur.

En particulier, le compteur (« nonce ») n'est pas utilisé. Cela permet une plus grande flexibilité dans la création de contrats : Vous pouvez dériver l'adresse du nouveau contrat avant qu'il ne soit créé. En outre, vous pouvez vous fier à cette adresse également dans le cas où le créateur contrat crée d'autres contrats entre-temps.

Le principal cas d'utilisation ici est celui des contrats qui agissent en tant que juges pour les interactions hors chaîne, qui n'ont besoin d'être créés que s'il y a un différend.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract D {
    uint public x;
    constructor(uint a) {
        x = a;
    }
}

contract C {
    function createDSalted(bytes32 salt, uint arg) public {
        // Cette expression compliquée vous indique simplement comment l'adresse
        // peut être précalculée. Elle n'est là qu'à titre d'illustration.
        // En fait, vous n'avez besoin que de `new D{salt : salt}(arg)`.
        address predictedAddress = address(uint160(uint(keccak256(abi.encodePacked(
            bytes1(0xff),
            address(this),
            salt,
            keccak256(abi.encodePacked(
                type(D).creationCode,
                arg
            )))
        )))

        D d = new D{salt: salt}(arg);
        require(address(d) == predictedAddress);
    }
}
```

Avertissement : Il existe quelques particularités en ce qui concerne la création salée. Un contrat peut être recréé à la même adresse après avoir été détruit. Pourtant, il est possible pour ce contrat nouvellement créé d'avoir un bytecode déployé différent, même si le bytecode de création a été le même (ce qui est une exigence parce que sinon l'adresse changerait). Ceci est dû au fait que le constructeur peut interroger l'état externe qui pourrait avoir changé entre les deux créations et l'incorporer dans le bytecode déployé avant qu'il ne soit stocké.

3.8.4 Ordre d'évaluation des expressions

L'ordre d'évaluation des expressions n'est pas spécifié (de manière plus formelle, l'ordre dans lequel les enfants d'un noeud de l'arbre des expressions sont évalués n'est pas spécifié, mais ils sont bien sûr évalués avant le noeud lui-même). Il est seulement garanti que les instructions sont exécutées dans l'ordre et que le court-circuitage des expressions booléennes est effectué.

3.8.5 Affectation

Déstructurer les affectations et renvoyer des valeurs multiples

Solidity autorise en interne les types tuple, c'est-à-dire une liste d'objets potentiellement différents dont le nombre est une constante à la constante au moment de la compilation. Ces tuples peuvent être utilisés pour retourner plusieurs valeurs en même temps. Celles-ci peuvent alors être affectées à des variables nouvellement déclarées soit à des variables préexistantes (ou à des valeurs LV en général).

Les tuples ne sont pas des types à proprement parler dans Solidity, ils ne peuvent être utilisés que pour former des groupements syntaxiques d'expressions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    uint index;

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

    function g() public {
        // Variables déclarées avec le type et assignées à partir du tuple retourné,
        // il n'est pas nécessaire de spécifier tous les éléments (mais le nombre doit
        ↪ correspondre).
        (uint x, , uint y) = f();
        // Truc commun pour échanger des valeurs -- ne fonctionne pas pour les types de
        ↪ stockage sans valeur.
        (x, y) = (y, x);
        // Les composants peuvent être laissés de côté (également pour les déclarations
        ↪ de variables).
        (index, , ) = f(); // Sets the index to 7
    }
}
```

Il n'est pas possible de mélanger les déclarations de variables et les affectations non déclarées. Par exemple, l'exemple suivant n'est pas valide : `(x, uint y) = (1, 2);`

Note : Avant la version 0.5.0, il était possible d'assigner à des tuples de taille plus petite, soit en remplissant le côté gauche ou le côté droit (celui qui était vide). Ceci est maintenant interdit, donc les deux côtés doivent avoir le même nombre de composants.

Avertissement : Soyez prudent lorsque vous assignez à plusieurs variables en même temps lorsque des types de référence sont impliqués, car cela pourrait conduire à un comportement de copie inattendu.

Complications pour les tableaux et les structures

La sémantique des affectations est plus compliquée pour les types non-valeurs comme les tableaux et les structs, y compris les octets et les chaînes, voir *L'emplacement des données et le comportement d'affectation* pour plus de détails.

Dans l'exemple ci-dessous, l'appel à `g(x)` n'a aucun effet sur `x` parce qu'il crée une copie indépendante de la valeur de stockage en mémoire. Cependant, `h(x)` modifie avec succès `x` car seule une référence et non une copie est transmise.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract C {
    uint[20] x;

    function f() public {
        g(x);
        h(x);
    }

    function g(uint[20] memory y) internal pure {
        y[2] = 3;
    }

    function h(uint[20] storage y) internal {
        y[3] = 4;
    }
}
```

3.8.6 Champ d'application et déclarations

Une variable qui est déclarée aura une valeur initiale par défaut dont la représentation en octets est constituée de zéros. Les « valeurs par défaut » des variables sont l'« état zéro » typique de leur type. Par exemple, la valeur par défaut d'un `bool` est `false`. La valeur par défaut des types `uint` ou `int` est `0`. Pour les tableaux de taille statique et les types `bytes1` à `bytes32`, chaque élément sera initialisé à la valeur par défaut correspondant à son à son type. Pour les tableaux de taille dynamique, les octets et `string`, la valeur par défaut est un tableau ou une chaîne vide. Pour le type `enum`, la valeur par défaut est son premier membre.

Le scoping dans Solidity suit les règles de scoping répandues de C99 (et de nombreux autres langages) : Les variables sont visibles à partir du point juste après leur déclaration jusqu'à la fin du plus petit bloc `{ }` qui contient la déclaration. Les variables déclarées dans la partie d'initialisation d'une boucle `for` font exception à cette partie d'initialisation d'une boucle `for` ne sont visibles que jusqu'à la fin de la boucle `for`.

Les variables qui sont des paramètres (paramètres de fonction, paramètres de modificateur, paramètres de capture, ...) sont visibles à l'intérieur du bloc de code qui suit - le corps de la fonction/modificateur pour un paramètre de fonction et de modificateur et le bloc `catch` pour un paramètre `catch`.

Les variables et autres éléments déclarés en dehors d'un bloc de code, par exemple les fonctions, les contrats, les types définis par l'utilisateur, etc., sont visibles avant même d'avoir été déclarés. Cela signifie que vous pouvez utiliser des variables d'état avant qu'elles ne soient déclarées et appeler des fonctions de manière récursive.

En conséquence, les exemples suivants compileront sans avertissement, puisque les deux variables ont le même nom mais des portées disjointes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
contract C {
    function minimalScoping() pure public {
        {
            uint same;
            same = 1;
        }

        {
            uint same;
            same = 3;
        }
    }
}
```

Comme exemple spécial des règles de scoping de C99, notez que dans ce qui suit, la première affectation à `x` va en fait affecter la variable externe et non la variable interne. Dans tous les cas, vous obtiendrez un avertissement sur le fait que la variable externe est cachée.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
// Ceci signalera un avertissement
contract C {
    function f() pure public returns (uint) {
        uint x = 1;
        {
            x = 2; // ceci sera assigné à la variable externe
            uint x;
        }
        return x; // x a la valeur 2
    }
}
```

Avertissement : Avant la version 0.5.0, Solidity suivait les mêmes règles de portée que le langage JavaScript, c'est-à-dire qu'une variable déclarée n'importe où dans une fonction avait une portée pour l'ensemble de la fonction, indépendamment de l'endroit où elle était déclarée. L'exemple suivant montre un extrait de code qui utilisait pour compiler mais qui conduit à une erreur à partir de la version 0.5.0.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
// Cela ne compilerera pas
contract C {
    function f() pure public returns (uint) {
        x = 2;
        uint x;
        return x;
    }
}
```

3.8.7 Arithmétique vérifiée ou non vérifiée

Un débordement ou un sous-débordement est la situation où la valeur résultante d'une opération arithmétique, lorsqu'elle est exécutée sur un entier non limité, tombe en dehors de la plage du type de résultat.

Avant la version 0.8.0 de Solidity, les opérations arithmétiques s'emballaient toujours en cas de débordement ou de sous-débordement, ce qui a conduit à l'utilisation répandue de bibliothèques qui vérifiaient des vérifications supplémentaires.

Depuis la version 0.8.0 de Solidity, toutes les opérations arithmétiques s'inversent par défaut en cas de dépassement inférieur ou supérieur, rendant ainsi inutile l'utilisation de ces bibliothèques.

Pour obtenir le comportement précédent, un bloc `unchecked` peut être utilisé :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract C {
    function f(uint a, uint b) pure public returns (uint) {
        // Cette soustraction se terminera par un dépassement de capacité.
        unchecked { return a - b; }
    }
    function g(uint a, uint b) pure public returns (uint) {
        // Cette soustraction s'inversera en cas de dépassement de capacité.
        return a - b;
    }
}
```

L'appel à `f(2, 3)` retournera $2^{256}-1$, alors que `g(2, 3)` provoquera une assertion qui échoue.

Le bloc « non vérifié » peut être utilisé partout à l'intérieur d'un bloc, mais pas en remplacement pour un bloc. Il ne peut pas non plus être imbriqué.

Le paramètre `n` affecte que les déclarations qui sont syntaxiquement à l'intérieur du bloc. Les fonctions appelées à l'intérieur d'un bloc « non vérifié » n'héritent pas de cette propriété.

Note : Pour éviter toute ambiguïté, vous ne pouvez pas utiliser `_`; à l'intérieur d'un bloc non vérifié.

Les opérateurs suivants provoqueront une assertion d'échec en cas de débordement ou de sous-débordement et s'enrouleront sans erreur s'ils sont utilisés à l'intérieur d'un bloc non vérifié :

`++`, `--`, `+`, binaire `-`, unaire `-`, `*`, `/`, `%`, `**`

`+=`, `-=`, `*=`, `/=`, `%=`

Avertissement : Il n'est pas possible de désactiver la vérification de la division par zéro ou modulo par zéro en utilisant le bloc `unchecked`.

Note : Les opérateurs binaires n'effectuent pas de vérification de dépassement de capacité ou de sous-dépassement. Ceci est particulièrement visible lors de l'utilisation de décalages binaires (`<<`, `>>`, `<=`, `>=`) à la place de la division d'entiers et de la multiplication par une puissance de 2. Par exemple, `type(uint256).max << 3` ne s'inverse pas alors que `type(uint256).max * 8` le ferait.

Note : La deuxième instruction dans `int x = type(int).min ; -x`; entraînera un dépassement de capacité car

l'intervalle négatif peut contenir une valeur de plus que l'intervalle positif.

Les conversions de type explicites seront toujours tronquées et ne provoqueront jamais une assertion d'échec à l'exception de la conversion d'un entier en un type enum.

3.8.8 Gestion des erreurs : Assert, Require, Revert et Exceptions

Solidity utilise des exceptions de retour à l'état initial pour gérer les erreurs. Une telle exception annule toutes les modifications apportées à l'état dans l'appel actuel (et tous ses sous-appels) et signale une erreur à l'appelant.

Lorsque des exceptions se produisent dans un sous-appel, elles « remontent » (c'est-à-dire que les exceptions sont rejetées) automatiquement à moins qu'elles ne soient capturées dans une instruction `try/catch`. Les exceptions à cette règle sont `send` et les fonctions de bas niveau `call`, `delegatecall` et `staticcall` : elles retournent `false` comme première valeur de retour en cas d'une exception, au lieu de « bouillonner ».

Avertissement : Les fonctions de bas niveau `call`, `delegatecall` et `staticcall` retournent `true` comme première valeur de retour si le compte appelé est inexistant, ce qui fait partie de la conception de l'EVM. L'existence du compte doit être vérifiée avant l'appel si nécessaire.

Les exceptions peuvent contenir des données d'erreur qui sont renvoyées à l'appelant sous la forme de *error instances*. Les erreurs intégrées « Erreur(string) » et « Panique(uint256) » sont utilisées par des fonctions spéciales, comme expliqué ci-dessous. `Error` est utilisé pour les conditions d'erreurs « normales ». Tandis que `Panic` est utilisé pour les erreurs qui ne devraient pas être présentes dans un code sans bogues.

Panique via « Assert » et erreur via « Require ».

Les fonctions pratiques `assert` et `require` peuvent être utilisées pour vérifier les conditions et lancer une exception si la condition n'est pas remplie.

La fonction `assert` crée une erreur de type `Panic(uint256)`. La même erreur est créée par le compilateur dans certaines situations, comme indiqué ci-dessous.

`Assert` ne doit être utilisée que pour tester les erreurs internes et pour vérifier les invariants. Un code qui fonctionne correctement ne devrait jamais créer un `Panic`, même pas sur une entrée externe invalide. Si cela se produit, alors il y a un bogue dans votre contrat que vous devez corriger. Les outils d'analyse du langage peuvent évaluer votre contrat pour identifier les conditions et les appels de fonction qui provoquent une panique.

Une exception de panique est générée dans les situations suivantes. Le code d'erreur fourni avec les données d'erreur indique le type de panique.

1. `0x00` : Utilisé pour les paniques génériques insérées par le compilateur.
2. `0x01` : Si vous appelez `assert` avec un argument qui évalue à `false`.
3. `0x11` : Si une opération arithmétique résulte en un débordement ou un sous-débordement en dehors d'un bloc « non vérifié { ... } ».
4. `0x12` : Si vous divisez ou modulez par zéro (par exemple, `5 / 0` ou `23 % 0`).
5. `0x21` : Si vous convertissez une valeur trop grande ou négative en un type d'enum.
6. `0x22` : Si vous accédez à un tableau d'octets de stockage qui est incorrectement codé.
7. `0x31` : Si vous appelez `.pop()` sur un tableau vide.
8. `0x32` : Si vous accédez à un tableau, à `bytesN` ou à une tranche de tableau à un index hors limites ou négatif (c'est-à-dire `x[i]` où `i >= x.length` ou `i < 0`).
9. `0x41` : Si vous allouez trop de mémoire ou créez un tableau trop grand.

10. 0x51 : Si vous appelez une variable zéro initialisée de type fonction interne.

La fonction `require` crée soit une erreur sans aucune donnée, soit une erreur de type `Error(string)`. Elle doit être utilisée pour garantir des conditions valides qui ne peuvent pas être détectées avant le moment de l'exécution. Cela inclut les conditions sur les entrées ou les valeurs de retour des appels à des contrats externes.

Note : Il n'est actuellement pas possible d'utiliser des erreurs personnalisées en combinaison avec `require`. Veuillez utiliser `if (!condition) revert CustomError();` à la place.

Une exception `Error(string)` (ou une exception sans données) est générée par le compilateur dans les situations suivantes :

1. Appeler `require(x)` où `x` est évalué à `false`.
2. Si vous utilisez `revert()` ou `revert("description")`.
3. Si vous effectuez un appel de fonction externe ciblant un contrat qui ne contient pas de code.
4. Si votre contrat reçoit de l'Ether via une fonction publique sans modificateur `payable` (y compris le constructeur et la fonction de repli).
5. Si votre contrat reçoit de l'Ether via une fonction publique `getter`.

Dans les cas suivants, les données d'erreur de l'appel externe (s'il est fourni) sont transférées. Cela signifie qu'il peut soit causer une *Error* ou une *Panic* (ou toute autre donnée) :

1. Si un `.transfer()` échoue.
2. Si vous appelez une fonction via un appel de message mais qu'elle ne se termine pas correctement (c'est-à-dire qu'elle tombe en panne sèche, qu'il n'y a pas de `throw` elle-même une exception), sauf lorsqu'une opération de bas niveau `call`, `send`, `delegatecall`, `callcode` ou `staticcall` est utilisé. Les opérations de bas niveau ne lèvent jamais d'exceptions mais indiquent les échecs en retournant `false`.
3. Si vous créez un contrat en utilisant le mot-clé `new` mais que le contrat création *ne se termine pas correctement*.

Vous pouvez éventuellement fournir une chaîne de message pour `require`, mais pas pour `assert`.

Note : Si vous ne fournissez pas un argument de type chaîne à `require`, il se retournera avec des données d'erreur vides, sans même inclure le sélecteur d'erreur.

L'exemple suivant montre comment vous pouvez utiliser `require` pour vérifier les conditions sur les entrées et `assert` pour vérifier les erreurs internes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract Sharer {
    function sendHalf(address payable addr) public payable returns (uint balance) {
        require(msg.value % 2 == 0, "Even value required.");
        uint balanceBeforeTransfer = address(this).balance;
        addr.transfer(msg.value / 2);
        // Puisque le transfert lève une exception en cas d'échec et que
        // ne peut pas rappeler ici, il ne devrait pas y avoir de moyen pour nous
        // d'avoir encore la moitié de l'argent.
        assert(address(this).balance == balanceBeforeTransfer - msg.value / 2);
        return address(this).balance;
    }
}
```

En interne, Solidity effectue une opération de retour en arrière (instruction `0xfd`). Cela provoque l'EVM à revenir sur toutes les modifications apportées à l'état. La raison de ce retour en arrière est qu'il n'y a pas de moyen sûr de poursuivre l'exécution, parce qu'un effet attendu ne s'est pas produit. Parce que nous voulons conserver l'atomicité des transactions, l'action la plus sûre est d'annuler tous les changements et de rendre la transaction entière (ou au moins l'appel) sans effet.

Dans les deux cas, l'appelant peut réagir à de tels échecs en utilisant `try/catch`, mais les changements dans l'appelant seront toujours annulés.

Note : Les exceptions de panique utilisaient l'opcode `invalid` avant Solidity 0.8.0, qui consommait tout le gaz disponible pour l'appel. Les exceptions qui utilisent `require` consommaient tout le gaz jusqu'à la version Metropolis.

revert

Une réversion directe peut être déclenchée à l'aide de l'instruction `revert` et de la fonction `revert`.

L'instruction `revert` prend une erreur personnalisée comme argument direct sans parenthèses :

```
revert CustomError(arg1, arg2);
```

Pour des raisons de rétrocompatibilité, il existe également la fonction `revert()`, qui utilise des parenthèses et accepte une chaîne de caractères :

```
revert(); revert(« description »);
```

Les données d'erreur seront renvoyées à l'appelant et pourront être capturées à cet endroit. L'utilisation de `revert()` provoque un `revert` sans aucune donnée d'erreur alors que `revert("description")` créera une erreur `Error(string)`.

L'utilisation d'une instance d'erreur personnalisée sera généralement beaucoup plus économique qu'une description sous forme de chaîne, car vous pouvez utiliser le nom de l'erreur pour la décrire, qui est encodé dans seulement quatre octets. Une description plus longue peut être fournie via `NatSpec`, ce qui n'entraîne aucun coût.

L'exemple suivant montre comment utiliser une chaîne d'erreur et une instance d'erreur personnalisée avec `revert` et l'équivalent `require` :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract VendingMachine {
    address owner;
    error Unauthorized();
    function buy(uint amount) public payable {
        if (amount > msg.value / 2 ether)
            revert("Not enough Ether provided.");
        // Autre façon de faire :
        require(
            amount <= msg.value / 2 ether,
            "Not enough Ether provided."
        );
        // Effectuer l'achat.
    }
    function withdraw() public {
        if (msg.sender != owner)
            revert Unauthorized();
    }
}
```

(suite sur la page suivante)

(suite de la page précédente)

```

    payable(msg.sender).transfer(address(this).balance);
  }
}

```

Les deux façons de faire `si (!condition) revert(...);` et `require(condition, ...);` sont équivalentes tant que les arguments de `revert` et `require` n'ont pas d'effets secondaires, par exemple si ce ne sont que des chaînes de caractères.

Note : La fonction `require` est évaluée comme n'importe quelle autre fonction. Cela signifie que tous les arguments sont évalués avant que la fonction elle-même ne soit exécutée. En particulier, dans `require(condition, f())` la fonction `f` est exécutée même si `condition` est vraie.

La chaîne fournie est *abi-encoded* comme s'il s'agissait d'un appel à une fonction `Error(string)`. Dans l'exemple ci-dessus, `revert("Not enough Ether provided.");` renvoie l'hexadécimal suivant comme données de retour d'erreur :

```

0x08c379a0 // Sélecteur de
↳ fonction pour Error(string)
0x000000000000000000000000000000000000000000000000000000000000000020 // Décalage des
↳ données
0x00000000000000000000000000000000000000000000000000000000000000001a // Longueur de la
↳ chaîne
0x4e6f7420656e6f7567682045746865722070726f76696465642e000000000000 // Données en chaîne

```

Le message fourni peut être récupéré par l'appelant à l'aide de `try/catch` comme indiqué ci-dessous.

Note : Il existait auparavant un mot-clé appelé « `throw` » avec la même sémantique que « `reverse()` », qui a été déprécié dans la version 0.4.13 et supprimé dans la version 0.5.0.

try/catch

Il existait auparavant un mot-clé appelé « `throw` » avec la même sémantique que `reverse()`.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;

interface DataFeed { function getData(address token) external returns (uint value); }

contract FeedConsumer {
    DataFeed feed;
    uint errorCount;
    function rate(address token) public returns (uint value, bool success) {
        // Désactiver définitivement le mécanisme s'il y a
        // plus de 10 erreurs.
        require(errorCount < 10);
        try feed.getData(token) returns (uint v) {
            return (v, true);
        } catch Error(string memory /*reason*/) {

```

(suite sur la page suivante)

```

    // Ceci est exécuté dans le cas où
    // le revert a été appelé dans getData
    // et qu'une chaîne de raison a été fournie.
    errorCount++;
    return (0, false);
} catch Panic(uint /*errorCode*/) {
    // Ceci est exécuté en cas de panique,
    // c'est-à-dire une erreur grave comme une division par zéro
    // ou un dépassement de capacité. Le code d'erreur peut être utilisé
    // pour déterminer le type d'erreur.
    errorCount++;
    return (0, false);
} catch (bytes memory /*lowLevelData*/) {
    // Ceci est exécuté au cas où revert() a été utilisé.
    errorCount++;
    return (0, false);
}
}
}

```

Le mot-clé `try` doit être suivi d'une expression représentant un appel de fonction externe ou une création de contrat (`new ContractName()`). Les erreurs à l'intérieur de l'expression ne sont pas prises en compte (par exemple s'il s'agit d'une expression complexe qui implique aussi des appels de fonctions internes), seul un retour en arrière se produisant dans l'appel externe lui-même. La partie `returns` (qui est optionnelle) qui suit déclare des variables de retour correspondant aux types retournés par l'appel externe. Dans le cas où il n'y a pas eu d'erreur ces variables sont assignées et l'exécution du contrat continue à l'intérieur du premier bloc de succès. Si la fin du bloc de succès est atteinte, l'exécution continue après les blocs `catch`.

Solidity prend en charge différents types de blocs `catch` en fonction du type d'erreur :

- `catch Error(string memory reason) { ... }` : Cette clause `catch` est exécutée si l'erreur a été provoquée par `revert("reasonString")` ou par `require(false, "reasonString")` (ou une erreur interne qui provoque une telle exception).
- `catch Panic(uint errorCode) { ... }` : Si l'erreur a été causée par une panique, c'est-à-dire par un `assert` défaillant, division par zéro, un accès invalide à un tableau, un débordement arithmétique et autres, cette clause `catch` sera exécutée.
- `catch (bytes memory lowLevelData) { ... }` : Cette clause est exécutée si la signature de l'erreur signature d'erreur ne correspond à aucune autre clause, s'il y a eu une erreur lors du décodage du message d'erreur, ou si aucune donnée d'erreur n'a été fournie avec l'exception. La variable déclarée donne accès aux données d'erreur de bas niveau dans ce cas.
- `catch { ... }` : Si vous n'êtes pas intéressé par les données d'erreur, vous pouvez simplement utiliser `catch { ... }` (même comme seule clause `catch`) au lieu de la clause précédente.

Il est prévu de supporter d'autres types de données d'erreur dans le futur. Les chaînes « Erreur » et « Panique » sont actuellement analysées telles quelles et ne sont pas traitées comme des identifiants.

Afin d'attraper tous les cas d'erreur, vous devez avoir au moins la clause suivante `catch { ... }` ou la clause `catch (bytes memory lowLevelData) { ... }`.

Les variables déclarées dans la clause `returns` et la clause `catch` sont uniquement dans le bloc qui suit.

Note : Si une erreur se produit pendant le décodage des données de retour dans un énoncé `try/catch`, cela provoque une exception dans le contrat en cours d'exécution et, pour cette raison, elle n'est pas attrapée dans la clause `catch`. S'il y a une erreur pendant le décodage de `catch Error(string memory reason)` et qu'il existe une clause `catch` de bas niveau, cette erreur y est attrapée.

Note : Si l'exécution atteint un bloc de capture, alors les effets de changement d'état de l'appel externe ont été annulés. Si l'exécution atteint le bloc de succès, les effets n'ont pas été annulés. Si les effets ont été inversés, alors l'exécution continue soit dans un bloc catch ou bien l'exécution de l'instruction try/catch elle-même s'inverse (par exemple, en raison d'échecs de décodage comme indiqué ci-dessus ou en raison de l'absence d'une clause catch de bas niveau).

Note : Les raisons de l'échec d'un appel peuvent être multiples. Ne supposez pas que le message d'erreur provient directement du contrat appelé : L'erreur peut s'être produite plus bas dans la chaîne d'appels et le contrat appelé n'a fait que la transmettre. De même, elle peut être due à une situation de panne sèche et non d'une condition d'erreur délibérée : L'appelant conserve toujours au moins 1/64ème du gaz dans un appel et donc l'appelant a encore du gaz.

3.9 Contrats

Les contrats dans Solidity sont similaires aux classes dans les langages orientés objet. Ils contiennent des données persistantes dans des variables d'état, et des fonctions qui peuvent modifier ces variables. L'appel d'une fonction sur un contrat (instance) différent va effectuer un appel de fonction EVM et donc un changement de contexte de telle sorte que les variables d'état dans le contrat appelant sont inaccessibles. Un contrat et ses fonctions doivent être appelés pour que quelque chose se produise. Il n'y a pas de concept de « cron » dans Ethereum pour appeler une fonction à un événement particulier automatiquement.

3.9.1 Création de contrats

Les contrats peuvent être créés « de l'extérieur » via des transactions Ethereum ou à partir de contrats Solidity.

Des IDE, tels que [Remix](#), rendent le processus de création transparent à l'aide d'éléments d'interface utilisateur.

Une façon de créer des contrats de façon programmatique sur Ethereum est via l'API JavaScript [web3.js](#). Elle dispose d'une fonction appelée `web3.eth.Contract` pour faciliter la création de contrats.

Lorsqu'un contrat est créé, son *constructeur* (une fonction déclarée avec la fonction le mot-clé `constructor`) est exécutée une fois.

Un constructeur est facultatif. Un seul constructeur est autorisé, ce qui signifie que la surcharge n'est pas supportée.

Après l'exécution du constructeur, le code final du contrat est stocké sur la blockchain. Ce code comprend toutes les fonctions publiques et externes ainsi que toutes les fonctions qui sont accessibles à partir de là par des appels de fonction. Le code déployé n'inclut pas le code du constructeur ou les fonctions internes appelées uniquement depuis le constructeur.

En interne, les arguments des constructeurs sont passés *ABI encodé* après le code du contrat lui-même, mais vous n'avez pas à vous en soucier si vous utilisez `web3.js`.

Si un contrat souhaite créer un autre contrat, le code source (et le binaire) du contrat créé doit être connu du créateur. Cela signifie que les dépendances cycliques de création sont impossibles.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract OwnedToken {
    // `TokenCreator` est un type de contrat qui est défini ci-dessous.
    // Il est possible d'y faire référence tant qu'il n'est pas utilisé
```

(suite sur la page suivante)

```
// pour créer un nouveau contrat.
TokenCreator creator;
address owner;
bytes32 name;

// Il s'agit du constructeur qui enregistre le
// créateur et le nom attribué.
constructor(bytes32 _name) {
    // Les variables d'état sont accessibles via leur nom
    // et non pas via, par exemple, `this.owner`. Les fonctions peuvent
    // être accédées directement ou via `this.f`,
    // mais ce dernier fournit une vue externe
    // à la fonction. En particulier dans le constructeur,
    // vous ne devriez pas accéder aux fonctions de manière externe,
    // car la fonction n'existe pas encore.
    // Voir la section suivante pour plus de détails.
    owner = msg.sender;

    // Nous effectuons une conversion de type explicite de `address`
    // vers `TokenCreator` et nous supposons que le type de
    // contrat appelant est `TokenCreator`, mais il n'existe
    // aucun moyen réel de le vérifier.
    // Cette opération ne crée pas de nouveau contrat.
    creator = TokenCreator(msg.sender);
    name = _name;
}

function changeName(bytes32 newName) public {
    // Seul le créateur peut modifier le nom.
    // Nous comparons le contrat en fonction de son
    // adresse qui peut être récupérée par
    // conversion explicite en adresse.
    if (msg.sender == address(creator))
        name = newName;
}

function transfer(address newOwner) public {
    // Seul le propriétaire actuel peut transférer le jeton.
    if (msg.sender != owner) return;

    // Nous demandons au contrat de création si le transfert
    // doit avoir lieu en utilisant une fonction du
    // contrat `TokenCreator` défini ci-dessous. Si
    // l'appel échoue (par exemple à cause d'une panne sèche),
    // l'exécution échoue également ici.
    if (creator.isTokenTransferOK(owner, newOwner))
        owner = newOwner;
}
}

contract TokenCreator {
```

(suite sur la page suivante)

(suite de la page précédente)

```

function createToken(bytes32 name)
    public
    returns (OwnedToken tokenAddress)
{
    // Crée un nouveau contrat `Token` et retourne son adresse.
    // Du côté de JavaScript, le type de retour
    // de cette fonction est `address`, puisque c'est
    // le type le plus proche disponible dans l'ABI.
    return new OwnedToken(name);
}

function changeName(OwnedToken tokenAddress, bytes32 name) public {
    // Encore une fois, le type externe de `tokenAddress` est
    // simplement `address`.
    tokenAddress.changeName(name);
}

// Effectuer des vérifications pour déterminer si le transfert d'un jeton vers
// le contrat `OwnedToken` doit être effectué.
function isTokenTransferOK(address currentOwner, address newOwner)
    public
    pure
    returns (bool ok)
{
    // Vérifier une condition arbitraire pour voir si le transfert doit avoir lieu.
    return keccak256(abi.encodePacked(currentOwner, newOwner))[0] == 0x7f;
}
}

```

3.9.2 Visibilité et Getters

Solidity connaît deux types d'appels de fonction : internes qui ne créent pas d'appel EVM réel (également appelé « appel de message ») et les externes qui le font. Pour cette raison, il existe quatre types de visibilité pour les fonctions et les variables d'état.

Les fonctions doivent être spécifiées comme étant `external`, `public`, `internal` ou `private`. Pour les variables d'état, `external` n'est pas possible.

external Les fonctions externes font partie de l'interface du contrat, ce qui signifie qu'elles peuvent être appelées depuis d'autres contrats et via des transactions. Une fonction externe `f` ne peut pas être appelée en interne (c'est-à-dire que `f()` ne fonctionne pas, mais `this.f()` fonctionne).

public Les fonctions publiques font partie de l'interface du contrat et peuvent être appelées soit en interne, soit via des messages. Pour les variables d'état publiques, une fonction getter automatique (voir ci-dessous) est générée.

internal Ces fonctions et variables d'état ne peuvent être accessibles qu'en interne (c'est à dire depuis le contrat en cours ou des contrats qui en dérivent), sans utiliser `this`. C'est le niveau de visibilité par défaut des variables d'état.

private Les fonctions privées et les variables d'état ne sont visibles que pour le contrat dans lequel elles sont définies et non dans des contrats dérivés.

Note : Tout ce qui est à l'intérieur d'un contrat est visible pour tous les observateurs externes à la blockchain. Rendre quelque chose `private` empêche seulement les autres contrats de lire ou de modifier l'information, mais elle sera

toujours visible pour le monde entier en dehors de la blockchain.

Le spécificateur de visibilité est donné après le type pour les variables d'état et entre la liste des paramètres et la liste de paramètres de retour pour les fonctions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

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

Dans l'exemple suivant, D, peut appeler `c.setData()` pour récupérer la valeur de `data` dans le stockage d'état, mais ne peut pas appeler `f`. Le contrat E est dérivé du contrat C et peut donc appeler `compute`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    uint private data;

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

// Cela ne compilera pas
contract D {
    function readData() public {
        C c = new C();
        uint local = c.f(7); // erreur : le membre `f` n'est pas visible
        c.setData(3);
        local = c.getData();
        local = c.compute(3, 5); // erreur : le membre `compute` n'est pas visible
    }
}

contract E is C {
    function g() public {
        C c = new C();
        uint val = compute(3, 5); // accès au membre interne (du contrat dérivé au
↳ contrat parent)
    }
}
```

Fonctions Getter

Le compilateur crée automatiquement des fonctions getter pour toutes les variables d'état **publiques**. Pour le contrat donné ci-dessous, le compilateur générera une fonction appelée `data` qui ne prend aucun arguments et retourne un `uint`, la valeur de la variable d'état `data`. Les variables d'état peuvent être initialisées lorsqu'elles sont déclarées.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    uint public data = 42;
}

contract Caller {
    C c = new C();
    function f() public view returns (uint) {
        return c.data();
    }
}
```

Les fonctions getter ont une visibilité externe. Si le symbole est accédé en interne (c'est-à-dire sans `this.`), il est évalué comme une variable d'état. S'il est accédé en externe (c'est-à-dire avec `this.`), il est évalué comme une fonction.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract C {
    uint public data;
    function x() public returns (uint) {
        data = 3; // accès interne
        return this.data(); // accès externe
    }
}
```

Si vous avez une variable d'état public de type tableau, alors vous pouvez seulement récupérer les éléments uniques du tableau via la fonction getter générée. Ce mécanisme existe pour éviter des coûts de gaz élevés lors du retour d'un tableau entier. Vous pouvez utiliser pour spécifier l'élément individuel à retourner, par exemple `myArray(0)`. Si vous voulez retourner un tableau entier en un seul appel, vous devez alors écrire une fonction, par exemple :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract arrayExample {
    // variable d'état publique
    uint[] public myArray;

    // Fonction Getter générée par le compilateur
    /*
    function myArray(uint i) public view returns (uint) {
        return myArray[i];
    }
    */

    // fonction qui retourne le tableau entier
```

(suite sur la page suivante)

(suite de la page précédente)

```

function getArray() public view returns (uint[] memory) {
    return myArray;
}

```

Maintenant vous pouvez utiliser `getArray()` pour récupérer le tableau entier, au lieu de `myArray(i)`, qui retourne un seul élément par appel.

L'exemple suivant est plus complexe :

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Complex {
    struct Data {
        uint a;
        bytes3 b;
        mapping (uint => uint) map;
        uint[3] c;
        uint[] d;
        bytes e;
    }
    mapping (uint => mapping(bool => Data[])) public data;
}

```

Il génère une fonction de la forme suivante. Le mappage et les tableaux (à l'exception des tableaux d'octets) dans la structure sont omis parce qu'il n'y a pas de bonne façon de sélectionner les membres individuels de la structure ou de fournir une clé pour le mappage :

```

function data(uint arg1, bool arg2, uint arg3)
    public
    returns (uint a, bytes3 b, bytes memory e)
{
    a = data[arg1][arg2][arg3].a;
    b = data[arg1][arg2][arg3].b;
    e = data[arg1][arg2][arg3].e;
}

```

3.9.3 Modificateurs de fonction

Les modificateurs peuvent être utilisés pour changer le comportement des fonctions de manière déclarative. Par exemple, vous pouvez utiliser un modificateur pour vérifier automatiquement une condition avant d'exécuter la fonction.

Les modificateurs sont des propriétés héritables des contrats et peuvent être remplacées par des contrats dérivés, mais uniquement s'ils sont marqués `virtual`. Pour plus de détails, veuillez consulter [Modifier Overriding](#).

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

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

```

(suite sur la page suivante)

(suite de la page précédente)

```

// Ce contrat définit uniquement un modificateur mais ne l'utilise pas.
// mais ne l'utilise pas : il sera utilisé dans les contrats dérivés.
// Le corps de la fonction est inséré là où apparaît le symbole spécial
// `_;` dans la définition d'un modificateur.
// Cela signifie que si le propriétaire appelle cette fonction,
// la fonction est exécutée et sinon, une exception est
// levée.
modifieur onlyOwner {
    require(
        msg.sender == owner,
        "Seul le propriétaire peut appeler cette fonction."
    );
    -;
}

contract destructible is owned {
    // Ce contrat hérite du modificateur `onlyOwner` de la fonction
    // `owned` et l'applique à la fonction `destroy`, qui
    // fait que les appels à `destroy` n'ont d'effet que si
    // ils sont effectués par le propriétaire stocké.
    function destroy() public onlyOwner {
        selfdestruct(owner);
    }
}

contract priced {
    // Les modificateurs peuvent recevoir des arguments :
    modifieur costs(uint price) {
        if (msg.value >= price) {
            -;
        }
    }
}

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

    constructor(uint initialPrice) { price = initialPrice; }

    // Il est important de fournir également
    // le mot-clé `payable` ici, sinon la fonction
    // rejetera automatiquement tout l'Ether qui lui sera envoyé.
    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }

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

```

(suite sur la page suivante)

```

}

contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(
            !locked,
            "Reentrant call."
        );
        locked = true;
        _;
        locked = false;
    }

    /// Cette fonction est protégée par un mutex, ce qui signifie que
    /// les appels réentrants provenant de `msg.sender.call` ne peuvent pas appeler `f` à
    ↪ nouveau.
    /// L'instruction `return 7` attribue 7 à la valeur de retour mais
    /// exécute l'instruction `locked = false` dans le modificateur.
    function f() public noReentrancy returns (uint) {
        (bool success,) = msg.sender.call("");
        require(success);
        return 7;
    }
}

```

Si vous voulez accéder à un modificateur `m` défini dans un contrat `C`, vous pouvez utiliser `C.m` pour le le référencer sans recherche virtuelle. Il est seulement possible d'utiliser les modificateurs définis dans le contrat actuel ou ses contrats de base. Les modificateurs peuvent aussi être définis dans des bibliothèques, mais leur utilisation est limitée aux fonctions de la même bibliothèque.

Plusieurs modificateurs sont appliqués à une fonction en les spécifiant dans une séparée par des espaces et sont évaluées dans l'ordre présenté.

Les modificateurs ne peuvent pas accéder ou modifier implicitement les arguments et les valeurs de retour des fonctions qu'ils modifient. Leurs valeurs ne peuvent leur être transmises que de manière explicite au moment de l'invocation.

Les retours explicites d'un modificateur ou d'un corps de fonction ne quittent que le modificateur ou du corps de la fonction actuelle. Les variables de retour sont assignées et le flux de contrôle continue après le `_` du modificateur précédent.

Avertissement : Dans une version antérieure de Solidity, les instructions `return` dans les fonctions ayant des modificateurs se comportaient différemment.

Un retour explicite d'un modificateur avec `return`; n'affecte pas les valeurs retournées par la fonction. Le modificateur peut toutefois choisir de ne pas exécuter du tout le corps de la fonction et, dans ce cas, les variables `return` sont placées à leur valeur par défaut comme si la fonction avait un corps vide.

Le symbole `_` peut apparaître plusieurs fois dans le modificateur. Chaque occurrence est remplacée par le corps de la fonction.

Les expressions arbitraires sont autorisées pour les arguments du modificateur et dans ce contexte, tous les symboles visibles de la fonction sont visibles dans le modificateur. Les symboles introduits dans le modificateur ne sont pas visibles dans la fonction (car ils pourraient être modifiés par la surcharge).

3.9.4 Variables d'état constantes et immuables

Les variables d'état peuvent être déclarées comme `constant` ou `immutable`. Dans les deux cas, les variables ne peuvent pas être modifiées après la construction du contrat. Pour les variables `constant`, la valeur doit être fixée à la compilation, alors que pour les variables `immutable`, elle peut encore être assignée au moment de la construction.

Il est également possible de définir des variables `constant` au niveau du fichier.

Le compilateur ne réserve pas d'emplacement pour ces variables, et chaque occurrence est remplacée par la valeur correspondante.

Comparé aux variables d'état régulières, les coûts de gaz des variables constantes et immuables sont beaucoup plus faibles. Pour une variable constante, l'expression qui lui est assignée est copiée à tous les endroits où elle est accédée et est également réévaluée à chaque fois. Cela permet des optimisations locales. Les variables immuables sont évaluées une seule fois au moment de la construction et leur valeur est copiée à tous les endroits du code où elles sont accédées. Pour ces valeurs, 32 octets sont réservés, même si elles pourraient tenir dans moins d'octets. Pour cette raison, les valeurs constantes peuvent parfois être moins chères que les valeurs immuables.

Tous les types de constantes et d'immuables ne sont pas encore implémentés. Les seuls types supportés sont *strings* (uniquement pour les constantes) et *value types*.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.4;

uint constant X = 32**22 + 8;

contract C {
    string constant TEXT = "abc";
    bytes32 constant MY_HASH = keccak256("abc");
    uint immutable decimals;
    uint immutable maxBalance;
    address immutable owner = msg.sender;

    constructor(uint _decimals, address _reference) {
        decimals = _decimals;
        // Les affectations aux immuables peuvent même accéder à l'environnement.
        maxBalance = _reference.balance;
    }

    function isBalanceTooHigh(address _other) public view returns (bool) {
        return _other.balance > maxBalance;
    }
}
```

Constant

Pour les variables `constant`, la valeur doit être une constante au moment de la compilation et elle doit être assignée à l'endroit où la variable est déclarée. Toute expression qui accède au stockage, aux données de la blockchain (par exemple, `block.timestamp`, `address(this).balance` ou `block.number`) ou aux données d'exécution (`msg.value` ou `gasleft()`) ou fait des appels à des contrats externes est interdit. Les expressions qui pourraient avoir un effet secondaire sur l'allocation de mémoire sont autorisées, mais celles qui pourraient avoir un effet secondaire sur d'autres objets mémoire ne le sont pas. Les fonctions intégrées `keccak256`, `sha256`, `ripemd160`, `ecrecover`, `addmod` et `mulmod` sont autorisées (même si, à l'exception de `keccak256`, ils appellent des contrats externes).

La raison pour laquelle les effets secondaires sur l'allocateur de mémoire sont autorisés est qu'il devrait être possible

de construire des objets complexes comme par exemple des tables de consultation. Cette fonctionnalité n'est pas encore totalement utilisable.

Immutable

Les variables déclarées comme `immutable` sont un peu moins restreintes que celles déclarées comme `constant` : Les variables immuables peuvent se voir attribuer une valeur arbitraire dans le constructeur du contrat ou au moment de leur déclaration. Elles ne peuvent être assignées qu'une seule fois et peuvent, à partir de ce moment, être lues même pendant la construction.

Le code de création du contrat généré par le compilateur modifiera le code d'exécution du contrat avant qu'il ne soit retourné en remplaçant toutes les références aux immutables par les valeurs qui leur sont attribuées. Ceci est important si vous comparez le code d'exécution généré par le compilateur avec celui réellement stocké dans la blockchain.

Note : Les immutables qui sont affectés lors de leur déclaration ne sont considérés comme initialisés que lorsque le constructeur du contrat s'exécute. Cela signifie que vous ne pouvez pas initialiser les immutables en ligne avec une valeur qui dépend d'un autre immuable. Vous pouvez cependant le faire à l'intérieur du constructeur du contrat.

Il s'agit d'une protection contre les différentes interprétations concernant l'ordre de l'initialisation des variables d'état et de l'exécution du constructeur, en particulier en ce qui concerne l'héritage.

3.9.5 Fonctions

Les fonctions peuvent être définies à l'intérieur et à l'extérieur des contrats.

Les fonctions hors contrat, aussi appelées « fonctions libres », ont toujours une valeur implicite `internal`. *visibilité* implicite. Leur code est inclus dans tous les contrats qui les appellent, comme pour les fonctions internes des bibliothèques.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

function sum(uint[] memory _arr) pure returns (uint s) {
    for (uint i = 0; i < _arr.length; i++)
        s += _arr[i];
}

contract ArrayExample {
    bool found;
    function f(uint[] memory _arr) public {
        // Cela appelle la fonction free en interne.
        // Le compilateur ajoutera son code au contrat.
        uint s = sum(_arr);
        require(s >= 10);
        found = true;
    }
}
```

Note : Les fonctions définies en dehors d'un contrat sont toujours exécutées dans le contexte d'un contrat. Elles ont toujours accès à la variable `this`, peuvent appeler d'autres contrats, leur envoyer de l'Ether et détruire le contrat qui

les a appelées, entre autres choses. La principale différence avec les fonctions définies à l'intérieur d'un contrat est que les fonctions libres n'ont pas d'accès direct aux variables de stockage et aux fonctions qui ne sont pas dans leur portée.

Paramètres des fonctions et variables de retour

Les fonctions prennent des paramètres typés en entrée et peuvent, contrairement à beaucoup d'autres langages, renvoyer un nombre arbitraire de valeurs en sortie.

Paramètres des fonctions

Les paramètres de fonction sont déclarés de la même manière que les variables, et le nom des paramètres non utilisés peuvent être omis.

Par exemple, si vous voulez que votre contrat accepte un type d'appel externe avec deux entiers, vous utiliserez quelque chose comme ce qui suit :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Simple {
    uint sum;
    function taker(uint _a, uint _b) public {
        sum = _a + _b;
    }
}
```

Les paramètres de fonction peuvent être utilisés comme n'importe quelle autre variable locale et ils peuvent également être affectés.

Note : Une *fonction externe* ne peut pas accepter un tableau multidimensionnel comme paramètre d'entrée. Cette fonctionnalité est possible si vous activez le codeur ABI v2 en ajoutant `pragma abicoder v2;` à votre fichier source.

Une *fonction interne* peut accepter un tableau multidimensionnel sans activer la fonction.

Variables de retour

Les variables de retour de fonction sont déclarées avec la même syntaxe après le mot-clé `returns`.

Par exemple, supposons que vous vouliez renvoyer deux résultats : la somme et le produit de deux entiers passés comme paramètres de la fonction, vous utiliserez quelque chose comme :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Simple {
    function arithmetic(uint _a, uint _b)
        public
        pure
        returns (uint o_sum, uint o_product)
    {
```

(suite sur la page suivante)

```
    o_sum = _a + _b;
    o_product = _a * _b;
  }
}
```

Les noms des variables de retour peuvent être omis. Les variables de retour peuvent être utilisées comme toute autre variable locales et sont initialisées avec leur *valeur par défaut* et ont cette valeur jusqu'à ce qu'elles soient (ré)assignées.

Vous pouvez soit assigner explicitement aux variables de retour et ensuite laisser la fonction comme ci-dessus, ou vous pouvez fournir des valeurs de retour (soit une seule, soit *multiple ones*) directement avec l'instruction `return`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Simple {
    function arithmetic(uint _a, uint _b)
        public
        pure
        returns (uint o_sum, uint o_product)
    {
        return (_a + _b, _a * _b);
    }
}
```

Si vous utilisez un `return` précoce pour quitter une fonction qui a des variables de retour, vous devez fournir des valeurs de retour avec l'instruction `return`.

Note : Vous ne pouvez pas retourner certains types à partir de fonctions non internes, notamment les tableaux dynamiques multidimensionnels et les structs. Si vous activez le ABI coder v2 en ajoutant `pragma abicoder v2;` à votre fichier source, alors plus de types sont disponibles, mais les types `mapping` sont toujours limités à l'intérieur d'un seul contrat et vous ne pouvez pas les transférer.

Renvoi de valeurs multiples

Lorsqu'une fonction possède plusieurs types de retour, l'instruction `return (v0, v1, ..., vn)` peut être utilisée pour retourner plusieurs valeurs. Le nombre de composants doit être le même que le nombre de variables de retour et leurs types doivent correspondre, éventuellement après une conversion *implicite*.

Mutabilité de l'État

Voir les fonctions

Les fonctions peuvent être déclarées vues, auquel cas elles promettent de ne pas modifier l'état.

Note : Si la cible EVM du compilateur est Byzantium ou plus récente (par défaut), l'opcode `STATICCALL` est utilisé lorsque les fonctions `view` sont appelées, ce qui impose à l'état de rester non modifié dans le cadre de l'exécution de l'EVM. Pour les fonctions de bibliothèque `view`, `DELEGATECALL` est utilisé, car il n'existe pas de combinaison de `DELEGATECALL` et de `STATICCALL`. Cela signifie que les fonctions de la bibliothèque `view` n'ont pas de contrôles d'exécution qui empêchent les états. Ceci ne devrait pas avoir d'impact négatif sur la sécurité car le code de la bibliothèque

est généralement connu au moment de la compilation et le vérificateur statique effectue des vérifications au moment de la compilation.

Les instructions suivantes sont considérées comme modifiant l'état :

1. Écriture dans les variables d'état
2. *Émettre des événements*
3. *Créer d'autres contrats*
4. Utiliser `selfdestruct`
5. Envoyer de l'Ether via des appels
6. Appeler une fonction qui n'est pas marquée `view` ou `pure`
7. Utiliser des appels de bas niveau
8. Utilisation d'un assemblage en ligne contenant certains opcodes

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    function f(uint a, uint b) public view returns (uint) {
        return a * (b + 42) + block.timestamp;
    }
}
```

Note : `constant` sur les fonctions était un alias de `view`, mais cela a été abandonné dans la version 0.5.0.

Note : Les méthodes Getter sont automatiquement marquées `view`.

Note : Avant la version 0.5.0, le compilateur n'utilisait pas l'opcode `STATICCALL` pour les fonctions `view`. Cela permettait des modifications d'état dans les fonctions `view` par l'utilisation de conversions de types explicites invalides. En utilisant `STATICCALL` pour les fonctions `view`, les modifications de l'état sont empêchées au niveau de l'EVM.

Fonctions pures

Les fonctions peuvent être déclarées `pure`, auquel cas elles promettent de ne pas lire ou modifier l'état. En particulier, il devrait être possible d'évaluer une fonction `pure` à la compilation seulement ses entrées et `msg.data`, mais sans aucune connaissance de l'état actuel de la blockchain. Cela signifie que la lecture de variables `immutable` peut être une opération non `pure`.

Note : Si la cible EVM du compilateur est Byzantium ou plus récente (par défaut), l'opcode `STATICCALL` est utilisé, ce qui ne garantit pas que l'état ne soit pas lu, mais au moins qu'il ne soit pas modifié.

En plus de la liste des instructions modifiant l'état expliquée ci-dessus, les suivantes sont considérées comme lisant l'état :

1. Lecture des variables d'état
2. Accès à `adresse(this).balance` ou `<adresse>.balance`

3. Accéder à l'un des membres de `block`, `tx`, `msg` (à l'exception de `msg.sig` et `msg.data`)
4. L'appel de toute fonction non marquée `pure`
5. L'utilisation d'un assemblage en ligne qui contient certains opcodes

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

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

Les fonctions pures sont en mesure d'utiliser les fonctions `revert()` et `require()` pour revenir sur des changements d'état potentiels lorsqu'une *erreur se produit*.

Revenir en arrière sur un changement d'état n'est pas considéré comme une « modification d'état », car seuls les changements d'état effectués précédemment dans du code qui n'avait pas la restriction `view` ou `pure` sont inversés et ce code a la possibilité d'attraper le `revert` et de ne pas le transmettre.

Ce comportement est également en accord avec l'opcode `STATICCALL`.

Avertissement : Il est impossible d'empêcher les fonctions de lire l'état au niveau de l'EVM, il est seulement possible de les empêcher d'écrire dans l'état (c'est-à-dire que seul `view` peut être imposé au niveau de l'EVM, `pure` ne peut pas).

Note : Avant la version 0.5.0, le compilateur n'utilisait pas l'opcode `STATICCALL` pour les programmes `pure`. Ceci permettait des modifications d'état dans les fonctions `pure` par l'utilisation de conversions de types explicites invalides. En utilisant `STATICCALL` pour les fonctions `pure`, les modifications de l'état sont empêchées au niveau de l'EVM.

Note : Avant la version 0.4.17, le compilateur n'imposait pas que `pure` ne lise pas l'état. C'est un contrôle de type à la compilation, qui peut être contourné en faisant des conversions explicites invalides entre les types de contrat, car le compilateur peut vérifier que le type du contrat ne fait pas d'opérations de changement d'état, mais il ne peut pas vérifier que le contrat qui sera appelé au moment de l'exécution est effectivement de ce type.

Fonctions spéciales

Fonction de réception d'Ether

Un contrat peut avoir au maximum une fonction `receive`, déclarée à l'aide des éléments suivants `receive()` `external payable { ... }` (sans le mot-clé `function`). Cette fonction ne peut pas avoir d'arguments, ne peut rien retourner et doit avoir une visibilité `external` et une mutabilité de l'état `payable`. Elle peut être virtuelle, peut être surchargée et peut avoir des modificateurs.

La fonction de réception est exécutée lors d'un appel au contrat avec des données d'appel vides. C'est la fonction qui est exécutée lors des transferts d'Ether (par exemple via `.send()` ou `.transfer()`). Si cette fonction n'existe pas, mais qu'une fonction `payable de repli` existe, la fonction de repli sera appelée lors d'un transfert d'Ether simple. Si aucune fonction de réception d'Ether ni aucune fonction de repli payable n'est présente, le contrat ne peut pas recevoir d'Ether par le biais de transactions normales et lance une exception.

Dans le pire des cas, la fonction `receive` ne peut compter que sur le fait que 2300 gaz soient disponible (par exemple lorsque `send` ou `transfer` est utilisé), ce qui laisse peu de place pour effectuer d'autres opérations que la journalisation de base. Les opérations suivantes consommeront plus de gaz que l'allocation de 2300 gaz :

- Écriture dans le stockage
- Création d'un contrat
- Appeler une fonction externe qui consomme une grande quantité de gaz
- Envoi d'éther

Avertissement : Les contrats qui reçoivent de l'Ether directement (sans appel de fonction, c'est-à-dire en utilisant `send` ou `transfer`) mais qui ne définissent pas de fonction de réception d'Ether ou de fonction de repli payable, lancer une exception en renvoyant l'Ether (ceci était différent avant Solidity v0.4.0). Donc si vous voulez que votre contrat reçoive de l'Ether, vous devez implémenter une fonction de réception d'Ether (l'utilisation de fonctions de repli payantes pour recevoir de l'éther n'est pas recommandée, car elle n'échouerait pas en cas de confusion d'interface).

Avertissement : Un contrat sans fonction de réception d'Ether peut recevoir de l'Ether en tant que destinataire d'une transaction *coinbase* (c.à.d *récompense de bloc miner*) ou en tant que destination d'une *selfdestruct*.

Un contrat ne peut pas réagir à de tels transferts d'Ether et ne peut donc pas les rejeter. Il s'agit d'un choix de conception de l'EVM, Solidity ne peut pas le contourner.

Cela signifie également que `address(this).balance` peut être plus élevé que la somme d'une comptabilité manuelle implémentée dans un contrat (par exemple, en ayant un compteur mis à jour dans la fonction de réception d'Ether).

Ci-dessous vous pouvez voir un exemple d'un contrat Sink qui utilise la fonction `receive`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// Ce contrat garde tout l'Ether qui lui est envoyé sans aucun moyen
// de le récupérer.
contract Sink {
    event Received(address, uint);
    receive() external payable {
        emit Received(msg.sender, msg.value);
    }
}
```

Fonction de repli

Un contrat peut avoir au maximum une fonction `fallback`, déclarée en utilisant soit `fallback () external [payable]`, soit `fallback (bytes calldata _input) external [payable] returns (bytes memory _output)` (dans les deux cas sans le mot-clé `function`). Cette fonction doit avoir une visibilité `external`. Une fonction de repli peut être virtuelle, peut remplacer et peut avoir des modificateurs.

La fonction de repli est exécutée lors d'un appel au contrat si aucune des autres fonction ne correspond à la signature de la fonction donnée, ou si aucune donnée n'est fournie et qu'il n'existe pas de *fonction de réception d'éther*. La fonction de repli reçoit toujours des données, mais pour recevoir également de l'Ether elle doit être marquée `payable`.

Si la version avec paramètres est utilisée, `_input` contiendra les données complètes envoyées au contrat (égal à `msg.data`) et peut retourner des données dans `_output`. Les données retournées ne seront pas codées par l'ABI. Au lieu de

cela, elles seront retournées sans modifications (même pas de remplissage).

Dans le pire des cas, si une fonction de repli payable est également utilisée à la place d'une fonction de réception, elle ne peut compter que sur le gaz 2300 disponible (voir *fonction de réception d'éther* pour une brève description des implications de ceci).

Comme toute fonction, la fonction de repli peut exécuter des opérations complexes tant qu'il y a suffisamment de gaz qui lui est transmis.

Avertissement : Une fonction de repli payable est également exécutée pour les transferts d'Ether simples, si aucune fonction de réception d'Ether n'est présente. Il est recommandé de toujours définir une fonction de réception Ether de réception, si vous définissez une fonction de repli payable afin de distinguer les transferts Ether des confusions d'interface.

Note : Si vous voulez décoder les données d'entrée, vous pouvez vérifier les quatre premiers octets pour le sélecteur de fonction et ensuite vous pouvez utiliser `abi.decode` avec la syntaxe array slice pour décoder les données codées par ABI : `(c, d) = abi.decode(_input[4 :], (uint256, uint256));` Notez que cette méthode ne doit être utilisée qu'en dernier recours, et que les fonctions appropriées doivent être utilisées à la place.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

contract Test {
    uint x;
    // Cette fonction est appelée pour tous les messages envoyés à
    // ce contrat (il n'y a pas d'autre fonction).
    // L'envoi d'Ether à ce contrat provoquera une exception,
    // car la fonction de repli n'a pas le modificateur `payable`.
    fallback() external { x = 1; }
}

contract TestPayable {
    uint x;
    uint y;
    // Cette fonction est appelée pour tous les messages envoyés à
    // ce contrat, sauf les transferts Ether simples
    // (il n'y a pas d'autre fonction que la fonction de réception).
    // Tout appel à ce contrat avec des calldata non vides exécutera
    // la fonction de repli (même si Ether est envoyé avec l'appel).
    fallback() external payable { x = 1; y = msg.value; }

    // Cette fonction est appelée pour les transferts Ether simples, c'est à dire
    // pour chaque appel avec des données d'appel vides.
    receive() external payable { x = 2; y = msg.value; }
}

contract Caller {
    function callTest(Test test) public returns (bool) {
        (bool success,) = address(test).call(abi.encodeWithSignature(
↪ "nonExistingFunction()"));
        require(success);
    }
}
```

(suite sur la page suivante)

(suite de la page précédente)

```

    // il en résulte que test.x devient == 1.

    // address(test) ne permettra pas d'appeler directement `send`, puisque `test` n
    ↪ a pas de payable
    // fonction de repli.
    // Il doit être converti en adresse payable pour pouvoir appeler `send`.
    address payable testPayable = payable(address(test));

    // Si quelqu'un envoie de l'Ether à ce contrat,
    // le transfert échouera, c'est à dire que cela renvoie false ici.
    return testPayable.send(2 ether);
}

function callTestPayable(TestPayable test) public returns (bool) {
    (bool success,) = address(test).call(abi.encodeWithSignature(
    ↪ "nonExistingFunction()"));
    require(success);
    // le résultat est que test.x devient == 1 et test.y devient 0.
    (success,) = address(test).call{value: 1}(abi.encodeWithSignature(
    ↪ "nonExistingFunction()"));
    require(success);
    // le résultat est que test.x devient == 1 et test.y devient 1.

    // Si quelqu'un envoie de l'Ether à ce contrat, la fonction de réception de
    ↪ TestPayable sera appelée.
    // Comme cette fonction écrit dans le stockage, elle prend plus d'éther que ce
    ↪ qui est disponible avec un
    // simple `send` ou `transfer`. Pour cette raison, nous devons utiliser un
    ↪ appel de bas niveau.
    (success,) = address(test).call{value: 2 ether}("");
    require(success);
    // le résultat est que test.x devient == 2 et test.y devient 2 ether.

    return true;
}
}

```

Surcharge des fonctions

Un contrat peut avoir plusieurs fonctions du même nom mais avec des types de paramètres différents. Ce processus est appelé « surcharge » et s'applique également aux fonctions héritées. L'exemple suivant montre la surcharge de la fonction `f` dans la portée du contrat `A`.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract A {
    function f(uint _in) public pure returns (uint out) {
        out = _in;
    }
}

```

(suite sur la page suivante)

(suite de la page précédente)

```
function f(uint _in, bool _really) public pure returns (uint out) {
    if (_really)
        out = _in;
}
```

Les fonctions surchargées sont également présentes dans l'interface externe. C'est une erreur si deux fonctions visibles de l'extérieur diffèrent par leurs types Solidity mais pas par leurs types externes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

// This will not compile
contract A {
    function f(B _in) public pure returns (B out) {
        out = _in;
    }

    function f(address _in) public pure returns (address out) {
        out = _in;
    }
}

contract B {
}
```

Les deux surcharges de fonction `f` ci-dessus finissent par accepter le type d'adresse pour l'ABI bien qu'ils ils sont considérés comme différents dans Solidity.

Résolution des surcharges et correspondance des arguments

Les fonctions surchargées sont sélectionnées en faisant correspondre les déclarations de fonction dans la portée actuelle aux arguments fournis dans l'appel de fonction. Les fonctions sont sélectionnées comme candidates à la surcharge si tous les arguments peuvent être implicitement convertis dans les types attendus. S'il n'y a pas exactement un candidat, la résolution échoue.

Note : Les paramètres de retour ne sont pas pris en compte pour la résolution des surcharges.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract A {
    function f(uint8 _in) public pure returns (uint8 out) {
        out = _in;
    }

    function f(uint256 _in) public pure returns (uint256 out) {
        out = _in;
    }
}
```

Appeler `f(50)` créerait une erreur de type puisque `50` peut être implicitement converti à la fois en types `uint8` et `uint256`. D'un autre côté, `f(256)` se résoudrait en une surcharge `f(uint256)` puisque `256` ne peut pas être implicitement converti en `uint8`.

3.9.6 Événements

Les événements Solidity offrent une abstraction au-dessus de la fonctionnalité de journalisation de l'EVM. Les applications peuvent s'abonner et écouter ces événements via l'interface RPC d'un client Ethereum.

Les événements sont des membres héritables des contrats. Lorsque vous les appelez, ils font en sorte que les arguments dans le journal de la transaction, une structure de données spéciale dans la blockchain. Ces journaux sont associés à l'adresse du contrat, sont incorporés dans la blockchain, et y restent aussi longtemps qu'un bloc est accessible (pour toujours à partir de maintenant, mais cela pourrait changer avec Serenity). Le journal et ses données d'événement ne sont pas accessibles à partir des contrats (même pas depuis le contrat qui les a créés).

Il est possible de demander une preuve Merkle pour les journaux. Si une entité externe fournit une telle preuve à un contrat, celui-ci peut vérifier que le journal existe réellement dans la blockchain. Vous devez fournir des en-têtes de bloc car le contrat ne peut voir que les 256 derniers hachages de blocs.

Vous pouvez ajouter l'attribut `indexed` à un maximum de trois paramètres qui les ajoutent à une structure de données spéciale appelée «*topics*» au lieu de la partie données du journal. Un topic ne peut contenir qu'un seul mot (32 octets), donc si vous utilisez un *type de référence* pour un argument indexé, le hachage Keccak-256 de la valeur est stocké comme un sujet à la place.

Tous les paramètres sans l'attribut `indexed` sont *ABI-encodés* dans la partie données du journal.

Les sujets vous permettent de rechercher des événements, par exemple en filtrant une séquence de blocs pour certains événements. Vous pouvez également filtrer les événements en fonction de l'adresse du contrat qui a émis l'événement.

Par exemple, le code ci-dessous utilise le contrat `web3.js subscribe("logs")`. La *méthode* pour filtrer les journaux qui correspondent à un sujet avec une certaine valeur d'adresse :

```
var options = {
  fromBlock: 0,
  address: web3.eth.defaultAccount,
  topics: ["0x0000000000000000000000000000000000000000000000000000000000000000", null,
↪null]
};
web3.eth.subscribe('logs', options, function (error, result) {
  if (!error)
    console.log(result);
})
.on("data", function (log) {
  console.log(log);
})
.on("changed", function (log) {
});
```

Le hachage de la signature de l'événement est l'un des sujets, sauf si vous avez déclaré l'événement avec le spécificateur `anonymous`. Cela signifie qu'il n'est pas possible de filtrer les événements anonymes spécifiques par nom, vous pouvez seulement filtrer par l'adresse du contrat. L'avantage des événements anonymes est qu'ils sont moins chers à déployer et à appeler. Ils vous permettent également de déclarer quatre arguments indexés au lieu de trois.

Note : Comme le journal des transactions ne stocke que les données de l'événement et non le type, vous devez connaître le type de l'événement, y compris le paramètre qui est indexé et si l'événement est anonyme afin d'interpréter correc-

tement les données. En particulier, il est possible de « falsifier » la signature d'un autre événement en utilisant un événement anonyme.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.21 <0.9.0;

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

    function deposit(bytes32 _id) public payable {
        // Les événements sont émis en utilisant `emit`, suivi par
        // le nom de l'événement et les arguments
        // (le cas échéant) entre parenthèses. Toute invocation de ce type
        // (même profondément imbriquée) peut être détectée à partir de
        // l'API JavaScript en filtrant pour `Deposit`.
        emit Deposit(msg.sender, _id, msg.value);
    }
}
```

L'utilisation dans l'API JavaScript est la suivante :

```
var abi = /* abi tel que généré par le compilateur */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at("0x1234...ab67" /* address */);

var depositEvent = clientReceipt.Deposit();

// surveiller les changements
depositEvent.watch(function(error, result){
    // le résultat contient des arguments non indexés et des sujets
    // donnés à l'appel `Deposit`.
    if (!error)
        console.log(result);
});

// Ou passez un callback pour commencer à regarder immédiatement.
var depositEvent = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result);
});
```

Le résultat de l'opération ci-dessus ressemble à ce qui suit (découpé) :

```
{
  "returnValues": {
    "_from": "0x1111...FFFCCCC",
    "_id": "0x50...sd5adb20",
    "_value": "0x420042"
```

(suite sur la page suivante)

(suite de la page précédente)

```

},
"raw": {
  "data": "0x7f...91385",
  "topics": ["0xfd4...b4ead7", "0x7f...1a91385"]
}
}

```

Ressources supplémentaires pour comprendre les événements

- Documentation Javascript
- Exemple d'utilisation des événements
- Comment y accéder en js

3.9.7 Les erreurs et la déclaration de retour en arrière

Les erreurs dans Solidity fournissent un moyen pratique et efficace d'expliquer à l'utilisateur pourquoi une opération a échoué. Elles peuvent être définies à l'intérieur et à l'extérieur des contrats (y compris les interfaces et les bibliothèques).

Elles doivent être utilisées conjointement avec l'instruction *revert* qui provoque toutes les modifications de l'appel en cours et renvoie les données d'erreur à l'appelant.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

/// Solde insuffisant pour le transfert. Nécessaire `required` mais seulement
/// `available` disponible.
/// @param available disponible disponible.
/// @param required montant demandé pour le transfert.
error InsufficientBalance(uint256 available, uint256 required);

contract TestToken {
  mapping(address => uint) balance;
  function transfer(address to, uint256 amount) public {
    if (amount > balance[msg.sender])
      revert InsufficientBalance({
        available: balance[msg.sender],
        required: amount
      });
    balance[msg.sender] -= amount;
    balance[to] += amount;
  }
  // ...
}

```

Les erreurs ne peuvent pas être surchargées ou remplacées mais sont héritées. La même erreur peut être définie à plusieurs endroits, à condition que les champs d'application soient distincts. Les instances d'erreurs ne peuvent être créées qu'en utilisant les instructions *revert*.

L'erreur crée des données qui sont ensuite transmises à l'appelant avec l'opération *revert*, afin de retourner au composant hors chaîne ou de l'attraper dans une instruction *try/catch*. Notez qu'une erreur ne peut être attrapée que si elle provient d'un appel externe, les retours se produisant dans des appels internes ou à l'intérieur de la même fonction ne peuvent pas être attrapés.

Si vous ne fournissez pas de paramètres, l'erreur ne nécessite que quatre octets de données et vous pouvez utiliser *NatSpec* comme ci-dessus pour expliquer plus en détail les raisons de l'erreur, qui ne sont pas stockées dans la chaîne. Cela en fait une fonctionnalité de signalement d'erreur très bon marché et pratique à la fois.

Plus précisément, une instance d'erreur est codée par ABI de la même manière que un appel à une fonction du même nom et du même type le serait et est ensuite utilisé comme données de retour dans l'opcode `revert`. Cela signifie que les données consistent en un sélecteur de 4 octets suivi de données *ABI-encodées*. Le sélecteur est constitué des quatre premiers octets du keccak256-hash de la signature du type d'erreur.

Note : Il est possible qu'un contrat soit révoqué avec des erreurs différentes du même nom ou même avec des erreurs définies à des endroits différents qui sont indiscernables par l'appelant. Pour l'extérieur, c'est-à-dire l'ABI, seul le nom de l'erreur est pertinent, pas le contrat ou le fichier où elle est définie.

L'instruction `require(condition, "description");` serait équivalente à `if (!condition) revert Error("description")` si vous pouviez définir `error Error(string)`. Notez cependant que `Error` est un type intégré et ne peut être défini dans un code fourni par l'utilisateur.

De même, un échec de `assert` ou des conditions similaires se retourneront avec une erreur du type intégré `Panic(uint256)`.

Note : Les données d'erreur ne doivent être utilisées que pour donner une indication de l'échec, mais pas comme un moyen pour le flux de contrôle. La raison en est que les données de retour des appels internes sont propagées en retour dans la chaîne des appels externes par défaut. Cela signifie qu'un appel interne peut « forger » des données de retour qui semblent pouvoir provenir du contrat qui l'a appelé.

3.9.8 Héritage

Solidity prend en charge l'héritage multiple, y compris le polymorphisme.

Le polymorphisme signifie qu'un appel de fonction (interne et externe) exécute toujours la fonction du même nom (et des types de paramètres) dans le contrat le plus dérivé de la hiérarchie d'héritage. Ceci doit être explicitement activé sur chaque fonction de la hiérarchie en utilisant les mots-clés `virtual` et `override`. Voir *Remplacement de fonctions* pour plus de détails.

Il est possible d'appeler des fonctions plus haut dans la hiérarchie d'héritage en interne, en spécifiant explicitement le contrat en utilisant `ContractName.functionName()` ou en utilisant `super.functionName()` si vous souhaitez appeler la fonction à un niveau supérieur dans la hiérarchie d'héritage aplatie (voir ci-dessous).

Lorsqu'un contrat hérite d'autres contrats, un seul contrat unique est créé sur la blockchain, et le code de tous les contrats de base est compilé dans le contrat créé. Cela signifie que tous les appels internes aux fonctions des contrats de base utilisent également des appels de fonctions internes (`super.f(...)`) utilisera `JUMP` et non un appel de message).

Le shadowing de variables d'état est considéré comme une erreur. Un contrat dérivé peut seulement déclarer une variable d'état `x`, s'il n'y a pas de variable d'état visible avec le même nom dans l'une de ses bases.

Le système d'héritage général est très similaire à celui de Python <<https://docs.python.org/3/tutorial/classes.html#inheritance>>, surtout en ce qui concerne l'héritage multiple, mais il y a aussi quelques *différences*.

Les détails sont donnés dans l'exemple suivant.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
```

(suite sur la page suivante)

(suite de la page précédente)

```

contract Owned {
    constructor() { owner = payable(msg.sender); }
    address payable owner;
}

// Utilisez `s` pour dériver d'un autre contrat.
// Les contrats dérivés peuvent accéder à tous les membres non privés, y compris
// les fonctions internes et les variables d'état. Ceux-ci ne peuvent pas être
// accessibles en externe via `this`.
contract Destructible is Owned {
    // Le mot clé `virtual` signifie que la fonction peut modifier
    // son comportement dans les classes dérivées ("overriding").
    function destroy() virtual public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

// Ces contrats abstraits ne sont fournis que pour faire connaître
// l'interface au compilateur. Notez la fonction
// sans corps. Si un contrat n'implémente pas toutes les
// fonctions, il ne peut être utilisé que comme une interface.
abstract contract Config {
    function lookup(uint id) public virtual returns (address adr);
}

abstract contract NameReg {
    function register(bytes32 name) public virtual;
    function unregister() public virtual;
}

// L'héritage multiple est possible. Notez que `owned`
// est aussi une classe de base de `Destructible`, mais il n'y a qu'une seule instance de
↳ `owned`.
// Pourtant, il n'existe qu'une seule instance de `owned` (comme pour l'héritage virtuel en
↳ C++).
contract Named is Owned, Destructible {
    constructor(bytes32 name) {
        Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
        NameReg(config.lookup(1)).register(name);
    }

    // Les fonctions peuvent être remplacées par une autre fonction ayant le même nom et
    // le même nombre/types d'entrées. Si la fonction de remplacement a différents
    // types de paramètres de sortie différents, cela entraîne une erreur.
    // Les appels de fonction locaux et par message tiennent compte de ces surcharges.
    // Si vous voulez que la fonction soit prioritaire, vous devez utiliser le
    // mot-clé `override`. Vous devez à nouveau spécifier le mot-clé `virtual`
    // si vous voulez que cette fonction soit à nouveau surchargée.
}

```

(suite sur la page suivante)

```

function destroy() public virtual override {
    if (msg.sender == owner) {
        Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
        NameReg(config.lookup(1)).unregister();
        // Il est toujours possible d'appeler une
        // fonction spécifique surchargée.
        Destructible.destroy();
    }
}

// Si un constructeur prend un argument, il doit être
// fourni dans l'en-tête ou le modificateur-invocation-style à
// le constructeur du contrat dérivé (voir ci-dessous).
contract PriceFeed is Owned, Destructible, Named("GoldFeed") {
    function updateInfo(uint newInfo) public {
        if (msg.sender == owner) info = newInfo;
    }

    // Ici, nous ne spécifions que `override` et non `virtual`.
    // Cela signifie que les contrats dérivant de `PriceFeed`
    // ne peuvent plus modifier le comportement de `destroy`.
    function destroy() public override(Destructible, Named) { Named.destroy(); }
    function get() public view returns(uint r) { return info; }

    uint info;
}

```

Notez que ci-dessus, nous appelons `Destructible.destroy()` pour « faire suivre » la demande de destruction. La manière dont cela est fait est problématique, comme le montre l'exemple suivant :

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

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

contract Destructible is owned {
    function destroy() public virtual {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is Destructible {
    function destroy() public virtual override { /* do cleanup 1 */ Destructible.
->destroy(); }
}

contract Base2 is Destructible {

```

(suite sur la page suivante)

(suite de la page précédente)

```

    function destroy() public virtual override { /* do cleanup 2 */ Destructible.
    ↪destroy(); }
}

contract Final is Base1, Base2 {
    function destroy() public override(Base1, Base2) { Base2.destroy(); }
}

```

Un appel à `Final.destroy()` fera appel à `Base2.destroy` parce que nous le spécifions explicitement dans la surcharge finale, mais cette fonction contournera `Base1.destroy`. Le moyen de contourner ce problème est d'utiliser `super` :

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

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

contract Destructible is owned {
    function destroy() virtual public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is Destructible {
    function destroy() public virtual override { /* do cleanup 1 */ super.destroy(); }
}

contract Base2 is Destructible {
    function destroy() public virtual override { /* do cleanup 2 */ super.destroy(); }
}

contract Final is Base1, Base2 {
    function destroy() public override(Base1, Base2) { super.destroy(); }
}

```

Si `Base2` appelle une fonction de `super`, elle n'appelle pas simplement cette fonction sur l'un de ses contrats de base. Au contraire, elle appelle plutôt cette fonction sur le contrat de base suivant dans le d'héritage final, il appellera donc `Base1.destroy()` (notez que la séquence d'héritage finale est – en commençant par le contrat le plus dérivé : `Final`, `Base2`, `Base1`, `Destructible`, `owned`). La fonction réelle qui est appelée lors de l'utilisation de `super` est pas connue dans le contexte de la classe où elle est utilisée, bien que son type soit connu. Il en va de même pour la recherche ordinaire de méthode virtuelle ordinaire.

Remplacement des fonctions

Les fonctions de base peuvent être surchargées par les contrats hérités pour changer leur comportement si elles sont marquées comme `virtual`. La fonction de remplacement doit alors utiliser le mot-clé `override` dans l'en-tête de la fonction. La fonction de remplacement ne peut que changer la visibilité de la fonction de remplacement de `externe` à `public`. La mutabilité peut être changée en une mutabilité plus stricte en suivant l'ordre : `nonpayable` peut être remplacé par `view` et `pure`. `view` peut être remplacé par `pure`. `payable` est une exception et ne peut pas être changé en une autre mutabilité.

L'exemple suivant démontre la modification de la mutabilité et de la visibilité :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base
{
    function foo() virtual external view {}
}

contract Middle is Base {}

contract Inherited is Middle
{
    function foo() override public pure {}
}
```

Pour l'héritage multiple, les contrats de base les plus dérivés qui définissent la même doivent être spécifiés explicitement après le mot-clé `override`. En d'autres termes, vous devez spécifier tous les contrats de base qui définissent la même fonction et qui n'ont pas encore été remplacés par un autre contrat de base (sur un chemin quelconque du graphe d'héritage). De plus, si un contrat hérite de la même fonction à partir de plusieurs bases (sans lien), il doit explicitement la remplacer :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base1
{
    function foo() virtual public {}
}

contract Base2
{
    function foo() virtual public {}
}

contract Inherited is Base1, Base2
{
    // Dérive de plusieurs bases définissant foo(), nous devons donc explicitement
    // le surcharger
    function foo() public override(Base1, Base2) {}
}
```

Un spécificateur de surcharge explicite n'est pas nécessaire si la fonction est définie dans un contrat de base commun ou s'il existe une fonction unique dans un contrat de base commun qui prévaut déjà sur toutes les autres fonctions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract A { function f() public pure{} }
contract B is A {}
contract C is A {}
// Aucune surcharge explicite n'est requise
contract D is B, C {}
```

Plus formellement, il n'est pas nécessaire de surcharger une fonction (directement ou indirectement) héritée de bases multiples s'il existe un contrat de base qui fait partie de tous les chemins de surcharge pour la signature, et (1) cette base implémente la fonction et qu'aucun chemin depuis le contrat actuel vers la base ne mentionne une fonction avec cette signature ou (2) cette base n'implémente pas la fonction et il y a au plus une mention de la fonction dans tous les chemins allant du contrat actuel à cette base.

Dans ce sens, un chemin de surcharge pour une signature est un chemin à travers le graphe d'héritage qui commence au contrat considéré et se termine par un contrat mentionnant une fonction avec cette signature qui n'est pas surchargée.

Si vous n'indiquez pas qu'une fonction qui surcharge est `virtual`, les contrats dérivés ne peuvent plus modifier le comportement de cette fonction.

Note : Les fonctions ayant la visibilité `private` ne peuvent pas être `virtual`.

Note : Les fonctions sans implémentation doivent être marquées `virtual` en dehors des interfaces. Dans les interfaces, toutes les fonctions sont automatiquement considérées comme `virtual`.

Note : A partir de Solidity 0.8.8, le mot-clé `override` n'est pas nécessaire pour remplacer une fonction, au cas où la fonction est définie dans plusieurs bases.

Les variables d'état publiques peuvent remplacer les fonctions externes si les types de paramètres et de retour de la fonction correspondent à la fonction getter de la variable :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract A
{
    function f() external view virtual returns(uint) { return 5; }
}

contract B is A
{
    uint public override f;
}
```

Note : Si les variables d'état publiques peuvent remplacer les fonctions externes, elles ne peuvent pas elles-mêmes être surchargées.

Remplacement d'un modificateur

Les modificateurs de fonction peuvent se substituer les uns aux autres. Cela fonctionne de la même manière que la *superposition de fonctions* (sauf qu'il n'y a pas de surcharge pour les modificateurs). Le mot-clé `virtual` doit être utilisé sur le modificateur surchargé et le mot-clé `override` doit être utilisé dans le modificateur de surcharge :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base
{
    modifier foo() virtual {_;}
}

contract Inherited is Base
{
    modifier foo() override {_;}
}
```

En cas d'héritage multiple, tous les contrats de base directs doivent être spécifiés explicitement :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base1
{
    modifier foo() virtual {_;}
}

contract Base2
{
    modifier foo() virtual {_;}
}

contract Inherited is Base1, Base2
{
    modifier foo() override(Base1, Base2) {_;}
}
```

Constructeurs

Un constructeur est une fonction facultative déclarée avec le mot-clé `constructor` qui est exécutée lors de la création du contrat, et dans laquelle vous pouvez exécuter le code d'initialisation du contrat.

Avant que le code du constructeur ne soit exécuté, les variables d'état sont initialisées à leur valeur spécifiée si vous les initialisez en ligne, ou leur *valeur par défaut* si vous ne le faites pas.

Après l'exécution du constructeur, le code définitif du contrat est déployé sur la blockchain. Le déploiement du code coûte un gaz supplémentaire linéaire à la longueur du code. Ce code comprend toutes les fonctions qui font partie de l'interface publique et toutes les fonctions qui sont accessibles à partir de celle-ci par des appels de fonction. Il ne comprend pas le code du constructeur ni les fonctions internes qui ne sont appelées uniquement depuis le constructeur.

S'il n'y a pas de constructeur, le contrat prendra en charge le constructeur par défaut, qui est équivalent à `constructor() {}`. Par exemple :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

abstract contract A {
    uint public a;

    constructor(uint _a) {
        a = _a;
    }
}

contract B is A(1) {
    constructor() {}
}
```

Vous pouvez utiliser des paramètres internes dans un constructeur (par exemple des pointeurs de stockage). Dans ce cas, le contrat doit être marqué *abstract*, parce que ces paramètres ne peuvent pas se voir attribuer de valeurs valides de l'extérieur, mais uniquement par le biais des constructeurs des contrats dérivés.

Avertissement : Avant la version 0.4.22, les constructeurs étaient définis comme des fonctions portant le même nom que le contrat. Cette syntaxe a été dépréciée et n'est plus autorisée dans la version 0.5.0.

Avertissement : Avant la version 0.7.0, vous deviez spécifier la visibilité des constructeurs comme étant soit *internal* ou *public*.

Arguments pour les constructeurs de base

Les constructeurs de tous les contrats de base seront appelés en suivant les règles de linéarisation expliquées ci-dessous. Si les constructeurs de base ont des arguments, les contrats dérivés doivent tous les spécifier. Ceci peut être fait de deux manières :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

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

// Soit spécifier directement dans la liste d'héritage...
contract Derived1 is Base(7) {
    constructor() {}
}

// ou par un "modificateur" du constructeur dérivé.
contract Derived2 is Base {
    constructor(uint _y) Base(_y * _y) {}
}
```

L'une des façons est directement dans la liste d'héritage (`est Base(7)`). L'autre est dans la façon dont un modificateur est invoqué dans le cadre du constructeur dérivé (`Base(_y * _y)`). La première façon est plus pratique si l'argument du constructeur est une constante et définit le comportement du contrat ou le le décrit. La deuxième façon doit être utilisée si les arguments du constructeur de la base dépendent de ceux du contrat dérivé. Les arguments doivent être donnés soit dans la liste d'héritage ou dans le style modificateur dans le constructeur dérivé. Spécifier les arguments aux deux endroits est une erreur.

Si un contrat dérivé ne spécifie pas les arguments de tous les constructeurs de ses contrats de base, il sera considéré comme un contrat abstrait.

Héritage multiple et linéarisation

Les langages qui autorisent l'héritage multiple doivent faire face à plusieurs problèmes. L'un d'entre eux est le **problème du diamant**. Solidity est similaire à Python en ce qu'il utilise la « **C3 Linearization** » pour forcer un ordre spécifique dans le graphe acyclique dirigé (DAG) des classes de base. Cette propriété souhaitable de la monotonie, mais désapprouve certains graphes d'héritage. En particulier, l'ordre dans lequel les classes de base sont données dans la directive `s` est important : Vous devez lister les contrats de base directs dans l'ordre de « le plus similaire à la base » à « le plus dérivé ». Notez que cet ordre est l'inverse de celui utilisé en Python.

Une autre façon simplifiée d'expliquer ceci est que lorsqu'une fonction est appelée qui est définie plusieurs fois dans différents contrats, les bases données sont recherchées de droite à gauche (de gauche à droite en Python) de manière approfondie, s'arrêtant à la première correspondance. Si un contrat de base a déjà été recherché, il est ignoré.

Dans le code suivant, Solidity donnera l'erreur suivante erreur « Linearization of inheritance graph impossible » (« Linéarisation du graphe d'héritage impossible »).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract X {}
contract A is X {}
// Cela ne compilera pas
contract C is A, X {}
```

La raison en est que C demande à X de supplanter A (en spécifiant A, X dans cet ordre), mais A lui-même demande d'outrepasser X, ce qui est une contradiction qui ne peut être résolue.

En raison du fait que vous devez explicitement surcharger une fonction qui est héritée de plusieurs bases sans une surcharge unique, la linéarisation de C3 n'est pas trop importante en pratique.

Un domaine où la linéarisation de l'héritage est particulièrement importante et peut-être pas aussi claire est lorsqu'il y a plusieurs constructeurs dans la hiérarchie de l'héritage. Les constructeurs seront toujours exécutés dans l'ordre linéarisé, quel que soit l'ordre dans lequel leurs arguments sont fournis dans le constructeur du contrat hérité. Par exemple :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base1 {
    constructor() {}
}

contract Base2 {
    constructor() {}
}
```

(suite sur la page suivante)

(suite de la page précédente)

```

// Les constructeurs sont exécutés dans l'ordre suivant :
// 1 - Base1
// 2 - Base2
// 3 - Derived1
contract Derived1 is Base1, Base2 {
    constructor() Base1() Base2() {}
}

// Les constructeurs sont exécutés dans l'ordre suivant :
// 1 - Base2
// 2 - Base1
// 3 - Derived2
contract Derived2 is Base2, Base1 {
    constructor() Base2() Base1() {}
}

// Les constructeurs sont toujours exécutés dans l'ordre suivant :
// 1 - Base2
// 2 - Base1
// 3 - Derived3
contract Derived3 is Base2, Base1 {
    constructor() Base1() Base2() {}
}

```

Hériter de différents types de membres portant le même nom

C'est une erreur lorsque l'une des paires suivantes dans un contrat porte le même nom en raison de l'héritage :

- une fonction et un modificateur
- une fonction et un événement
- un événement et un modificateur

À titre d'exception, un getter de variable d'état peut remplacer une fonction externe.

3.9.9 Contrats abstraits

Les contrats doivent être marqués comme abstraits lorsqu'au moins une de leurs fonctions n'est pas implémentée. Les contrats peuvent être marqués comme abstraits même si toutes les fonctions sont implémentées.

Cela peut être fait en utilisant le mot-clé `abstract` comme le montre l'exemple suivant. Notez que ce contrat doit être défini comme abstrait, car la fonction `utterance()` a été définie, mais aucune implémentation n'a été fournie (aucun corps d'implémentation `{ }` n'a été donné).

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract Feline {
    function utterance() public virtual returns (bytes32);
}

```

Ces contrats abstraits ne peuvent pas être instanciés directement. Cela est également vrai si un contrat abstrait met en œuvre toutes les fonctions définies. L'utilisation d'un contrat abstrait comme classe de base est illustrée dans l'exemple suivant :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract Feline {
    function utterance() public pure virtual returns (bytes32);
}

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

Si un contrat hérite d'un contrat abstrait et qu'il n'implémente pas toutes les fonctions non implémentées en les surchargeant, il doit également être marqué comme abstrait.

Notez qu'une fonction sans implémentation est différente d'une *Fonction Type*, même si leur syntaxe est très similaire.

Exemple de fonction sans implémentation (une déclaration de fonction) :

```
function foo(address) external returns (address);
```

Exemple de déclaration d'une variable dont le type est un type de fonction :

```
function(address) external returns (address) foo;
```

Les contrats abstraits découplent la définition d'un contrat de son implémentation fournissant une meilleure extensibilité et auto-documentation et facilitant les modèles comme la méthode *Template* et supprimant la duplication du code. Les contrats abstraits sont utiles de la même façon que définir des méthodes dans une interface est utile. C'est un moyen pour le concepteur du contrat abstrait de dire « tout enfant de moi doit implémenter cette méthode ».

Note : Les contrats abstraits ne peuvent pas remplacer une fonction virtuelle implémentée par une fonction virtuelle non implémentée.

3.9.10 Interfaces

Les interfaces sont similaires aux contrats abstraits, mais aucune fonction ne peut y être implémentée. Il existe d'autres restrictions :

- Elles ne peuvent pas hériter d'autres contrats, mais elles peuvent hériter d'autres interfaces.
- Toutes les fonctions déclarées doivent être externes.
- Elles ne peuvent pas déclarer de constructeur.
- Elles ne peuvent pas déclarer de variables d'état.
- Elles ne peuvent pas déclarer de modificateurs.

Certaines de ces restrictions peuvent être levées à l'avenir.

Les interfaces sont fondamentalement limitées à ce que l'ABI du contrat peut représenter, et la conversion entre l'ABI et une interface devrait être possible sans aucune perte d'information.

Les interfaces sont désignées par leur propre mot-clé :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

interface Token {
    enum TokenType { Fungible, NonFungible }
    struct Coin { string obverse; string reverse; }
    function transfer(address recipient, uint amount) external;
}
```

Les contrats peuvent hériter d'interfaces comme ils le feraient pour d'autres contrats.

Toutes les fonctions déclarées dans les interfaces sont implicitement `virtual`, les fonctions qui les surchargent n'ont pas besoin du mot-clé `override`. Cela ne signifie pas automatiquement qu'une fonction surchargée peut être à nouveau surchargée. Cela n'est possible que si la fonction qui la surcharge est marquée `virtual`.

Les interfaces peuvent hériter d'autres interfaces. Les règles sont les mêmes que pour l'héritage normal.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

interface ParentA {
    function test() external returns (uint256);
}

interface ParentB {
    function test() external returns (uint256);
}

interface SubInterface is ParentA, ParentB {
    // Doit redéfinir test afin d'affirmer que les parents
    // sont compatibles.
    function test() external override(ParentA, ParentB) returns (uint256);
}
```

Les types définis dans les interfaces et autres structures de type contrat sont accessibles à partir d'autres contrats : `Token.TokenType` ou `Token.Coin`.

Avertissement : Les interfaces supportent les types `enum` depuis *Solidity version 0.5.0*, soyez sûr que le `pragma` version spécifie cette version au minimum.

3.9.11 Bibliothèques

Les bibliothèques sont similaires aux contrats, mais leur but est d'être déployées une seule fois à une adresse spécifique et leur code est réutilisé en utilisant le `DELEGATECALL` (`CALLCODE` jusqu'à Homestead) de l'EVM. Cela signifie que si des fonctions de bibliothèque sont appelées, leur code est exécuté dans le contexte du contrat d'appel, c'est-à-dire que `this` pointe vers le contrat appelant, et surtout le stockage du contrat appelant est accessible. Comme une bibliothèque est un morceau de code source isolé, elle ne peut accéder aux variables d'état du contrat d'appel que si elles sont explicitement fournies (elle n'aurait aucun moyen de les nommer, sinon). Les fonctions des bibliothèques ne peuvent être appelées directement (c'est-à-dire sans l'utilisation de `DELEGATECALL`) que si elles ne modifient pas l'état (c'est-à-dire si ce sont des fonctions `view` ou `pure`), parce que les bibliothèques sont supposées être sans état. En particulier, il n'est possible de détruire une bibliothèque.

Note : Jusqu'à la version 0.4.20, il était possible de détruire des bibliothèques en contournant le système de types de Solidity. A partir de cette version, les bibliothèques contiennent un *mécanisme* qui empêche les fonctions modifiant l'état d'être appelées directement (c'est-à-dire sans DELEGATECALL).

Les bibliothèques peuvent être vues comme des contrats de base implicites des contrats qui les utilisent. Elles ne seront pas explicitement visibles dans la hiérarchie de l'héritage, mais les appels aux fonctions des bibliothèques ressemblent aux appels aux fonctions des contrats de base explicites (en utilisant un accès qualifié comme `L.f()`). Bien sûr, les appels aux fonctions internes utilisent la convention d'appel interne, ce qui signifie que tous les types internes peuvent être passés et les types *stockés en mémoire* seront passés par référence et non copiés. Pour réaliser cela dans l'EVM, le code des fonctions de bibliothèques internes qui sont appelées à partir d'un contrat ainsi que toutes les fonctions appelées à partir de celui-ci seront incluses dans le contrat et un appel régulier JUMP sera utilisé au lieu d'un DELEGATECALL.

Note : L'analogie avec l'héritage s'effondre lorsqu'il s'agit de fonctions publiques. L'appel d'une fonction de bibliothèque publique avec `L.f()` entraîne un appel externe (DELEGATECALL pour être précis). En revanche, `A.f()` est un appel interne lorsque A est un contrat de base du contrat actuel.

L'exemple suivant illustre comment utiliser les bibliothèques (mais en utilisant une méthode manuelle, ne manquez pas de consulter *utiliser for* pour un exemple plus avancé pour implémenter un ensemble).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// Nous définissons un nouveau type de données struct qui sera utilisé pour
// contenir ses données dans le contrat d'appel.
struct Data {
    mapping(uint => bool) flags;
}

library Set {
    // Notez que le premier paramètre est de type
    // "référence de stockage" et donc seulement son adresse de stockage et pas
    // son contenu est transmis dans le cadre de l'appel. Il s'agit d'une
    // particularité des fonctions de bibliothèque. Il est idiomatique
    // d'appeler le premier paramètre `self` si la fonction peut
    // être vue comme une méthode de cet objet.
    function insert(Data storage self, uint value)
        public
        returns (bool)
    {
        if (self.flags[value])
            return false; // déjà là
        self.flags[value] = true;
        return true;
    }

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

(suite sur la page suivante)

(suite de la page précédente)

```

        return false; // pas là
        self.flags[value] = false;
        return true;
    }

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

contract C {
    Data knownValues;

    function register(uint value) public {
        // Les fonctions de la bibliothèque peuvent être appelées sans une
        // instance spécifique de la bibliothèque, puisque
        // l'"instance" sera le contrat en cours.
        require(Set.insert(knownValues, value));
    }
    // In this contract, we can also directly access knownValues.flags, if we want.
}

```

Bien sûr, vous n'êtes pas obligé de suivre cette voie pour utiliser des bibliothèques : elles peuvent aussi être utilisées sans définir de type de données struct. Les fonctions fonctionnent également sans paramètres de de référence de stockage, et elles peuvent avoir plusieurs paramètres de référence et dans n'importe quelle position.

Les appels à `Set.contains`, `Set.insert` et `Set.remove` sont tous compilés en tant qu'appels (DELEGATECALL) à un contrat/librairie externe. Si vous utilisez des bibliothèques, soyez conscient qu'un appel à une fonction externe réelle est effectué. `msg.sender`, `msg.value` et `this` garderont leurs valeurs dans cet appel. (avant Homestead, à cause de l'utilisation de CALLCODE, `msg.sender` et `msg.value` changeaient, cependant).

L'exemple suivant montre comment utiliser les *types stockés dans la mémoire* et les fonctions internes des bibliothèques afin d'implémenter des types personnalisés sans la surcharge des appels de fonctions externes :

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

struct bigint {
    uint[] limbs;
}

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

    function add(bigint memory _a, bigint memory _b) internal pure returns (bigint_
memory r) {

```

(suite sur la page suivante)

```

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);
    unchecked {
        r.limbs[i] = a + b + carry;

        if (a + b < a || (a + b == type(uint).max && carry > 0))
            carry = 1;
        else
            carry = 0;
    }
}
if (carry > 0) {
    // dommage, nous devons ajouter un membre
    uint[] memory newLimbs = new uint[](r.limbs.length + 1);
    uint i;
    for (i = 0; i < r.limbs.length; ++i)
        newLimbs[i] = r.limbs[i];
    newLimbs[i] = carry;
    r.limbs = newLimbs;
}

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

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

contract C {
    using BigInt for bigint;

    function f() public pure {
        bigint memory x = BigInt.fromUint(7);
        bigint memory y = BigInt.fromUint(type(uint).max);
        bigint memory z = x.add(y);
        assert(z.limb(1) > 0);
    }
}

```

Il est possible d'obtenir l'adresse d'une bibliothèque en convertissant le type de la bibliothèque en type `address`, c'est-à-dire en utilisant `address(LibraryName)`.

Comme le compilateur ne connaît pas l'adresse à laquelle la bibliothèque sera déployée, le code hexadécimal compilé contiendra des caractères de remplacement de la forme `__$30bbc0abd4d6364515865950d3e0d10953$__`. Le caractère de remplacement est un préfixe de 34 caractères de l'encodage hexadécimal du hachage keccak256 du nom de bibliothèque pleinement qualifié, qui serait par exemple `libraries/bigint.sol:BigInt` si la bibliothèque était stockée dans un fichier appelé `bigint.sol` dans un répertoire `libraries/`. Un tel bytecode est incomplet et ne devrait

pas être déployé. Les placeholders doivent être remplacés par des adresses réelles. Vous pouvez le faire soit en passant au compilateur lors de la compilation de la bibliothèque ou en utilisant l'éditeur de liens pour mettre à jour un binaire déjà compilé. Voir *Liens entre les bibliothèques* pour des informations sur la façon d'utiliser le compilateur en ligne de commande pour la liaison.

Par rapport aux contrats, les bibliothèques sont limitées de la manière suivante :

- elles ne peuvent pas avoir de variables d'état
- elles ne peuvent ni hériter ni être héritées
- elles ne peuvent pas recevoir d'éther
- elles ne peuvent pas être détruites

(Ces restrictions pourraient être levées ultérieurement).

Signatures de fonction et sélecteurs dans les bibliothèques

Bien que les appels externes à des fonctions de bibliothèques publiques ou externes soient possibles, la convention d'appel pour de tels appels est considérée comme interne à Solidity et n'est pas la même que celle spécifiée pour la fonction ordinaire du *contrat ABI*. Les fonctions de bibliothèque externes supportent plus de types d'arguments que les fonctions de contrat externes, par exemple les structs récursifs et les pointeurs de stockage. Pour cette raison, les signatures de fonctions utilisées pour calculer le sélecteur à 4 octets sont calculées selon un schéma de dénomination interne et les arguments de types non pris en charge par l'ABI du contrat utilisent un encodage interne.

Les identifiants suivants sont utilisés pour les types dans les signatures :

- Les types de valeurs, les `string` non stockés et les `bytes` non stockés utilisent les mêmes identifiants que dans l'ABI du contrat.
- Les types de tableaux non stockés suivent la même convention que dans l'ABI du contrat, c'est-à-dire `<type>[]` pour les tableaux dynamiques et `<type>[M]` pour les tableaux de taille fixe de `M` éléments.
- Les structures non stockées sont désignées par leur nom complet, c'est-à-dire `C.S` pour contrat `C { struct S { ... } }`.
- Les mappages de pointeurs de stockage utilisent `mapping(<keyType> => <valueType>) storage` où `<keyType>` et `<valueType>` sont les identificateurs des types de clé et de valeur du mappage, respectivement.
- Les autres types de pointeurs de stockage utilisent l'identificateur de type de leur type non stocké correspondant, mais ajoutent un espace unique suivi de `storage`.

Le codage des arguments est le même que pour l'ABI des contrats ordinaires, sauf pour les pointeurs de stockage, qui sont codés en tant que `uint256` faisant référence à l'emplacement de stockage vers lequel ils pointent.

Comme pour l'ABI du contrat, le sélecteur est constitué des quatre premiers octets du Keccak256-hash de la signature. Sa valeur peut être obtenue à partir de Solidity en utilisant le membre `.selector` comme suit :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.14 <0.9.0;

library L {
    function f(uint256) external {}
}

contract C {
    function g() public pure returns (bytes4) {
        return L.f.selector;
    }
}
```

Protection d'appel pour les bibliothèques

Comme mentionné dans l'introduction, si le code d'une bibliothèque est exécuté en utilisant un `CALL` au lieu d'un `DELEGATECALL` ou `CALLCODE`, il se réverbère sauf si une fonction `view` ou `pure` est appelée.

L'EVM ne fournit pas de moyen direct pour qu'un contrat puisse détecter s'il a été appelé en utilisant `CALL` ou non, mais un contrat peut utiliser l'opcode `ADDRESS` pour savoir « où » il est actuellement en cours d'exécution. Le code généré compare cette adresse à l'adresse utilisée au moment de la construction pour déterminer le mode d'appel.

Plus spécifiquement, le code d'exécution d'une bibliothèque commence toujours par une instruction `push`, qui est un zéro de 20 octets au moment de la compilation. Lorsque le code déployé s'exécute, cette constante est remplacée en mémoire par l'adresse actuelle et ce code modifié est stocké dans le contrat. Au moment de l'exécution, cela fait en sorte que l'adresse du moment du déploiement soit la première constante à être poussée sur la pile et le code du distributeur compare l'adresse actuelle à cette constante pour toute fonction non-visible et non pure.

Cela signifie que le code réel stocké sur la chaîne pour une bibliothèque est différent du code rapporté par le compilateur en tant que `deployedBytecode`.

3.9.12 Utiliser For

La directive `using A for B;` peut être utilisée pour attacher des fonctions (de la bibliothèque A) à n'importe quel type (B) dans le contexte d'un contrat. Ces fonctions recevront l'objet sur lequel elles sont appelées comme premier paramètre (comme la variable `self` en Python).

L'effet de `using A for *;` est que les fonctions de la bibliothèque A sont attachées à *tout* type.

Dans les deux cas, *toutes* les fonctions de la bibliothèque sont attachées, même celles pour lesquelles le type du premier paramètre ne correspond pas au type de l'objet. Le type est vérifié au moment où la fonction est appelée et la résolution de surcharge de fonction est effectuée.

La directive `using A pour B;` n'est active que dans le contrat actuel, y compris au sein de toutes ses fonctions, et n'a aucun effet en dehors du contrat dans lequel elle est utilisée. La directive ne peut être utilisée qu'à l'intérieur d'un contrat, et non à l'intérieur de l'une de ses fonctions.

Réécrivons l'exemple de l'ensemble à partir de la directive *Bibliothèques* de cette manière :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// Il s'agit du même code que précédemment, mais sans commentaires.
struct Data { mapping(uint => bool) flags; }

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

    function remove(Data storage self, uint value)
        public
```

(suite sur la page suivante)

(suite de la page précédente)

```

    returns (bool)
  {
    if (!self.flags[value])
      return false; // pas là
    self.flags[value] = false;
    return true;
  }

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

contract C {
  using Set for Data; // c'est le changement crucial
  Data knownValues;

  function register(uint value) public {
    // Ici, toutes les variables de type Data ont
    // des fonctions membres correspondantes.
    // L'appel de fonction suivant est identique à
    // `Set.insert(knownValues, value)`
    require(knownValues.insert(value));
  }
}

```

Il est également possible d'étendre les types élémentaires de cette manière :

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.8 <0.9.0;

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

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

```

(suite sur la page suivante)

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

function replace(uint _old, uint _new) public {
    // Cette opération effectue l'appel de la fonction de bibliothèque
    uint index = data.indexOf(_old);
    if (index == type(uint).max)
        data.push(_new);
    else
        data[index] = _new;
}
}
```

Notez que tous les appels de bibliothèque externes sont des appels de fonction EVM réels. Cela signifie que si vous passez des types de mémoire ou de valeur, une copie sera effectuée, même de la variable `self`. La seule situation où aucune copie ne sera effectuée est l'utilisation de variables de référence de stockage ou l'appel de fonctions de bibliothèque internes sont appelées.

3.10 Assemblage en ligne

Vous pouvez intercaler des instructions Solidity avec de l'assemblage en ligne dans un langage proche de celui de la machine virtuelle Ethereum. Cela vous donne un contrôle plus fin, ce qui est particulièrement utile lorsque vous améliorez le langage en écrivant des bibliothèques.

Le langage utilisé pour l'assemblage en ligne dans Solidity est appelé *Yul*. et il est documenté dans sa propre section. Cette section couvrira uniquement comment le code d'assemblage en ligne peut s'interfacer avec le code Solidity environnant.

Avvertissement : L'assemblage en ligne est un moyen d'accéder à la machine virtuelle d'Ethereum à un faible niveau. Cela contourne plusieurs fonctions importantes de sécurité et de vérification de Solidity. Vous ne devez l'utiliser que pour des tâches qui en ont besoin, et seulement si vous avez confiance en son utilisation.

Un bloc d'assemblage en ligne est marqué par `assembly { . . . }`, où le code à l'intérieur des accolades est du code dans le langage *Yul*.

Le code d'assemblage en ligne peut accéder aux variables locales de Solidity comme expliqué ci-dessous.

Les différents blocs d'assemblage en ligne ne partagent aucun espace de nom, c'est-à-dire qu'il n'est pas possible d'appeler une fonction Yul ou d'accéder à des variables Solidity.

3.10.1 Exemple

L'exemple suivant fournit du code de bibliothèque pour accéder au code d'un autre contrat et le et de le charger dans une variable bytes. Ceci est également possible avec « plain Solidity », en utilisant <adresse>.code. Mais le point important ici est que les bibliothèques d'assemblage réutilisables peuvent améliorer le langage Solidity sans changer le compilateur. langage Solidity sans changer de compilateur.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

library GetCode {
    function at(address _addr) public view returns (bytes memory o_code) {
        assembly {
            // retrieve the size of the code, this needs assembly
            let size := extcodesize(_addr)
            // allocate output byte array - this could also be done without assembly
            // by using o_code = new bytes(size)
            o_code := mload(0x40)
            // new "memory end" including padding
            mstore(0x40, add(o_code, and(add(add(size, 0x20), 0x1f), not(0x1f))))
            // store length in memory
            mstore(o_code, size)
            // actually retrieve the code, this needs assembly
            extcodecopy(_addr, add(o_code, 0x20), 0, size)
        }
    }
}
```

L'assemblage en ligne est également bénéfique dans les cas où l'optimiseur ne parvient pas à produire code efficace, par exemple :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

library VectorSum {
    // This function is less efficient because the optimizer currently fails to
    // remove the bounds checks in array access.
    function sumSolidity(uint[] memory _data) public pure returns (uint sum) {
        for (uint i = 0; i < _data.length; ++i)
            sum += _data[i];
    }

    // We know that we only access the array in bounds, so we can avoid the check.
    // 0x20 needs to be added to an array because the first slot contains the
    // array length.
    function sumAsm(uint[] memory _data) public pure returns (uint sum) {
        for (uint i = 0; i < _data.length; ++i) {
            assembly {
                sum := add(sum, mload(add(add(_data, 0x20), mul(i, 0x20))))
            }
        }
    }
}
```

(suite sur la page suivante)

```

// Same as above, but accomplish the entire code within inline assembly.
function sumPureAsm(uint[] memory _data) public pure returns (uint sum) {
    assembly {
        // Load the length (first 32 bytes)
        let len := mload(_data)

        // Skip over the length field.
        //
        // Keep temporary variable so it can be incremented in place.
        //
        // NOTE: incrementing _data would result in an unusable
        //       _data variable after this assembly block
        let data := add(_data, 0x20)

        // Iterate until the bound is not met.
        for
            { let end := add(data, mul(len, 0x20)) }
            lt(data, end)
            { data := add(data, 0x20) }
        {
            sum := add(sum, mload(data))
        }
    }
}

```

3.10.2 Accès aux variables, fonctions et bibliothèques externes

Vous pouvez accéder aux variables Solidity et autres identifiants en utilisant leur nom.

Les variables locales de type valeur sont directement utilisables dans l'assemblage en ligne. Elles peuvent à la fois être lues et assignées.

Les variables locales qui font référence à la mémoire sont évaluées à l'adresse de la variable en mémoire et non à la valeur elle-même. Ces variables peuvent également être assignées, mais notez qu'une assignation ne modifie que le pointeur et non les données. et qu'il est de votre responsabilité de respecter la gestion de la mémoire de Solidity. Voir *Conventions dans Solidity*.

De même, les variables locales qui font référence à des tableaux de données ou à des structures de données de taille statique sont évaluées à l'adresse de la variable dans `calldata`, et non à la valeur elle-même. La variable peut également être assignée à un nouveau décalage, mais notez qu'aucune validation pour assurer que la variable ne pointerait pas au-delà de `calldatasize()` n'est effectuée.

Pour les pointeurs de fonctions externes, l'adresse et le sélecteur de fonction peuvent être accessibles en utilisant `x`. `address` et `x.selector`. Le sélecteur est constitué de quatre octets alignés à droite. Les deux valeurs peuvent être assignées. Par exemple :

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.10 <0.9.0;

contract C {
    // Assigns a new selector and address to the return variable @fun
    function combineToFunctionPointer(address newAddress, uint newSelector) public pure_
    ↪ returns (function() external fun) {

```

(suite sur la page suivante)

(suite de la page précédente)

```

assembly {
    fun.selector := newSelector
    fun.address  := newAddress
}
}

```

Pour les tableaux de données dynamiques, vous pouvez accéder à leur offset (en octets) et leur longueur (nombre d'éléments) en utilisant `x.offset` et `x.length`. Les deux expressions peuvent également être assignées à, mais comme pour le cas statique, aucune validation ne sera effectuée pour s'assurer que la zone de données résultante est dans les limites de `calldata.size()`.

Pour les variables de stockage local ou les variables d'état, un seul identifiant `Yul` n'est pas suffisant, car elles n'occupent pas nécessairement un seul emplacement de stockage complet. Par conséquent, leur « adresse » est composée d'un slot et d'un byte-offset à l'intérieur de cet emplacement. Pour récupérer le slot pointé par la variable `x`, on utilise `x.slot`, et pour récupérer le byte-offset vous utilisez `x.offset`. L'utilisation de la variable `x` elle-même entraînera une erreur.

Vous pouvez également assigner à la partie `.slot` d'un pointeur de variable de stockage local. Pour celles-ci (structs, arrays ou mappings), la partie `.offset` est toujours zéro. Il n'est pas possible d'assigner à la partie `.slot` ou `.offset` d'une variable d'état, cependant.

Les variables locales de Solidity sont disponibles pour les affectations, par exemple :

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract C {
    uint b;
    function f(uint x) public view returns (uint r) {
        assembly {
            // We ignore the storage slot offset, we know it is zero
            // in this special case.
            r := mul(x, sload(b.slot))
        }
    }
}

```

Avertissement : Si vous accédez à des variables d'un type qui s'étend sur moins de 256 bits (par exemple `uint64`, `adresse`, ou `bytes16`), vous ne pouvez pas faire d'hypothèses sur les bits qui ne font pas partie du codage du type. En particulier, ne supposez pas qu'ils soient nuls. Pour être sûr, effacez toujours les données correctement avant de les utiliser dans un contexte où cela est important : `uint32 x = f() ; assembly { x := and(x, 0xffffffff) /* maintenant utiliser x */ }` Pour nettoyer les types signés, vous pouvez utiliser l'opcode `signextend` : `assembly { signextend(<nombre_bytes_de_x_moins_un>, x) }`

Depuis Solidity 0.6.0, le nom d'une variable d'assemblage en ligne ne peut pas suivre aucune déclaration visible dans la portée du bloc d'assemblage en ligne (y compris les déclarations de variables, de contrats et de fonctions).

Depuis la version 0.7.0 de Solidity, les variables et les fonctions déclarées à l'intérieur du bloc d'assemblage en ligne ne peuvent pas contenir `.`, mais l'utilisation de `.` est valide valide pour accéder aux variables Solidity depuis l'extérieur du bloc d'assemblage en ligne.

3.10.3 Choses à éviter

L'assemblage en ligne peut avoir une apparence de haut niveau, mais il est en fait extrêmement bas niveau. Les appels de fonction, les boucles, les ifs et les switches sont convertis par de simples règles de réécriture. règles de réécriture et après cela, la seule chose que l'assembleur fait pour vous est de réarranger opcodes de style fonctionnel, le comptage de la hauteur de la pile pour pour l'accès aux variables et la suppression des emplacements de pile pour les variables locales à l'assemblage lorsque la fin de leur bloc est atteinte.

3.10.4 Conventions dans Solidity

Contrairement à l'assemblage EVM, Solidity possède des types dont la taille est inférieure à 256 bits, par exemple uint24. Pour des raisons d'efficacité, la plupart des opérations arithmétiques ignorent le fait que les types peuvent être plus courts que 256 bits, et les bits d'ordre supérieur sont nettoyés lorsque cela est nécessaire, c'est-à-dire peu de temps avant qu'ils ne soient écrits en mémoire ou avant que les comparaisons ne soient effectuées. Cela signifie que si vous accédez à une telle variable à partir d'un assemblage en ligne, vous devrez peut-être d'abord nettoyer manuellement les bits d'ordre supérieur.

Solidity gère la mémoire de la manière suivante. Il existe un « pointeur de mémoire libre » à la position 0x40 dans la mémoire. Si vous voulez allouer de la mémoire, utilisez la mémoire à partir de l'endroit où pointe ce pointeur et mettez-la à jour. Il n'y a aucune garantie que la mémoire n'a pas été utilisée auparavant et vous ne pouvez donc pas supposer que son contenu est de zéro octet. Il n'existe pas de mécanisme intégré pour libérer la mémoire allouée. Voici un extrait d'assemblage que vous pouvez utiliser pour allouer de la mémoire qui suit le processus décrit ci-dessus.

```
function allocate(length) -> pos {
    pos := mload(0x40)
    mstore(0x40, add(pos, length))
}
```

Les 64 premiers octets de la mémoire peuvent être utilisés comme « espace de grattage » pour l'allocation à court terme. Les 32 octets après le pointeur de mémoire libre (c'est-à-dire, à partir de 0x60) sont censés être zéro de manière permanente et sont utilisés comme valeur initiale pour les tableaux de mémoire dynamique vides. Cela signifie que la mémoire allouable commence à 0x80, qui est la valeur initiale du pointeur de mémoire libre.

Les éléments des tableaux de mémoire dans Solidity occupent toujours des multiples de 32 octets (c'est même vrai pour les « octets »). Même vrai pour bytes1[], mais pas pour bytes et string). Les tableaux de mémoire multidimensionnels sont des pointeurs vers des tableaux de mémoire. La longueur d'un tableau dynamique est stockée dans le premier emplacement du tableau et suivie par les éléments du tableau.

Avertissement : Les tableaux de mémoire de taille statique n'ont pas de champ de longueur, mais celui-ci pourrait être ajouté ultérieurement pour permettre une meilleure convertibilité entre les tableaux de taille statique et dynamique. Pour permettre une meilleure convertibilité entre les tableaux de taille statique et dynamique. Donc ne vous y fiez pas.

3.11 Aide-mémoire

3.11.1 Ordre de Préséance des Opérateurs

Voici l'ordre de préséance des opérateurs, classés par ordre d'évaluation.

Prédominance	Description	Opérateur
1	Incrément et décrétement de Postfix	++, --
	Nouvelle expression	new <nomutilisateur>
	Subscription de tableau	<array>[<index>]
	Accès des membres	<objet>.<membre>
	Appel de type fonctionnel	<func>(<args...>)
	Parenthèses	(<déclaration>)
2	Préfixe d'incrément et de décrémentation	++, --
	Moins unaire	-
	Opérations unaires	delete
	Logique NON	!
	NON par bit	~
3	Exponentité	**
4	Multiplication, division et modulo	*, /, %
5	Addition et soustraction	+, -
6	Opérateurs de décalage par bit	<<, >>
7	ET par bit	&
8	XOR par bit	^
9	OU par bit	
10	Opérateurs d'inégalité	<, >, <=, >=
11	Opérateurs d'égalité	==, !=
12	ET logique	&&
13	OU logique	
14	Opérateur ternaire	<conditional> ? <if-true> : <if-false>
	Opérateurs d'assignation	=, =, ^=, &=, <<=, >>=, +=, -=, *=, /=, %=
15	Opérateur de virgule	,

3.11.2 Variables Globales

- `abi.decode(bytes memory encodedData, (...)) returns (...)` : *ABI*-décode les données fournies. Les types sont donnés entre parenthèses comme deuxième argument. Exemple : `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...)` returns (bytes memory) : *ABI*-encode les arguments donnés
- `abi.encodePacked(...)` returns (bytes memory) : Performe l'*encodage emballé* des arguments donnés. Notez que cet encodage peut être ambigu !
- `abi.encodeWithSelector(bytes4 selector, ...)` returns (bytes memory) : *ABI*-encode les arguments donnés en commençant par le deuxième et en ajoutant au début le sélecteur de quatre octets donné.
- `abi.encodeCall(function functionPointer, (...))` returns (bytes memory) : *ABI*-encode un appel à `functionPointer` avec les arguments trouvés dans le tuple. Effectue une vérification complète des types, en s'assurant que les types correspondent à la signature de la fonction. Le résultat est égal à `abi.encodeWithSelector(functionPointer.selector, (...))`

- `abi.encodeWithSignature(string memory signature, ...)` returns (bytes memory) : Equivalant à `abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)`
- `bytes.concat(...)` returns (bytes memory) : *Concatène un nombre variable d'arguments dans un tableau d'un octet*
- `block.basefee (uint)` : redevance de base du bloc actuel (EIP-3198 et EIP-1559)
- `block.chainid (uint)` : identifiant de la chaîne actuelle
- `block.coinbase (address payable)` : adresse du mineur du bloc actuel
- `block.difficulty (uint)` : difficulté actuelle du bloc
- `block.gaslimit (uint)` : limite de gaz du bloc actuel
- `block.number (uint)` : numéro du bloc actuel
- `block.timestamp (uint)` : Horodatage du bloc actuel
- `gasleft()` returns (uint256) : gaz résiduel
- `msg.data (bytes)` : données d'appel complètes
- `msg.sender (address)` : expéditeur du message (appel en cours)
- `msg.value (uint)` : nombre de wei envoyés avec le message
- `tx.gasprice (uint)` : prix du gaz de la transaction
- `tx.origin (address)` : expéditeur de la transaction (chaîne d'appel complète)
- `assert(bool condition)` : interrompt l'exécution et annule les changements d'état si la condition est « fausse » (à utiliser pour les erreurs internes).
- `require(bool condition)` : interrompt l'exécution et annuler les changements d'état si la condition est « fausse » (à utiliser pour une entrée malformée ou une erreur dans un composant externe)
- `require(bool condition, string memory message)` : interrompt l'exécution et annule les changements d'état si la condition est « fausse » (à utiliser en cas d'entrée malformée ou d'erreur dans un composant externe). Fournit également un message d'erreur.
- `revert()` : interrompt l'exécution et revenir sur les changements d'état
- `revert(string memory message)` : interrompt l'exécution et revenir sur les changements d'état en fournissant une chaîne explicative
- `blockhash(uint blockNumber)` returns (bytes32) : hachage du bloc donné - ne fonctionne que pour les 256 blocs les plus récents
- `keccak256(bytes memory)` returns (bytes32) : calculer le hachage Keccak-256 de l'entrée
- `sha256(bytes memory)` returns (bytes32) : calculer le hachage SHA-256 de l'entrée
- `ripemd160(bytes memory)` returns (bytes20) : calculer le hachage RIPEMD-160 de l'entrée
- `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s)` returns (address) : récupérer l'adresse associée à la clé publique de la signature de la courbe elliptique, renvoie zéro en cas d'erreur
- `addmod(uint x, uint y, uint k)` returns (uint) : compute $(x + y) \% k$ où l'addition est effectuée avec une précision arbitraire et ne s'arrête pas à 2^{256} . Affirmer que $k \neq 0$ à partir de la version 0.5.0.
- `mulmod(uint x, uint y, uint k)` returns (uint) : compute $(x * y) \% k$ où la multiplication est effectuée avec une précision arbitraire et ne s'arrête pas à 2^{256} . Affirmer que $k \neq 0$ à partir de la version 0.5.0.
- `this (current contract's type)` : le contrat en cours, explicitement convertible en « adresse » ou « adresse payable ».
- `super` : le contrat un niveau plus haut dans la hiérarchie d'héritage
- `selfdestruct(address payable recipient)` : détruire le contrat en cours, en envoyant ses fonds à l'adresse donnée
- `<address>.balance (uint256)` : solde de la *Address* dans Wei
- `<address>.code (bytes memory)` : le code à *Address* (peut être vide)
- `<address>.codehash (bytes32)` : le codehash de l'adresse *Address*
- `<address payable>.send(uint256 amount)` returns (bool) : envoie une quantité donnée de Wei à *Address*, renvoie *false* en cas d'échec
- `<address payable>.transfer(uint256 amount)` : envoie une quantité donnée de Wei à *Address*, lance en cas d'échec
- `type(C).name (string)` : le nom du contrat
- `type(C).creationCode (bytes memory)` : bytecode de création du contrat donné, voir *Type Information*.
- `type(C).runtimeCode (bytes memory)` : le bytecode d'exécution du contrat donné, voir *Type Information*.

- `type(I).interfaceId (bytes4)` : contenant l'identificateur d'interface EIP-165 de l'interface donnée, voir *Type Information*.
- `type(T).min (T)` : la valeur minimale représentable par le type entier T, voir *Type Information*.
- `type(T).max (T)` : la valeur maximale représentable par le type entier T, voir *Type Information*.

Note : Lorsque les contrats sont évalués hors chaîne plutôt que dans le contexte d'une transaction comprise dans un bloc, vous ne devez pas supposer que `block.*` et `tx.*` font référence à des valeurs d'un bloc ou d'une transaction d'un bloc ou d'une transaction spécifique. Ces valeurs sont fournies par l'implémentation EVM qui exécute le contrat et peuvent être arbitraires. `contract` et peuvent être arbitraires.

Note : Ne comptez pas sur `block.timestamp` ou `blockhash` comme source d'aléatoire, à moins que vous ne sachiez ce que vous faites.

L'horodatage et le hachage du bloc peuvent tous deux être influencés par les mineurs dans une certaine mesure. De mauvais acteurs dans la communauté minière peuvent par exemple exécuter une fonction de paiement de casino sur un hash choisi et réessayer un autre hash s'ils n'ont pas reçu d'argent.

L'horodatage du bloc actuel doit être strictement plus grand que l'horodatage du dernier bloc, mais la seule garantie est qu'il se situera quelque part entre les horodatages de deux blocs consécutifs dans la chaîne canonique.

Note : Les hachages des blocs ne sont pas disponibles pour tous les blocs pour des raisons d'évolutivité. Vous ne pouvez accéder qu'aux hachages des 256 blocs les plus récents. autres valeurs seront nulles.

Note : Dans la version 0.5.0, les alias suivants ont été supprimés : `suicide` comme alias pour `selfdestruct`, `msg.gas` comme alias pour `gasleft`, `block.blockhash` comme alias pour `blockhash` et `sha3` comme alias pour `keccak256`.

Note : Dans la version 0.7.0, l'alias `now` (pour `block.timestamp`) a été supprimé.

3.11.3 Spécification de la Visibilité des Fonctions

```
function myFunction() <visibility specifier> returns (bool) {
    return true;
}
```

- `public` : visible en externe et en interne (crée une *fonction réceptrice* pour les variables de stockage/d'état)
- `private` : uniquement visible dans le contrat en cours
- `external` : visible uniquement en externe (uniquement pour les fonctions) - c'est-à-dire qu'il ne peut être appelé que par message (via `this.func`)
- `internal` : uniquement visible en interne

3.11.4 Modificateurs

- `pure` pour les fonctions : Interdit la modification ou l'accès à l'état.
- `view` pour les fonctions : Interdit la modification de l'état.
- `payable` pour les fonctions : Leur permet de recevoir de l'Ether en même temps qu'un appel.
- `constant` pour les variables d'état : Ne permet pas l'affectation (sauf l'initialisation), n'occupe pas d'emplacement de stockage.
- `immutable` pour les variables d'état : Permet exactement une affectation au moment de la construction et est constante par la suite. Est stockée dans le code.
- `anonymous` pour les événements : Ne stocke pas la signature de l'événement comme sujet.
- `indexed` pour les paramètres d'événements : Stocke le paramètre en tant que sujet.
- `virtual` pour les fonctions et les modificateurs : Permet de modifier le comportement de la fonction ou du modificateur dans les contrats dérivés.
- `override` : Indique que cette fonction, ce modificateur ou cette variable d'état publique change le comportement d'une fonction ou d'un modificateur dans un contrat de base.

3.11.5 Mots clés réservés

Ces mots-clés sont réservés dans Solidity. Ils pourraient faire partie de la syntaxe à l'avenir :

`after`, `alias`, `apply`, `auto`, `byte`, `case`, `copyof`, `default`, `define`, `final`, `implements`, `in`, `inline`, `let`, `macro`, `match`, `mutable`, `null`, `of`, `partial`, `promise`, `reference`, `relocatable`, `sealed`, `sizeof`, `static`, `supports`, `switch`, `typedef`, `typeof`, `var`.

3.12 Utilisation du compilateur

3.12.1 Utilisation du compilateur en ligne de commande

Note : Cette section ne s'applique pas à *solcjs*, même s'il est utilisé en mode ligne de commande.

Utilisation de base

L'une des cibles de construction du référentiel Solidity est `solc`, le compilateur en ligne de commande de Solidity. L'utilisation de `solc --help` vous fournit une explication de toutes les options. Le compilateur peut produire diverses sorties, allant de simples binaires et assemblages sur un arbre syntaxique abstrait (parse tree) à des estimations de l'utilisation du gaz. Si vous voulez seulement compiler un seul fichier, vous le lancez comme `solc --bin sourceFile.sol` et il imprimera le binaire. Si vous voulez obtenir certaines des variantes de sortie plus avancées de `solc`, il est probablement préférable de lui dire de tout sortir dans des fichiers séparés en utilisant `solc -o outputDirectory --bin --ast-compact-json --asm sourceFile.sol`.

Options de l'optimiseur

Avant de déployer votre contrat, activez l'optimiseur lors de la compilation en utilisant `solc --optimize --bin sourceFile.sol`. Par défaut, l'optimiseur optimisera le contrat en supposant qu'il est appelé 200 fois au cours de sa durée de vie (plus précisément, il suppose que chaque opcode est exécuté environ 200 fois). Si vous voulez que le déploiement initial du contrat soit moins cher et que les exécutions de fonctions ultérieures soient plus coûteuses, définissez-le à `--optimize-runs=1`. Si vous vous attendez à de nombreuses transactions et que vous ne vous souciez pas des coûts de la taille de la sortie, définissez `--optimize-runs` à un nombre élevé. Ce paramètre a des effets sur les éléments suivants (cela pourrait changer dans le futur) :

- la taille de la recherche binaire dans la routine d'envoi des fonctions
- la façon dont les constantes comme les grands nombres ou les chaînes de caractères sont stockées.

Chemin de base et remappage des importations

Le compilateur en ligne de commande lira automatiquement les fichiers importés depuis le système de fichiers, mais il est également possible de fournir des redirections *path* en utilisant `prefix=path` de la manière suivante :

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

Ceci indique essentiellement au compilateur de rechercher tout ce qui commence par `github.com/ethereum/dapp-bin/` sous `/usr/local/lib/dapp-bin/`.

Lorsque vous accédez au système de fichiers pour rechercher des importations, les *chemins qui ne commencent pas par ./ ou ../* sont traités comme relatifs aux répertoires spécifiés en utilisant les options `--base-path` et `-include-path` (ou le répertoire de travail actuel si le chemin de base n'est pas spécifié). De plus, la partie du chemin ajoutée via ces options n'apparaîtra pas dans les métadonnées du contrat.

Pour des raisons de sécurité, le compilateur a des *restrictions sur les répertoires auxquels il peut accéder*. Les répertoires des fichiers sources spécifiés sur la ligne de commande et les chemins cibles des remappings sont automatiquement autorisés à être accédés par le lecteur de fichiers, mais tout le reste est rejeté par défaut. Des chemins supplémentaires (et leurs sous-répertoires) peuvent être autorisés via la commande `--allow-paths /sample/path,/another/sample/path`. Tout ce qui se trouve à l'intérieur du chemin spécifié par `--base-path` est toujours autorisé.

Ce qui précède n'est qu'une simplification de la façon dont le compilateur gère les chemins d'importation. Pour une explication détaillée avec des exemples et une discussion des cas de coin, veuillez vous référer à la section sur *résolution de chemin*.

Liens entre les bibliothèques

Si vos contrats utilisent *libraries*, vous remarquerez que le bytecode contient des sous-chaînes de la forme `__$53aea86b7d70b31448b230b20ae141a537$__`. Il s'agit de caractères de remplacement pour les adresses réelles des bibliothèques. Le placeholder est un préfixe de 34 caractères de l'encodage hexadécimal du hachage keccak256 du nom de bibliothèque entièrement qualifié. Le fichier de bytecode contiendra également des lignes de la forme `/ / <placeholder> -> <fq library name>` à la fin pour aider à identifier les bibliothèques que les placeholders représentent. Notez que le nom de bibliothèque pleinement qualifié est le chemin de son fichier source et le nom de la bibliothèque séparés par `:`. Vous pouvez utiliser `solc` comme linker, ce qui signifie qu'il insérera les adresses des bibliothèques pour vous à ces endroits :

Soit vous ajoutez `--libraries "file.sol:Math=0x1234567890123456789012345678901234567890 file.sol:Heap=0xabCD5678901234567890123458901234567890"` à votre commande pour fournir une adresse pour chaque bibliothèque (utilisez des virgules ou des espaces comme séparateurs) ou stockez la chaîne dans un fichier (une bibliothèque par ligne) et lancez `solc` en utilisant `-libraries fileName``.

Note : À partir de la version 0.8.1 de Solidity, on accepte `=` comme séparateur entre bibliothèque et adresse, et `:` comme séparateur est déprécié. Il sera supprimé à l'avenir. Actuellement,

```
--libraries "file.sol:Math:0x123456789012345678901234567890123458901234567890 file.  
sol:Heap:0xabCD567890123456789012345890123234567890" fonctionnera également.
```

Si `solc` est appelé avec l'option `--standard-json`, il attendra une entrée JSON (comme expliqué ci-dessous) sur l'entrée standard, et retournera une sortie JSON sur la sortie standard. C'est l'interface recommandée pour des utilisations plus complexes et particulièrement automatisées. Le processus se terminera toujours dans un état « success » et rapportera toute erreur via la sortie JSON. L'option `--base-path` est également traitée en mode `standard-json`.

Si `solc` est appelé avec l'option `--link`, tous les fichiers d'entrée sont interprétés comme des binaires non liés (encodés en hexadécimal) dans le format `__$53aea86b7d70b31448b230b20ae141a537$__` donné ci-dessus et sont liés in-place (si l'entrée est lue depuis `stdin`, elle est écrite sur `stdout`). Toutes les options sauf `--libraries` sont ignorées (y compris `-o`) dans ce cas.

Avertissement : La liaison manuelle des bibliothèques sur le bytecode généré est déconseillée car elle ne permet pas de mettre à jour les métadonnées du contrat. Puisque les métadonnées contiennent une liste de bibliothèques spécifiées au moment de la compilation et le bytecode contient un hash de métadonnées, vous obtiendrez des binaires différents, selon du moment où la liaison est effectuée.

Vous devez demander au compilateur de lier les bibliothèques au moment où un contrat est compilé, soit en utilisant l'option `--libraries` de `solc` ou la clé `libraries` si vous utilisez l'interface `standard-JSON` au compilateur.

Note : L'espace réservé à la bibliothèque était auparavant le nom pleinement qualifié de la bibliothèque elle-même au lieu du hash de celui-ci. Ce format est toujours pris en charge par `solc --link` mais le compilateur ne l'affichera plus. Ce changement a été fait pour réduire la probabilité de collision entre les bibliothèques, puisque seuls les 36 premiers caractères du nom de du nom complet de la bibliothèque pouvaient être utilisés.

3.12.2 Réglage de la version de l'EVM sur la cible

Lorsque vous compilez le code de votre contrat, vous pouvez spécifier la version de la machine virtuelle d'Ethereum pour laquelle compiler afin d'éviter des caractéristiques ou des comportements particuliers.

Avertissement : La compilation pour la mauvaise version EVM peut entraîner un comportement erroné, étrange et défaillant. Veuillez vous assurer, en particulier si vous exécutez une chaîne privée, que vous utilisez les versions EVM correspondantes.

Sur la ligne de commande, vous pouvez sélectionner la version EVM comme suit :

```
solc --evm-version <VERSION> contract.sol
```

Dans l'interface *standard JSON*, utilisez la clé `"evmVersion"` dans le champ `"settings"` :

```
{  
  "sources": { /* ... */ },  
  "settings": {  
    "optimizer": { /* ... */ },  
    "evmVersion": "<VERSION>"  
  }  
}
```

Options de la cible

Vous trouverez ci-dessous une liste des versions EVM cibles et des modifications relatives au compilateur introduites à chaque version. La rétrocompatibilité n'est pas garantie entre chaque version.

- **homestead**
 - (version la plus ancienne)
- **tangerineWhistle**
 - Le coût du gaz pour l'accès à d'autres comptes a augmenté, ce qui est pertinent pour l'estimation du gaz et l'optimiseur.
 - Tout le gaz est envoyé par défaut pour les appels externes, auparavant une certaine quantité devait être conservée.
- **spuriousDragon**
 - Le coût du gaz pour l'opcode `exp` a augmenté, ce qui est important pour l'estimation du gaz et l'optimiseur.
- **byzantium**
 - Les opcodes `returndatacopy`, `returndatasize` et `staticcall` sont disponibles en assembly.
 - L'opcode `staticcall` est utilisé lors de l'appel de fonctions de vue ou de fonctions pures non libérées, ce qui empêche les fonctions de modifier l'état au niveau de l'EVM, c'est-à-dire qu'il s'applique même lorsque vous utilisez des conversions de type invalides.
 - Il est possible d'accéder aux données dynamiques renvoyées par les appels de fonctions.
 - Introduction de l'opcode `revert`, ce qui signifie que `revert()` ne gaspillera pas de gaz.
- **constantinople**
 - Les opcodes `create2`, `extcodehash''`, ```shl`, `shr` et `sar` sont disponibles en assembleur.
 - Les opérateurs de décalage utilisent des opcodes de décalage et nécessitent donc moins de gaz.
- **petersburg**
 - Le compilateur se comporte de la même manière qu'avec constantinople.
- **istanbul**
 - Les opcodes `chainid` et ```selfbalance``` sont disponibles en assemblage.
- **berlin**
 - Les coûts du gaz pour `LOAD`, `*CALL`, `BALANCE`, `EXT` et `SELFDESTRUCT` ont augmenté. Le compilateur suppose des coûts de gaz froid pour de telles opérations. Ceci est pertinent pour l'estimation des gaz et l'optimiseur.
- **london (default)**
 - Le tarif de base du bloc ([EIP-3198](#) et [EIP-1559](#)) est accessible via le global `block.basefee` ou `basefee()` en assemblage inline.

3.12.3 Description JSON des entrées et sorties du compilateur

La manière recommandée de s'interfacer avec le compilateur Solidity, surtout pour les configurations plus complexes et automatisées est l'interface dite d'entrée-sortie JSON. La même interface est fournie par toutes les distributions du compilateur.

Les champs sont généralement susceptibles d'être modifiés, certains sont optionnels (comme indiqué), mais nous essayons de ne faire que des changements compatibles avec le passé.

L'API du compilateur attend une entrée au format JSON et produit le résultat de la compilation dans une sortie au format JSON. La sortie d'erreur standard n'est pas utilisée et le processus se terminera toujours dans un état de « succès », même s'il y a eu des erreurs. Les erreurs sont toujours signalées dans le cadre de la sortie JSON.

Les sous-sections suivantes décrivent le format à travers un exemple. Les commentaires ne sont bien sûr pas autorisés et sont utilisés ici uniquement à des fins explicatives.

Description de l'entrée

```

{
  // Requis : Langue du code source. Les langages actuellement pris en charge sont
  ↪ "Solidity" et "Yul".
  "language": "Solidity",
  // Requis
  "sources":
  {
    // Les clés ici sont les noms "globaux" des fichiers sources,
    // les importations peuvent utiliser d'autres fichiers via les remappings (voir ci-
    ↪ dessous).
    "myFile.sol":
    {
      // Facultatif : hachage keccak256 du fichier source.
      // Il est utilisé pour vérifier le contenu récupéré s'il est importé via des URL.
      "keccak256": "0x123...",
      // Obligatoire (sauf si "content" est utilisé, voir ci-dessous) : URL(s) vers le
      ↪ fichier source.
      // Les URL doivent être importées dans cet ordre et le résultat doit être vérifié
      ↪ par rapport à l'empreinte
      // le hachage keccak256 (si disponible). Si le hachage ne correspond pas ou si
      ↪ aucune des
      // URL n'aboutit à un succès, une erreur doit être signalée.
      // En utilisant l'interface en ligne de commande, seuls les chemins de systèmes de
      ↪ fichiers sont pris en charge.
      // Avec l'interface JavaScript, l'URL sera transmise au callback de lecture fourni
      ↪ par l'utilisateur.
      // Ainsi, toute URL prise en charge par le callback peut être utilisée.
      "urls":
      [
        "bzzr://56ab...",
        "ipfs://Qma...",
        "/tmp/path/to/file.sol"
        // Si des fichiers sont utilisés, leurs répertoires doivent être ajoutés à la
        ↪ ligne de commande via
        // `--allow-paths <path>`.
      ]
    },
    "destructible":
    {
      // Facultatif : keccak256 hash du fichier source
      "keccak256": "0x234...",
      // Obligatoire (sauf si "urls" est utilisé) : contenu littéral du fichier source
      "content": "contract destructible is owned { function shutdown() { if (msg.sender
      ↪ == owner) selfdestruct(owner); } }"
    }
  },
  // Optionnel
  "settings":
  {
    // Facultatif : Arrête la compilation après l'étape donnée. Actuellement, seul
    ↪ "parsing" est valide ici

```

(suite sur la page suivante)

(suite de la page précédente)

```

"stopAfter": "parsing",
// Facultatif : Liste triée de réaffectations
"remappings": [ ":g=/dir" ],
// Facultatif : Paramètres de l'optimiseur
"optimizer": {
  // Désactivé par défaut.
  // NOTE : enabled=false laisse toujours certaines optimisations activées. Voir les_
↳ commentaires ci-dessous.
  // ATTENTION : Avant la version 0.8.6, l'omission de la clé 'enabled' n'était pas_
↳ équivalente à la mise en place de la clé 'enabled'.
  // l'activer à false et désactiverait en fait toutes les optimisations.
  "enabled": true,
  // Optimisez en fonction du nombre de fois que vous avez l'intention d'exécuter le_
↳ code.
  // Les valeurs les plus basses optimisent davantage le coût de déploiement initial,
↳ les valeurs les plus élevées optimisent davantage les utilisations à haute fréquence.
  // Plus les valeurs sont faibles, plus l'optimisation est axée sur le coût du_
↳ déploiement initial.
  // Plus les valeurs sont élevées, plus l'optimisation est axée sur un usage_
↳ fréquent.
  "runs": 200,
  // Activez ou désactivez les composants de l'optimiseur en détail.
  // L'interrupteur "enabled" ci-dessus fournit deux valeurs par défaut qui peuvent_
↳ être
  // modifiables ici. Si "details" est donné, "enabled" peut être omis.
  "details": {
    // L'optimiseur de trou d'homme est toujours activé si aucun détail n'est donné,
    // utilisez les détails pour le désactiver.
    "peephole": true,
    // L'inliner est toujours activé si aucun détail n'est donné,
    // utilisez les détails pour le désactiver.
    "inliner": true,
    // L'enlèvement du jumpdest inutilisé est toujours activé si aucun détail n'est_
↳ donné,
    // utilisez les détails pour le désactiver.
    "jumpdestRemover": true,
    // Réorganise parfois les littéraux dans les opérations commutatives.
    "orderLiterals": false,
    // Supprime les blocs de code dupliqués
    "deduplicate": false,
    // L'élimination des sous-expressions communes, c'est l'étape la plus compliquée_
↳ mais
    // peut également fournir le gain le plus important.
    "cse": false,
    // Optimiser la représentation des nombres littéraux et des chaînes de_
↳ caractères dans le code.
    "constantOptimizer": false,
    // Le nouvel optimiseur Yul. Opère principalement sur le code du codeur ABI v2
    // et de l'assemblage en ligne.
    // Il est activé en même temps que le réglage de l'optimiseur global
    // et peut être désactivé ici.
    // Avant Solidity 0.6.0, il devait être activé par ce commutateur.

```

(suite sur la page suivante)

```

    "yul": false,
    // Tuning options for the Yul optimizer.
    "yulDetails": {
        // Améliore l'allocation des emplacements de pile pour les variables, peut
↳ libérer les emplacements de pile plus tôt.
        // Activé par défaut si l'optimiseur Yul est activé.
        "stackAllocation": true,
        // Sélectionnez les étapes d'optimisation à appliquer.
        // Facultatif, l'optimiseur utilisera la séquence par défaut si elle est omise.
        "optimizerSteps": "dhfoDgvulfntUtnIf..."
    }
},
// Version de l'EVM pour laquelle il faut compiler.
// Affecte la vérification de type et la génération de code. Peut être homestead,
// tangerineWhistle, spuriousDragon, byzantium, constantinople, petersburg, istanbul
↳ ou berlin.
"evmVersion": "byzantium",
// Facultatif : Modifier le pipeline de compilation pour passer par la
↳ représentation intermédiaire de Yul.
// Il s'agit d'une fonctionnalité hautement EXPERIMENTALE, à ne pas utiliser en
↳ production. Elle est désactivée par défaut.
"viaIR": true,
// Facultatif : Paramètres de débogage
"debug": {
    // Comment traiter les chaînes de motifs de retour (et d'exigence). Les paramètres
↳ sont
    // "default", "strip", "debug" et "verboseDebug".
    // "default" n'injecte pas les chaînes revert générées par le compilateur et
↳ conserve celles fournies par l'utilisateur.
    // "strip" supprime toutes les chaînes revert (si possible, c'est-à-dire si des
↳ littéraux sont utilisés) en conservant les effets secondaires.
    // "debug" injecte des chaînes pour les revert internes générés par le compilateur,
↳ implémenté pour les encodeurs ABI V1 et V2 pour le moment.
    // "verboseDebug" ajoute même des informations supplémentaires aux chaînes de
↳ revert fournies par l'utilisateur (pas encore implémenté).
    "revertStrings": "default",
    // Facultatif : quantité d'informations de débogage supplémentaires à inclure dans
↳ les commentaires de l'EVM
    // produit et dans le code Yul. Les composants disponibles sont :
    // - `location` : Annotations de la forme `@src <index>:<start>:<end>` indiquant
    // l'emplacement de l'élément correspondant dans le fichier Solidity original, où :
    // - `<index>` est l'index du fichier correspondant à l'annotation `@use-src`,
    // - `<start>` est l'indice du premier octet à cet emplacement,
    // - `<end>` est l'indice du premier octet après cet emplacement.
    // - `snippet` : Un extrait de code d'une seule ligne provenant de l'emplacement
↳ indiqué par `@src`.
    // L'extrait est cité et suit l'annotation `@src` correspondante.
    // - `*` : Valeur joker qui peut être utilisée pour tout demander.
    "debugInfo": ["location", "snippet"]
},
// Paramètres des métadonnées (facultatif)

```

(suite de la page précédente)

```

"metadata": {
  // Utiliser uniquement le contenu littéral et non les URL (faux par défaut)
  "useLiteralContent": true,
  // Utilisez la méthode de hachage donnée pour le hachage des métadonnées qui est
↳ ajouté au bytecode.
  // Le hachage des métadonnées peut être supprimé du bytecode via l'option "none".
  // Les autres options sont "ipfs" et "bzzr1".
  // Si l'option est omise, "ipfs" est utilisé par défaut.
  "bytecodeHash": "ipfs"
},
// Adresses des bibliothèques. Si toutes les bibliothèques ne sont pas données ici,
// il peut en résulter des objets non liés dont les données de sortie sont
↳ différentes.
"libraries": {
  // La clé de premier niveau est le nom du fichier source dans lequel la
↳ bibliothèque est utilisée.
  // Si des remappages sont utilisés, ce fichier source doit correspondre au chemin
↳ global
  // après que les remappages aient été appliqués.
  // Si cette clé est une chaîne vide, cela fait référence à un niveau global.
  "myFile.sol": {
    "MyLib": "0x123123..."
  }
},
// Ce qui suit peut être utilisé pour sélectionner les sorties souhaitées en se
↳ basant
// sur les noms de fichiers et de contrats.
// Si ce champ est omis, alors le compilateur charge et effectue une vérification de
↳ type,
// mais ne générera aucune sortie en dehors des erreurs.
// La clé de premier niveau est le nom du fichier et la clé de second niveau est le
↳ nom du contrat.
// Un nom de contrat vide est utilisé pour les sorties qui ne sont pas liées à un
↳ contrat
// mais à l'ensemble du fichier source, comme l'AST.
// Une étoile comme nom de contrat fait référence à tous les contrats du fichier.
// De même, une étoile comme nom de fichier correspond à tous les fichiers.
// Pour sélectionner toutes les sorties que le compilateur peut éventuellement
↳ générer, utilisez
// "outputSelection : { "*" : { "*" : [ "*" ], "" : [ "*" ] } }"
// mais notez que cela pourrait ralentir inutilement le processus de compilation.
//
// Les types de sortie disponibles sont les suivants :
//
// Niveau fichier (nécessite une chaîne vide comme nom de contrat) :
// ast - AST de tous les fichiers sources
//
// Niveau du contrat (nécessite le nom du contrat ou "*") :
// abi - ABI
// devdoc - Documentation du développeur (natspec)
// userdoc - Documentation utilisateur (natspec)
// metadata - Métadonnées

```

(suite sur la page suivante)

```

// ir - Représentation intermédiaire Yul du code avant optimisation
// irOptimized - Représentation intermédiaire après optimisation
// storageLayout - Emplacements, décalages et types des variables d'état du contrat.
// evm.assembly - Nouveau format d'assemblage
// evm.legacyAssembly - Ancien format d'assemblage en JSON
// evm.bytecode.functionDebugData - Informations de débogage au niveau des fonctions.
// evm.bytecode.object - Objet bytecode
// evm.bytecode.opcodes - Liste d'opcodes
// evm.bytecode.sourceMap - Cartographie de la source (utile pour le débogage)
// evm.bytecode.linkReferences - Références de liens (si objet non lié)
// evm.bytecode.generatedSources - Sources générées par le compilateur.
// evm.deployedBytecode* - Bytecode déployé (a toutes les options que evm.bytecode a)
// evm.deployedBytecode.immutableReferences - Correspondance entre les identifiants.
↳AST et les plages de bytecode qui font référence aux immutables.
// evm.methodIdentifiers - La liste des hachages de fonctions
// evm.gasEstimates - Estimations des gaz de fonction
// ewasm.wasm - Ewasm au format S-expressions de WebAssembly
// ewasm.wasm - Ewasm au format binaire WebAssembly
//
// Notez que l'utilisation d'un `evm`, `evm.bytecode`, `ewasm`, etc. sélectionnera.
↳chaque
// partie cible de cette sortie. De plus, `*` peut être utilisé comme un joker pour.
↳tout demander.
//
"outputSelection": {
  "*": {
    "*": [
      "metadata", "evm.bytecode" // Activez les sorties de métadonnées et de.
↳bytecode de chaque contrat.
      , "evm.bytecode.sourceMap" // Activez la sortie de la carte des sources pour.
↳chaque contrat.
    ],
    "" : [
      "ast" // Active la sortie AST de chaque fichier.
    ]
  },
  // Active la sortie de l'abi et des opcodes de MonContrat définis dans le fichier.
↳def.
  "def": {
    "MyContract": [ "abi", "evm.bytecode.opcodes" ]
  }
},
// L'objet modelChecker est expérimental et sujet à des modifications.
"modelChecker":
{
  // Choisir les contrats qui doivent être analysés comme ceux qui sont déployés.
  "contracts":
  {
    "source1.sol": ["contract1"],
    "source2.sol": ["contract2", "contract3"]
  },
  // Choisir si les opérations de division et de modulo doivent être remplacées par

```

(suite sur la page suivante)

(suite de la page précédente)

```

// multiplication avec des variables de type slack. La valeur par défaut est `true`.
// L'utilisation de `false` ici est recommandée si vous utilisez le moteur CHC
// et que vous n'utilisez pas Spacer comme solveur de Horn (en utilisant Eldarica,
↳par exemple).
// Voir la section Vérification formelle pour une explication plus détaillée de
↳cette option.
"divModWithSlacks": true,
// Choisissez le moteur de vérification de modèle à utiliser : all (par défaut),
↳bmc, chc, none.
"engine": "chc",
// Choisissez quels types d'invariants doivent être signalés à l'utilisateur :
↳contrat, réentrance.
"invariants": ["contract", "reentrancy"],
// Choisissez si vous souhaitez afficher toutes les cibles non prouvées. La valeur
↳par défaut est `false`.
"showUnproved": true,
// Choisissez les solveurs à utiliser, s'ils sont disponibles.
// Voir la section Vérification formelle pour la description des solveurs.
"solvers": ["cvc4", "smtlib2", "z3"],
// Choisissez les cibles à vérifier : constantCondition,
// underflow, overflow, divByZero, balance, assert, popEmptyArray, outOfBounds.
// Si l'option n'est pas donnée, toutes les cibles sont vérifiées par défaut,
// sauf underflow/overflow pour Solidity >=0.8.7.
// Voir la section Vérification formelle pour la description des cibles.
"targets": ["underflow", "overflow", "assert"],
// Délai d'attente pour chaque requête SMT en millisecondes.
// Si cette option n'est pas donnée, le SMTChecker utilisera une limite de
↳ressources déterministe
// par défaut.
// Un délai d'attente de 0 signifie qu'il n'y a aucune restriction de ressources ou
↳de temps pour les requêtes.
"timeout": 20000
}
}
}

```

Description de la sortie

```

{
// Facultatif : non présent si aucune erreur/avis/infos n'a été rencontrée.
"errors": [
{
// Facultatif : Emplacement dans le fichier source.
"sourceLocation": {
"file": "sourceFile.sol",
"start": 0,
"end": 100
},
// Facultatif : Autres lieux (par exemple, lieux de déclarations conflictuelles)
"secondarySourceLocations": [

```

(suite sur la page suivante)

```

    {
      "file": "sourceFile.sol",
      "start": 64,
      "end": 92,
      "message": "L'autre déclaration est ici :"
    }
  ],
  // Obligatoire : Type d'erreur, tel que "TypeError", "InternalCompilerError",
  ↪ "Exception", etc.
  // Voir ci-dessous pour la liste complète des types.
  "type": "TypeError",
  // Obligatoire : Composant d'où provient l'erreur, tel que "general", "ewasm", etc.
  "component": "general",
  // Obligatoire ("error", "warning" ou "info", mais veuillez noter que cela
  ↪ pourrait être étendu à l'avenir)
  "severity": "error",
  // Facultatif : code unique pour la cause de l'erreur.
  "errorCode": "3141",
  // Obligatoire
  "message": "Mot clé invalide",
  // Facultatif : le message formaté avec l'emplacement de la source
  "formattedMessage": "sourceFile.sol:100: Invalid keyword"
}
],
// Il contient les sorties au niveau du fichier.
// Il peut être limité/filtré par les paramètres outputSelection.
"sources": {
  "sourceFile.sol": {
    // Identifiant de la source (utilisé dans les cartes de sources)
    "id": 1,
    // L'objet AST
    "ast": {}
  }
},
// Il contient les sorties au niveau du contrat.
// Il peut être limité/filtré par les paramètres outputSelection.
"contracts": {
  "sourceFile.sol": {
    // Si la langue utilisée ne comporte pas de noms de contrat, ce champ doit être
    ↪ égal à une chaîne vide.
    "ContractName": {
      // L'ABI du contrat Ethereum. S'il est vide, il est représenté comme un tableau
      ↪ vide.
      // See https://docs.soliditylang.org/en/develop/abi-spec.html
      "abi": [],
      // Voir la documentation sur la sortie des métadonnées (chaîne JSON sérialisée).
      "metadata": "{/* ... */}",
      // Documentation utilisateur (natspec)
      "userdoc": {},
      // Documentation pour les développeurs (natspec)
      "devdoc": {},
      // Représentation intermédiaire (chaîne de caractères)

```

(suite sur la page suivante)

(suite de la page précédente)

```

"ir": "",
// Voir la documentation sur l'agencement du stockage.
"storageLayout": {"storage": [/* ... */], "types": {/* ... */} },
// Sorties liées à l'EVM
"evm": {
  // Assemblée (chaîne de caractères)
  "assembly": "",
  // Assemblage à l'ancienne (objet)
  "legacyAssembly": {},
  // Bytecode et détails connexes.
  "bytecode": {
    // Débogage des données au niveau des fonctions.
    "functionDebugData": {
      // Suit maintenant un ensemble de fonctions incluant des fonctions
↳ définies par l'utilisateur.
      // L'ensemble ne doit pas nécessairement être complet.
      "@mint_13": { // Nom interne de la fonction
        "entryPoint": 128, // Décalage d'octet dans le bytecode où la fonction
↳ commence (facultatif)
        "id": 13, // AST ID de la définition de la fonction ou null pour les
↳ fonctions internes au compilateur (facultatif)
        "parameterSlots": 2, // Nombre d'emplacements de pile EVM pour les
↳ paramètres de fonction (facultatif)
        "returnSlots": 1 // Nombre d'emplacements de pile EVM pour les valeurs de
↳ retour (facultatif)
      }
    },
    // Le bytecode sous forme de chaîne hexagonale.
    "object": "00fe",
    // Liste des opcodes (chaîne de caractères)
    "opcodes": "",
    // Le mappage de la source sous forme de chaîne. Voir la définition du
↳ mappage de la source.
    "sourceMap": "",
    // Tableau des sources générées par le compilateur. Actuellement, il ne
    // contient qu'un seul fichier Yul.
    "generatedSources": [{
      // Yul AST
      "ast": {/* ... */},
      // Fichier source sous sa forme texte (peut contenir des commentaires)
      "contents": "{ function abi_decode(start, end) -> data { data :=
↳ calldataload(start) } }",
      // ID du fichier source, utilisé pour les références aux sources, même
↳ "namespace" que les fichiers sources de Solidity.
      "id": 2,
      "language": "Yul",
      "name": "#utility.yul"
    }
  ]},
  // S'il est donné, il s'agit d'un objet non lié.
  "linkReferences": {
    "libraryFile.sol": {
      // Décalage des octets dans le bytecode.

```

(suite sur la page suivante)

```
        // La liaison remplace les 20 octets qui s'y trouvent.
        "Library1": [
            { "start": 0, "length": 20 },
            { "start": 200, "length": 20 }
        ]
    }
},
"deployedBytecode": {
    /* ..., */ // La même disposition que ci-dessus.
    "immutableReferences": {
        // Il existe deux références à l'immuable avec l'ID AST 3, toutes deux d'une
↪ longueur de 32 octets. L'une se trouve
        // à l'offset 42 du bytecode, l'autre à l'offset 80 du bytecode.
        "3": [{ "start": 42, "length": 32 }, { "start": 80, "length": 32 }]
    }
},
// La liste des hachages de fonctions
"methodIdentifiers": {
    "delegate(address)": "5c19a95c"
},
// Estimation des gaz de fonction
"gasEstimates": {
    "creation": {
        "codeDepositCost": "420000",
        "executionCost": "infinite",
        "totalCost": "infinite"
    },
    "external": {
        "delegate(address)": "25000"
    },
    "internal": {
        "heavyLifting()": "infinite"
    }
}
},
// Sorties liées à l'Ewasm
"ewasm": {
    // Format des expressions S
    "wast": "",
    // Format binaire (chaîne hexagonale)
    "wasm": ""
}
}
}
```

Types d'erreurs

1. `JSONError` : L'entrée JSON n'est pas conforme au format requis, par exemple, l'entrée n'est pas un objet JSON, la langue n'est pas supportée, etc.
2. `IOError` : Erreurs de traitement des entrées/sorties et des importations, telles qu'une URL non résoluble ou une erreur de hachage dans les sources fournies.
3. `ParserError` : Le code source n'est pas conforme aux règles du langage.
4. `DocstringParsingError` : Les balises NatSpec du bloc de commentaires ne peuvent pas être analysées.
5. `SyntaxError` : Erreur de syntaxe, comme l'utilisation de « continue » en dehors d'une boucle « for ».
6. `DeclarationError` : Noms d'identifiants invalides, impossibles à résoudre ou contradictoires. Par exemple, « Identifiant non trouvé ».
7. `TypeError` : Erreur dans le système de types, comme des conversions de types invalides, des affectations invalides, etc.
8. `UnimplementedFeatureError` : La fonctionnalité n'est pas supportée par le compilateur, mais devrait l'être dans les futures versions.
9. `InternalCompilerError` : Bogue interne déclenché dans le compilateur - il doit être signalé comme un problème.
10. `Exception` : Echec inconnu lors de la compilation - ceci devrait être signalé comme un problème.
11. `CompilerError` : Utilisation non valide de la pile du compilateur - ceci devrait être signalé comme un problème.
12. `FatalError` : Une erreur fatale n'a pas été traitée correctement - ceci devrait être signalé comme un problème.
13. `Warning` : Un avertissement, qui n'a pas arrêté la compilation, mais qui devrait être traité si possible.
14. `Info` : Une information que le compilateur pense que l'utilisateur pourrait trouver utile, mais qui n'est pas dangereuse et ne doit pas nécessairement être traitée.

3.12.4 Outils de compilation

`solidity-upgrade`

`solidity-upgrade` peut vous aider à mettre à jour semi-automatiquement vos contrats en fonction des changements de langue. Bien qu'il n'implémente pas et ne puisse pas implémenter tous changements requis pour chaque version de rupture, il prend en charge ceux qui, autrement, nécessiteraient de nombreux ajustements manuels répétitifs.

Note : `solidity-upgrade` effectue une grande partie du travail, mais vos contrats nécessiteront très probablement d'autres ajustements manuels. Nous vous recommandons d'utiliser un système de contrôle de version pour vos fichiers. Cela permet de réviser et éventuellement de revenir en arrière sur les modifications apportées.

Avertissement : `solidity-upgrade` n'est pas considéré comme complet ou exempt de bogues, donc veuillez l'utiliser avec précaution.

Comment cela fonctionne

Vous pouvez passer un ou plusieurs fichiers sources Solidity à `solidity-upgrade [files]`. Si ceux-ci utilisent l'instruction `import` qui fait référence à des fichiers en dehors du répertoire du fichier source actuel, vous devez spécifier des répertoires qui sont autorisés à lire et à importer des fichiers, en passant l'instruction `--allow-paths [directory]`. Vous pouvez ignorer les fichiers manquants en passant `--ignore-missing`.

`solidity-upgrade` est basé sur `libsolidity` et peut analyser vos fichiers sources, et peut y trouver des mises à jour applicables.

Les mises à jour de source sont considérées comme de petits changements textuels à votre code source. Elles sont appliquées à une représentation en mémoire des fichiers sources donnés. Le fichier source correspondant est mis à jour par défaut, mais vous pouvez passer la commande `--dry-run` pour simuler l'ensemble du processus de mise à jour sans écrire dans aucun fichier.

Le processus de mise à jour lui-même a deux phases. Dans la première phase, les fichiers sources sont analysés, et puisqu'il n'est pas possible de mettre à jour le code source à ce niveau, les erreurs sont collectées et peuvent être enregistrées en passant `--verbose`. Aucune mise à jour de la source n'est disponible à ce stade.

Dans la deuxième phase, toutes les sources sont compilées et tous les modules d'analyse de mise à niveau activés sont exécutés en même temps que la compilation. Par défaut, tous les modules disponibles sont activés. Veuillez lire la documentation sur les *modules disponibles* pour plus de détails.

Cela peut entraîner des erreurs de compilation qui peuvent être corrigées par des mises à jour des sources. Si aucune erreur ne se produit, aucune mise à niveau des sources n'est signalée et vous avez terminé. Si des erreurs se produisent et qu'un module de mise à niveau a signalé une mise à niveau de la source, la première source, la première signalée est appliquée et la compilation est déclenchée à nouveau pour tous les fichiers sources donnés. L'étape précédente est répétée aussi longtemps que des mises à jour de sources sont signalées. Si des erreurs surviennent encore, vous pouvez les enregistrer en passant le paramètre `--verbose`. Si aucune erreur ne se produit, vos contrats sont à jour et peuvent être compilés avec la dernière version du compilateur.

Modules de mise à niveau disponibles

Module	Version	Description
constructor	0.5.0	Les constructeurs doivent maintenant être définis à l'aide du mot-clé « constructeur ».
visibility	0.5.0	La visibilité explicite des fonctions est désormais obligatoire, La valeur par défaut est <code>public</code> .
abstract	0.6.0	Le mot-clé <code>abstract</code> doit être utilisé si le contrat ne met pas en œuvre toutes ses fonctions.
virtual	0.6.0	Fonctions sans implémentation en dehors d'un doivent être marquées <code>virtual</code> .
override	0.6.0	Lorsque vous remplacez une fonction ou un modificateur, la nouvelle fonction le mot clé <code>override</code> doit être utilisé.
dotsyntax	0.7.0	La syntaxe suivante est obsolète : <code>f.gas(...)</code> , <code>f.value(...)</code> et <code>(new C).value(...)</code> . Remplacez ces appels par <code>f{gas: ..., value: ...}</code> et <code>(new C){value: ...}</code> .
now	0.7.0	Le mot clé <code>now</code> est obsolète. Utilisez <code>block.timestamp</code> à la place.
constructor-visibility	0.7.0	Supprime la visibilité des constructeurs.

Veillez lire [0.5.0 notes de mise à jour](#), [0.6.0 notes de mise à jour](#), [0.7.0 notes de mise à jour](#) et [0.8.0 notes de mise à jour](#) pour plus de détails.

Synopsis

```
Usage: solidity-upgrade [options] contract.sol
```

Allowed options:

```
--help           Show help message and exit.
--version        Show version and exit.
--allow-paths path(s)
                  Allow a given path for imports. A list of paths can be
                  supplied by separating them with a comma.
--ignore-missing Ignore missing files.
--modules module(s)
                  Only activate a specific upgrade module. A list of
                  modules can be supplied by separating them with a comma.
--dry-run        Apply changes in-memory only and don't write to input
                  file.
```

(suite sur la page suivante)

<code>--verbose</code>	Print logs, errors and changes. Shortens output of upgrade patches.
<code>--unsafe</code>	Accept <i>*unsafe*</i> changes.

Rapports de bogue / Demandes de fonctionnalités

Si vous avez trouvé un bogue ou si vous avez une demande de fonctionnalité, veuillez [déposer une question](#) sur Github.

Exemple

Supposons que vous ayez le contrat suivant dans `Source.sol` :

```
pragma solidity >=0.6.0 <0.6.4;
// Ceci ne compilera pas après la version 0.7.0.
// SPDX-License-Identifier: GPL-3.0
contract C {
    // FIXME : supprimer la visibilité du constructeur et rendre le contrat
    ↪️abstrait.
    constructor() internal {}
}

contract D {
    uint time;

    function f() public payable {
        // FIXME : remplacer maintenant par block.timestamp
        time = now;
    }
}

contract E {
    D d;

    // FIXME : supprimer la visibilité du constructeur
    constructor() public {}

    function g() public {
        // FIXME : change .value(5) => {value : 5}
        d.f.value(5)();
    }
}
```

Changements requis

Le contrat ci-dessus ne sera pas compilé à partir de la version 0.7.0. Pour mettre le contrat à jour avec la version actuelle de Solidity, les modules de mise à jour suivants doivent être exécutés : `constructor-visibility`, `now` et `dotsyntax`. Veuillez lire la documentation sur *modules disponibles* pour plus de détails.

Exécution de la mise à niveau

Il est recommandé de spécifier explicitement les modules de mise à niveau en utilisant l'argument `--modules`.

```
solidity-upgrade --modules constructor-visibility,now,dotsyntax Source.sol
```

The command above applies all changes as shown below. Please review them carefully (the pragmas will have to be updated manually.)

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
abstract contract C {
    // FIXME : supprimer la visibilité du constructeur et rendre le contrat
    ↪abstrait.
    constructor() {}
}

contract D {
    uint time;

    function f() public payable {
        // FIXME : remplacer maintenant par block.timestamp
        time = block.timestamp;
    }
}

contract E {
    D d;

    // FIXME : supprimer la visibilité du constructeur
    constructor() {}

    function g() public {
        // FIXME : change .value(5) => {value : 5}
        d.f{value: 5}();
    }
}
```

3.13 Analyse de la sortie du compilateur

Il est souvent utile d'examiner le code d'assemblage généré par le compilateur. Le binaire généré, c'est-à-dire la sortie de `solc --bin contract.sol`, est généralement difficile à lire. Il est recommandé d'utiliser l'indicateur `--asm` pour analyser la sortie de l'assemblage. Même pour les gros contrats, regarder une visualisation de l'assemblage avant et après un changement est souvent très instructif.

Considérons le contrat suivant (nommé, disons `contract.sol`) :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
contract C {
    function one() public pure returns (uint) {
        return 1;
    }
}
```

Voici le résultat de l'opération « `solc --asm contract.sol` ».

```
===== contract.sol:C =====
EVM assembly:
    /* "contract.sol":0:86 contract C {... */
    mstore(0x40, 0x80)
    callvalue
    dup1
    iszero
    tag_1
    jumpi
    0x00
    dup1
    revert
tag_1:
    pop
    dataSize(sub_0)
    dup1
    dataOffset(sub_0)
    0x00
    codecopy
    0x00
    return
stop

sub_0: assembly {
    /* "contract.sol":0:86 contract C {... */
    mstore(0x40, 0x80)
    callvalue
    dup1
    iszero
    tag_1
    jumpi
    0x00
    dup1
    revert
```

(suite sur la page suivante)

(suite de la page précédente)

```

tag_1:
  pop
  jumpi(tag_2, lt(calldatasize, 0x04))
  shr(0xe0, calldataload(0x00))
  dup1
  0x901717d1
  eq
  tag_3
  jumpi
tag_2:
  0x00
  dup1
  revert
  /* "contract.sol":17:84  function one() public pure returns (uint) {... */
tag_3:
  tag_4
  tag_5
  jump // in
tag_4:
  mload(0x40)
  tag_6
  swap2
  swap1
  tag_7
  jump // in
tag_6:
  mload(0x40)
  dup1
  swap2
  sub
  swap1
  return
tag_5:
  /* "contract.sol":53:57  uint */
  0x00
  /* "contract.sol":76:77  1 */
  0x01
  /* "contract.sol":69:77  return 1 */
  swap1
  pop
  /* "contract.sol":17:84  function one() public pure returns (uint) {... */
  swap1
  jump // out
  /* "#utility.yul":7:125  */
tag_10:
  /* "#utility.yul":94:118  */
  tag_12
  /* "#utility.yul":112:117  */
  dup2
  /* "#utility.yul":94:118  */
  tag_13
  jump // in

```

(suite sur la page suivante)

```
tag_12:
  /* "#utility.yul":89:92 */
  dup3
  /* "#utility.yul":82:119 */
  mstore
  /* "#utility.yul":72:125 */
  pop
  pop
  jump // out
  /* "#utility.yul":131:353 */
tag_7:
  0x00
  /* "#utility.yul":262:264 */
  0x20
  /* "#utility.yul":251:260 */
  dup3
  /* "#utility.yul":247:265 */
  add
  /* "#utility.yul":239:265 */
  swap1
  pop
  /* "#utility.yul":275:346 */
tag_15
  /* "#utility.yul":343:344 */
  0x00
  /* "#utility.yul":332:341 */
  dup4
  /* "#utility.yul":328:345 */
  add
  /* "#utility.yul":319:325 */
  dup5
  /* "#utility.yul":275:346 */
tag_10
  jump // in
tag_15:
  /* "#utility.yul":229:353 */
  swap3
  swap2
  pop
  pop
  jump // out
  /* "#utility.yul":359:436 */
tag_13:
  0x00
  /* "#utility.yul":425:430 */
  dup2
  /* "#utility.yul":414:430 */
  swap1
  pop
  /* "#utility.yul":404:436 */
  swap2
  swap1
```

(suite sur la page suivante)

(suite de la page précédente)

```

    pop
    jump // out

    auxdata: ↵
↵0xa2646970667358221220a5874f19737ddd4c5d77ace1619e5160c67b3d4bedac75fce908fed32d98899864736f6c6378273
}

```

Alternativement, la sortie ci-dessus peut également être obtenue à partir de [Remix](#), sous l'option « Compilation Details » après avoir compilé un contrat.

Remarquez que la sortie asm commence par le code de création / constructeur. Le code de déploiement est fourni comme partie du sous-objet (dans l'exemple ci-dessus, il fait partie du sous-objet `sub_0`). Le champ `auxdata` `` correspond au contrat `:ref:metadata <encodage des métadonnées dans le bytecode>`. Les commentaires dans la sortie de l'assemblage pointent vers la emplacement de la source. Notez que `` `#utility.yul` est un fichier généré en interne de fonctions utilitaires qui peut être obtenu en utilisant les drapeaux `--combined-json generated-sources, generated-sources-runtime`.

De même, l'assemblage optimisé peut être obtenu avec la commande `solc --optimize --asm contract.sol`. Souvent, il est intéressant de voir si deux sources différentes dans Solidity aboutissent au même code optimisé. le même code optimisé. Par exemple, pour voir si les expressions $(a * b) / c$, $a * b / c$ génèrent le même bytecode. Cela peut être facilement fait en prenant un diff de la sortie assembleur correspondante, après avoir éventuellement supprimé les commentaires. d'assemblage correspondant, après avoir éventuellement supprimé les commentaires qui font référence aux emplacements des sources.

Note : La sortie `--asm` n'est pas conçue pour être lisible par une machine. Par conséquent, il peut y avoir des changements de rupture sur la sortie entre les versions mineures de `solc`.

3.14 Changements apportés au Codegen basé sur Solidity IR

Solidity peut générer du bytecode EVM de deux manières différentes : Soit directement de Solidity vers les opcodes EVM (« old codegen »), soit par le biais d'une représentation intermédiaire (« IR ») dans Yul (« new codegen » ou « IR-based codegen »).

Le générateur de code basé sur l'IR a été introduit dans le but non seulement de permettre génération de code plus transparente et plus vérifiable, mais aussi de permettre des passes d'optimisation plus puissantes qui couvrent plusieurs fonctions.

Actuellement, le générateur de code basé sur IR est toujours marqué comme expérimental, mais il supporte toutes les fonctionnalités du langage et a fait l'objet de nombreux tests. Nous considérons donc qu'il est presque prêt à être utilisé en production.

Vous pouvez l'activer sur la ligne de commande en utilisant `--experimental-via-ir` ou avec l'option `{"viaIR" : true}` dans le standard-json et nous encourageons tout le monde à l'essayer !

Pour plusieurs raisons, il existe de minuscules différences sémantiques entre l'ancien générateur de code basé sur l'IR, principalement dans des domaines où nous ne nous attendons pas à ce que les gens se fient à ce comportement de toute façon. Cette section met en évidence les principales différences entre l'ancien et le générateur de code basé sur la RI.

3.14.1 Changements uniquement sémantiques

Cette section énumère les changements qui sont uniquement sémantiques, donc potentiellement cacher un comportement nouveau et différent dans le code existant.

- Lorsque les structures de stockage sont supprimées, chaque emplacement de stockage qui contient un membre de la structure est entièrement mis à zéro. Auparavant, l'espace de remplissage n'était pas modifié. Par conséquent, si l'espace de remplissage dans une structure est utilisé pour stocker des données (par exemple, dans le contexte d'une mise à jour de contrat), vous devez être conscient que `delete` effacera maintenant aussi le membre ajouté (alors qu'il n'aurait pas été effacé dans le passé).

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1;

contract C {
    struct S {
        uint64 y;
        uint64 z;
    }
    S s;
    function f() public {
        // ...
        delete s;
        // s occupe seulement les 16 premiers octets de l'emplacement de 32 octets.
        // delete écrira zéro dans l'emplacement complet
    }
}
```

Nous avons le même comportement pour la suppression implicite, par exemple lorsque le tableau de structs est raccourci.

- Les modificateurs de fonction sont mis en œuvre d'une manière légèrement différente en ce qui concerne les paramètres de fonction et les variables de retour. Cela a notamment un effet si le caractère générique `_`; est évalué plusieurs fois dans un modificateur. Dans l'ancien générateur de code, chaque paramètre de fonction et variable de retour a un emplacement fixe sur la pile. Si la fonction est exécutée plusieurs fois parce que `_`; est utilisé plusieurs fois ou utilisé dans une boucle, alors un changement de la valeur du paramètre de fonction ou de la variable de retour est visible lors de la prochaine exécution de la fonction. Le nouveau générateur de code implémente les modificateurs à l'aide de fonctions réelles et transmet les paramètres de fonction. Cela signifie que plusieurs évaluations du corps d'une fonction obtiendront les mêmes valeurs pour les paramètres, et l'effet sur les variables de retour est qu'elles sont réinitialisées à leur valeur par défaut (zéro) à chaque exécution.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0;
contract C {
    function f(uint _a) public pure mod() returns (uint _r) {
        _r = _a++;
    }
    modifier mod() { _; _; }
}
```

Si vous exécutez `f(0)` dans l'ancien générateur de code, il retournera 2, alors qu'il retournera 1 en utilisant le nouveau générateur de code.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

contract C {
```

(suite sur la page suivante)

(suite de la page précédente)

```

bool active = true;
modifier mod()
{
    _;
    active = false;
    _;
}
function foo() external mod() returns (uint ret)
{
    if (active)
        ret = 1; // Same as `return 1`
}
}

```

La fonction C. `foo()` renvoie les valeurs suivantes :

- Ancien générateur de code : 1 comme variable de retour est initialisé à 0 une seule fois avant la première évaluation `_;` et ensuite écrasée par la variable `return 1;`. Elle n'est pas initialisée à nouveau pour la seconde évaluation et `foo()` ne l'assigne pas explicitement non plus (à cause de `active == false`), il garde donc sa première valeur.
- Nouveau générateur de code : 0 car tous les paramètres, y compris les paramètres de retour, seront ré-initialisés avant chaque évaluation `_;`.
- L'ordre d'initialisation des contrats a changé en cas d'héritage. L'ordre était auparavant le suivant :
 - Toutes les variables d'état sont initialisées à zéro au début.
 - Évaluer les arguments du constructeur de base du contrat le plus dérivé au contrat le plus basique.
 - Initialiser toutes les variables d'état dans toute la hiérarchie d'héritage, de la plus basique à la plus dérivée.
 - Exécuter le constructeur, s'il est présent, pour tous les contrats dans la hiérarchie linéarisée du plus bas au plus dérivé.
- Nouvel ordre :
 - Toutes les variables d'état sont initialisées à zéro au début.
 - Évaluer les arguments du constructeur de base du contrat le plus dérivé au contrat le plus basique.
 - Pour chaque contrat dans l'ordre du plus basique au plus dérivé dans la hiérarchie linéarisée, exécuter :
 1. Si elles sont présentes à la déclaration, les valeurs initiales sont assignées aux variables d'état.
 2. Le constructeur, s'il est présent.

Cela entraîne des différences dans certains contrats, par exemple :

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1;

contract A {
    uint x;
    constructor() {
        x = 42;
    }
    function f() public view returns(uint256) {
        return x;
    }
}

contract B is A {
    uint public y = f();
}

```

Auparavant, y était fixé à 0. Cela est dû au fait que nous initialisons d'abord les variables d'état : D'abord,

x est mis à 0, et lors de l'initialisation de y, f() renvoie 0, ce qui fait que y est également 0. Avec les nouvelles règles, y sera fixé à 42. Nous commençons par initialiser x à 0, puis nous appelons le constructeur de A qui fixe x à 42. Enfin, lors de l'initialisation de y, f() renvoie 42, ce qui fait que y est 42.

- La copie de tableaux d'« octets » de la mémoire vers le stockage est implémentée d'une manière différente. L'ancien générateur de code copie toujours des mots entiers, alors que le nouveau coupe le tableau d'octets après sa fin. L'ancien comportement peut conduire à ce que des données sales soient copiées après la fin du tableau (mais toujours dans le même emplacement de stockage). Cela entraîne des différences dans certains contrats, par exemple :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;

contract C {
    bytes x;
    function f() public returns (uint _r) {
        bytes memory m = "tmp";
        assembly {
            mstore(m, 8)
            mstore(add(m, 32), "deadbeef15dead")
        }
        x = m;
        assembly {
            _r := sload(x.slot)
        }
    }
}

Auparavant, `f()` renvoyait
→ `0x646561646265656663135646561640000000000000000000000000000000000000000010`
```

(il a une longueur correcte, et les 8 premiers éléments sont corrects, mais ensuite il contient des données sales qui ont été définies via l'assemblage). Maintenant, il renvoie 0x64656164626565660000000000000000000000000000000000000000000000000000000010 (il a une longueur correcte, et des éléments corrects, mais il ne contient pas de données superflues).

- Pour l'ancien générateur de code, l'ordre d'évaluation des expressions n'est pas spécifié. Pour le nouveau générateur de code, nous essayons d'évaluer dans l'ordre de la source (de gauche à droite), mais nous ne le garantissons pas. Cela peut conduire à des différences sémantiques. Par exemple :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function preincr_u8(uint8 _a) public pure returns (uint8) {
        return ++_a + _a;
    }
}
```

La fonction preincr_u8(1) retourne les valeurs suivantes :

- Ancien générateur de code : 3 (1 + 2) mais la valeur de retour n'est pas spécifiée en général.
 - Nouveau générateur de code : 4 (2 + 2) mais la valeur de retour n'est pas garantie
- D'autre part, les expressions des arguments de fonction sont évaluées dans le même ordre par les deux générateurs de code, à l'exception des fonctions globales addmod et mulmod. Par exemple :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function add(uint8 _a, uint8 _b) public pure returns (uint8) {
        return _a + _b;
    }
    function g(uint8 _a, uint8 _b) public pure returns (uint8) {
        return add(++_a + ++_b, _a + _b);
    }
}
```

La fonction `g(1, 2)` renvoie les valeurs suivantes :

— Ancien générateur de code : `10` (`add(2 + 3, 2 + 3)`) mais la valeur de retour n'est pas spécifiée en général.

— Nouveau générateur de code : `10` mais la valeur de retour n'est pas garantie

Les arguments des fonctions globales `addmod` et `mulmod` sont évalués de droite à gauche par l'ancien générateur de code et de gauche à droite par le nouveau générateur de code. Par exemple :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function f() public pure returns (uint256 aMod, uint256 mMod) {
        uint256 x = 3;
        // Old code gen: add/mulmod(5, 4, 3)
        // New code gen: add/mulmod(4, 5, 5)
        aMod = addmod(++x, ++x, x);
        mMod = mulmod(++x, ++x, x);
    }
}
```

La fonction `f()` renvoie les valeurs suivantes :

— Ancien générateur de code : `» aMod = 0 »` et `» mMod = 2 «`.

— Nouveau générateur de code : `» aMod = 4 »` et `» mMod = 0 «`.

- Le nouveau générateur de code impose une limite dure de `type(uint64).max (0xffffffffffffffff)` pour le pointeur de mémoire libre. Les allocations qui augmenteraient sa valeur au-delà de cette limite. L'ancien générateur de code n'a pas n'a pas cette limite.

Par exemple :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >0.8.0;
contract C {
    function f() public {
        uint[] memory arr;
        // allocation size: 576460752303423481
        // assumes freeMemPtr points to 0x80 initially
        uint solYulMaxAllocationBeforeMemPtrOverflow = (type(uint64).max - 0x80 -
↪31) / 32;
        // freeMemPtr overflows UINT64_MAX
        arr = new uint[](solYulMaxAllocationBeforeMemPtrOverflow);
    }
}
```

La fonction `f()` se comporte comme suit :

- Ancien générateur de code : manque de gaz lors de la mise à zéro du contenu du tableau après la grande allocation de mémoire.

- Nouveau générateur de code : retour en arrière en raison d'un débordement du pointeur de mémoire libre (ne tombe pas en panne sèche).

3.14.2 Internes

Pointeurs de fonctions internes

L'ancien générateur de code utilise des décalages de code ou des balises pour les valeurs des pointeurs de fonctions internes. Ceci est particulièrement compliqué car ces offsets sont différents au moment de la construction et après le déploiement et les valeurs peuvent traverser cette frontière via le stockage. Pour cette raison, les deux offsets sont codés au moment de la construction dans la même valeur (dans différents octets).

Dans le nouveau générateur de code, les pointeurs de fonction utilisent des ID internes qui sont alloués en séquence. Comme les appels via des pointeurs de fonction doivent toujours utiliser une fonction de distribution interne qui utilise l'instruction `switch` pour sélectionner la bonne fonction.

L'ID 0 est réservé aux pointeurs de fonction non initialisés qui provoquent une panique dans la fonction de répartition lorsqu'ils sont appelés.

Dans l'ancien générateur de code, les pointeurs de fonctions internes sont initialisés avec une fonction spéciale qui provoque toujours une panique. Cela provoque une écriture en mémoire au moment de la construction pour les pointeurs de fonctions internes en mémoire.

Nettoyage

L'ancien générateur de code n'effectue le nettoyage qu'avant une opération dont le résultat pourrait être affecté par les valeurs des bits sales. Le nouveau générateur de code effectue le nettoyage après toute opération qui peut entraîner des bits sales. L'espoir est que l'optimiseur sera suffisamment puissant pour éliminer les opérations de nettoyage redondantes.

Par exemple :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function f(uint8 _a) public pure returns (uint _r1, uint _r2)
    {
        _a = ~_a;
        assembly {
            _r1 := _a
        }
        _r2 = _a;
    }
}
```

La fonction `f(1)` renvoie les valeurs suivantes :

- Ancien générateur de code : (fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe, 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000fe)
- Nouveau générateur de codes : (000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000fe, 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000fe)

Notez que, contrairement au nouveau générateur de code, l'ancien générateur de code n'effectue pas de nettoyage après l'affectation bit-non (`_a = ~_a`). Il en résulte que des valeurs différentes sont assignées (dans le bloc d'assemblage en ligne) à la valeur de retour `_r1` entre l'ancien et le nouveau générateur de code. Cependant, les deux générateurs de code effectuent un nettoyage avant que la nouvelle valeur de `_a` soit assignée à `_r2`.

3.15 Disposition des variables d'état dans le stockage

Les variables d'état des contrats sont stockées dans le stockage d'une manière compacte telle que plusieurs valeurs utilisent parfois le même emplacement de stockage. À l'exception des tableaux de taille dynamique et des mappings (voir ci-dessous), les données sont stockées contiguës, élément après élément, en commençant par la première variable d'état, qui est stockée dans l'emplacement 0. Pour chaque variable une taille en octets est déterminée en fonction de son type. Les éléments multiples et contigus qui nécessitent moins de 32 octets sont regroupés dans un seul emplacement de stockage si possible, selon les règles suivantes :

- Le premier élément d'un emplacement de stockage est stocké avec un alignement d'ordre inférieur.
- Les types de valeurs n'utilisent que le nombre d'octets nécessaires à leur stockage.
- Si un type de valeur ne tient pas dans la partie restante d'un emplacement de stockage, il est stocké dans l'emplacement de stockage suivant.
- Les structures et les tableaux de données commencent toujours par un nouvel emplacement et leurs éléments sont serrés selon ces règles.
- Les éléments qui suivent les données de structure ou de tableau commencent toujours par un nouvel emplacement de stockage.

Pour les contrats qui utilisent l'héritage, l'ordre des variables d'état est déterminé par l'ordre linéaire C3 des contrats, en commençant par le contrat le plus basique. Si les règles ci-dessus le permettent, les variables d'état de différents contrats partagent le même emplacement de stockage.

Les éléments des structs et des arrays sont stockés les uns après les autres, comme des valeurs individuelles.

Avertissement : Lorsque vous utilisez des éléments qui sont plus petits que 32 octets, la consommation de gaz de votre contrat peut être plus élevée. Cela est dû au fait que l'EVM fonctionne sur 32 octets à la fois. Par conséquent, si l'élément est plus petit que cela, l'EVM doit utiliser plus d'opérations pour réduire la taille de l'élément de 32 octets à la taille souhaitée.

Il peut être avantageux d'utiliser des types de taille réduite si vous traitez des valeurs de stockage car le compilateur regroupera plusieurs éléments dans un emplacement de stockage et combinera ainsi plusieurs lectures ou écritures en une seule opération. Si vous ne lisez pas ou n'écrivez pas toutes les valeurs d'un slot en même temps, cela peut avoir l'effet inverse : Lorsqu'une valeur est écrite dans un emplacement de stockage à valeurs multiples, l'emplacement de stockage doit être lu en premier et ensuite combinée avec la nouvelle valeur, de sorte que les autres données du même emplacement ne soient pas détruites.

Lorsqu'il s'agit d'arguments de fonction ou de valeurs, il n'y a pas d'avantage inhérent car le compilateur n'empaquette pas ces valeurs.

Enfin, pour permettre à l'EVM d'optimiser cela, assurez-vous d'essayer d'ordonner vos variables de stockage et les membres de `struct`, de manière à ce qu'ils puissent être empaquetés de façon serrée. Par exemple, déclarer vos variables de stockage dans l'ordre suivant : `uint128, uint128, uint256` au lieu de `uint128, uint256, uint128`, car la première n'occupera que deux emplacements de stockage, alors que l'autre en occupera trois.

Note : La disposition des variables d'état dans le stockage est considérée comme faisant partie de l'interface externe de Solidity, en raison du fait que les pointeurs de stockage peuvent être transmis aux bibliothèques. Cela signifie que tout changement des règles décrites dans cette section est considéré comme un changement de rupture du langage et, en raison de sa nature critique, doit être considéré très attentivement avant d'être exécuté.

3.15.1 Mappings et tableaux dynamiques

En raison de leur taille imprévisible, les mappings et les types de tableaux de taille dynamique ne peuvent être stockés qu'« entre » les variables d'état qui les précèdent et les suivent. Au lieu de cela, ils sont considérés comme n'occupant que 32 octets au regard des *règles ci-dessus* et les éléments qu'ils contiennent sont stockés à partir d'un différent emplacement de stockage qui est calculé à l'aide d'un hachage Keccak-256.

Supposons que l'emplacement de stockage du mappage ou du tableau finisse par être un slot p après avoir appliqué *les règles de disposition du stockage*. Pour les tableaux dynamiques, ce slot stocke le nombre d'éléments dans le tableau (les tableaux d'octets et les chaînes de caractères sont une exception, voir *ci-dessous*). Pour les mappings, le slot reste vide, mais il est toujours nécessaire pour garantir que même s'il y a deux mappings l'un à côté de l'autre, leur contenu se retrouve à des emplacements de stockage différents.

Les données du tableau sont situées à partir de $\text{keccak256}(p)$ et sont disposées de la même manière que les données de tableau de taille statique : Un élément après l'autre, partageant potentiellement des emplacements de stockage si les éléments ne dépassent pas 16 octets. Les tableaux dynamiques de tableaux dynamiques appliquent cette règle de manière récursive. L'emplacement de l'élément $x[i][j]$, où le type de x est $\text{uint24}[][]$, est calculé comme suit (en supposant à nouveau que x est lui-même stocké dans l'emplacement p) : L'emplacement est $\text{keccak256}(\text{keccak256}(p) + i) + \text{floor}(j / \text{floor}(256 / 24))$ et l'élément peut être obtenu à partir des données de l'emplacement v en utilisant $(v \gg ((j \% \text{floor}(256 / 24))) * 24)) \& \text{type}(\text{uint24}).\text{max}$.

La valeur correspondant à une clé de mappage k est située à $\text{keccak256}(h(k) \cdot p)$ où \cdot est la concaténation et h est une fonction qui est appliquée à la clé en fonction de son type :

- pour les types de valeurs, h compacte la valeur à 32 octets de la même manière que lors du stockage de la valeur en mémoire.
- pour les chaînes de caractères et les tableaux d'octets, h calcule le hachage keccak256 des données non paginées.

Si la valeur du mappage est un type non-valeur, l'emplacement calculé marque le début des données. Si la valeur est de type struct, par exemple, vous devez ajouter un offset correspondant au membre struct pour atteindre le membre.

À titre d'exemple, considérons le contrat suivant :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract C {
    struct S { uint16 a; uint16 b; uint256 c; }
    uint x;
    mapping(uint => mapping(uint => S)) data;
}
```

Calculons l'emplacement de stockage de $\text{data}[4][9].c$. La position de la cartographie elle-même est 1 (la variable x de 32 octets la précède). Cela signifie que $\text{data}[4]$ est stocké à $\text{keccak256}(\text{uint256}(4) \cdot \text{uint256}(1))$. Le type des données[4] est à nouveau un mappage et les données de $\text{data}[4][9]$ commencent à l'emplacement $\text{keccak256}(\text{uint256}(9) \cdot \text{keccak256}(\text{uint256}(4) \cdot \text{uint256}(1)))$. Le décalage de l'emplacement du membre c dans la structure S est 1 parce que a et b sont emballés dans un seul slot. Cela signifie que l'emplacement de $\text{data}[4][9].c$ est $\text{keccak256}(\text{uint256}(9) \cdot \text{keccak256}(\text{uint256}(4) \cdot \text{uint256}(1))) + 1$. Le type de la valeur est uint256 , elle utilise donc un seul slot.

bytes et string

`bytes` et `string` sont encodés de manière identique. En général, le codage est similaire à celui de `bytes1[]`, dans le sens où il y a un slot pour le tableau lui-même et une zone de données qui est calculée en utilisant un hachage `keccak256` de la position de ce slot. Cependant, pour les valeurs courtes (inférieures à 32 octets), les éléments du tableau sont stockés avec la longueur dans le même slot.

En particulier, si les données ont une longueur maximale de 31 octets, les éléments sont stockés dans les octets d'ordre supérieur (alignés à gauche) et l'octet d'ordre inférieur stocke la valeur `longueur * 2`. Pour les tableaux d'octets qui stockent des données d'une longueur de 32 octets ou plus, l'emplacement principal `p` stocke la valeur `length * 2 + 1` et les données sont stockées comme d'habitude dans `keccak256(p)`. Cela signifie que vous pouvez distinguer un tableau court d'un tableau long en vérifiant si le bit le plus bas est activé : court (non activé) et long (activé).

Note : La gestion des slots codés de manière invalide n'est actuellement pas prise en charge mais pourrait être ajoutée à l'avenir. Si vous compilez via le pipeline expérimental du compilateur basé sur l'IR, la lecture d'un slot non codé invalide entraîne une erreur `Panic(0x22)`.

3.15.2 Sortie JSON

La disposition de stockage d'un contrat peut être demandée via l'interface *standard JSON*. La sortie est un objet JSON contenant deux clés, `storage` et `types`. L'objet `storage` est un tableau où chaque élément a la forme suivante :

```
{
  "astId": 2,
  "contract": "fileA:A",
  "label": "x",
  "offset": 0,
  "slot": "0",
  "type": "t_uint256"
}
```

L'exemple ci-dessus est la disposition de stockage du contrat `A { uint x ; }` de l'unité source `fileA` et :

- `astId` est l'identifiant du noeud AST de la déclaration de la variable d'état.
- `contract` est le nom du contrat, y compris son chemin d'accès comme préfixe
- `label` est le nom de la variable d'état
- `offset` est le décalage en octets dans le slot de stockage selon l'encodage
- `slot` est l'emplacement de stockage où la variable d'état réside ou commence. Cette adresse nombre peut être très grand, c'est pourquoi sa valeur JSON est représentée sous forme de chaîne de caractères.
- `type` est un identifiant utilisé comme clé pour les informations sur le type de la variable (décrit dans ce qui suit).

Le type donné, dans ce cas `t_uint256`, représente un élément de la liste des types, qui a la forme :

```
{
  "encoding": "inplace",
  "label": "uint256",
  "numberOfBytes": "32",
}
```

où :

- `encoding` : comment les données sont codées dans le stockage, où les valeurs possibles sont :
 - `inplace` : les données sont disposées de manière contiguë dans le stockage (voir *ci-dessus*).
 - `mapping` : Méthode basée sur le hachage Keccak-256 (voir *ci-dessus*).

- `dynamic_array` : Méthode basée sur le hachage Keccak-256 (voir *ci-dessus*).
- `bytes` : slot unique ou méthode basée sur le hachage Keccak-256 selon la taille des données (voir *ci-dessus*).
- `label` est le nom canonique du type.
- `numberOfBytes` est le nombre d'octets utilisés (sous forme de chaîne décimale). Notez que si `numberOfBytes` > 32, cela signifie que plus d'un slot est utilisé.

Certains types ont des informations supplémentaires en plus des quatre ci-dessus. Les mappings contiennent leurs types `key` et `value` (encore une fois en faisant référence à une entrée dans ce mappage des types), les tableaux ont leur type base, et les structures listent leurs membres dans le même format que le stockage de premier niveau (voir *ci-dessus*).

Note : Le format de sortie JSON de la disposition de stockage d'un contrat est encore considéré comme expérimental, et est susceptible d'être modifié dans les versions de Solidity qui ne sont pas en rupture.

L'exemple suivant montre un contrat et sa disposition de stockage, contenant des types de valeur et de référence, des types codés emballés et des types imbriqués.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;
contract A {
    struct S {
        uint128 a;
        uint128 b;
        uint[2] staticArray;
        uint[] dynArray;
    }

    uint x;
    uint y;
    S s;
    address addr;
    mapping (uint => mapping (address => bool)) map;
    uint[] array;
    string s1;
    bytes b1;
}
```

```
{
  "storage": [
    {
      "astId": 15,
      "contract": "fileA:A",
      "label": "x",
      "offset": 0,
      "slot": "0",
      "type": "t_uint256"
    },
    {
      "astId": 17,
      "contract": "fileA:A",
      "label": "y",
      "offset": 0,
      "slot": "1",
      "type": "t_uint256"
    }
  ]
}
```

(suite sur la page suivante)

(suite de la page précédente)

```

},
{
  "astId": 20,
  "contract": "fileA:A",
  "label": "s",
  "offset": 0,
  "slot": "2",
  "type": "t_struct(S)13_storage"
},
{
  "astId": 22,
  "contract": "fileA:A",
  "label": "addr",
  "offset": 0,
  "slot": "6",
  "type": "t_address"
},
{
  "astId": 28,
  "contract": "fileA:A",
  "label": "map",
  "offset": 0,
  "slot": "7",
  "type": "t_mapping(t_uint256,t_mapping(t_address,t_bool))"
},
{
  "astId": 31,
  "contract": "fileA:A",
  "label": "array",
  "offset": 0,
  "slot": "8",
  "type": "t_array(t_uint256)dyn_storage"
},
{
  "astId": 33,
  "contract": "fileA:A",
  "label": "s1",
  "offset": 0,
  "slot": "9",
  "type": "t_string_storage"
},
{
  "astId": 35,
  "contract": "fileA:A",
  "label": "b1",
  "offset": 0,
  "slot": "10",
  "type": "t_bytes_storage"
}
],
"types": {
  "t_address": {

```

(suite sur la page suivante)

```
"encoding": "inplace",
"label": "address",
"numberOfBytes": "20"
},
"t_array(t_uint256)2_storage": {
  "base": "t_uint256",
  "encoding": "inplace",
  "label": "uint256[2]",
  "numberOfBytes": "64"
},
"t_array(t_uint256)dyn_storage": {
  "base": "t_uint256",
  "encoding": "dynamic_array",
  "label": "uint256[]",
  "numberOfBytes": "32"
},
"t_bool": {
  "encoding": "inplace",
  "label": "bool",
  "numberOfBytes": "1"
},
"t_bytes_storage": {
  "encoding": "bytes",
  "label": "bytes",
  "numberOfBytes": "32"
},
"t_mapping(t_address,t_bool)": {
  "encoding": "mapping",
  "key": "t_address",
  "label": "mapping(address => bool)",
  "numberOfBytes": "32",
  "value": "t_bool"
},
"t_mapping(t_uint256,t_mapping(t_address,t_bool))": {
  "encoding": "mapping",
  "key": "t_uint256",
  "label": "mapping(uint256 => mapping(address => bool))",
  "numberOfBytes": "32",
  "value": "t_mapping(t_address,t_bool)"
},
"t_string_storage": {
  "encoding": "bytes",
  "label": "string",
  "numberOfBytes": "32"
},
"t_struct(S)13_storage": {
  "encoding": "inplace",
  "label": "struct A.S",
  "members": [
    {
      "astId": 3,
      "contract": "fileA:A",
```

(suite sur la page suivante)

(suite de la page précédente)

```
    "label": "a",
    "offset": 0,
    "slot": "0",
    "type": "t_uint128"
  },
  {
    "astId": 5,
    "contract": "fileA:A",
    "label": "b",
    "offset": 16,
    "slot": "0",
    "type": "t_uint128"
  },
  {
    "astId": 9,
    "contract": "fileA:A",
    "label": "staticArray",
    "offset": 0,
    "slot": "1",
    "type": "t_array(t_uint256)2_storage"
  },
  {
    "astId": 12,
    "contract": "fileA:A",
    "label": "dynArray",
    "offset": 0,
    "slot": "3",
    "type": "t_array(t_uint256)dyn_storage"
  }
],
"numberOfBytes": "128"
},
"t_uint128": {
  "encoding": "inplace",
  "label": "uint128",
  "numberOfBytes": "16"
},
"t_uint256": {
  "encoding": "inplace",
  "label": "uint256",
  "numberOfBytes": "32"
}
}
```

3.16 Mise en page en mémoire

Solidity réserve quatre emplacements de 32 octets, avec des plages d'octets spécifiques (y compris les points de terminaison) utilisées comme suit :

- `0x00 - 0x3f` (64 octets) : espace de grattage pour les méthodes de hachage
- `0x40 - 0x5f` (32 octets) : taille de la mémoire actuellement allouée (alias pointeur de mémoire libre)
- `0x60 - 0x7f` (32 octets) : emplacement zéro

L'espace d'effacement peut être utilisé entre les instructions (c'est-à-dire dans l'assemblage en ligne). L'emplacement zéro est utilisé comme valeur initiale pour les tableaux de mémoire dynamique et ne devrait jamais être écrit dans (le pointeur de mémoire libre pointe initialement sur `0x80`).

Solidity place toujours les nouveaux objets sur le pointeur de mémoire libre et la mémoire n'est jamais libérée (cela pourrait changer à l'avenir).

Les éléments des tableaux de mémoire dans Solidity occupent toujours des multiples de 32 octets (ceci est même vrai pour `bytes1[]`, mais pas pour `bytes` et `string`). Les tableaux de mémoire multidimensionnels sont des pointeurs vers des tableaux de mémoire. La longueur d'un tableau dynamique est stockée dans le premier emplacement du tableau, suivie des éléments du tableau.

Avertissement : Il y a certaines opérations dans Solidity qui nécessitent une zone de mémoire temporaire plus grande que 64 octets et qui ne peuvent donc pas être placées dans l'espace scratch. Elles seront placées là où la mémoire libre pointe, mais étant donné leur courte durée de vie, le pointeur n'est pas mis à jour. La mémoire peut être mise à zéro. Pour cette raison, il ne faut pas s'attendre à ce que la mémoire libre pointe vers une mémoire mise à zéro.

Bien que cela puisse sembler être une bonne idée d'utiliser `msize` pour arriver à une zone de mémoire définitivement mise à zéro, l'utilisation d'un tel pointeur de façon non-temporelle sans mettre à jour le pointeur de mémoire libre peut avoir des résultats inattendus.

3.16.1 Différences par rapport à l'agencement du stockage

Comme décrit ci-dessus, la disposition en mémoire est différente de la disposition en *storage*. Vous trouverez ci-dessous quelques exemples.

Exemple de différence dans les tableaux

Le tableau suivant occupe 32 octets (1 emplacement) en stockage, mais 128 octets (4 éléments de 32 octets chacun) en mémoire.

```
uint8[4] a;
```

Exemple d'écart de structure

La structure suivante occupe 96 octets (3 emplacements de 32 octets) en stockage, mais 128 octets (4 éléments de 32 octets chacun) en mémoire.

```
struct S {
    uint a;
    uint b;
    uint8 c;
```

(suite sur la page suivante)

(suite de la page précédente)

```

uint8 d;
}

```

3.17 Mise en page des données d'appel

Les données d'entrée pour un appel de fonction sont supposées être dans le format défini par l'*ABI specification*. Entre autres, la spécification ABI exige que les arguments soient complétés par des multiples de 32 octets. Les appels de fonctions internes utilisent une convention différente.

Les arguments du constructeur d'un contrat sont directement ajoutés à la fin du code du contrat, également en codage ABI. Le constructeur y accède par le biais d'un décalage codé en dur, et non pas en utilisant l'opcode `codesize`, puisque celui-ci change bien sûr lors de l'ajout de données au code.

3.18 Nettoyer les variables

Lorsqu'une valeur est inférieure à 256 bits, dans certains cas, les bits restants doivent être nettoyés. Le compilateur Solidity est conçu pour nettoyer ces bits restants avant toute opération qui pourraient être affectés par les déchets potentiels dans les bits restants. Par exemple, avant d'écrire une valeur en mémoire, les bits restants doivent être effacés car le contenu de la mémoire peut être utilisé pour le calcul hachages ou être envoyés en tant que données d'un appel de fonction. De même, avant de stocker une valeur dans le stockage, les bits restants doivent être nettoyés car sinon des valeurs brouillées peuvent être observées.

Notez que l'accès via `assembly` dans le code Solidity n'est pas considéré comme une telle opération : Si vous utilisez `assembly` dans votre code pour accéder aux variables Solidity plus court que 256 bits, le compilateur ne garantit pas que la valeur est correctement nettoyée.

De plus, nous ne nettoyons pas les bits si l'opération suivante n'est pas affectée par l'opération actuelle. Par exemple, puisque tout valeurs non nul est considérée comme `true` par l'instruction `JUMPI`, nous ne nettoyons pas les valeurs booléennes avant qu'elles ne soient utilisées comme condition pour `JUMPI`.

En plus du principe ci-dessus, le compilateur Solidity nettoie les données d'entrée lorsqu'elles sont chargées sur la stack.

Différents types ont des règles différentes pour nettoyer les valeurs non valides :

Type	Valeurs valides	Valeurs invalides
enum of n members	0 until n - 1	exception
bool	0 or 1	1
signed integers	sign-extended word	currently silently wraps ; in the future exceptions will be thrown*
unsigned integers	higher bits zeroed	currently silently wraps ; in the future exceptions will be thrown*

— enveloppe actuellement silencieusement ; à l'avenir, des exceptions seront levées

3.19 Source Mappings (SourceMap de Compilation)

Dans le cadre de l'output AST, le compilateur fournit une plage du source code qui est représenté par le nœud respectif dans l'AST. Cela peut être utilisé à diverses fins allant des outils d'analyse statique qui rapportent les erreurs basées sur l'AST et les outils de débogage qui mettent en évidence les variables locales et leurs usages.

De plus, le compilateur peut également générer un mappage à partir du bytecode à la plage du code source qui a généré l'instruction. C'est important pour les outils d'analyse statique qui fonctionnent au niveau du bytecode et pour afficher la position actuelle dans le code source à l'intérieur d'un débogueur ou pour la gestion des points d'arrêt. Cette cartographie contient également d'autres informations, comme le type de saut et la profondeur du modificateur (voir ci-dessous).

Les deux types de SourceMap utilisent des identifiants entiers pour faire référence aux fichiers source. L'identifiant d'un fichier source est stocké dans `output['sources'][sourceName]['id']` où `output` est la sortie de l'interface de compilation standard-json analysée en tant que JSON. Pour certaines routines utilitaires, le compilateur génère des fichiers source « internes » qui ne font pas partie de l'entrée d'origine mais sont référencés à partir de la SourceMap. Ces fichiers sources ainsi que leurs identifiants peuvent être obtenu via `output['contracts'][sourceName][contractName]['evm']['bytecode']['generatedSources']`.

Note : Dans le cas d'instructions qui ne sont associées à aucun fichier source particulier, le mappage source attribue un identifiant entier de `-1`. Cela peut arriver pour sections de bytecode issues d'instructions d'assemblage en ligne générées par le compilateur.

Les SourceMap à l'intérieur de l'AST utilisent la notation suivantes

`s:l:f`

Où `s` est le décalage d'octet au début de la plage dans le fichier source, `l` est la longueur de la plage source en octets et `f` est la source indice mentionné ci-dessus.

L'encodage dans le mappage source pour le bytecode est plus compliqué : C'est une liste de `s:l:f:j:m` séparés par `;`. Chacun de ces correspond à une instruction, c'est-à-dire que vous ne pouvez pas utiliser le décalage d'octet mais vous devez utiliser l'offset d'instruction (les instructions push sont plus longues qu'un seul octet). Les champs `s`, `l` et `f` sont comme ci-dessus. `j` peut être soit `i`, `o` ou `-` signifiant si une instruction de saut va dans une fonction, return depuis une fonction ou est un saut régulier dans le cadre de par ex. une boucle. Le dernier champ, `m`, est un entier qui indique la « profondeur du modificateur ». Cette profondeur est augmenté chaque fois que l'instruction d'espace réservé (`_`) est entrée dans un modificateur et diminué quand il est à nouveau laissé. Cela permet aux débogueurs de suivre les cas délicats comme quand le même modificateur est utilisé deux fois ou bien que plusieurs déclarations d'espace réservé ont été utilisé dans un seul modificateur.

Afin de compresser ces SourceMap pour le bytecode, les règles suivantes sont utilisées :

- Si un champ est vide, la valeur de l'élément précédent est utilisée.
- S'il manque un `:`, tous les champs suivants sont considérés comme vides.

Cela signifie que les SourceMap suivants représentent les mêmes informations :

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

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

Il est important de noter que lorsque la commande interne *verbatim* est utilisée, les SourceMap seront invalides : la fonction intégrée est considérée comme un seul instruction au lieu de potentiellement multiples.

3.20 L'optimiseur

Le compilateur Solidity utilise deux modules d'optimisation différents : L'« ancien » optimiseur qui opère au niveau de l'opcode et le « nouvel » optimiseur qui opère sur le code Yul IR.

L'optimiseur basé sur les opcodes applique un ensemble de [règles de simplification](#) aux opcodes. Il combine également des ensembles de codes égaux et supprime le code inutilisé.

L'optimiseur basé sur Yul est beaucoup plus puissant, car il peut travailler à travers les appels de fonctions. Par exemple, les sauts arbitraires ne sont pas possibles dans Yul, il est possible de calculer les effets secondaires de chaque fonction. Considérons deux appels de fonction, où la première ne modifie pas le stockage et la seconde le fait. Si leurs arguments et valeurs de retour ne dépendent pas les uns des autres, nous pouvons réordonner les appels de fonction. De même, si une fonction est sans effet secondaire et que son résultat est multiplié par zéro, on peut supprimer complètement l'appel de fonction.

Actuellement, le paramètre `--optimize` active l'optimiseur basé sur le code optique pour le bytecode généré et l'optimiseur Yul pour le code Yul généré en interne, par exemple pour ABI coder v2. On peut utiliser `solc --ir-optimized --optimize` pour produire un Yul expérimental optimisé pour une source Solidity. De même, on peut utiliser `solc --strict-assembly --optimize` pour un mode Yul autonome.

Vous pouvez trouver plus de détails sur les deux modules d'optimisation et leurs étapes d'optimisation ci-dessous.

3.20.1 Avantages de l'optimisation du code Solidity

Globalement, l'optimiseur tente de simplifier les expressions compliquées, ce qui réduit à la fois la taille du code et le coût d'exécution, c'est-à-dire qu'il peut réduire le gaz nécessaire au déploiement du contrat ainsi qu'aux appels externes faits au contrat. Il spécialise également les fonctions ou les met en ligne. En particulier, l'inlining de fonctions est une opération qui peut entraîner un code beaucoup plus gros, mais elle est souvent effectuée car elle permet d'obtenir des simplifications supplémentaires.

3.20.2 Différences entre le code optimisé et le code non optimisé

En général, la différence la plus visible est que les expressions constantes sont évaluées au moment de la compilation. En ce qui concerne la sortie ASM, on peut également noter une réduction des blocs de code équivalents ou dupliqués (comparez la sortie des drapeaux `--asm` et `--asm --optimize`). Cependant, lorsqu'il s'agit de la représentation Yul/intermédiaire, il peut y avoir des différences significatives, par exemple, les fonctions peuvent être inlined, combinées ou réécrites pour redondances, etc. (comparez la sortie entre les drapeaux `--ir` et `--optimize --ir-optimized`).

3.20.3 Exécution des paramètres de l'optimiseur

Le nombre d'exécutions (`--optimize-runs`) spécifie approximativement combien de fois chaque opcode du code déployé sera exécuté pendant la durée de vie du contrat. Cela signifie qu'il s'agit d'un paramètre de compromis entre la taille du code (coût de déploiement) et le coût d'exécution du code (coût après déploiement). Un paramètre « runs » de « 1 » produira un code court mais coûteux. En revanche, un paramètre « runs » plus grand produira un code plus long mais plus efficace en termes de gaz. La valeur maximale du paramètre est $2^{32}-1$.

Note : Une idée fausse courante est que ce paramètre spécifie le nombre d'itérations de l'optimiseur. Ce n'est pas vrai : l'optimiseur s'exécutera toujours autant de fois qu'il peut encore améliorer le code.

3.20.4 Module d'optimisation basé sur l'opcode

Le module d'optimisation basé sur le code opcode opère sur le code assembleur. Il divise la séquence d'instructions en blocs de base aux JUMPs et JUMPDESTs. À l'intérieur de ces blocs, l'optimiseur analyse les instructions et enregistre chaque modification de la pile, de la mémoire ou du stockage sous la forme d'une expression constituée d'une instruction et une liste d'arguments qui sont des pointeurs vers d'autres expressions.

De plus, l'optimiseur basé sur les opcodes utilise un composant appelé « CommonSubexpressionEliminator » qui, entre autres, trouve les expressions qui sont toujours égales (sur chaque entrée) et les combine en une classe d'expressions. Il essaie d'abord de trouver chaque nouvelle expression dans une liste d'expressions déjà connues. Si aucune correspondance n'est trouvée, il simplifie l'expression selon des règles comme `constant + constant = sum_of_constants` ou `X * 1 = X`. Comme il s'agit d'un processus récursif, nous pouvons également appliquer cette dernière règle si le deuxième facteur est une expression plus complexe dont nous savons que l'évaluation est toujours égale à un.

Certaines étapes de l'optimiseur suivent symboliquement les emplacements de stockage et de mémoire. Par exemple, cela est utilisée pour calculer les hachages Keccak-256 qui peuvent être évalués lors de la compilation. Considérons la séquence :

```
PUSH 32
PUSH 0
CALLDATALOAD
PUSH 100
DUP2
MSTORE
KECCAK256
```

ou l'équivalent Yul

```
let x := calldataload(0)
mstore(x, 100)
let value := keccak256(x, 32)
```

Dans ce cas, l'optimiseur suit la valeur à un emplacement mémoire `calldataload(0)` et réalise que le hachage Keccak-256 peut être évalué au moment de la compilation. Cela ne fonctionne que s'il n'y a pas autre instruction qui modifie la mémoire entre le `mstore` et le `keccak256`. Donc s'il y a une instruction qui écrit dans la mémoire (ou le stockage), alors nous devons effacer la connaissance de la mémoire (ou stockage) actuelle. Il y a cependant une exception à cet effacement, lorsque nous pouvons facilement voir que l'instruction n'écrit pas à un certain endroit.

Par exemple,

```
let x := calldataload(0)
mstore(x, 100)
// Emplacement de la mémoire de la connaissance actuelle x -> 100
let y := add(x, 32)
// N'efface pas la connaissance que x -> 100, puisque y n'écrit pas dans [x, x + 32)
mstore(y, 200)
// Ce Keccak-256 peut maintenant être évalué.
let value := keccak256(x, 32)
```

Par conséquent, les modifications apportées aux emplacements de stockage et de mémoire, par exemple à l'emplacement 1, doivent effacer la connaissance des emplacements de stockage ou de mémoire qui peuvent être égaux à 1. Plus précisément, pour le stockage, l'optimiseur doit effacer toute connaissance des emplacements symboliques, qui peuvent être égaux à 1. Et, pour la mémoire, l'optimiseur doit effacer toute connaissance des emplacements symboliques qui ne sont pas au moins 32 octets. Si `m` représente un emplacement arbitraire, alors la décision d'effacement est prise en calculant la valeur `sub(1, m)`. Pour le stockage, si cette valeur s'évalue à un littéral qui est non-zéro, alors la connaissance de `m` sera conservée. Pour la mémoire, si la valeur correspond à une valeur littérale comprise entre 32 et $2^{*}256$

- 32, alors la connaissance de `m` sera conservée. Dans tous les autres cas, la connaissance de `m` sera effacée.

Après ce processus, nous savons quelles expressions doivent se trouver sur la pile à la fin, et nous avons une liste des modifications de la mémoire et du stockage. Ces informations sont stockées avec les blocs de base et est utilisée pour les relier. En outre, les connaissances sur la configuration de la pile, du stockage et de la mémoire sont transmises au(x) bloc(s) suivant(s).

Si nous connaissons les cibles de toutes les instructions `JUMP` et `JUMPI`, nous pouvons construire un graphe complet du flux de contrôle du programme. S'il y a seulement une cible que nous ne connaissons pas (cela peut arriver car en principe, les cibles de saut peuvent être calculées à partir des entrées), nous devons effacer toute connaissance sur l'état d'entrée d'un bloc car il peut être la cible du `JUMP` inconnu. Si le module d'optimisation basé sur les opcodes d'opération trouve un `JUMPI` dont la condition s'évalue à une constante, il le transforme en un saut inconditionnel.

Comme dernière étape, le code de chaque bloc est re-généré. L'optimiseur crée un graphe de dépendance à partir des expressions sur la pile à la fin du bloc, et il abandonne toute opération qui ne fait pas partie de ce graphe. Il génère du code qui applique les modifications à la mémoire et au stockage dans l'ordre dans lequel elles ont été faites dans le code d'origine (en abandonnant les modifications qui ne sont pas nécessaires). Enfin, il génère toutes les valeurs qui doivent se trouver sur la pile au bon endroit.

Ces étapes sont appliquées à chaque bloc de base et le code nouvellement généré est utilisé comme remplacement s'il est plus petit. Si un bloc de base est divisé à un `JUMPI` et que pendant l'analyse, la condition s'évalue à une constante, le `JUMPI` est remplacé en fonction de la valeur de la constante. Ainsi, un code comme

```
uint x = 7;
data[7] = 9;
if (data[x] != x + 2) // cette condition n'est jamais vraie
    return 2;
else
    return 1;
```

se simplifie comme suit :

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

Double simple

Depuis la version 0.8.2 de Solidity, il existe une autre étape d'optimisation qui remplace certains sauts vers des blocs contenant des instructions « simples » se terminant par un « saut » par une copie de ces instructions. Cela correspond à l'inlining de petites fonctions simples de Solidity ou de Yul. En particulier, la séquence `PUSHTAG(tag) JUMP` peut être remplacée, dès lors que le `JUMP` est marqué comme un saut « dans » une fonction et que derrière le `tag` se trouve un bloc de base (comme décrit ci-dessus pour la fonction « `CommonSubexpressionEliminator` ») qui se termine par un autre `JUMP` marqué comme étant un saut « hors » d'une fonction.

En particulier, considérez l'exemple prototypique suivant d'assemblage généré pour un appel à une fonction interne de Solidity :

```
tag_return
tag_f
jump      // sur
tag_return:
...opcodes après l'appel à f...

tag_f:
```

(suite sur la page suivante)

(suite de la page précédente)

```
...corps de fonction f...
jump    // hors
```

Tant que le corps de la fonction est un bloc de base continu, le « Inliner » peut remplacer `tag_f jump` par le bloc à `tag_f`, ce qui donne :

```
tag_return
...corps de fonction f...
jump
tag_return:
...opcodes après l'appel à f...

tag_f:
...corps de fonction f...
jump    // hors
```

Maintenant, idéalement, les autres étapes de l'optimiseur décrites ci-dessus auront pour résultat de déplacer le `push` de la balise de retour vers le saut restant, résultant en :

```
...corps de fonction f...
tag_return
jump
tag_return:
...opcodes après l'appel à f...

tag_f:
...corps de fonction f...
jump    // out
```

Dans cette situation, le « PeepholeOptimizer » supprimera le saut de retour. Idéalement, tout ceci peut être fait pour toutes les références à `tag_f` en le laissant inutilisé, s.t. il peut être enlevé, donnant :

```
...corps de fonction f...
...opcodes après l'appel à f...
```

Ainsi, l'appel à la fonction `f` est inlined et la définition originale de `f` peut être supprimée.

Un tel inlining est tenté chaque fois qu'une heuristique suggère que l'inlining est moins coûteux sur la durée de vie d'un contrat que de ne pas le faire. Cette heuristique dépend de la taille du corps de la fonction, du nombre d'autres références à sa balise (approximativement le nombre d'appels à la fonction) et le nombre prévu d'exécutions du contrat (le paramètre « runs » de l'optimiseur global).

3.20.5 Module optimiseur basé sur Yul

L'optimiseur basé sur Yul se compose de plusieurs étapes et composants qui transforment tout l'AST d'une manière sémantiquement équivalente. L'objectif est d'obtenir un code plus court ou au moins légèrement plus long, mais qui permettra d'autres étapes d'optimisation.

Avertissement : L'optimiseur étant en cours de développement, les informations fournies ici peuvent être obsolètes. Si vous dépendez d'une certaine fonctionnalité, veuillez contacter l'équipe directement.

L'optimiseur suit actuellement une stratégie purement avide et ne fait aucun retour en arrière.

Tous les composants du module optimiseur basé sur Yul sont expliqués ci-dessous. Les étapes de transformation suivantes sont les principaux composants :

- Transformation SSA
- Éliminateur de sous-expression commune
- Simplificateur d'expression
- Éliminateur d'assignation redondante
- Inliner complet

Étapes de l'optimiseur

Il s'agit d'une liste de toutes les étapes de l'optimiseur basé sur Yul, classées par ordre alphabétique. Vous pouvez trouver plus d'informations sur les étapes individuelles et leur séquence ci-dessous.

- *BlockFlattener*.
- *CircularReferencesPruner*.
- *CommonSubexpressionEliminator*.
- *ConditionalSimplifier*.
- *ConditionalUnsimplifier*.
- *ControlFlowSimplifier*.
- *DeadCodeEliminator*.
- *EqualStoreEliminator*.
- *EquivalentFunctionCombiner*.
- *ExpressionJoiner*.
- *Expression Simplifier*.
- *ExpressionSplitter*.
- *ForLoopConditionIntoBody*.
- *ForLoopConditionOutOfBody*.
- *ForLoopInitRewriter*.
- *ExpressionInliner*.
- *FullInliner*.
- *FunctionGrouper*.
- *FunctionHoister*.
- *FunctionSpecializer*.
- *LiteralRematerialiser*.
- *LoadResolver*.
- *LoopInvariantCodeMotion*.
- *RedundantAssignEliminator*.
- *ReasoningBasedSimplifier*.
- *Rematerialiser*.
- *SSAReverser*.
- *SSATransform*.
- *StructuralSimplifier*.
- *UnusedFunctionParameterPruner*.
- *UnusedPruner*.
- *VarDeclInitializer*.

Sélection des optimisations

Par défaut, l'optimiseur applique sa séquence prédéfinie d'étapes d'optimisation à l'assemblage généré. Vous pouvez remplacer cette séquence et fournir la vôtre en utilisant l'option `--yul-optimizations` :

```
solc --optimize --ir-optimized --yul-optimizations 'dhfoD[xarrsclMcCTU]uljmul'
```

La séquence à l'intérieur de [...] sera appliquée plusieurs fois dans une boucle jusqu'à ce que le code Yul reste inchangé ou jusqu'à ce que le nombre maximum de tours (actuellement 12) ait été atteint.

Les abréviations disponibles sont listées dans les docs [Yul optimizer](#).

Prétraitement

Les composants de prétraitement effectuent des transformations pour mettre le programme dans une certaine forme normale avec laquelle il est plus facile de travailler. Cette forme normale est conservée pendant le reste du processus d'optimisation.

Disambiguateur

Le désambiguïsateur prend un AST et retourne une copie fraîche où tous les identifiants ont des noms uniques dans l'AST d'entrée. C'est une condition préalable pour toutes les autres étapes de l'optimiseur. Un des avantages est que la recherche d'identificateurs n'a pas besoin de prendre en compte les scopes, ce qui simplifie l'analyse nécessaire pour les autres étapes.

Toutes les étapes suivantes ont la propriété que tous les noms restent uniques. Cela signifie que si un nouvel identifiant doit être introduit, un nouveau nom unique est généré.

FunctionHoister

Le hoister de fonction déplace toutes les définitions de fonction à la fin du bloc le plus haut. Il s'agit d'une transformation sémantiquement équivalente tant qu'elle est effectuée après l'étape de désambiguïsation. La raison en est que le déplacement d'une définition vers un bloc de niveau supérieur ne peut pas diminuer sa visibilité et il est impossible de référencer des variables définies dans une autre fonction.

L'avantage de cette étape est que les définitions de fonctions peuvent être recherchées plus facilement, et les fonctions peuvent être optimisées de manière isolée sans avoir à traverser complètement l'AST.

FunctionGrouper

Le grouper de fonctions doit être appliqué après le désambiguïsateur et le hachoir de fonctions. Son effet est que tous les éléments les plus hauts qui ne sont pas des définitions de fonction sont déplacés dans un seul bloc qui est la première déclaration du bloc racine.

Après cette étape, un programme a la forme normale suivante :

```
{ I F... }
```

Où I est un bloc (potentiellement vide) qui ne contient aucune définition de fonction (même pas de manière récursive), et F est une liste de définitions de fonctions telle qu'aucune fonction ne contient une définition de fonction.

L'avantage de cette étape est que nous savons toujours où commence la liste des fonctions.

ForLoopConditionIntoBody

Cette transformation déplace la condition d’itération de boucle d’une boucle for dans le corps de la boucle. Nous avons besoin de cette transformation car *ExpressionSplitter* ne s’appliquera pas aux expressions de condition d’itération (le C dans l’exemple suivant).

```
for { Init... } C { Post... } {
  Body...
}
```

est transformé en

```
for { Init... } 1 { Post... } {
  if iszero(C) { break }
  Body...
}
```

Cette transformation peut également être utile lorsqu’elle est couplée avec *LoopInvariantCodeMotion*, puisque les invariants des conditions invariantes de la boucle peuvent alors être pris en dehors de la boucle.

ForLoopInitRewriter

Cette transformation permet de déplacer la partie d’initialisation d’une boucle for avant la boucle :

```
for { Init... } C { Post... } {
  Body...
}
```

est transformé en

```
Init...
for {} C { Post... } {
  Body...
}
```

Cela facilite le reste du processus d’optimisation car nous pouvons ignorer les règles de scoping compliquées du bloc d’initialisation de la boucle for.

VarDeclInitializer

Cette étape réécrit les déclarations de variables afin qu’elles soient toutes initialisées. Les déclarations comme `let x, y` sont divisées en plusieurs déclarations.

Pour l’instant, elle ne supporte que l’initialisation avec le littéral zéro.

Transformation Pseudo-SSA

Le but de ce composant est de mettre le programme sous une forme plus longue, afin que les autres composants puissent plus facilement travailler avec lui. La représentation finale sera similaire à une forme SSA (static-single-assignment), à la différence qu'elle ne fait pas appel à des fonctions « phi » explicites qui combinent les valeurs provenant de différentes branches du flux de contrôle, car une telle fonctionnalité n'existe pas dans le langage Yul. Au lieu de cela, lors de la fusion du flux de contrôle, si une variable est réaffectée dans l'une des branches, une nouvelle variable SSA est déclarée pour contenir sa valeur actuelle, de sorte que les expressions suivantes ne doivent toujours faire référence qu'à des variables SSA.

Un exemple de transformation est le suivant :

```
{
  let a := calldataload(0)
  let b := calldataload(0x20)
  if gt(a, 0) {
    b := mul(b, 0x20)
  }
  a := add(a, 1)
  sstore(a, add(b, 0x20))
}
```

Lorsque toutes les étapes de transformation suivantes sont appliquées, le programme aura l'aspect suivant comme suit :

```
{
  let _1 := 0
  let a_9 := calldataload(_1)
  let a := a_9
  let _2 := 0x20
  let b_10 := calldataload(_2)
  let b := b_10
  let _3 := 0
  let _4 := gt(a_9, _3)
  if _4
  {
    let _5 := 0x20
    let b_11 := mul(b_10, _5)
    b := b_11
  }
  let b_12 := b
  let _6 := 1
  let a_13 := add(a_9, _6)
  let _7 := 0x20
  let _8 := add(b_12, _7)
  sstore(a_13, _8)
}
```

Notez que la seule variable qui est réassignée dans cet extrait est `b`. Cette réaffectation ne peut être évitée car `b` a des valeurs différentes en fonction du flux de contrôle. Toutes les autres variables ne changent jamais de valeur une fois qu'elles sont définies. L'avantage de cette propriété est que les variables peuvent être déplacées librement et les références à celles-ci peuvent être échangées par leur valeur initiale (et vice-versa), tant que ces valeurs sont encore valables dans le nouveau contexte.

Bien sûr, le code ici est loin d'être optimisé. Au contraire, il est beaucoup plus long. L'espoir est que ce code soit plus facile à travailler et que, de plus, il y a des étapes d'optimisation qui annulent ces changements et rendent le code plus

compact à la fin.

ExpressionSplitter

Le séparateur d'expression transforme des expressions comme `add(mload(0x123), mul(mload(0x456), 0x20))` en une séquence de déclarations de variables uniques auxquelles sont attribuées des sous-expressions de cette expression, de sorte que chaque appel de fonction n'a que des variables comme arguments.

Ce qui précède serait transformé en

```
{
  let _1 := 0x20
  let _2 := 0x456
  let _3 := mload(_2)
  let _4 := mul(_3, _1)
  let _5 := 0x123
  let _6 := mload(_5)
  let z := add(_6, _4)
}
```

Notez que cette transformation ne change pas l'ordre des opcodes ou des appels de fonction.

Elle n'est pas appliquée à la condition d'itération de la boucle, car le flux de contrôle de la boucle ne permet pas ce « contournement » des expressions internes dans tous les cas. Nous pouvons contourner cette limitation en appliquant la condition-boucle-for-dans-corps pour déplacer la condition d'itération dans le corps de la boucle.

Le programme final doit être sous une forme telle que (à l'exception des conditions de boucle) les appels de fonction ne peuvent pas être imbriqués dans des expressions et tous les arguments des appels de fonction doivent être des variables.

Les avantages de cette forme sont qu'il est beaucoup plus facile de réorganiser la séquence des opcodes et il est également plus facile d'effectuer l'inlining des appels de fonction. En outre, il est plus simple de remplacer des parties individuelles d'expressions ou de réorganiser l'« arbre d'expression ». L'inconvénient est qu'un tel code est beaucoup plus difficile à lire pour les humains.

SSATransform

Cette étape tente de remplacer les affectations répétées à existantes par des déclarations de nouvelles variables. Les réaffectations sont toujours présentes, mais toutes les références aux variables réaffectées sont remplacées par des variables nouvellement déclarées.

Exemple :

```
{
  let a := 1
  mstore(a, 2)
  a := 3
}
```

est transformé en

```
{
  let a_1 := 1
  let a := a_1
  mstore(a_1, 2)
}
```

(suite sur la page suivante)

```

let a_3 := 3
a := a_3
}

```

Sémantique exacte :

Pour toute variable a qui est assignée quelque part dans le code (les variables qui sont déclarées avec une valeur et ne sont jamais réassignées ne sont pas modifiées), effectuez les transformations suivantes :

- remplacer `let a := v` par `let a_i := v let a := a_i`
- remplacer `a := v` par `let a_i := v a := a_i` où i est un nombre tel que a_i est encore inutilisé.

En outre, enregistrez toujours la valeur actuelle de i utilisée pour a et remplacez chaque référence à a par a_i . Le mappage de la valeur courante est effacé pour une variable a à la fin de chaque bloc dans lequel elle a été affectée et à la fin du bloc d'initialisation de la boucle `for` si elle est affectée à l'intérieur du corps de la boucle `for` ou du bloc `post`. Si la valeur d'une variable est effacée selon la règle ci-dessus et que la variable est déclarée en dehors du bloc, une nouvelle variable SSA sera créée à l'endroit où le flux de contrôle se rejoint, cela inclut le début du bloc `post-boucle/corps` et l'emplacement juste après l'instruction `If/Switch/ForLoop/Block`.

Après cette étape, il est recommandé d'utiliser le Redundant Assign Eliminator pour supprimer les assignations intermédiaires inutiles.

Cette étape donne de meilleurs résultats si le séparateur d'expressions et l'éliminateur de sous-expressions communes sont exécutés juste avant, car elle ne génère alors pas de quantités excessives de variables. D'autre part, l'éliminateur de sous-expressions communes pourrait être plus efficace s'il était exécuté après la transformation SSA.

RedundantAssignEliminator

La transformation SSA génère toujours une affectation de la forme `a := a_i`, même si cela n'est pas nécessaire dans de nombreux cas, comme dans l'exemple suivant :

```

{
  let a := 1
  a := mload(a)
  a := sload(a)
  sstore(a, 1)
}

```

La transformation SSA convertit cet extrait en ce qui suit :

```

{
  let a_1 := 1
  let a := a_1
  let a_2 := mload(a_1)
  a := a_2
  let a_3 := sload(a_2)
  a := a_3
  sstore(a_3, 1)
}

```

L'éliminateur d'assignations redondantes supprime les trois assignations à a , car la valeur de a n'est pas utilisée et transforme ainsi ce cet extrait en une forme SSA stricte :

```

{
  let a_1 := 1

```

(suite de la page précédente)

```

let a_2 := mload(a_1)
let a_3 := sload(a_2)
sstore(a_3, 1)
}

```

Bien sûr, les parties complexes pour déterminer si une affectation est redondante ou non sont liées à la jonction du flux de contrôle.

Le composant fonctionne en détail comme suit :

L'AST est parcouru deux fois : dans une étape de collecte d'informations et dans l'étape de suppression proprement dite. Pendant la collecte d'informations, nous maintenons une correspondance entre les instructions d'affectation et les trois états « unused », « undecided » et « used » qui signifie si la valeur assignée sera utilisée ultérieurement par une référence à la variable.

Lorsqu'une affectation est visitée, elle est ajoutée au mappage dans l'état « undecided » (voir la remarque sur les boucles for ci-dessous), et chaque autre affectation à la même variable qui est toujours dans l'état « undecided » est changée en « unused ». Lorsqu'une variable est référencée, l'état de toute affectation à cette variable qui se trouve encore dans l'état « undecided » est changé en « used ».

Aux points où le flux de contrôle se divise, une copie de la cartographie est remise à chaque branche. Aux points où le flux de contrôle se rejoint, les deux mappings provenant des deux branches sont combinés de la manière suivante : Les déclarations qui ne figurent que dans un seul mappage ou qui ont le même état sont utilisées sans modification. Les valeurs conflictuelles sont résolues de la manière suivante :

- « unused », « undecided » -> « undecided »
- « unused », « used » -> « used »
- « undecided », « used » -> « used »

Pour les boucles for, la condition, le corps et la partie post sont visités deux fois, en tenant compte du flux de contrôle de jonction à la condition. En d'autres termes, nous créons trois chemins de flux de contrôle : zéro parcours de la boucle, un parcours et deux parcours, puis nous les combinons à la fin.

Il n'est pas nécessaire de simuler une troisième exécution ou même plus, ce qui peut être vu comme suit :

L'état d'une affectation au début de l'itération entraînera de manière déterministe un état de cette affectation à la fin de l'itération. Soit cette fonction de mappage d'état soit appelée f . La combinaison des trois états différents états différents unused, undecided et used, comme expliqué ci-dessus, est l'opération max, où unused = 0, undecided = 1 et used = 2.

La bonne méthode serait de calculer

```
max(s, f(s), f(f(s)), f(f(f(s))), ...)
```

comme état après la boucle. Puisque f a juste une plage de trois valeurs différentes, en l'itérant, on doit atteindre un cycle après au plus trois itérations, et donc $f(f(f(s)))$ doit être égal à l'une des valeurs s , $f(s)$, ou $f(f(s))$. et donc

```
max(s, f(s), f(f(s))) = max(s, f(s), f(f(s)), f(f(f(s))), ...).
```

En résumé, exécuter la boucle au maximum deux fois est suffisant car il n'y a que trois états différents.

Pour les instructions switch qui ont un cas « par défaut », il n'y a pas de flux de contrôle qui saute le switch.

Lorsqu'une variable sort de sa portée, toutes les instructions qui se trouvent encore dans l'état « undecided » sont transformées en « unused », sauf si la variable est le paramètre de retour d'une fonction - dans ce cas, l'état passe à « used ».

Dans la deuxième traversée, toutes les affectations qui sont dans l'état « unused » sont supprimées.

Cette étape est généralement exécutée juste après la transformation SSA pour compléter la génération du pseudo-SSA.

Outils

Movability

Movability est une propriété d'une expression. Elle signifie en gros que l'expression est sans effet secondaire et que son évaluation ne dépend que des valeurs des variables et de l'état des constantes d'appel de l'environnement. La plupart des expressions sont mobiles. Les parties suivantes rendent une expression non-mobile :

- les appels de fonction (cela pourrait être assoupli à l'avenir si toutes les instructions de la fonction sont mobiles)
- les opcodes qui ont (peuvent avoir) des effets secondaires (comme `call` ou `selfdestruct`)
- les opcodes qui lisent ou écrivent des informations de mémoire, de stockage ou d'état externe
- les opcodes qui dépendent de l'ordinateur actuel, de la taille de la mémoire ou de la taille des données de retour.

DataflowAnalyzer

L'analyseur de flux de données n'est pas une étape d'optimisation en soi mais est utilisé comme un outil par d'autres composants. Tout en parcourant l'AST, il suit la valeur actuelle de chaque variable, tant que cette valeur est une expression mobile. Il enregistre les variables qui font partie de l'expression qui est actuellement assignée à chaque autre variable. Lors de chaque affectation à une variable `a`, la valeur courante stockée de `a` est mise à jour et toutes les valeurs stockées de toutes les variables `b` sont effacées chaque fois que `a` fait partie de l'expression actuellement stockée pour `b`.

Aux jonctions du flux de contrôle, la connaissance des variables est effacée si elles ont été ou seraient affectées dans l'un des chemins du flux de contrôle. Par exemple, en entrant dans une boucle `for`, on efface toutes les variables qui seront affectées pendant le bloc `body` ou le bloc `post`.

Simplifications à l'échelle de l'expression

Ces passes de simplification modifient les expressions et les remplacent par des expressions équivalentes et, espérons-le, plus simples.

CommonSubexpressionEliminator

Cette étape utilise l'analyseur de flux de données et remplace les sous-expressions qui correspondent syntaxiquement à la valeur actuelle d'une variable par une référence à cette variable. Il s'agit d'une transformation d'équivalence car ces sous-expressions doivent être déplaçables.

Toutes les sous-expressions qui sont elles-mêmes des identificateurs sont remplacées par leur valeur courante si la valeur est un identificateur.

La combinaison des deux règles ci-dessus permet de calculer une valeur locale numérotation, ce qui signifie que si deux variables ont la même valeur, l'une d'entre elles sera toujours inutilisée. L'élagueur d'inutilisation ou l'éliminateur d'assignations redondantes Redundant Assign Eliminator seront alors en mesure d'éliminer complètement de telles variables.

Cette étape est particulièrement efficace si le séparateur d'expression est exécuté avant. Si le code est sous forme de pseudo-SSA, les valeurs des variables sont disponibles pendant un temps plus long et donc nous avons une plus grande chance que les expressions soient remplaçables.

Le simplifieur d'expression sera capable d'effectuer de meilleurs remplacements si l'éliminateur de sous-expressions communes a été exécuté juste avant lui.

Expression Simplifier

Le simplificateur d'expression utilise l'analyseur de flux de données et utilise d'une liste de transformations d'équivalence sur des expressions comme $X + 0 \rightarrow X$ pour simplifier le code.

Il essaie de faire correspondre des motifs comme $X + 0$ sur chaque sous-expression. Au cours de la procédure de correspondance, il résout les variables en fonction de leur variables actuellement assignées afin de pouvoir faire correspondre des motifs plus profondément imbriqués, même lorsque le code est sous forme de pseudo-SSA.

Certains motifs comme $X - X \rightarrow 0$ ne peuvent être appliqués qu'à condition que que l'expression X est mobile, parce que sinon, cela supprimerait ses effets secondaires potentiels. Puisque les références aux variables sont toujours mobiles, même si leur valeur actuelle ne l'est pas, le simplificateur d'expression est encore plus puissant sous forme fractionnée ou pseudo-SSA.

LiteralRematerialiser

À documenter.

LoadResolver

Étape d'optimisation qui remplace les expressions de type `sload(x)` et `mload(x)` par la valeur actuellement stockée dans le stockage resp. La mémoire, si elle est connue.

Fonctionne mieux si le code est sous forme SSA.

Prérequis : Disambiguator, ForLoopInitRewriter.

ReasoningBasedSimplifier

Cet optimiseur utilise les solveurs SMT pour vérifier si les conditions `if` sont constantes.

- Si `constraints AND condition` est UNSAT, la condition n'est jamais vraie et le corps entier peut être supprimé.
- Si `constraints AND NOT condition` est UNSAT, la condition est toujours vraie et peut être remplacée par `1`.

Les simplifications ci-dessus ne peuvent être appliquées que si la condition est mobile.

Elles ne sont efficaces que sur le dialecte EVM, mais peuvent être utilisées sans danger sur les autres dialectes.

Prérequis : Disambiguator, SSATransform.

Simplifications à l'échelle de la déclaration

CircularReferencesPruner

Cette étape supprime les fonctions qui s'appellent les unes les autres mais qui ne sont ni référencées de manière externe ni référencées depuis le contexte le plus externe.

ConditionalSimplifier

Le simplificateur conditionnel insère des affectations aux variables de condition si la valeur peut être déterminée à partir du flux de contrôle.

Détruit le formulaire SSA.

Actuellement, cet outil est très limité, surtout parce que nous n'avons pas encore de support pour les types booléens. Puisque les conditions vérifient seulement si les expressions sont non nulles, nous ne pouvons pas attribuer une valeur spécifique.

Fonctions actuelles :

- switch cases : insérer « <condition> := <caseLabel> »
- après une instruction if avec un flux de contrôle terminant, insérez « <condition> := 0 »

Fonctionnalités futures :

- permettre les remplacements par « 1 »
- prise en compte de la terminaison des fonctions définies par l'utilisateur

Fonctionne mieux avec le formulaire SSA et si la suppression du code mort a été exécutée auparavant.

Prérequis : Disambiguator

ConditionalUnSimplifier

Inverse du simplificateur conditionnel.

ControlFlowSimplifier

Simplifie plusieurs structures de flux de contrôle :

- remplacer if par un corps vide par pop(condition)
- supprimer le cas vide de switch par défaut
- supprimer le cas vide du switch si aucun cas par défaut n'existe
- remplacer switch sans cas par pop(expression)
- transformer un switch avec un seul cas en if
- remplacer un switch avec un seul cas par défaut avec pop(expression) et body
- remplacer le switch avec const expr par le cas body correspondant
- remplacer for par un flux de contrôle terminant et sans autre break/continue par if
- supprimer leave à la fin d'une fonction.

Aucune de ces opérations ne dépend du flux de données. Le StructuralSimplifier effectue des tâches similaires qui dépendent du flux de données.

Le ControlFlowSimplifier enregistre la présence ou l'absence de `break` et `continue` pendant sa traversée.

Prérequis : Disambiguator, FunctionHoister, ForLoopInitRewriter Important : Introduit les opcodes EVM et ne peut donc être utilisé que sur du code EVM pour le moment.

DeadCodeEliminator

Cette étape d'optimisation supprime le code inaccessible.

Le code inaccessible est tout code à l'intérieur d'un bloc qui est précédé d'une commande `leave`, `return`, `invalid`, `break`, `continue`, `selfdestruct` ou `revert`.

Les définitions de fonctions sont conservées car elles peuvent être appelées par du code précédent et sont donc considérées comme accessibles.

Parce que les variables déclarées dans le bloc `init` d'une boucle `for` ont leur portée étendue au corps de la boucle, nous avons besoin que `ForLoopInitRewriter` soit exécuté avant cette étape.

Prérequis : `ForLoopInitRewriter`, `Function Hoister`, `Function Groupier`

EqualStoreEliminator

Cette étape supprime les appels à `mstore(k, v)` et `sstore(k, v)` s'il y avait un appel précédent à `mstore(k, v)` / `sstore(k, v)`, aucun autre magasin entre les deux et les valeurs de `k` et `v` n'ont pas changé.

Cette simple étape est efficace si elle est exécutée après la transformation SSA et l'éliminateur de sous-expression commune, parce que SSA s'assurera que les variables ne changeront pas et l'éliminateur de sous-expression commune réutilise exactement la même variable si la valeur est connue pour être la même.

Prérequis : `Désambiguïsateur`, `ForLoopInitRewriter`

UnusedPruner

Cette étape supprime les définitions de toutes les fonctions qui ne sont jamais référencées.

Elle supprime également la déclaration des variables qui ne sont jamais référencées. Si la déclaration affecte une valeur qui n'est pas déplaçable, l'expression est conservée, mais sa valeur est supprimée.

Toutes les déclarations d'expressions mobiles (expressions qui ne sont pas assignées) sont supprimées.

StructuralSimplifier

Il s'agit d'une étape générale qui permet d'effectuer différents types de simplifications au niveau structurel :

- remplacer l'instruction `if` avec un corps vide par `pop(condition)`
- remplacer l'instruction `if` avec une condition vraie par son corps
- supprimer l'instruction `if` avec une condition fausse
- transformer un `switch` avec un seul cas en `if`
- remplacer le commutateur avec un seul cas par défaut par `pop(expression)` et son corps
- remplacer le commutateur avec une expression littérale par le corps du cas correspondant
- remplacer la boucle `for` avec une fausse condition par sa partie initialisation.

Ce composant utilise le `Dataflow Analyzer`.

BlockFlattener

Cette étape élimine les blocs imbriqués en insérant l’instruction du bloc interne à l’endroit approprié du bloc externe. Elle dépend du FunctionGrouper et n’aplatit pas le bloc le plus extérieur pour conserver la forme produite par le FunctionGrouper.

```
{
  {
    let x := 2
    {
      let y := 3
      mstore(x, y)
    }
  }
}
```

est transformé en

```
{
  {
    let x := 2
    let y := 3
    mstore(x, y)
  }
}
```

Tant que le code est désambiguïsé, cela ne pose pas de problème car la portée des variables ne peut que croître.

LoopInvariantCodeMotion

Cette optimisation déplace les déclarations de variables SSA mobiles en dehors de la boucle.

Seules les déclarations au niveau supérieur dans le corps ou le post-bloc d’une boucle sont prises en compte à l’intérieur de branches conditionnelles ne seront pas déplacées hors de la boucle.

Exigences :

- Le Disambiguator, ForLoopInitRewriter et FunctionHoister doivent être exécutés en amont.
- Le séparateur d’expression et la transformation SSA doivent être exécutés en amont pour obtenir un meilleur résultat.

Optimisations au niveau des fonctions

FunctionSpecializer

Cette étape spécialise la fonction avec ses arguments littéraux.

Si une fonction, disons, `function f(a, b) { sstore (a, b) }`, est appelée avec des arguments littéraux, par exemple, `f(x, 5)`, où `x` est un identificateur, elle peut être spécialisée en créant une nouvelle fonction `f_1` qui ne prend qu’un seul argument, c’est-à-dire,

```
function f_1(a_1) {
  let b_1 := 5
```

(suite sur la page suivante)

(suite de la page précédente)

```
sstore(a_1, b_1)
}
```

D'autres étapes d'optimisation permettront de simplifier davantage la fonction. L'étape d'optimisation est principalement utile pour les fonctions qui ne seraient pas inlined.

Prérequis : Disambiguator, FunctionHoister

LiteralRematerialiser est recommandé comme prérequis, même s'il n'est pas nécessaire pour la l'exactitude.

UnusedFunctionParameterPruner

Cette étape supprime les paramètres inutilisés dans une fonction.

Si un paramètre est inutilisé, comme c et y dans, fonction `f(a,b,c) -> x, y { x := div(a,b) }`, on supprime le paramètre et créons une nouvelle fonction de « liaison » comme suit :

```
function f(a,b) -> x { x := div(a,b) }
function f2(a,b,c) -> x, y { x := f(a,b) }
```

et remplace toutes les références à f par f2. L'inliner doit être exécuté ensuite pour s'assurer que toutes les références à f2 sont remplacées par f.

Conditions préalables : Disambiguator, FunctionHoister, LiteralRematerialiser.

L'étape LiteralRematerialiser n'est pas nécessaire pour l'exactitude. Elle permet de traiter des cas tels que : fonction `f(x) -> y { revert(y, y) }` où le littéral y sera remplacé par sa valeur 0, ce qui nous permet de réécrire la fonction.

EquivalentFunctionCombiner

Si deux fonctions sont syntaxiquement équivalentes, tout en autorisant le renommage de variables mais pas de réorganisation, toute référence à l'une des fonctions est remplacée par l'autre.

La suppression effective de la fonction est effectuée par l'élagueur inutilisé.

Mise en ligne des fonctions

ExpressionInliner

Ce composant de l'optimiseur effectue une mise en ligne restreinte des fonctions en mettant en ligne les fonctions qui peuvent être inlined à l'intérieur des expressions fonctionnelles, c'est-à-dire les fonctions qui :

- retournent une seule valeur
- ont un corps tel que `r := <expression fonctionnelle>`
- ne font ni référence à elles-mêmes ni à r dans la partie droite.

De plus, pour tous les paramètres, tous les éléments suivants doivent être vrais :

- L'argument est mobile.
- Le paramètre est soit référencé moins de deux fois dans le corps de la fonction, soit l'argument est plutôt bon marché (« coût » d'au plus 1, comme une constante jusqu'à 0xff).

Exemple : La fonction à inliner a la forme de fonction `f(...) -> r { r := E }` où E est une expression qui ne fait pas référence à r et tous les arguments de l'appel de fonction sont des expressions mobiles.

Le résultat de cet inlining est toujours une seule expression.

Ce composant ne peut être utilisé que sur des sources ayant des noms uniques.

FullInliner

Le Full Inliner remplace certains appels de certaines fonctions par le corps de la fonction. Ceci n'est pas très utile dans la plupart des cas, car cela ne fait qu'augmenter la taille du code sans en tirer aucun avantage. De plus, le code est généralement très coûteux et nous préférons souvent avoir un code plus court qu'un code plus efficace. Dans certains cas, cependant, l'inlining d'une fonction peut avoir des effets positifs sur les étapes suivantes de l'optimiseur. C'est le cas si l'un des arguments de la fonction est une constante, par exemple.

Pendant l'inlining, une heuristique est utilisée pour déterminer si l'appel de fonction doit être inline ou non. L'heuristique actuelle n'inline pas les « grandes » fonctions, à moins que la fonction appelée est minuscule. Les fonctions qui ne sont utilisées qu'une seule fois sont inlined, ainsi que les fonctions de taille moyenne, tandis que les appels de fonction avec des arguments constants permettent des fonctions légèrement plus grandes.

À l'avenir, nous pourrions inclure un composant de retour en arrière qui, au lieu d'inliner immédiatement une fonction, ne fait que la spécialiser, ce qui signifie qu'une copie de la fonction est générée où un certain paramètre est toujours remplacé par une constante. Après cela, nous pouvons exécuter l'optimiseur sur cette fonction spécialisée. Si cela résulte en des gains importants, la fonction spécialisée est conservée, sinon la fonction originale est utilisée à la place.

Nettoyage

Le nettoyage est effectué à la fin de l'exécution de l'optimiseur. Il essaie de combiner à nouveau les expressions divisées en expressions profondément imbriquées, améliore également la « compilabilité » pour les machines à pile en éliminant les variables autant que possible.

ExpressionJoiner

C'est l'opération inverse du séparateur d'expression. Elle transforme une séquence de déclarations de variables qui ont exactement une référence en une expression complexe. Cette étape préserve entièrement l'ordre des appels de fonctions et des exécutions d'opcodes. Elle n'utilise aucune information concernant la commutativité des opcodes; si le déplacement de la valeur d'une variable vers son lieu d'utilisation devait changer l'ordre d'un appel de fonction ou d'une exécution d'opcode, la transformation n'est pas effectuée.

Notez que le composant ne déplacera pas la valeur d'une affectation de variable ou une variable qui est référencée plus d'une fois.

Le snippet `let x := add(0, 2) let y := mul(x, mload(2))` n'est pas transformé, car il entraînerait l'ordre d'appel des opcodes `add` et `mload` - même si cela ne ferait pas de différence car `add` est mobile.

Lorsque l'on réordonne les opcodes de cette manière, les références de variables et les littéraux sont ignorés. Pour cette raison, l'extrait `let x := add(0, 2) let y := mul(x, 3)` est transformé en `let y := mul(x, 3)`. même si l'opcode `add` serait exécuté après l'évaluation du code serait exécuté après l'évaluation du littéral `3`.

SSAReverser

Il s'agit d'un petit pas qui permet d'inverser les effets de la transformation SSA si elle est combinée avec l'Éliminateur de sous-expression commune et l'Éliminateur d'élagueurs inutilisés.

La forme SSA que nous générons est préjudiciable à la génération de code sur l'EVM et sur WebAssembly car elle génère de nombreuses variables locales. Il serait préférable de réutiliser les variables existantes avec des affectations au lieu de de nouvelles déclarations de variables.

La transformation SSA réécrit

```
let a := calldataload(0)
mstore(a, 1)
```

à

```
let a_1 := calldataload(0)
let a := a_1
mstore(a_1, 1)
let a_2 := calldataload(0x20)
a := a_2
```

Le problème est qu'au lieu de `a`, la variable `a_1` est utilisée chaque fois que `a` est référencé. La transformation SSA modifie les déclarations de cette forme en échangeant simplement la déclaration et l'affectation. L'extrait ci-dessus est transformé en

```
let a := calldataload(0)
let a_1 := a
mstore(a_1, 1)
a := calldataload(0x20)
let a_2 := a
```

Il s'agit d'une transformation d'équivalence très simple, mais lorsque nous lançons maintenant l'éliminateur de sous-expression commune Common Subexpression Eliminator, il remplacera toutes les occurrences de `a_1` par `a` (jusqu'à ce que `a` soit réassigné). L'élagueur inutilisé va ensuite éliminer alors la variable `a_1` et inversera ainsi complètement la transformation SSA.

StackCompressor

Un problème qui rend la génération de code pour la machine virtuelle d'Ethereum est le fait qu'il y a une limite stricte de 16 emplacements pour atteindre la pile d'expression. Cela se traduit plus ou moins par une limite de 16 variables locales. Le compresseur de pile prend le code Yul et le compile en bytecode EVM. Chaque fois que la différence de pile est trop importante, il enregistre la fonction dans laquelle cela s'est produit.

Pour chaque fonction qui a causé un tel problème, le Rematerialiser est appelé avec une demande spéciale pour éliminer agressivement des variables spécifiques triées par le coût de leurs valeurs.

En cas d'échec, cette procédure est répétée plusieurs fois.

Rematerialiser

L'étape de rematérialisation tente de remplacer les références de variables par l'expression qui a été affectée en dernier lieu à la variable. Ceci n'est bien sûr bénéfique que si cette expression est comparativement bon marché à évaluer. En outre, elle n'est sémantiquement équivalente que si la valeur de l'expression n'a pas changé entre le point d'affectation et le point d'utilisation. Le principal avantage de cette étape est qu'elle peut économiser des emplacements de pile si elle conduit à l'élimination complète d'une variable (voir ci-dessous), mais elle peut aussi sauver un opcode DUP sur l'EVM si l'expression est très bon marché.

Le rematérialisateur utilise l'analyseur de flux de données pour suivre les valeurs actuelles des variables, qui sont toujours mobiles. Si la valeur est très bon marché ou si l'élimination de la variable a été explicitement demandée, la référence de la variable est remplacée par sa valeur actuelle.

ForLoopConditionOutOfBody

Inverse la transformation de ForLoopConditionIntoBody.

Pour tout mobile *c*, il se transforme en

```
for { ... } 1 { ... } {  
  if iszero(c) { break }  
  ...  
}
```

en

```
for { ... } c { ... } {  
  ...  
}
```

et il tourne

```
for { ... } 1 { ... } {  
  if c { break }  
  ...  
}
```

en

```
for { ... } iszero(c) { ... } {  
  ...  
}
```

Le `LiteralRematerialiser` doit être exécuté avant cette étape.

Spécifique à WebAssembly

MainFunction

Change le bloc le plus haut en une fonction avec un nom spécifique (« main ») qui n'a ni entrées ni sorties.

Dépend du Function Groupier.

3.21 Métadonnées du contrat

Le compilateur Solidity génère automatiquement un fichier JSON, le contrat qui contient des informations sur le contrat compilé. Vous pouvez utiliser ce fichier pour interroger la version du compilateur, les sources utilisées, l'ABI et la documentation NatSpec, pour interagir de manière plus sûre avec le contrat et vérifier son code source.

Le compilateur ajoute par défaut le hash IPFS du fichier de métadonnées à la fin du bytecode (pour plus de détails, voir ci-dessous) de chaque contrat, de sorte que vous pouvez le fichier de manière authentifiée sans avoir à recourir à un fournisseur de données centralisé. Les autres options disponibles sont le hachage Swarm et ne pas ajouter le hachage des métadonnées au bytecode. Elles peuvent être configurées via l'interface *Standard JSON Interface*.

Vous devez publier le fichier de métadonnées sur IPFS, Swarm, ou un autre service pour que que d'autres puissent y accéder. Vous créez le fichier en utilisant la commande `solc --metadata`, qui génère un fichier appelé `ContractName_meta.json`. Ce fichier contient les références IPFS et Swarm au code source et le fichier de métadonnées.

Le fichier de métadonnées a le format suivant. L'exemple ci-dessous est présenté de manière lisible par l'homme. Des métadonnées correctement formatées doivent utiliser correctement les guillemets, réduire les espaces blancs au minimum et trier les clés de tous les objets pour arriver à un formatage unique. Les commentaires ne sont pas autorisés et ne sont utilisés ici qu'à des fins explicatives.

```
{
  // Obligatoire : La version du format de métadonnées
  "version": "1",
  // Obligatoire : Langue du code source, sélectionne essentiellement une "sous-version"
  // de la spécification
  "language": "Solidity",
  // Obligatoire : Détails sur le compilateur, le contenu est spécifique
  // au langage.
  "compiler": {
    // Requis pour Solidity : Version du compilateur
    "version": "0.4.6+commit.2dabdf0.Emscripten.clang",
    // Facultatif : hachage du binaire du compilateur qui a produit cette sortie.
    "keccak256": "0x123..."
  },
  // Requis : Fichiers source de compilation/unités de source, les clés sont des noms de
  ↪ fichiers.
  "sources":
  {
    "myFile.sol": {
      // Requis : keccak256 hash du fichier source
      "keccak256": "0x123...",
      // Obligatoire (sauf si "content" est utilisé, voir ci-dessous) : URL(s) triée(s)
      // vers le fichier source, le protocole est plus ou moins arbitraire, mais une
      // une URL Swarm est recommandée
    }
  }
}
```

(suite sur la page suivante)

```

    "urls": [ "bzzr://56ab..." ],
    // Facultatif : Identifiant de la licence SPDX tel qu'indiqué dans le fichier.
↪source.
    "license": "MIT"
  },
  "destructible": {
    // Requis : keccak256 hash du fichier source
    "keccak256": "0x234...",
    // Obligatoire (sauf si "url" est utilisé) : contenu littéral du fichier source.
    "content": "contract destructible is owned { function destroy() { if (msg.sender.
↪== owner) selfdestruct(owner); } }"
  }
},
// Requis : Paramètres du compilateur
"settings":
{
  // Requis pour Solidity : Liste triée de réaffectations
  "remappings": [ ":g=/dir" ],
  // Facultatif : Paramètres de l'optimiseur. Les champs "enabled" et "runs" sont.
↪obsolètes
  // et ne sont fournis que pour des raisons de compatibilité ascendante.
  "optimizer": {
    "enabled": true,
    "runs": 500,
    "details": {
      // peephole a la valeur par défaut "true".
      "peephole": true,
      // la valeur par défaut de l'inliner est "true".
      "inliner": true,
      // jumpdestRemover a la valeur par défaut "true".
      "jumpdestRemover": true,
      "orderLiterals": false,
      "deduplicate": false,
      "cse": false,
      "constantOptimizer": false,
      "yul": true,
      // Facultatif : Présent uniquement si "yul" est "true".
      "yulDetails": {
        "stackAllocation": false,
        "optimizerSteps": "dhfoDgvulfntUtnIf..."
      }
    }
  }
},
"metadata": {
  // Reflète le paramètre utilisé dans le json d'entrée, la valeur par défaut est.
↪false.
  "useLiteralContent": true,
  // Reflète le paramètre utilisé dans le json d'entrée, la valeur par défaut est
↪"ipfs".
  "bytecodeHash": "ipfs"
},
// Requis pour Solidity : Fichier et nom du contrat ou de la bibliothèque pour.
↪lesquels ces

```

(suite sur la page suivante)

(suite de la page précédente)

```

// métadonnées est créée pour.
"compilationTarget": {
  "myFile.sol": "MyContract"
},
// Requis pour Solidity : Adresses des bibliothèques utilisées
"libraries": {
  "MyLib": "0x123123..."
}
},
// Requis : Informations générées sur le contrat.
"output":
{
  // Requis : Définition ABI du contrat
  "abi": [/* ... */],
  // Requis : Documentation du contrat par l'utilisateur de NatSpec
  "userdoc": [/* ... */],
  // Requis : Documentation du contrat par le développeur NatSpec
  "devdoc": [/* ... */]
}
}

```

Avertissement : Comme le bytecode du contrat résultant contient le hachage des métadonnées par défaut, toute modification des métadonnées peut entraîner une modification du bytecode. Cela inclut changement de nom de fichier ou de chemin, et puisque les métadonnées comprennent un hachage de toutes les sources utilisées, un simple changement d'espace résulte en des métadonnées différentes, et un bytecode différent.

Note : La définition ABI ci-dessus n'a pas d'ordre fixe. Il peut changer avec les versions du compilateur. Cependant, à partir de la version 0.5.12 de Solidity, le tableau maintient un certain ordre.

3.21.1 Encodage du hachage des métadonnées dans le bytecode

Parce que nous pourrions supporter d'autres façons de récupérer le fichier de métadonnées à l'avenir, le mappage {"ipfs" : <Hachage IPFS>, "solc" : <version du compilateur>} est stockée CBOR-encodé. Puisque la cartographie peut contenir plus de clés (voir ci-dessous) et que le début de cet encodage n'est pas facile à trouver, sa longueur est ajoutée dans un encodage big-endian de deux octets. La version actuelle du compilateur Solidity ajoute généralement l'élément suivant à la fin du bytecode déployé.

```

0xa2
0x64 'i' 'p' 'f' 's' 0x58 0x22 <34 octets hachage IPFS>
0x64 's' 'o' 'l' 'c' 0x43 <Codage de la version sur 3 octets>
0x00 0x33

```

Ainsi, afin de récupérer les données, la fin du bytecode déployé peut être vérifiée, pour correspondre à ce modèle et utiliser le hachage IPFS pour récupérer le fichier.

Alors que les versions de solc utilisent un encodage de 3 octets de la version comme indiqué ci-dessus (un octet pour chaque numéro de version majeure, mineure et de patch), les versions préversées utiliseront à la place une chaîne de version complète incluant le hachage du commit et la date de construction.

Note : Le mappage CBOR peut également contenir d'autres clés, il est donc préférable de décoder complètement les données plutôt que de se fier à ce qu'elles commencent par `0xa264`. Par exemple, si des fonctionnalités expérimentales qui affectent la génération de code sont utilisées, le mappage contiendra également `"experimental" : true`.

Note : Le compilateur utilise actuellement le hachage IPFS des métadonnées par défaut, mais il peut aussi utiliser le hachage `bzzr1` ou un autre hachage à l'avenir, donc ne vous ne comptez pas sur cette séquence pour commencer avec `0xa2 0x64 'i' 'p' 'f' 's'`. Nous ajouterons peut-être des données supplémentaires à cette structure CBOR.

3.21.2 Utilisation pour la génération automatique d'interface et NatSpec

Les métadonnées sont utilisées de la manière suivante : Un composant qui veut interagir avec un contrat (par exemple Mist ou tout autre porte-monnaie) récupère le code du contrat, à partir de là, le hachage IPFS/Swarm d'un fichier qui est ensuite récupéré. Ce fichier est décodé en JSON dans une structure comme ci-dessus.

Le composant peut alors utiliser l'ABI pour générer automatiquement une interface utilisateur rudimentaire pour le contrat.

En outre, le portefeuille peut utiliser la documentation utilisateur NatSpec pour afficher un message de confirmation à l'utilisateur chaque fois qu'il interagit avec le contrat, ainsi qu'une demande d'autorisation pour la signature de la transaction.

Pour plus d'informations, lisez *Format de la spécification en langage naturel d'Ethereum (NatSpec)*.

3.21.3 Utilisation pour la vérification du code source

Afin de vérifier la compilation, les sources peuvent être récupérées sur IPFS/Swarm via le lien dans le fichier de métadonnées. Le compilateur de la version correcte (qui est vérifié pour faire partie des compilateurs « officiels ») est invoqué sur cette entrée avec les paramètres spécifiés. Le bytecode résultant est comparé aux données de la transaction de création ou aux données de l'opcode CREATE. Cela vérifie automatiquement les métadonnées puisque leur hachage fait partie du bytecode. Les données en excès correspondent aux données d'entrée du constructeur, qui doivent être décodées selon l'interface et présentées à l'utilisateur.

Dans le référentiel [sourcify \(npm package\)](#) vous pouvez voir un exemple de code qui montre comment utiliser cette fonctionnalité.

3.22 Spécification ABI pour les contrats

3.22.1 Conception de base

L'interface binaire d'application de contrat (ABI) est le moyen standard d'interagir avec les contrats dans l'écosystème Ethereum, à la fois depuis l'extérieur de la blockchain et pour l'interaction entre les contrats. de l'extérieur de la blockchain que pour l'interaction entre contrats. Les données sont codées en fonction de leur type, comme décrit dans cette spécification. L'encodage n'est pas autodécrit et nécessite donc un schéma pour être décodé.

Nous supposons que les fonctions d'interface d'un contrat sont fortement typées, connues au moment de la compilation et statiques. Nous supposons que tous les contrats auront les définitions d'interface de tous les contrats qu'ils appellent disponibles au moment de la compilation.

Cette spécification ne concerne pas les contrats dont l'interface est dynamique ou connue uniquement au moment de l'exécution.

3.22.2 Sélecteur de fonctions

Les quatre premiers octets des données d'appel d'une fonction spécifient la fonction à appeler. Il s'agit des premiers (gauche, ordre supérieur en big-endian) quatre octets du hachage Keccak-256 de la signature de la fonction. La signature est définie comme l'expression canonique du prototype de base sans spécificateur d'emplacement de données, c'est-à-dire qu'il s'agit de l'expression canonique de la fonction. spécificateur d'emplacement de données, c'est-à-dire le nom de la fonction avec la liste des types de paramètres entre parenthèses. Les types de paramètres sont séparés par une simple virgule - aucun espace n'est utilisé.

Note : Le type de retour d'une fonction ne fait pas partie de cette signature. Dans *Solidity's function overloading* les types de retour ne sont pas pris en compte. La raison est de garder la résolution d'appel de fonction indépendante du contexte. La description *JSON de l'ABI* contient cependant des entrées et des sorties.

3.22.3 Codage des arguments

À partir du cinquième octet, les arguments codés suivent. Ce codage est également utilisé à d'autres d'autres endroits, par exemple les valeurs de retour et les arguments d'événements sont codés de la même manière, sans les quatre octets spécifiant la fonction.

3.22.4 Types

Les types élémentaires suivants existent :

- `uint<M>` : type de nombre entier non signé de M bits, $0 < M \leq 256$, $M \% 8 == 0$. Par exemple, `uint32`, `uint8`, `uint256`.
- `int<M>` : type d'entier signé en complément à deux de M bits, $0 < M \leq 256$, $M \% 8 == 0$.
- `address` : équivalent à `uint160`, sauf pour l'interprétation supposée et le typage du langage. Pour calculer le sélecteur de fonction, on utilise `address`.
- `uint`, `int` : synonymes de `uint256`, `int256` respectivement. Pour calculer le sélecteur de fonction sélecteur de fonction, `uint256` et `int256` doivent être utilisés.
- `bool` : équivalent à `uint8` restreint aux valeurs 0 et 1. Pour le calcul du sélecteur de fonction, `bool` est utilisé.
- `fixed<M>x<N>` : nombre décimal signé en virgule fixe de M bits, $8 \leq M \leq 256$, $M \% 8 == 0$, et $0 < N \leq 80$, qui désigne la valeur v comme $v / (10^{**} N)$.
- `ufixed<M>x<N>` : variante non signée de `fixed<M>x<N>`.
- `fixed`, `ufixed` : synonymes de `fixed128x18`, `ufixed128x18` respectivement. Pour calculer le sélecteur de fonction, il faut utiliser `fixed128x18`` et `ufixed128x18``.
- `bytes<M>` : type binaire de M octets, $0 < M \leq 32$.
- `function` : une adresse (20 octets) suivie d'un sélecteur de fonction (4 octets). Encodé de manière identique à `bytes24`.

Le type de tableau (de taille fixe) suivant existe :

- `<type>[M]` : un tableau de longueur fixe de M éléments, $M \geq 0$, du type donné.

Les types de taille non fixe suivants existent :

- `bytes` : séquence d'octets de taille dynamique.
- `string` : chaîne unicode de taille dynamique supposée être encodée en UTF-8.
- `<type>[]` : un tableau de longueur variable d'éléments du type donné.

Les types peuvent être combinés en un tuple en les mettant entre parenthèses, séparés par des virgules :

- `(T1, T2, ..., Tn)` : tuple constitué des types `T1`, ..., `Tn`, $n \geq 0$

Il est possible de former des tuples de tuples, des tableaux de tuples et ainsi de suite. Il est également possible de former des n-uplets zéro (où $n == 0$).

Correspondance entre Solidity et les types ABI

Solidity supporte tous les types présentés ci-dessus avec les mêmes noms, à l'exception des tuples. L'exception des tuples. Par contre, certains types Solidity ne sont pas supportés par l'ABI par l'ABI. Le tableau suivant montre sur la colonne de gauche les types Solidity qui ne font pas partie de l'ABI et, dans la colonne de droite, les types ABI qui les représentent.

Solidity	ABI
adresse payable<address>` ``address`	
<i>contract</i>	address
<i>enum</i>	uint8
<i>types de valeurs définies par l'utilisateur</i>	
<i>struct</i>	tuple

Avertissement : Avant la version 0.8.0 les enums pouvaient avoir plus de 256 membres et étaient représentés par le plus petit type de plus petit type d'entier juste assez grand pour contenir la valeur de n'importe quel membre.

3.22.5 Critères de conception pour l'encodage

Le codage est conçu pour avoir les propriétés suivantes, qui sont particulièrement utiles si certains arguments sont des tableaux imbriqués :

1. Le nombre de lectures nécessaires pour accéder à une valeur est au plus égal à la profondeur de la valeur dans la structure du tableau d'arguments. dans la structure du tableau d'arguments, c'est-à-dire que quatre lectures sont nécessaires pour récupérer `a_i[k][l][r]`. Dans une version précédente de l'ABI, le nombre de lectures était linéairement proportionnel au nombre total de paramètres dynamiques dans le pire des cas. dynamiques dans le pire des cas.
2. Les données d'une variable ou d'un élément de tableau ne sont pas entrelacées avec d'autres données et elles sont relocalisables, c'est-à-dire qu'elles n'utilisent que des « adresses » relatives.

3.22.6 Spécification formelle de l'encodage

Nous distinguons les types statiques et dynamiques. Les types statiques sont codés sur place et les types dynamiques sont dynamiques sont codés à un emplacement alloué séparément après le bloc actuel.

Définition : Les types suivants sont appelés « dynamiques » :

- bytes
- Chaîne de caractères
- T[] pour tout T
- T[k] pour tout T dynamique et tout $k \geq 0$
- (T_1, \dots, T_k) si T_i est dynamique pour tout $1 \leq i \leq k$

Tous les autres types sont dits « statiques ».

Définition : `len(a)` est le nombre d'octets dans une chaîne binaire `a`. Le type de `len(a)` est supposé être `uint256`.

Nous définissons `enc`, le codage réel, comme une correspondance entre les valeurs des types ABI et les chaînes binaires telles que `len(enc(X))` dépend de la valeur de `X` si et seulement si le type de `X` est dynamique.

Définition : Pour toute valeur ABI `X`, on définit récursivement `enc(X)`, en fonction du type de `X`. du type de `X` qui est

- (T_1, \dots, T_k) pour $k \geq 0$ et tout type T_1, \dots, T_k
 $enc(X) = head(X(1)) \dots head(X(k)) \ tail(X(1)) \dots \ tail(X(k))$
où $X = (X(1), \dots, X(k))$ et tête » et » queue » sont définies comme suit pour » T_i » :

si T_i est statique :

$\text{head}(X(i)) = \text{enc}(X(i))$ et $\text{tail}(X(i)) = ""$ (la chaîne vide)

sinon, c'est-à-dire si T_i est dynamique :

$\text{head}(X(i)) = \text{enc}(\text{len}(\text{head}(X(1)) \dots \text{head}(X(k)) \text{tail}(X(1)) \dots \text{tail}(X(i-1))))$
 $\text{tail}(X(i)) = \text{enc}(X(i))$

Notez que dans le cas dynamique, $\text{head}(X(i))$ est bien défini car les longueurs des parties de tête parties de la tête ne dépendent que des types et non des valeurs. La valeur de $\text{head}(X(i))$ est le décalage du début de $\text{tail}(X(i))$. du début de $\text{tail}(X(i))$ par rapport au début de $\text{enc}(X)$.

— $T[k]$ pour tout T et k :

$\text{enc}(X) = \text{enc}([X[0], \dots, X[k-1]])$

c'est-à-dire qu'il est codé comme s'il s'agissait d'un tuple avec k éléments du même type.

— $T[]$ où X a k éléments (k est supposé être de type `uint256`) :

$\text{enc}(X) = \text{enc}(k) \text{enc}([X[0], \dots, X[k-1]])$

c'est-à-dire qu'il est encodé comme s'il s'agissait d'un tableau de taille statique k , préfixé par le le nombre d'éléments.

— `bytes`, de longueur k (qui est supposé être de type `uint256`) :

$\text{enc}(X) = \text{enc}(k) \text{pad_right}(X)$, c'est-à-dire que le nombre d'octets est codé sous forme de `uint256` suivi de la valeur réelle de X en tant que séquence d'octets, suivie par le nombre minimal d'octets zéro pour que $\text{len}(\text{enc}(X))$ soit un multiple de 32.

— Chaîne de caractères :

$\text{enc}(X) = \text{enc}(\text{enc_utf8}(X))$, c'est-à-dire que X est codé en UTF-8 et que cette valeur est interprétée comme étant du type `bytes` et encodée plus loin. Notez que la longueur utilisée dans ce codage est le nombre d'octets de la chaîne encodée en UTF-8, et non son nombre de caractères.

— `uint<M>` : $\text{enc}(X)$ est le codage big-endian de X , complété du côté gauche par des octets zéro. d'ordre supérieur (gauche) avec des octets zéro de sorte que la longueur soit de 32 octets.

— Adresse : comme dans le cas de `uint160`.

— `int<M>` : $\text{enc}(X)$ est le code de complément à deux big-endian de X , complété sur le côté supérieur (gauche) par des octets `0xff` pour les X négatifs et par des octets zéro pour les X non négatifs, de sorte que la longueur soit de 32 octets.

— `bool` : comme dans le cas de `uint8`, où 1 est utilisé pour `vrai` et 0 pour `false`.

— `fixed<M>x<N>` : $\text{enc}(X)$ est $\text{enc}(X * 10^{**N})$ où $X * 10^{**N}$ est interprété comme un `int256`.

— `fixed` : comme dans le cas `fixed128x18`

— `ufixed<M>x<N>` : $\text{enc}(X)$ est $\text{enc}(X * 10^{**N})$ où $X * 10^{**N}$ est interprété comme un `uint256`.

— `ufixed` : comme dans le cas `ufixed128x18`

— `bytes<M>` : $\text{enc}(X)$ est la séquence d'octets dans X remplie de zéros de queue jusqu'à une longueur de 32 octets.

Notez que pour tout X , $\text{len}(\text{enc}(X))$ est un multiple de 32.

3.22.7 Sélecteur de fonctions et codage des arguments

En somme, un appel à la fonction f avec les paramètres a_1, \dots, a_n est encodé comme suit

`function_selector(f) enc((a_1, ..., a_n))`

et les valeurs de retour v_1, \dots, v_k de f sont codées en tant que

`enc((v_1, ..., v_k))`

c'est-à-dire que les valeurs sont combinées en un tuple et codées.

(suite de la page précédente)

```

13 - 000000000000000000000000000000000000000000000000000000000000000000000000e0 - décalage pour
    ↪ "trois"
14 - 0000000000000000000000000000000000000000000000000000000000000003 - compte pour "un"
15 - 6f6e650000000000000000000000000000000000000000000000000000000000 - codage de "un"
16 - 0000000000000000000000000000000000000000000000000000000000000003 - compte pour "deux"
    ↪ ""
17 - 74776f000000000000000000000000000000000000000000000000000000000000 - codage de "deux"
18 - 0000000000000000000000000000000000000000000000000000000000000005 - compte pour
    ↪ "trois"
19 - 7468726565000000000000000000000000000000000000000000000000000000 - codage de "trois"

```

Le décalage `f` pointe vers le début du contenu du tableau `[[1, 2], [3]]` qui est la ligne 2 (64 octets); donc `f = 0x0000000000000000000000000000000000000000000000000000000000000040`.

Le décalage `g` pointe vers le début du contenu du tableau `["one", "two", "three"]` qui est la ligne 10 (320 octets); donc `g = 0x000000000000000000000000000000000000000000000000000000000000140`.

3.22.10 Événements

Les événements sont une abstraction du protocole de journalisation et de surveillance des événements d'Ethereum. Les entrées de journal fournissent l'adresse du contrat du contrat, une série de quatre sujets maximum et des données binaires de longueur arbitraire. Les événements exploitent la fonction existante ABI existante afin d'interpréter ceci (avec une spécification d'interface) comme une structure correctement typée.

Étant donné un nom d'événement et une série de paramètres d'événement, nous les divisons en deux sous-séries : celles qui sont indexées et celles qui ne le sont pas. ceux qui ne le sont pas. Ceux qui sont indexés, dont le nombre peut aller jusqu'à 3 (pour les événements non anonymes) ou 4 (pour les événements anonymes), sont utilisés avec le hachage Keccak de la signature de l'événement pour former les sujets de l'entrée du journal. Ceux qui ne sont pas indexés forment le tableau d'octets de l'événement.

En fait, une entrée de journal utilisant cette ABI est décrite comme suit :

- `address` : l'adresse du contrat (intrinsèquement fournie par Ethereum);
- `topics[0]` : `keccak(EVENT_NAME+ "("+EVENT_ARGS.map(canonical_type_of).join(",")+")")` (`canonical_type_of` est une fonction qui renvoie simplement le nom du contrat. est une fonction qui renvoie simplement le type canonique d'un argument donné, par exemple, pour `uint` indexé `foo`, elle renverrait retournerait `uint256`). Cette valeur n'est présente dans `topics[0]` que si l'événement n'est pas déclaré comme anonyme;
- `topics[n]` : `abi_encode(EVENT_INDEXED_ARGS[n - 1])` si l'événement n'est pas déclaré comme étant anonyme. ou `abi_encode(EVENT_INDEXED_ARGS[n])` s'il l'est (`EVENT_INDEXED_ARGS` est la série des `EVENT_ARGS` qui sont sont indexés);
- `data` : qui ne sont pas indexés, `abi_encode` est la fonction d'encodage ABI utilisée pour retourner une série de valeurs typées d'une fonction, comme décrit ci-dessus).

Pour tous les types d'une longueur maximale de 32 octets, le tableau `EVENT_INDEXED_ARGS` contient la valeur directement, avec un padding ou une extension de signe (pour les entiers signés) à 32 octets, comme pour le codage ABI normal. Cependant, pour tous les types « complexes » ou de longueur dynamique, y compris tous les tableaux, `string`, `bytes` et `structs`, `EVENT_INDEXED_ARGS` contiendra le hachage *Keccak* d'une valeur spéciale encodée sur place (voir *Codage des paramètres d'événements indexés*), plutôt que la valeur encodée directement. Cela permet aux applications d'interroger efficacement les valeurs de types de longueur dynamique dynamiques (en définissant le hachage de la valeur encodée comme sujet), mais les applications ne peuvent pas de décoder les valeurs indexées qu'elles n'ont pas demandées. Pour les types de longueur dynamique, les développeurs d'applications doivent faire un compromis entre la recherche rapide de valeurs prédéterminées prédéterminées (si l'argument est indexé) et la lisibilité de valeurs arbitraires (ce qui exige que les arguments ne soient pas indexés). que les arguments ne soient pas indexés). Les développeurs peuvent surmonter ce compromis et atteindre à la fois recherche efficace et la lisibilité arbitraire en définissant

des événements avec deux arguments - un indexés, l'autre non - destinés à contenir la même valeur.

3.22.11 Erreurs

En cas d'échec à l'intérieur d'un contrat, celui-ci peut utiliser un opcode spécial pour interrompre l'exécution et annuler tous les changements d'état. En plus de ces effets, des données descriptives peuvent être retournées à l'appelant. Ces données descriptives sont le codage d'une erreur et de ses arguments de la même manière que les données d'un appel de fonction.

A titre d'exemple, considérons le contrat suivant dont la fonction `transfer` se retourne toujours se retourne avec une erreur personnalisée de « solde insuffisant » :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract TestToken {
    error InsufficientBalance(uint256 available, uint256 required);
    function transfer(address /*to*/, uint amount) public pure {
        revert InsufficientBalance(0, amount);
    }
}
```

Les données de retour seraient codées de la même manière que l'appel de fonction `InsufficientBalance(0, amount)` à la fonction `InsufficientBalance(uint256, uint256)`, c'est-à-dire `0xcf479181, uint256(0), uint256(montant)`.

Les sélecteurs d'erreur `0x00000000` et `0xffffffff` sont réservés pour une utilisation future.

Avertissement : Ne faites jamais confiance aux données d'erreur. Par défaut, les données d'erreur remontent à travers la chaîne d'appels externes, ce qui signifie que ce qui signifie qu'un contrat peut recevoir une erreur qui n'est définie dans aucun des contrats qu'il appelle directement. De plus, tout contrat peut simuler n'importe quelle erreur en renvoyant des données qui correspondent à une signature d'erreur, même si l'erreur n'est définie nulle part.

3.22.12 JSON

Le format JSON de l'interface d'un contrat est donné par un tableau de descriptions de fonctions, d'événements et d'erreurs. Une description de fonction est un objet JSON avec les champs :

- `type` : "fonction", "constructeur", "receive" (la fonction « *receive Ether* ») ou "fallback" (la fonction « *default* »);
- `name` : le nom de la fonction;
- `inputs` : un tableau d'objets, chacun d'entre eux contenant :
 - `name` : le nom du paramètre.
 - `type` : le type canonique du paramètre (plus bas).
 - `components` : utilisé pour les types de tuple (plus bas).
- `outputs` : un tableau d'objets similaires aux `inputs`.
- `stateMutability` : une chaîne avec l'une des valeurs suivantes : `pure` (spécifié pour ne pas lire l'état de la blockchain), `view` (*spécifié pour ne pas modifier l'état de la blockchain state*), `nonpayable` (la fonction n'accepte pas les Ether - la valeur par défaut) et `payable` (la fonction accepte les Ether).

Le constructeur et la fonction de repli n'ont jamais de `name` ou de `outputs`. La fonction de repli n'a pas non plus de `inputs`.

Note : Envoyer un Ether non nul à une fonction non payante inversera la transaction.

Note : L'état de mutabilité « non-payable » est reflété dans Solidity en ne spécifiant pas de modificateur d'état du tout, un modificateur d'état mutable.

Une description d'événement est un objet JSON avec des champs assez similaires :

- `type` : toujours « événement ».
- `name` : le nom de l'événement.
- `inputs` : un tableau d'objets, chacun d'entre eux contenant :
 - `name` : le nom du paramètre.
 - `type` : le type canonique du paramètre (plus bas).
 - `components` : utilisé pour les types de tuple (plus bas).
 - `indexed` : `true` si le champ fait partie des sujets du journal, `false` s'il fait partie du segment de données du journal.
- `anonymous` : `true` si l'événement a été déclaré comme `"anonymous"`.

Les erreurs se présentent comme suit :

- `type` : toujours `"erreur"`.
- `name` : le nom de l'erreur.
- `inputs` : un tableau d'objets, chacun d'entre eux contenant :
 - `name` : le nom du paramètre.
 - `type` : le type canonique du paramètre (plus bas).
 - `components` : utilisé pour les types de tuple (plus bas).

Note : Il peut y avoir plusieurs erreurs avec le même nom et même avec une signature identique signature identique dans le tableau JSON, par exemple si les erreurs proviennent de différents fichiers différents dans le contrat intelligent ou sont référencées à partir d'un autre contrat intelligent. Pour l'ABI, seul le nom de l'erreur elle-même est pertinent et non l'endroit où elle est définie.

Par exemple,

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract Test {
    constructor() { b = hex"12345678901234567890123456789012"; }
    event Event(uint indexed a, bytes32 b);
    event Event2(uint indexed a, bytes32 b);
    error InsufficientBalance(uint256 available, uint256 required);
    function foo(uint a) public { emit Event(a, b); }
    bytes32 b;
}
```

donnerait le JSON :

```
[{
  "type": "error",
  "inputs": [{"name": "available", "type": "uint256"}, {"name": "required", "type": "uint256"}],
  "name": "InsufficientBalance"
}, {
```

(suite sur la page suivante)

(suite de la page précédente)

```

"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": "function",
"inputs": [{"name": "a", "type": "uint256"}],
"name": "foo",
"outputs": []
}]

```

Handling tuple types

Bien que les noms ne fassent intentionnellement pas partie de l'encodage ABI, il est tout à fait logique de les inclure dans le JSON pour pouvoir l'afficher à l'utilisateur final. La structure est imbriquée de la manière suivante :

Un objet avec des membres `name`, `type`` et potentiellement `components`` décrit une variable typée. Le type canonique est déterminé jusqu'à ce qu'un type de tuple soit atteint et la description de la chaîne de caractères jusqu'à ce point est stockée dans `l'objet``. jusqu'à ce point est stockée dans le préfixe `type`` avec le mot `tuple`, c'est-à-dire que ce sera `tuple` suivi par une séquence de `[]` et de `[k]` avec des entiers `k`. Les composants du tuple sont ensuite stockés dans le membre `components`, qui est de type tableau et a la même structure que l'objet de niveau supérieur, sauf que `indexed` n'y est pas autorisé.

A titre d'exemple, le code

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.5 <0.9.0;
pragma abicoder v2;

contract Test {
    struct S { uint a; uint[] b; T[] c; }
    struct T { uint x; uint y; }
    function f(S memory, T memory, uint) public pure {}
    function g() public pure returns (S memory, T memory, uint) {}
}

```

donnerait le JSON :

```

[
  {
    "name": "f",
    "type": "function",
    "inputs": [
      {
        "name": "s",
        "type": "tuple",

```

(suite sur la page suivante)

```
"components": [
  {
    "name": "a",
    "type": "uint256"
  },
  {
    "name": "b",
    "type": "uint256[]"
  },
  {
    "name": "c",
    "type": "tuple[]",
    "components": [
      {
        "name": "x",
        "type": "uint256"
      },
      {
        "name": "y",
        "type": "uint256"
      }
    ]
  }
],
{
  "name": "t",
  "type": "tuple",
  "components": [
    {
      "name": "x",
      "type": "uint256"
    },
    {
      "name": "y",
      "type": "uint256"
    }
  ]
},
{
  "name": "a",
  "type": "uint256"
}
],
"outputs": []
}
```

3.22.13 Mode de codage strict

Le mode d'encodage strict est le mode qui conduit exactement au même encodage que celui défini dans la spécification formelle ci-dessus. Cela signifie que les décalages doivent être aussi petits que possible tout en ne créant pas de chevauchements dans les zones de données autorisées.

Habituellement, les décodeurs ABI sont écrits de manière simple en suivant simplement les pointeurs de décalage, mais certains décodeurs peuvent appliquer un mode strict. Le décodeur Solidity ABI n'applique pas actuellement le mode strict, mais l'encodeur crée toujours des données en mode strict.

3.22.14 Mode Packed non standard

Grâce à `abi.encodePacked()`, Solidity prend en charge un mode packed non standard dans lequel :

- les types plus courts que 32 octets sont concaténés directement, sans remplissage ni extension de signe
- les types dynamiques sont encodés in-place et sans la longueur.
- les éléments de tableaux sont rembourrés, mais toujours encodés in-place.

De plus, les structs ainsi que les tableaux imbriqués ne sont pas supportés.

A titre d'exemple, l'encodage de `int16(-1)`, `bytes1(0x42)`, `uint16(0x03)`, `string("Hello, world !")` donne le résultat suivant :

```
0xffff42000348656c6c6f2c20776f726c6421
  ^^^^                               int16(-1)
    ^^                               bytes1(0x42)
      ^^^^                           uint16(0x03)
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^ string("Hello, world!") sans champ de longueur
```

Plus précisément :

- Pendant l'encodage, tout est encodé sur place. Cela signifie qu'il n'y a pas de distinction entre la tête et la queue, comme dans l'encodage ABI, et la longueur d'un tableau n'est pas encodée.
- Les arguments directs de `abi.encodePacked` sont encodés sans padding, tant qu'ils ne sont pas des tableaux (ou des `string` ou des `bytes`).
- L'encodage d'un tableau est la concaténation de l'encodage de ses éléments **avec*****. codage de ses éléments **avec** remplissage.
- Les types de taille dynamique comme `string`, `bytes` ou `uint[]` sont encodés sans leur champ de longueur.
- L'encodage de `string` ou `bytes` n'applique pas de remplissage à la fin sauf s'il s'agit d'une partie d'un tableau ou d'une structure (dans ce cas, il s'agit d'un multiple de 32 octets). 32 octets).

En général, l'encodage est ambigu dès qu'il y a deux éléments de taille dynamique, à cause du champ de longueur manquant.

Si le remplissage est nécessaire, des conversions de type explicites peuvent être utilisées : `abi.encodePacked(uint16(0x12)) == hex "0012"`.

Puisque le codage empacoté n'est pas utilisé lors de l'appel de fonctions, il n'y a pas de prise en charge particulière pour faire précéder un sélecteur de fonction. Comme l'encodage est ambigu, il n'y a pas de fonction de décodage.

Avertissement : Si vous utilisez `keccak256(abi.encodePacked(a, b))` et que `a` et `b` sont tous deux des types dynamiques, il est facile de créer des collisions dans la valeur de hachage en déplaçant des parties de `a` dans `b` et vice-versa. Plus précisément, `abi.encodePacked("a", "bc") == abi.encodePacked("ab", "c")`. Si vous utilisez `abi.encodePacked` pour des signatures, l'authentification ou l'intégrité de données d'utiliser toujours les mêmes types et de vérifier qu'au plus l'un d'entre eux est dynamique. À moins qu'il n'y ait une raison impérative, `abi.encode` devrait être préféré.

3.22.15 Codage des paramètres d'événements indexés

Les paramètres d'événements indexés qui ne sont pas des types de valeur, c'est-à-dire les tableaux et les stockés directement, mais un keccak256-hash d'un encodage est stocké. Ce codage est défini comme suit :

- l'encodage d'une valeur de type `bytes` et `chaîne` est juste le contenu de la chaîne de caractères sans aucun padding ou préfixe de longueur.
- l'encodage d'une structure est la concaténation de l'encodage de ses membres, toujours complétés par un multiple de 32 octets (même `bytes` et `string`).
- Le codage d'un tableau (de taille dynamique ou statique) est le suivant concaténation des encodages de ses éléments, toujours complétés par un multiple de 32 de 32 octets (même `bytes` et `string`) et sans préfixe de longueur.

Dans l'exemple ci-dessus, comme d'habitude, un nombre négatif est paddé par extension de signe et non paddé à zéro. Les types `bytesNN` sont paddés à droite tandis que les types `uintNN` / `intNN` sont paddés à gauche.

Avertissement : Le codage d'une structure est ambigu s'il contient plus d'un tableau de taille dynamique. dynamique. Pour cette raison, vérifiez toujours à nouveau les données de l'événement et ne vous fiez pas au résultat de la recherche basé uniquement sur les paramètres indexés.

3.23 Solidity v0.5.0 Changements de rupture

Cette section met en évidence les principaux changements introduits dans la version 0.5.0 de Solidity, ainsi que les raisons de ces changements et la façon de mettre à jour le code concerné. Pour la liste complète, consultez [le journal des modifications de la version](#).

Note : Les contrats compilés avec Solidity v0.5.0 peuvent toujours s'interfacer avec des contrats et même des bibliothèques compilés avec des versions plus anciennes sans avoir à les recompiler ou à les redéployer. Il suffit de modifier les interfaces pour inclure les emplacements des données et les spécificateurs de visibilité et de mutabilité. Voir la section *Interopérabilité avec les contrats plus anciens* en dessous.

3.23.1 Changements uniquement sémantiques

Cette section énumère les changements qui sont uniquement sémantiques, donc potentiellement cacher un comportement nouveau et différent dans le code existant.

- Le décalage signé vers la droite utilise maintenant le décalage arithmétique approprié, c'est-à-dire qu'il arrondit vers l'infini négatif au lieu d'arrondir vers zéro. l'infini négatif, au lieu d'arrondir vers zéro. Les décalages signés et non signés auront des opcodes dédiés dans Constantinople, et sont émulés par Solidity pour le moment. Solidity pour le moment.
- La déclaration `continue` dans une boucle `do...while` saute maintenant au comportement commun dans de tels cas. Auparavant, il sautait vers le corps de la boucle. Ainsi, si la condition est fausse, la boucle se termine.
- Les fonctions `.call()`, `.delegatecall()` et `.staticcall()` ne tamponnent plus lorsqu'on leur donne un seul paramètre `bytes`.
- Les fonctions `Pure` et `View` sont désormais appelées en utilisant l'opcode `STATICCALL` au lieu de `CALL` si la version de l'EVM est Byzantium ou ultérieure. Cela interdit les changements d'état au niveau de l'EVM.
- L'encodeur ABI pallie désormais correctement les tableaux d'octets et les chaînes de caractères des données d'appel (`msg.data` et paramètres de fonctions externes) lorsqu'ils sont utilisés dans des appels externes et dans `abi.encode`. Pour un encodage non codé, utilisez `abi.encodePacked`.

- Le décodeur ABI revient en arrière au début des fonctions et dans `abi.decode()` si les données d'appel passées sont trop courtes ou pointent hors des limites. Notez que les bits d'ordre supérieur sales sont toujours simplement ignorés.
- Transférer tout le gaz disponible avec des appels de fonctions externes à partir de Tangerine Whistle.

3.23.2 Changements sémantiques et syntaxiques

Cette section met en évidence les changements qui affectent la syntaxe et la sémantique.

- Les fonctions `.call()`, `.delegatecall()`, `staticcall()`, `keccak256()`, `sha256()` et `ripemd160()` n'acceptent plus qu'un seul argument `bytes`. `unique`, `bytes`. De plus, l'argument n'est pas paddé. Ceci a été changé pour rendre plus explicite et clair la façon dont les arguments sont concaténés. Changez chaque `.call()` (et famille) en un `.call("")` et chaque `.call(signature, a, b, c)` en utilisant `.call(abi.encodeWithSignature(signature, a, b, c))` (le dernier ne fonctionne que pour les types dernière ne fonctionne que pour les types de valeurs). Changez chaque `keccak256(a, b, c)` en `keccak256(abi.encodePacked(a, b, c))`. Même s'il ne s'agit pas d'une il est suggéré que les développeurs changent `x.call(bytes4(keccak256("f(uint256)")), a, b)` en `x.call(abi.encodeWithSignature("f(uint256)", a, b))`.
- Les fonctions `.call()`, `.delegatecall()` et `.staticcall()` retournent maintenant `(bool, bytes memory)` pour donner accès aux données de retour. Modifier `bool success = otherContract.call("f")` en `(bool success, bytes memory données) = otherContract.call("f")`.
- Solidity met désormais en œuvre les règles de délimitation du style C99 pour les locales de fonctions, c'est-à-dire que les variables ne peuvent être utilisées que déclarées et seulement dans le même périmètre ou dans des périmètres imbriqués. Les variables déclarées dans le bloc d'initialisation d'une boucle `for` sont valides en tout point de la boucle. `boucle`.

3.23.3 Exigences d'explicitation

Cette section liste les modifications pour lesquelles le code doit être plus explicite. Pour la plupart des sujets, le compilateur fournira des suggestions.

- La visibilité explicite des fonctions est maintenant obligatoire. Ajouter `public` à chaque fonction et constructeur fonction et constructeur, et `external` à chaque fonction de fallback ou d'interface d'interface qui ne spécifie pas déjà sa visibilité.
- La localisation explicite des données pour toutes les variables de type `struct`, `array` ou `mapping` est maintenant obligatoire. Ceci s'applique également aux paramètres des fonctions et aux de retour. Par exemple, changez `uint[] x = m_x` en `uint[] storage x = m_x`, et fonction `f(uint[][] x)` en fonction `f(uint[][] mémoire x)` où « memory » est l'emplacement des données et peut être remplacé par « storage » ou « calldata ». `calldata` en conséquence. Notez que les fonctions externes requièrent des paramètres dont l'emplacement des données est `calldata`.
- Les types de contrats n'incluent plus les membres `addresses` afin de séparer les espaces de noms. Par conséquent, il est maintenant nécessaire de convertir explicitement les valeurs du type de contrat en `addresses` avant d'utiliser une membre `address`. Exemple : si `c` est un contrat, changez `c.transfert(...)` en `adresse(c).transfert(...)`, et `c.balance` en `adresse(c).balance`.
- Les conversions explicites entre des types de contrats non liés sont désormais interdites. Vous pouvez seulement convertir un type de contrat en l'un de ses types de base ou ancêtres. Si vous êtes sûr que un contrat est compatible avec le type de contrat vers lequel vous voulez le convertir, bien qu'il n'en hérite pas. bien qu'il n'en hérite pas, vous pouvez contourner ce problème en convertissant d'abord en `adresse`. Exemple : si `A` et `B` sont des types de contrat, `B` n'hérite pas de `A` et `b` est un contrat de type `B`, vous pouvez toujours convertir `b` en type `A` en utilisant `A(adresse(b))`. Notez que vous devez toujours faire attention aux fonctions de repli payantes correspondantes, comme expliqué ci-dessous.
- Le type « adresse » a été divisé en « adresse » et « adresse payable », où seule « l'adresse payable » fournit la fonction « transfert ». Un site Une « adresse payable » peut être directement convertie en une « adresse », mais l'inverse n'est pas autorisé. l'inverse n'est pas autorisé. La conversion de `adresse` en `adresse payable`

est possible par conversion via `uint160`. Si `c` est un contrat, `address(c)` résulte en `address payable` seulement si `c` possède une fonction de repli payable. Si vous utilisez le modèle *withdraw pattern*, vous n'avez probablement pas à modifier votre code car `transfer` est uniquement utilisé sur `msg.sender` au lieu des adresses stockées et `msg.sender` est une adresse payable.

- Les conversions entre `bytesX` et `uintY` de taille différente sont maintenant interdites en raison du remplissage de `bytesX` à droite et du remplissage de `uintY` à gauche, ce qui peut entraîner des résultats de conversion inattendus. La taille doit maintenant être ajustée dans le type avant la conversion. Par exemple, vous pouvez convertir un `bytes4` (4 octets) en un `uint64` (8 octets) en convertissant d'abord le `bytes4` en un `uint64`. en convertissant d'abord la variable `bytes4` en `bytes8`, puis en `uint64`. Vous obtenez le inverse en convertissant en `uint32`. Avant la version 0.5.0, toute conversion entre `bytesX` et `uintY` passait par `uint8X`. Par exemple, `uint8(bytes3(0x291807))` sera converti en `uint8(uint24(bytes3(0x291807)))` (le résultat est `0x07`).
- L'utilisation de `msg.value` dans des fonctions non payantes (ou son introduction par le biais d'un modificateur) est interdit par mesure de sécurité. Transformez la fonction en payante » ou créez une nouvelle fonction interne pour la logique du programme qui utilise `msg.value`.
- Pour des raisons de clarté, l'interface de la ligne de commande exige maintenant - si l'entrée standard est utilisée comme source.

3.23.4 Éléments dépréciés

Cette section liste les changements qui déprécient des fonctionnalités ou des syntaxes antérieures. Notez que plusieurs de ces changements étaient déjà activés dans le mode expérimental `v0.5.0`.

Interfaces en ligne de commande et JSON

- L'option de ligne de commande `--formal` (utilisée pour générer la sortie de Why3 pour une vérification formelle plus poussée) était dépréciée et est maintenant supprimée. Un nouveau module de vérification formelle, le `SMTChecker`, est activé via `pragma experimental SMTChecker`;
- L'option de ligne de commande `--julia` a été renommée en `--yul` en raison du changement de nom du langage intermédiaire `Yul`, en raison du changement de nom du langage intermédiaire « Julia » en « Yul ».
- Les options de ligne de commande `--clone-bin` et `--combined-json clone-bin` ont été supprimées.
- Les remappages avec un préfixe vide ne sont pas autorisés.
- Les champs AST JSON `constant` et `payable` ont été supprimés. L'adresse informations sont maintenant présentes dans le champ `stateMutability`.
- Le champ JSON AST `isConstructor` du noeud `FunctionDefinition` a été remplacé par un champ appelé `Functions`. a été remplacé par un champ appelé `kind` qui peut avoir la valeur `constructor`, `fallback` ou `function`.
- Dans les fichiers hexadécimaux binaires non liés, les adresses des bibliothèques sont maintenant les 36 premiers caractères hexadécimaux de la clé. sont désormais les 36 premiers caractères hexadécimaux du hachage `keccak256` du nom de bibliothèque nom de bibliothèque entièrement qualifié, entouré de `$. . $`. Auparavant, seul le nom complet de la bibliothèque était utilisé. Cela réduit les risques de collisions, en particulier lorsque de longs chemins sont utilisés. Les fichiers binaires contiennent maintenant aussi une liste de correspondances entre ces caractères de remplacement vers les noms pleinement qualifiés.

Constructeurs

- Les constructeurs doivent désormais être définis à l'aide du mot clé « constructeur ».
- L'appel de constructeurs de base sans parenthèses est désormais interdit.
- La spécification des arguments des constructeurs de base plusieurs fois dans la même hiérarchie d'héritage est maintenant interdit.
- L'appel d'un constructeur avec des arguments mais avec un nombre d'arguments incorrect est maintenant désapprouvé. Si vous souhaitez seulement spécifier une relation d'héritage sans donner d'arguments, ne fournissez pas de parenthèses du tout.

Fonctions

- La fonction `callcode` est maintenant désapprouvée (en faveur de `delegatecall`). Il est toujours possible de l'utiliser via l'assemblage en ligne.
- La fonction `suicide` n'est plus autorisée (au profit de `selfdestruct`).
- `sha3` n'est plus autorisé (au profit de `keccak256`).
- `throw` est maintenant désapprouvé (en faveur de `revert`, `require` et de `assert`).

Conversions

- Les conversions explicites et implicites des littéraux décimaux en types `bytesXX`` sont maintenant désactivées. est désormais interdit.
- Les conversions explicites et implicites de littéraux hexadécimaux en types `bytesXX`` de taille différente sont désormais interdites. de taille différente sont désormais interdites.

Littéraux et suffixes

- L'unité de dénomination « années » n'est plus autorisée en raison de complications et de confusions concernant les années bissextiles. complications et de confusions concernant les années bissextiles.
- Les points de fin de ligne qui ne sont pas suivis d'un nombre ne sont plus autorisés.
- La combinaison de nombres hexadécimaux avec des unités (par exemple, « `0x1e wei` ») n'est plus autorisée. interdites.
- Le préfixe `0X` pour les nombres hexadécimaux n'est plus autorisé, seul `0x` est possible.

Variables

- La déclaration de structures vides n'est plus autorisée pour des raisons de clarté.
- Le mot clé « `var` » n'est plus autorisé pour favoriser l'explicitation.
- Les affectations entre les tuples avec un nombre différent de composants sont maintenant interdites. désapprouvé.
- Les valeurs des constantes qui ne sont pas des constantes de compilation ne sont pas autorisées.
- Les déclarations multi-variables avec un nombre de valeurs non concordant sont maintenant désapprouvées.
- Les variables de stockage non initialisées ne sont plus autorisées.
- Les composants de tuple vides ne sont plus admis.
- La détection des dépendances cycliques dans les variables et les structures est limitée en récursion à 256. récursion à 256.
- Les tableaux de taille fixe avec une longueur de zéro ne sont plus autorisés.

Syntaxe

- L'utilisation de `constant` comme modificateur de mutabilité de l'état de la fonction est désormais interdite.
- Les expressions booléennes ne peuvent pas utiliser d'opérations arithmétiques.
- L'opérateur unaire « `+` » n'est plus autorisé.
- Les littéraux ne peuvent plus être utilisés avec `abi.encodePacked` sans conversion préalable vers un type explicite.
- Les déclarations de retour vides pour les fonctions avec une ou plusieurs valeurs de retour ne sont plus désormais interdites.
- La syntaxe « loose assembly », c'est-à-dire les étiquettes de saut, est maintenant totalement interdite, les sauts et les instructions non fonctionnelles ne peuvent plus être utilisés. Utilisez les nouvelles fonctions `while`, `switch` et `if` à la place.
- Les fonctions sans implémentation ne peuvent plus utiliser de modificateurs.
- Les types de fonctions avec des valeurs de retour nommées ne sont plus autorisés.
- Les déclarations de variables d'une seule déclaration à l'intérieur de corps `if/while/for` qui ne sont pas qui ne sont pas des blocs ne sont plus autorisées.
- Nouveaux mots-clés : `calldata` et `constructor`.
- Nouveaux mots-clés réservés : `alias`, `apply`, `auto`, `copyof`, `définir''`, ```immutable''`, ```implements''`, ```macro''`, ```mutable''`, ```override`, `partiel`, `promise`, `reference`, `sealed`, ```sizeof''`, ```supports''`, ```typedef''` et ```unchecked''`.

3.23.5 Interopérabilité avec les anciens contrats

Il est toujours possible de s'interfacer avec des contrats écrits pour des versions de Solidity antérieures à la v0.5.0 (ou l'inverse) en définissant des interfaces pour eux. Considérons que vous avez le contrat suivant, antérieur à la version 0.5.0, déjà déployé :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.4.25;
// This will report a warning until version 0.4.25 of the compiler
// This will not compile after 0.5.0
contract OldContract {
    function someOldFunction(uint8 a) {
        //...
    }
    function anotherOldFunction() constant returns (bool) {
        //...
    }
    // ...
}
```

Il ne compilera plus avec Solidity v0.5.0. Cependant, vous pouvez lui définir une interface compatible :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
interface OldContract {
    function someOldFunction(uint8 a) external;
    function anotherOldFunction() external returns (bool);
}
```

Notez que nous n'avons pas déclaré « `anotherOldFunction` » comme étant « `view` », bien qu'elle soit déclarée « `constante` » dans le contrat original. Cela est dû au fait qu'à partir de la version 0.5.0 de Solidity, l'option `staticcall` est utilisée pour appeler les fonctions `view`. Avant la v0.5.0, le mot-clé `constant` n'était

pas appliqué, donc appeler une fonction déclarée constante avec `staticcall` peut encore se retourner, puisque la fonction constant peut encore tenter de modifier le stockage. Par conséquent, lorsque vous définissez une pour des contrats plus anciens, vous ne devriez utiliser `view` à la place de `constant` que si vous êtes absolument sûr que la fonction fonctionnera avec `staticcall`.

Avec l'interface définie ci-dessus, vous pouvez maintenant facilement utiliser le contrat pré-0.5.0 déjà déployé :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

interface OldContract {
    function someOldFunction(uint8 a) external;
    function anotherOldFunction() external returns (bool);
}

contract NewContract {
    function doSomething(OldContract a) public returns (bool) {
        a.someOldFunction(0x42);
        return a.anotherOldFunction();
    }
}
```

De même, les bibliothèques pré-0.5.0 peuvent être utilisées en définissant les fonctions de la bibliothèque sans implémentation et en en fournissant l'adresse de la bibliothèque pré-0.5.0 lors de l'édition de liens (voir `:ref:`\`commandline-compiler`` pour savoir comment utiliser le pour savoir comment utiliser le compilateur en ligne de commande pour l'édition de liens) :

```
// This will not compile after 0.6.0
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.5.0;

library OldLibrary {
    function someFunction(uint8 a) public returns(bool);
}

contract NewContract {
    function f(uint8 a) public returns (bool) {
        return OldLibrary.someFunction(a);
    }
}
```

3.23.6 Exemple

L'exemple suivant montre un contrat et sa version mise à jour pour Solidity v0.5.0 avec certaines des modifications énumérées dans cette section.

Ancienne version :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.4.25;
// This will not compile after 0.5.0

contract OtherContract {
```

(suite sur la page suivante)

```
uint x;
function f(uint y) external {
    x = y;
}
function() payable external {}
}

contract Old {
    OtherContract other;
    uint myNumber;

    // Function mutability not provided, not an error.
    function someInteger() internal returns (uint) { return 2; }

    // Function visibility not provided, not an error.
    // Function mutability not provided, not an error.
    function f(uint x) returns (bytes) {
        // Var is fine in this version.
        var z = someInteger();
        x += z;
        // Throw is fine in this version.
        if (x > 100)
            throw;
        bytes memory b = new bytes(x);
        y = -3 >> 1;
        // y == -1 (wrong, should be -2)
        do {
            x += 1;
            if (x > 10) continue;
            // 'Continue' causes an infinite loop.
        } while (x < 11);
        // Call returns only a Bool.
        bool success = address(other).call("f");
        if (!success)
            revert();
        else {
            // Local variables could be declared after their use.
            int y;
        }
        return b;
    }

    // No need for an explicit data location for 'arr'
    function g(uint[] arr, bytes8 x, OtherContract otherContract) public {
        otherContract.transfer(1 ether);

        // Since uint32 (4 bytes) is smaller than bytes8 (8 bytes),
        // the first 4 bytes of x will be lost. This might lead to
        // unexpected behavior since bytesX are right padded.
        uint32 y = uint32(x);
        myNumber += y + msg.value;
    }
}
```

(suite de la page précédente)

}

Nouvelle version :

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.5.0;
// This will not compile after 0.6.0

contract OtherContract {
    uint x;
    function f(uint y) external {
        x = y;
    }
    function() payable external {}
}

contract New {
    OtherContract other;
    uint myNumber;

    // Function mutability must be specified.
    function someInteger() internal pure returns (uint) { return 2; }

    // Function visibility must be specified.
    // Function mutability must be specified.
    function f(uint x) public returns (bytes memory) {
        // The type must now be explicitly given.
        uint z = someInteger();
        x += z;
        // Throw is now disallowed.
        require(x <= 100);
        int y = -3 >> 1;
        require(y == -2);
        do {
            x += 1;
            if (x > 10) continue;
            // 'Continue' jumps to the condition below.
        } while (x < 11);

        // Call returns (bool, bytes).
        // Data location must be specified.
        (bool success, bytes memory data) = address(other).call("f");
        if (!success)
            revert();
        return data;
    }

    using address_make_payable for address;
    // Data location for 'arr' must be specified
    function g(uint[] memory /* arr */, bytes8 x, OtherContract otherContract, address_
↪unknownContract) public payable {
        // 'otherContract.transfer' is not provided.

```

(suite sur la page suivante)

```
// Since the code of 'OtherContract' is known and has the fallback
// function, address(otherContract) has type 'address payable'.
address(otherContract).transfer(1 ether);

// 'unknownContract.transfer' is not provided.
// 'address(unknownContract).transfer' is not provided
// since 'address(unknownContract)' is not 'address payable'.
// If the function takes an 'address' which you want to send
// funds to, you can convert it to 'address payable' via 'uint160'.
// Note: This is not recommended and the explicit type
// 'address payable' should be used whenever possible.
// To increase clarity, we suggest the use of a library for
// the conversion (provided after the contract in this example).
address payable addr = unknownContract.make_payable();
require(addr.send(1 ether));

// Since uint32 (4 bytes) is smaller than bytes8 (8 bytes),
// the conversion is not allowed.
// We need to convert to a common size first:
bytes4 x4 = bytes4(x); // Padding happens on the right
uint32 y = uint32(x4); // Conversion is consistent
// 'msg.value' cannot be used in a 'non-payable' function.
// We need to make the function payable
myNumber += y + msg.value;
}
}

// We can define a library for explicitly converting ``address``
// to ``address payable`` as a workaround.
library address_make_payable {
    function make_payable(address x) internal pure returns (address payable) {
        return address(uint160(x));
    }
}
```

3.24 Solidity v0.6.0 Changements de rupture

Cette section met en évidence les principaux changements de rupture introduits dans Solidity version 0.6.0, ainsi que le raisonnement derrière ces changements et la façon de mettre à jour code affecté. Pour la liste complète, consultez le changelog de la version.

3.24.1 Changements dont le compilateur pourrait ne pas être averti

Cette section liste les changements pour lesquels le comportement de votre code pourrait changer sans que le compilateur vous en avertisse.

- Le type résultant d'une exponentiation est le type de la base. Il s'agissait auparavant du plus petit type qui peut contenir à la fois le type de la base et le type de l'exposant, comme pour les opérations symétriques. symétriques. De plus, les types signés sont autorisés pour la base de l'exponentiation.

3.24.2 Exigences d'explicitation

Cette section liste les changements pour lesquels le code doit être plus explicite, mais la sémantique ne change pas. Pour la plupart des sujets, le compilateur fournira des suggestions.

- Les fonctions ne peuvent maintenant être surchargées que lorsqu'elles sont marquées avec la clef `virtual` ou définies dans une interface. Les fonctions sans implémentation en dehors d'une interface doivent être marquées `virtual`. Lorsqu'on surcharge une fonction ou un modificateur, le nouveau mot-clé `override` doit être utilisé. doit être utilisé. Lorsqu'on remplace une fonction ou un modificateur défini dans plusieurs bases parallèles bases parallèles, toutes les bases doivent être listées entre parenthèses après le mot-clé comme ceci : `override(Base1, Base2)`.
- L'accès des membres à `length` des tableaux est maintenant toujours en lecture seule, même pour les tableaux de stockage. Il n'est plus possible de plus possible de redimensionner des tableaux de stockage en assignant une nouvelle valeur à leur longueur. Utilisez `push()`, `push(value)` ou `pop()` à la place, ou assignez un tableau complet, qui écrasera bien sûr le contenu existant. La raison derrière cela est d'éviter les collisions de stockage de gigantesques de stockage gigantesques.
- Le nouveau mot-clé `abstract` peut être utilisé pour marquer les contrats comme étant abstraits. Il doit être utilisé si un contrat n'implémente pas toutes ses fonctions. Les contrats abstraits ne peuvent pas être créés en utilisant l'opérateur `new`, et il n'est pas possible de générer du bytecode pour eux pendant la compilation.
- Les bibliothèques doivent implémenter toutes leurs fonctions, pas seulement les fonctions internes.
- Les noms des variables déclarées en `inline assembly` ne peuvent plus se terminer par `_slot` ou `_offset`.
- Les déclarations de variables dans l'assemblage en ligne ne peuvent plus suivre une déclaration en dehors du bloc d'assemblage en ligne. Si le nom contient un point, son préfixe jusqu'au point ne doit pas entrer en conflit avec une déclaration en dehors du bloc d'assemblage en ligne. d'assemblage.
- Le shadowing de variables d'état est désormais interdit. Un contrat dérivé peut seulement déclarer une variable d'état `x`, que s'il n'y a pas de variable d'état visible avec le même nom d'état visible portant le même nom dans l'une de ses bases.

3.24.3 Changements sémantiques et syntaxiques

Cette section liste les changements pour lesquels vous devez modifier votre code et il fait quelque chose d'autre après.

- Les conversions de types de fonctions externes en adresse sont maintenant interdites. A la place, les types de fonctions externes Au lieu de cela, les types de fonctions externes ont un membre appelé `address`, similaire au membre `selector` existant.
- La fonction `push(value)` pour les tableaux de stockage dynamique ne retourne plus la nouvelle longueur (elle ne retourne rien).
- La fonction sans nom communément appelée « fonction de repli » a été divisée en une nouvelle fonction de repli définie à l'aide de la fonction de repli. nouvelle fonction de repli définie à l'aide du mot-clé `fallback` et une fonction de réception d'éther définie à l'aide du mot-clé `receive`.
 - Si elle est présente, la fonction de réception de l'éther est appelée chaque fois que les données d'appel sont vides (que l'éther soit reçu ou non). (que l'éther soit reçu ou non). Cette fonction est implicitement payable.
 - La nouvelle fonction de repli est appelée lorsqu'aucune autre fonction ne correspond (si la fonction `receive ether` n'existe pas, cela inclut les appels avec des données d'appel vides). Vous pouvez rendre cette fonction payable ou non. Si elle n'est pas « payante », alors les transactions ne correspondant à aucune autre fonction

qui envoie une valeur seront inversées. Vous n'aurez besoin d'implémenter implémenter la nouvelle fonction de repli que si vous suivez un modèle de mise à niveau ou de proxy.

3.24.4 Nouvelles fonctionnalités

Cette section énumère des choses qui n'étaient pas possibles avant la version 0.6.0 de Solidity ou qui étaient plus difficiles à réaliser.

- L'instruction *try/catch* vous permet de réagir à l'échec d'appels externes.
- Les types `struct` et `enum` peuvent être déclarés au niveau du fichier.
- Les tranches de tableau peuvent être utilisées pour les tableaux de données d'appel, par exemple «`abi.decode(msg.data[4:], (uint, uint))`». est un moyen de bas niveau pour décoder les données utiles de l'appel de fonction.
- Natspec prend en charge les paramètres de retour multiples dans la documentation du développeur, en appliquant le même contrôle de nommage que `@param`.
- Yul et Inline Assembly ont une nouvelle instruction appelée `leave` qui quitte la fonction courante.
- Les conversions de adresse en adresse payable sont maintenant possibles via `payable(x)`, où `x` doit être de type adresse.

3.24.5 Changements d'interface

Cette section liste les changements qui ne sont pas liés au langage lui-même, mais qui ont un effet sur les interfaces du compilateur. Ces modifications peuvent changer la façon dont vous utilisez le compilateur sur la ligne de commande, la façon dont vous utilisez son interface programmable, ou la façon dont vous analysez la sortie qu'il produit. ou comment vous analysez la sortie qu'il produit.

Nouveau rapporteur d'erreurs

Un nouveau rapporteur d'erreur a été introduit, qui vise à produire des messages d'erreur plus accessibles sur la ligne de commande. Il est activé par défaut, mais si vous passez `--old-reporter`, vous revenez à l'ancien rapporteur d'erreurs, qui est déprécié.

Options de hachage des métadonnées

Le compilateur ajoute maintenant le hash `IPFS` du fichier de métadonnées à la fin du bytecode par défaut. (pour plus de détails, voir la documentation sur *contract metadata*). Avant la version 0.6.0, le compilateur ajoutait la balise `Swarm` hash par défaut, et afin de toujours supporter ce comportement, la nouvelle option de ligne de commande `--metadata-hash` a été introduite. Elle permet de sélectionner le hachage à produire et à ajouter ajouté, en passant soit `ipfs` soit `swarm` comme valeur à l'option de ligne de commande `--metadata-hash`. Passer la valeur `none` supprime complètement le hachage.

Ces changements peuvent également être utilisés via l'interface *Standard JSON Interface* et affecter les métadonnées JSON générées par le compilateur.

La façon recommandée de lire les métadonnées est de lire les deux derniers octets pour déterminer la longueur de l'encodage CBOR et d'effectuer un décodage correct sur ce bloc de données, comme expliqué dans la section *metadata*.

Optimiseur de Yul

Avec l'optimiseur de bytecode hérité, l'optimiseur *Yul* est maintenant activé par défaut lorsque vous appelez le compilateur avec `-optimize`. avec `--optimize`. Il peut être désactivé en appelant le compilateur avec `-no-optimize-yul`. Ceci affecte principalement le code qui utilise ABI coder v2.

Modifications de l'API C

Le code client qui utilise l'API C de `libsolc` a maintenant le contrôle de la mémoire utilisée par le compilateur. Pour rendre ce changement cohérent, `solidity_free` a été renommé en `solidity_reset`, les fonctions `solidity_alloc` et `solidity_free` ont été modifiées. `solidity_free` ont été ajoutées et `solidity_compile` retourne maintenant une chaîne de caractères qui doit être explicitement libérée par la fonction `solidity_free()`.

3.24.6 Comment mettre à jour votre code

Cette section donne des instructions détaillées sur la façon de mettre à jour le code antérieur pour chaque changement de rupture.

- Changez `address(f)` en `f.address` pour que `f` soit de type fonction externe.
- Remplacer fonction `()` externe [payable] `{ ... }` par soit `receive()` externe [payable] `{ .. }`, `fallback()` externe [payable] `{ ... }` ou les deux. } ou les deux. Préférez l'utilisation d'une fonction `receive` uniquement, lorsque cela est possible.
- Remplacez `uint length = array.push(value)` par `array.push(value);`. La nouvelle longueur peut être accessible via `array.length`.
- Changez `array.length++` en `array.push()` pour augmenter, et utilisez `pop()` pour diminuer la longueur d'un tableau de stockage.
- Pour chaque paramètre de retour nommé dans la documentation `@dev` d'une fonction, définissez une entrée `@return` contenant le nom du paramètre. qui contient le nom du paramètre comme premier mot. Par exemple, si vous avez une fonction « `f()` » définie comme suit comme « fonction `f()` public returns (uint value) » et une annotation `@dev`, documentez ses paramètres de retour comme suit de retour comme suit : `@return value` La valeur de retour. Vous pouvez mélanger des paramètres de retour nommés et non nommés documentation tant que les annotations sont dans l'ordre où elles apparaissent dans le type de retour du tuple.
- Choisissez des identifiants uniques pour les déclarations de variables dans l'assemblage en ligne qui n'entrent pas en conflit avec les déclarations en dehors de l'assemblage en ligne. avec des déclarations en dehors du bloc d'assemblage en ligne.
- Ajoutez « `virtual` » à chaque fonction non interface que vous avez l'intention de remplacer. Ajoutez `virtual` à toutes les fonctions sans implémentation en dehors des interfaces. à toutes les fonctions sans implémentation en dehors des interfaces. Pour l'héritage simple, ajoutez `override` à chaque fonction de remplacement. Pour l'héritage multiple, ajoutez `override(A, B, ..)`, où vous listez entre parenthèses tous les contrats qui définissent la fonction surchargée. Lorsque plusieurs bases définissent la même fonction, le contrat qui hérite doit remplacer toutes les fonctions conflictuelles.

3.25 Solidity v0.7.0 Changements de rupture

Cette section met en évidence les principaux changements de rupture introduits dans Solidity version 0.7.0, ainsi que le raisonnement derrière ces changements et la façon de mettre à jour code affecté. Pour la liste complète, consultez le changelog de la version <https://github.com/ethereum/solidity/releases/tag/v0.7.0>.

3.25.1 Changements silencieux de la sémantique

- L'exponentiation et les décalages de littéraux par des non-littéraux (par exemple, `1 << x` ou `2 ** x`) utiliseront toujours soit le type `uint256` (pour les littéraux non négatifs), soit le type `int256` (pour les littéraux négatifs) pour effectuer l'opération. Auparavant, l'opération était effectuée dans le type de la quantité de décalage / l'exposant, ce qui peut être trompeur.

3.25.2 Modifications de la syntaxe

- Dans les appels de fonctions externes et de création de contrats, l'éther et le gaz sont maintenant spécifiés en utilisant une nouvelle syntaxe : `x.f{gaz : 10000, valeur : 2 ether}(arg1, arg2)`. L'ancienne syntaxe `- x.f.gas(10000).value(2 ether)(arg1, arg2)` – provoquera une erreur.
- La variable globale `now` est obsolète, `block.timestamp` devrait être utilisée à la place. L'identifiant unique `now` est trop générique pour une variable globale et pourrait donner l'impression qu'elle change pendant le traitement de la transaction, alors que `block.timestamp` reflète correctement le fait qu'il s'agit d'une propriété du bloc.
- Les commentaires NatSpec sur les variables ne sont autorisés que pour les variables d'état publiques et non pour les variables locales ou internes.
- Le jeton `wei` est maintenant un mot-clé (utilisé pour spécifier, par exemple, `2 wei` comme un nombre) et ne peut pas être utilisé comme un identifiant.
- Les chaînes de caractères ne peuvent plus contenir que des caractères ASCII imprimables, ce qui inclut une variété de séquences d'échappement, telles que les hexadécimales. séquences d'échappement, telles que les échappements hexadécimaux (`\xff`) et unicode (`\u20ac`).
- Les chaînes littérales Unicode sont désormais prises en charge pour accueillir les séquences UTF-8 valides. Ils sont identifiés avec le préfixe `unicode` : `unicode "Hello "`.
- Mutabilité d'état : La mutabilité d'état des fonctions peut maintenant être restreinte pendant l'héritage : Les fonctions avec une mutabilité d'état par défaut peuvent être remplacées par des fonctions `pure''` et `view''`. tandis que les fonctions `view''` peuvent être remplacées par des fonctions `pure`. En même temps, les variables d'état publiques sont considérées comme `view` et même `pure` si elles sont constantes. si elles sont des constantes.

Assemblage en ligne

- Interdire `.` dans les noms de fonctions et de variables définies par l'utilisateur dans l'assemblage en ligne. C'est toujours valable si vous utilisez Solidity en mode Yul-only.
- L'emplacement et le décalage de la variable pointeur de stockage `x` sont accessibles via `x.slot` et `x.offset`. et `x.offset` au lieu de `x_slot` et `x_offset`.

3.25.3 Suppression des fonctionnalités inutilisées ou dangereuses

Mappages en dehors du stockage

- Si une structure ou un tableau contient un mappage, il ne peut être utilisé que dans le stockage. Auparavant, les membres du mappage étaient ignorés en mémoire, ce qui est déroutant et source d'erreurs. ce qui est déroutant et source d'erreurs.
- Les affectations aux structures ou tableaux dans le stockage ne fonctionnent pas s'ils contiennent des mappings. mappings. Auparavant, les mappings étaient ignorés silencieusement pendant l'opération de copie, ce qui ce qui est trompeur et source d'erreurs.

Fonctions et événements

- La visibilité (`public / internal` ``) n'est plus nécessaire pour les constructeurs : Pour empêcher un contrat d'être créé, il peut être marqué ```abstract`. Cela rend le concept de visibilité pour les constructeurs obsolète.
- Contrôleur de type : Désaccorder `virtual` pour les fonctions de bibliothèque : Puisque les bibliothèques ne peuvent pas être héritées, les fonctions de bibliothèque ne devraient pas être virtuelles.
- Plusieurs événements avec le même nom et les mêmes types de paramètres dans la même hiérarchie d'héritage sont interdits. même hiérarchie d'héritage sont interdits.
- utiliser `A pour B` n'affecte que le contrat dans lequel il est mentionné. Auparavant, l'effet était hérité. Maintenant, vous devez répéter l'instruction « `using` » dans tous les contrats dérivés qui font usage de cette instruction. dans tous les contrats dérivés qui utilisent cette fonctionnalité.

Expressions

- Les décalages par des types signés ne sont pas autorisés. Auparavant, les décalages par des montants négatifs étaient autorisés, mais ils étaient annulés à l'exécution.
- Les dénominations `finney` et ```szabo``` sont supprimées. Elles sont rarement utilisées et ne rendent pas le montant réel facilement visible. A la place, des valeurs explicites valeurs explicites comme « `1e20` » ou le très commun « `gwei` » peuvent être utilisées.

Déclarations

- Le mot-clé `var` ne peut plus être utilisé. Auparavant, ce mot-clé était analysé mais donnait lieu à une erreur de type et à une suggestion sur le type à utiliser. une suggestion sur le type à utiliser. Maintenant, il résulte en une erreur d'analyse.

3.25.4 Changements d'interface

- JSON AST : Marquer les littéraux de chaînes hexagonales avec `kind : "hexString"`.
- JSON AST : Les membres avec la valeur `null` sont supprimés de la sortie JSON.
- NatSpec : Les constructeurs et les fonctions ont une sortie `userdoc` cohérente.

3.25.5 Comment mettre à jour votre code

Cette section donne des instructions détaillées sur la façon de mettre à jour le code antérieur pour chaque changement de rupture.

- Changez `x.f.value(...)` en `x.f{value : ...}`. De même, `(new C).value(...)` en `nouveau C{valeur : ...}` et `x.f.gas(...).valeur(...)` en `x.f{gas : ..., valeur : ...}`.
- Remplacez `now` par `block.timestamp`.
- Changez les types de l'opérande droit dans les opérateurs de décalage en types non signés. Par exemple, remplacez `x >> (256 - y)` par `x >> uint(256 - y)`.
- Répétez les déclarations utilisant `A pour B` dans tous les contrats dérivés si nécessaire.
- Supprimez le mot-clé « `public` » de chaque constructeur.
- Supprimer le mot-clé « `interne` » de chaque constructeur et ajouter « `abstrait` » au contrat (s'il n'est pas déjà présent).
- Changez les suffixes `_slot` et `_offset` `` dans l'assemblage en ligne en ```.slot` et ```.offset```, respectivement.

3.26 Solidity v0.8.0 Changements de rupture

Cette section met en évidence les principaux changements de rupture introduits dans Solidity version 0.8.0. Pour la liste complète, consultez le changelog de la version 0.8.0.

3.26.1 Changements silencieux de la sémantique

Cette section répertorie les modifications où le code existant change de comportement sans que le compilateur vous en informe.

- Les opérations arithmétiques s'inversent en cas de sous-dépassement et de dépassement. Vous pouvez utiliser `unchecked { ... }` pour utiliser le comportement d'enveloppement précédent. Les vérifications pour le débordement sont très communes, donc nous les avons faites par défaut pour augmenter la lisibilité du code, même si cela entraîne une légère augmentation du coût de l'essence.
- ABI coder v2 est activé par défaut. Vous pouvez choisir d'utiliser l'ancien comportement en utilisant `pragma abicoder v1;`. Le pragma `pragma experimental ABIEncoderV2;` est toujours valide, mais il est déprécié et n'a aucun effet. Si vous voulez être explicite, veuillez utiliser le pragma `pragma abicoder v2;` à la place. Notez que ABI coder v2 supporte plus de types que v1 et effectue plus de contrôles d'intégrité sur les entrées. ABI coder v2 rend certains appels de fonctions plus coûteux et il peut aussi faire des appels de contrats réversibles qui n'étaient pas réversibles avec ABI coder v1 lorsqu'ils contiennent des données qui ne sont pas conformes aux types de paramètres. types de paramètres.
- L'exponentiation est associative à droite, c'est-à-dire que l'expression `a**b**c` est interprétée comme `a**(b**c)`. Avant la version 0.8.0, elle était interprétée comme `(a**b)**c`. C'est la façon courante d'analyser l'opérateur d'exponentiation.
- Les assertions qui échouent et d'autres vérifications internes comme la division par zéro ou le dépassement arithmétique n'utilisent pas l'opcode invalide mais plutôt l'opcode de retour. Plus précisément, ils utiliseront des données d'erreur égales à un appel de fonction à `Panic(uint256)` avec un code d'erreur spécifique aux circonstances. aux circonstances. Cela permettra d'économiser du gaz sur les erreurs tout en permettant aux outils d'analyse statique de distinguer ces situations d'un retour sur invalidité. distinguer ces situations d'un retour en arrière sur une entrée invalide, comme un `require` échoué.
- Si l'on accède à un tableau d'octets en stockage dont la longueur est mal codée, une panique est provoquée. Un contrat ne peut pas se retrouver dans cette situation à moins que l'assemblage en ligne soit utilisé pour modifier la représentation brute des tableaux d'octets de stockage.
- Si des constantes sont utilisées dans les expressions de longueur de tableau, les versions précédentes de Solidity utilisaient une précision arbitraire dans toutes les branches de l'arbre d'évaluation. dans toutes les branches de l'arbre d'évaluation. Maintenant, si des variables constantes sont utilisées comme expressions intermédiaires, leurs valeurs seront correctement arrondies de la même manière que lorsqu'elles sont utilisées dans des expressions d'exécution.
- Le type `byte` a été supprimé. C'était un alias de `bytes1`.

3.26.2 Nouvelles restrictions

Cette section énumère les changements qui pourraient empêcher les contrats existants de se compiler.

- Il existe de nouvelles restrictions liées aux conversions explicites de littéraux. Le comportement précédent dans les cas suivants était probablement ambigu :
 1. Les conversions explicites de littéraux négatifs et de littéraux plus grands que `type(uint160).max` en adresse sont interdites.
 2. Les conversions explicites entre des littéraux et un type de nombre entier T ne sont autorisées que si le littéral se situe entre `type(T).min` et `type(T).max`. En particulier, remplacez les utilisations de `uint(-1)` par `type(uint)`. par `type(uint).max`.

3. Les conversions explicites entre les littéraux et les énumérations ne sont autorisées que si le littéral peut représenter une valeur de l'énumération.
 4. Les conversions explicites entre les littéraux et le type adresse (par exemple `address(literal)`) ont le type `address`. type adresse au lieu de adresse payable. On peut obtenir un type d'adresse payable en utilisant une conversion explicite, c'est-à-dire `payable(literal)`.
- *Les littéraux d'adresse* ont le type `address` au lieu de `address payable`. Ils peuvent être convertis en `adresse payable` en utilisant une conversion explicite, par exemple `payable(0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF)`.
 - Il y a de nouvelles restrictions sur les conversions de type explicites. La conversion n'est autorisée que lorsqu'il y a lorsqu'il y a au plus un changement de signe, de largeur ou de catégorie de type (`int`, `address`, `bytesNN`, etc.). Pour effectuer plusieurs changements, il faut utiliser plusieurs conversions. Utilisons la notation `T(S)` pour désigner la conversion explicite `T(x)`, où, `T` et `S` sont des types, et `x` est une variable arbitraire de type `S`. Un exemple d'une telle exemple d'une telle conversion non autorisée serait `uint16(int8)` puisqu'elle change à la fois la largeur (8 bits à 16 bits) et le signe (d'entier signé à entier non signé). Pour effectuer la conversion, il faut passer par un type intermédiaire. passer par un type intermédiaire. Dans l'exemple précédent, ce serait `uint16(uint8(int8))` ou `uint16(int16(int8))`. Notez que les deux façons de convertir produiront des résultats différents, par ex, pour `-1`. Voici quelques exemples de conversions qui ne sont pas autorisées par cette règle.
 - `address(uint)` et `uint(address)` : conversion à la fois de la catégorie de type et de la largeur. Remplacez-les par `address(uint160(uint))` et `uint(uint160(address))` respectivement.
 - `payable(uint160)`, `payable(bytes20)` et `payable(integer-literal)` : conversion de la catégorie de type et de la la catégorie de type et la mutabilité d'état. Remplacez-les par `payable(address(uint160))`, `payable(address(bytes20))` et `payable(address(integer-literal))` respectivement. Notez que `payable(0)` est valide et constitue une exception à la règle.
 - `int80(bytes10)` et `bytes10(int80)` : conversion de la catégorie de type et du signe. Remplacez-les par `int80(uint80(bytes10))` et `bytes10(uint80(int80))` respectivement.
 - `Contract(uint)` : convertit à la fois la catégorie de type et le signe. Remplacez-la par `Contract(adresse(uint160(uint)))`.
 Ces conversions ont été interdites pour éviter toute ambiguïté. Par exemple, dans l'expression `uint16 x = uint16(int8(-1))`, la valeur de `x` dépendrait de la conversion du signe ou de la largeur appliquée en premier lieu. a été appliquée en premier.
 - Les options d'appel de fonction ne peuvent être données qu'une seule fois, c'est-à-dire que `c.f{gas : 10000}{value : 1}()` est invalide et doit être changé en `c.f{gas : 10000, value : 1}()`.
 - Les fonctions globales `log0`, `log1`, `log2`, `log3` et `log4` ont été supprimées. Ce sont des fonctions de bas niveau qui étaient largement inutilisées. Leur comportement est accessible depuis l'assemblage en ligne.
 - Les définitions de `enum` ne peuvent pas contenir plus de 256 membres. Cela permet de supposer que le type sous-jacent dans l'ABI est toujours `uint8`.
 - Les déclarations portant les noms « `this` », « `super` » et « `_` » ne sont pas autorisées, à l'exception des fonctions et événements publics. fonctions et événements publics. Cette exception a pour but de permettre la déclaration d'interfaces de contrats implémentées dans des langages autres que Solidity qui autorisent de tels noms de fonctions.
 - Suppression de la prise en charge des séquences d'échappement `b`, `f` et `v''` dans le code. Elles peuvent toujours être insérées par le biais d'échappements hexadécimaux, par exemple, respectivement, `" ``X08`, `» X0c` et `» X0b`.
 - Les variables globales `tx.origin` et `msg.sender` ont le type `address` au lieu de `adresse payable`. On peut les convertir en `adresse payable` en utilisant une conversion explicite, c'est-à-dire `payable(tx.origin)` ou `payable(msg.sender)`. Ce changement a été fait car le compilateur ne peut pas déterminer si ces adresses sont payables ou non. sont payables ou non, donc il faut maintenant une conversion explicite pour rendre cette exigence visible.
 - La conversion explicite en type `adresse` retourne toujours un type `adresse` non payable. Dans En particulier, les conversions explicites suivantes ont le type `adresse` au lieu de ```adresse payable` » :
 - `adresse(u)` où `u` est une variable de type `uint160`. On peut convertir `u` dans le type `adresse payable`

- en utilisant deux conversions explicites, c'est-à-dire, `payable(adresse(u))`.
- `adresse(b)` où `b` est une variable de type `bytes20`. On peut convertir `b` dans le type `adresse payable` en utilisant deux conversions explicites, c'est-à-dire, `payable(adresse(b))`.
- `adresse(c)` où `c` est un contrat. Auparavant, le type de retour de cette conversion dépendait de la possibilité pour le contrat de recevoir de l'Ether (soit en ayant une fonction de réception ou une fonction de repli payable). La conversion `payable(c)` a le type `adresse payable` et n'est autorisée que si le contrat "`c`" peut recevoir de l'Ether. En général, on peut convertir `c` en type `adresse payable` en utilisant la conversion explicite suivante : `payable(adresse(c))`. Notez que `address(this)` tombe sous la même catégorie que `address(c)` et les mêmes règles s'appliquent pour elle.
- La construction de « chainid » dans l'assemblage en ligne est maintenant considérée comme une « vue » au lieu d'une « pure ».
- La négation unaire ne peut plus être utilisée sur les entiers non signés, seulement sur les entiers signés.

3.26.3 Changements d'interface

- La sortie de `--combined-json` a changé : Les champs JSON `abi`, `devdoc`, `userdoc` et `storage-layout` sont maintenant des sous-objets. Avant la version 0.8.0, ils étaient sérialisés sous forme de chaînes de caractères.
- L'« ancien AST » a été supprimé (`--ast-json` sur l'interface de la ligne de commande et `legacyAST` pour le JSON standard). Utilisez l'« AST compact » (`--ast-compact--json` resp. `AST`) en remplacement.
- L'ancien rapporteur d'erreurs (`--old-reporter`) a été supprimé.

3.26.4 Comment mettre à jour votre code

- Si vous comptez sur l'arithmétique enveloppante, entourez chaque opération de `unchecked { ... }`.
- Optionnel : Si vous utilisez `SafeMath` ou une bibliothèque similaire, changez `x.add(y)` en `x + y`, `x.mul(y)` en `x * y` etc.
- Ajoutez `pragma abicoder v1` ; si vous voulez rester avec l'ancien codeur ABI.
- Supprimez éventuellement `pragma experimental ABIEncoderV2` ou `pragma abicoder v2` car ils sont redondants.
- Changez `byte` en `bytes1`.
- Ajouter des conversions de types explicites intermédiaires si nécessaire.
- Combinez `c.f{gas : 10000}{value : 1}()` en `c.f{gas : 10000, value : 1}()`.
- Remplacez `msg.sender.transfer(x)` par `payable(msg.sender).transfer(x)` ou utilisez une variable stockée de type `adresse payable`.
- Remplacez `x*y**z` par `(x*y)**z`.
- Utilisez l'assemblage en ligne en remplacement de `log0, ..., log4`.
- Négation des entiers non signés en les soustrayant de la valeur maximale du type et en ajoutant 1 (par exemple, `type(uint256).max - x + 1`, tout en s'assurant que `x` n'est pas zéro)

3.27 Format NatSpec

Les contrats Solidity peuvent utiliser une forme spéciale de commentaires pour fournir une documentation riche pour les fonctions, les variables de retour et autres. Cette forme spéciale est nommé Ethereum Natural Language Specification Format (NatSpec).

Note : NatSpec a été inspiré par [Doxygen](#). Bien qu'il utilise des commentaires et des balises de style Doxygen, il n'y a aucune intention de garder une compatibilité stricte avec Doxygen. Veuillez examiner attentivement les balises supportées listées ci-dessous.

Cette documentation est segmentée en messages destinés aux développeurs et en messages destinés aux l'utilisateur finaux. Ces messages peuvent être présentés à l'utilisateur final (l'humain) au moment où il interagit avec le contrat (c'est-à-dire lorsqu'il signe une transaction).

Il est recommandé que les contrats Solidity soient entièrement annotés à l'aide de NatSpec pour toutes les interfaces publiques (tout ce qui se trouve dans l'ABI).

NatSpec inclut le formatage des commentaires que l'auteur du contrat intelligent utilisera et qui sont compris par le compilateur Solidity. Ils sont également détaillés ci-dessous sortie du compilateur Solidity, qui extrait ces commentaires dans un format lisible par la machine.

NatSpec peut également inclure des annotations utilisées par des outils tiers. Celles-ci sont très probablement via la balise `@custom:<name>`, et un bon cas d'utilisation est celui des outils d'analyse et de vérification.

3.27.1 Exemple de documentation

La documentation est insérée au-dessus de chaque contrat, interface, fonction, et event en utilisant le format de notation Doxygen. Une variable d'état public est équivalente à une fonction pour les besoins de NatSpec.

- **Pour Solidity, vous pouvez choisir `///` pour les commentaires d'une ou plusieurs lignes** commentaires, ou `/**` et se terminant par `*/`.
- **Pour Vyper, utilisez `"""` indenté jusqu'au contenu intérieur avec des** commentaires. Voir la documentation de Vyper.

L'exemple suivant montre un contrat et une fonction utilisant toutes les balises disponibles.

Note : Le compilateur Solidity n'interprète les balises que si elles sont externes ou publiques. Vous pouvez utiliser des commentaires similaires pour vos fonctions internes et privées, mais elles ne seront pas interprétées.

Ceci pourrait changer à l'avenir.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.2 < 0.9.0;

/// @title Un simulateur pour les arbres
/// @author Larry A. Gardner
/// @notice Vous ne pouvez utiliser ce contrat que pour la simulation la plus
↳élémentaire.
/// Tous les appels de fonctions sont actuellement implémentés sans effets
↳secondaires.
/// @custom:experimental Il s'agit d'un contrat expérimental.
contract Tree {
    /// @notice Calculer l'âge de l'arbre en années, arrondi à l'unité supérieure,
↳pour les arbres vivants.
    /// @dev L'algorithme d'Alexandr N. Tetearing pourrait améliorer la précision.
    /// @param rings Le nombre de cernes de l'échantillon dendrochronologique.
    /// @return Âge en années, arrondi au chiffre supérieur pour les années
↳partielles
    function age(uint256 rings) external virtual pure returns (uint256) {
        return rings + 1;
    }

    /// @notice Renvoie le nombre de feuilles de l'arbre.
    /// @dev Renvoie uniquement un nombre fixe.
    function leaves() external virtual pure returns(uint256) {
```

(suite sur la page suivante)

```

    return 2;
  }
}

contract Plant {
  function leaves() external virtual pure returns(uint256) {
    return 3;
  }
}

contract KumquatTree is Tree, Plant {
  function age(uint256 rings) external override pure returns (uint256) {
    return rings + 2;
  }

  /// Retourne le nombre de feuilles que possède ce type d'arbre spécifique.
  /// @inheritdoc Arbre
  function leaves() external override(Tree, Plant) pure returns(uint256) {
    return 3;
  }
}

```

3.27.2 Tags

Toutes les balises sont facultatives. Le tableau suivant explique le but de chaque balise NatSpec et où elle peut être utilisée. Dans un cas particulier, si aucune balise n'est utilisée, le compilateur Solidity interprétera un commentaire `///` ou `/**` de la même manière que s'il était balisé avec `@notice`.

Tag		Contexte
@title	Un titre qui doit décrire le contrat/interface	contract, library, interface
@author	Le nom de l'auteur	contract, library, interface
@notice	Expliquer à un utilisateur final ce que cela fait	contract, library, interface, function, public state variable, event
@dev	Expliquez à un développeur tout détail supplémentaire	contract, library, interface, function, state variable, event
@param	Documente un paramètre comme dans Doxygen (doit être suivi du nom du paramètre)	function, event
@return	Documente les variables de retour de la fonction d'un contrat	function, public state variable
@inheritdoc	Copie toutes les étiquettes manquantes de la fonction de base (doit être suivi du nom du contrat).	function, public state variable
@custom: ..	Balise personnalisée, la sémantique est définie par l'application.	everywhere

Si votre fonction renvoie plusieurs valeurs, comme `(int quotient, int remainder)`, alors utilisez plusieurs instructions `return` dans le même format que les instructions `@param`.

Les balises personnalisées commencent par `@custom:` et doivent être suivies d'une ou plusieurs lettres minuscules ou d'un trait d'union. Elles ne peuvent cependant pas commencer par un trait d'union. Elles peuvent être utilisées partout et font partie de la documentation du développeur.

Expressions dynamiques

Le compilateur Solidity fera passer la documentation NatSpec de votre code source Solidity jusqu'à la sortie JSON, comme décrit dans ce guide. Le consommateur de ce JSON, par exemple le logiciel client de l'utilisateur final, peut le présenter directement à l'utilisateur final ou appliquer un prétraitement.

Par exemple, certains logiciels clients effectueront un rendu :

```
///@notice Cette fonction va multiplier `a` par 7
```

to the end-user as :

```
Cette fonction va multiplier 10 par 7
```

Si une fonction est appelée et que la valeur 10 est attribuée à l'entrée a.

La spécification de ces expressions dynamiques n'entre pas dans le cadre de la documentation de Solidity. et vous pouvez en savoir plus à l'adresse suivante [le projet radspec](#).

Notes sur l'héritage

Les fonctions sans NatSpec hériteront automatiquement de la documentation de leur fonction de base. Les exceptions à cette règle sont :

- Lorsque les noms des paramètres sont différents.
- Quand il y a plus d'une fonction de base.
- Quand il y a une balise explicite `@inheritdoc` qui spécifie quel contrat doit être utilisé pour hériter.

3.27.3 Sortie de documentation

Lorsqu'elle est analysée par le compilateur, une documentation telle que celle de l'exemple ci-dessus produira deux fichiers JSON différents. L'un est destiné à être consommé par l'utilisateur final comme un avis lorsqu'une fonction est exécutée et l'autre à être utilisé par le développeur.

Si le contrat ci-dessus est enregistré sous le nom de `ex1.sol`, alors vous pouvez générer la documentation en utilisant :

```
solc --userdoc --devdoc ex1.sol
```

Et la sortie est ci-dessous.

Note : À partir de la version 0.6.11 de Solidity, la sortie NatSpec contient également un champ `version` et un champ `kind`. Actuellement, la `version` est fixée à 1 et le `kind` doit être l'un de `user` ou `dev`. Dans le futur, il est possible que de nouvelles versions soient introduites et que les anciennes soient supprimées.

Documentation pour les utilisateurs

La documentation ci-dessus produira la documentation utilisateur suivante Fichier JSON en sortie :

```
{
  "version" : 1,
  "kind" : "user",
  "methods" :
  {
    "age(uint256)" :
    {
      "notice" : "Calculez l'âge de l'arbre en années, arrondi au chiffre supérieur,
↳pour les arbres vivants."
    }
  },
  "notice" : "Vous pouvez utiliser ce contrat uniquement pour la simulation la plus
↳basique"
}
```

Notez que la clé permettant de trouver les méthodes est la signature canonique de la fonction telle que définie dans le *Contrat ABI* et non le simple nom de la fonction.

Documentation pour les développeurs

Outre le fichier de documentation utilisateur, un fichier JSON de documentation pour les développeurs doit également être produit et doit ressembler à ceci :

```
{
  "version" : 1,
  "kind" : "dev",
  "author" : "Larry A. Gardner",
  "details" : "Tous les appels de fonction sont actuellement mis en œuvre sans effets
↳secondaires",
  "custom:experimental" : "Il s'agit d'un contrat expérimental.",
  "methods" :
  {
    "age(uint256)" :
    {
      "details" : "L'algorithme d'Alexandr N. Tetearing pourrait augmenter la précision",
      "params" :
      {
        "rings" : "Le nombre de cernes de l'échantillon dendrochronologique"
      },
      "return" : "âge en années, arrondi au chiffre supérieur pour les années incomplètes
↳"
    }
  },
  "title" : "Un simulateur pour les arbres"
}
```

3.28 Considérations de sécurité

Alors qu'il est généralement assez facile de construire un logiciel qui fonctionne comme prévu, il est beaucoup plus difficile de vérifier que personne ne peut l'utiliser d'une manière **non** prévue.

Dans Solidity, cela est encore plus important car vous pouvez utiliser des contrats intelligents pour gérer des jetons ou, éventuellement, des choses encore plus précieuses. De plus, chaque exécution d'un contrat intelligent se fait en public et, en plus de cela, le code source est souvent disponible.

Bien sûr, il faut toujours tenir compte de l'importance de l'enjeu : Vous pouvez comparer un contrat intelligent avec un service web qui est ouvert au public (et donc, également aux acteurs malveillants) et peut-être même open source. Si vous ne stockez que votre liste de courses sur ce service web, vous n'aurez peut-être pas à prendre trop de précautions, mais si vous gérez votre compte bancaire en utilisant ce service web, vous devriez être plus prudent.

Cette section énumère quelques pièges et recommandations générales en matière de sécurité mais ne peut, bien entendu, jamais être complète. Gardez également à l'esprit que même si le code de votre smart contrat intelligent est exempt de bogues, le compilateur ou la plateforme elle-même peuvent en bug. Une liste de certains bogues du compilateur liés à la sécurité et connus du public peut être trouvée dans la *liste des bugs connus*, qui est également lisible par machine. Notez qu'il existe un programme de prime de bogue qui couvre le générateur de code du compilateur Solidity.

Comme toujours, avec la documentation open source, merci de nous aider à étendre cette section (surtout, quelques exemples ne feraient pas de mal) !

NOTE : En plus de la liste ci-dessous, vous pouvez trouver plus de recommandations de sécurité et de meilleures pratiques dans la liste de connaissances de [Guy Lando](#) et le [repo GitHub de Consensys](#).

3.28.1 Pièges

Information privée et aléatoire

Tout ce que vous utilisez dans un contrat intelligent est visible publiquement, même les variables locales et les variables d'état marquées `private`.

L'utilisation de nombres aléatoires dans les contrats intelligents est assez délicat si vous ne voulez pas que les mineurs soient capables de tricher.

Ré-entrée en scène

Toute interaction d'un contrat (A) avec un autre contrat (B) et tout transfert d'Ether transmet le contrôle à ce contrat (B). Il est donc possible pour B de rappeler A avant que cette interaction ne soit terminée. Pour donner un exemple, le code suivant contient un bug (il ne s'agit que d'un extrait et non d'un contrat complet) :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// CE CONTRAT CONTIENT UN BUG - NE PAS UTILISER
contract Fund {
    /// @dev Cartographie des parts d'éther du contrat.
    mapping(address => uint) shares;
    /// Retirez votre part.
    function withdraw() public {
        if (payable(msg.sender).send(shares[msg.sender]))
            shares[msg.sender] = 0;
    }
}
```

(suite sur la page suivante)

```
}
}
```

Le problème n'est pas trop grave ici en raison du gaz limité dans le cadre de `send`, mais il expose quand même une faiblesse : Le transfert d'éther peut toujours inclure l'exécution de code, donc le destinataire pourrait être un contrat qui appelle dans `withdraw`. Cela lui permettrait d'obtenir de multiples remboursements et de récupérer tout l'Ether du contrat. En particulier, le contrat suivant permettra à un attaquant de rembourser plusieurs fois car il utilise `call` qui renvoie tout le gaz restant par défaut :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

// CE CONTRAT CONTIENT UN BUG - NE PAS UTILISER
contract Fund {
    /// @dev Cartographie des parts d'éther du contrat.
    mapping(address => uint) shares;
    /// Retirez votre part.
    function withdraw() public {
        (bool success,) = msg.sender.call{value: shares[msg.sender]}("");
        if (success)
            shares[msg.sender] = 0;
    }
}
```

Pour éviter la ré-entrance, vous pouvez utiliser le modèle Checks-Effects-Interactions comme comme indiqué ci-dessous :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Fund {
    /// @dev Cartographie des parts d'éther du contrat.
    mapping(address => uint) shares;
    /// Retirez votre part.
    function withdraw() public {
        uint share = shares[msg.sender];
        shares[msg.sender] = 0;
        payable(msg.sender).transfer(share);
    }
}
```

Notez que la ré-entrance n'est pas seulement un effet du transfert d'Ether mais de tout appel de fonction sur un autre contrat. De plus, vous devez également prendre en compte les situations de multi-contrats. Un contrat appelé pourrait modifier l'état d'un autre contrat dont vous dépendez.

Limite et boucles de gaz

Les boucles qui n'ont pas un nombre fixe d'itérations, par exemple les boucles qui dépendent de valeurs de stockage, doivent être utilisées avec précaution : En raison de la limite de gaz de bloc, les transactions ne peuvent consommer qu'une certaine quantité de gaz. Que ce soit explicitement ou simplement en raison du fonctionnement normal, le nombre d'itérations d'une boucle peut dépasser la limite de gaz en bloc, ce qui peut entraîner que le contrat complet soit bloqué à un certain point. Cela peut ne pas s'appliquer aux fonctions `view` qui sont uniquement exécutées pour lire les données de la blockchain. Cependant, de telles fonctions peuvent être appelées par d'autres contrats dans le cadre d'opérations sur la blockchain et les bloquer. Veuillez être explicite sur ces cas dans la documentation de vos contrats.

Envoi et réception d'Ether

- Ni les contrats ni les « comptes externes » ne sont actuellement capables d'empêcher que quelqu'un leur envoie de l'Ether. Les contrats peuvent réagir et rejeter un transfert régulier, mais il existe des moyens de déplacer de l'Ether sans créer un appel de message. Une façon est de simplement « miner vers » l'adresse du contrat et la seconde façon est d'utiliser `selfdestruct(x)`.
- Si un contrat reçoit de l'Ether (sans qu'une fonction soit appelée), soit la *receive Ether*, soit la fonction *fallback* est exécutée. S'il n'a ni fonction de réception ni fonction de repli, l'éther sera rejeté (en lançant une exception). Pendant l'exécution d'une de ces fonctions, le contrat ne peut compter que sur le « supplément de gaz » qui lui est transmis (2300 gaz) dont il dispose à ce moment-là. Cette allocation n'est pas suffisante pour modifier le stockage (ne considérez pas cela comme acquis, l'allocation pourrait changer avec les futures hard forks). Pour être sûr que votre contrat peut recevoir de l'Ether de cette manière, vérifiez les exigences en matière de gaz des fonctions de réception et de repli (par exemple dans la section « details » de Remix).
- Il existe un moyen de transmettre plus de gaz au contrat récepteur en utilisant `addr.call{value : x}("")`. C'est essentiellement la même chose que `addr.transfer(x)`, sauf qu'elle transmet tout le gaz restant et donne la possibilité au destinataire d'effectuer des actions plus coûteuses (et il renvoie un code d'échec au lieu de propager automatiquement l'erreur). Cela peut inclure le rappel dans le contrat d'envoi ou d'autres changements d'état auxquels vous n'auriez peut-être pas pensé. Cela permet donc une grande flexibilité pour les utilisateurs honnêtes mais aussi pour les acteurs malveillants.
- Utilisez les unités les plus précises possibles pour représenter le montant du wei, car vous perdez tout ce qui est arrondi en raison d'un manque de précision.
- Si vous voulez envoyer des Ether en utilisant `address.transfer`, il y a certains détails à connaître :
 1. Si le destinataire est un contrat, il provoque l'exécution de sa fonction de réception ou de repli qui peut, à son tour, rappeler le contrat émetteur.
 2. L'envoi d'Ether peut échouer si la profondeur d'appel dépasse 1024. Puisque l'appelant a le contrôle total de la profondeur d'appel, il peut faire échouer le transfert ; tenez compte de cette possibilité ou utilisez `send` et assurez-vous de toujours vérifier sa valeur de retour. Mieux encore, écrivez votre contrat en utilisant un modèle où le destinataire peut retirer de l'Ether à la place.
 3. L'envoi d'Ether peut également échouer parce que l'exécution du contrat du destinataire nécessite plus que la quantité d'essence allouée (explicitement en utilisant *require*, *assert*, *revert* ou parce que l'opération est trop coûteuse) - il « tombe en panne sèche » (OOG). Si vous utilisez `transfer` ou `send` avec une vérification de la valeur de retour, cela pourrait être un moyen pour le destinataire de bloquer la progression du contrat d'envoi. Là encore, la meilleure pratique consiste à :ref:`utiliser un motif « withdraw » plutôt qu'un motif « send » <withdrawal_pattern>`.

Profondeur de la pile d'appel

Les appels de fonctions externes peuvent échouer à tout moment parce qu'ils dépassent la limite de taille de la pile d'appels de 1024. Dans de telles situations, Solidity lève une exception. Les acteurs malveillants pourraient être en mesure de forcer la pile d'appels à une valeur élevée avant d'interagir avec votre contrat. Notez que, depuis que [Tangerine Whistle](#) hardfork, la règle 63/64 rend l'attaque de la profondeur de la pile d'appels impraticable. Notez également que la pile d'appel et la pile d'expression ne sont pas liées, même si toutes deux ont une limite de taille de 1024 emplacements de pile.

Notez que `.send()` ne lève **pas** d'exception si la pile d'appels est épuisée, mais renvoie plutôt `false` dans ce cas. Les fonctions de bas niveau `.call()`, `.delegatecall()` et `.staticcall()` se comportent de la même manière.

Procurations autorisées

Si votre contrat peut agir comme un proxy, c'est-à-dire s'il peut appeler des contrats arbitraires avec des données fournies par l'utilisateur, alors l'utilisateur peut essentiellement assumer l'identité du contrat proxy. Même si vous avez mis en place d'autres mesures de protection, il est préférable de construire votre système de contrat de telle sorte que le proxy n'a aucune autorisation (même pas pour lui-même). Si nécessaire, vous pouvez y parvenir en utilisant un deuxième proxy :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract ProxyWithMoreFunctionality {
    PermissionlessProxy proxy;

    function callOther(address _addr, bytes memory _payload) public
        returns (bool, bytes memory) {
        return proxy.callOther(_addr, _payload);
    }
    // Autres fonctions et autres fonctionnalités
}

// Il s'agit du contrat complet, il n'a pas d'autre fonctionnalités et
// ne nécessite aucun privilège pour fonctionner.
contract PermissionlessProxy {
    function callOther(address _addr, bytes memory _payload) public
        returns (bool, bytes memory) {
        return _addr.call(_payload);
    }
}
```

tx.origin

N'utilisez jamais `tx.origin` pour l'autorisation. Disons que vous avez un contrat de portefeuille comme celui-ci :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// CE CONTRAT CONTIENT UN BUG - NE PAS UTILISER
contract TxUserWallet {
    address owner;

    constructor() {
```

(suite sur la page suivante)

(suite de la page précédente)

```

    owner = msg.sender;
}

function transferTo(address payable dest, uint amount) public {
    // LE BOGUE EST ICI, vous devez utiliser msg.sender au lieu de tx.origin
    require(tx.origin == owner);
    dest.transfer(amount);
}
}

```

Maintenant, quelqu'un vous incite à envoyer de l'Ether à l'adresse de ce portefeuille d'attaque :

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

interface TxUserWallet {
    function transferTo(address payable dest, uint amount) external;
}

contract TxAttackWallet {
    address payable owner;

    constructor() {
        owner = payable(msg.sender);
    }

    receive() external payable {
        TxUserWallet(msg.sender).transferTo(owner, msg.sender.balance);
    }
}

```

Si votre porte-monnaie avait vérifié l'autorisation de `msg.sender`, il aurait obtenu l'adresse du porte-monnaie attaqué, au lieu de l'adresse du propriétaire. Mais en vérifiant `tx.origin`, il obtient l'adresse originale qui a déclenché la transaction, qui est toujours l'adresse du propriétaire. Le porte-monnaie attaqué draine instantanément tous vos fonds.

Complément à deux / Débordements / Débordements

Comme dans de nombreux langages de programmation, les types entiers de Solidity ne sont pas réellement des entiers. Ils ressemblent à des entiers lorsque les valeurs sont petites, mais ne peuvent pas représenter des nombres arbitrairement grands.

Le code suivant provoque un dépassement de capacité parce que le résultat de l'addition est trop grand pour être stocké dans le type `uint8` :

```

uint8 x = 255;
uint8 y = 1;
return x + y;

```

Solidity a deux modes dans lesquels il traite ces débordements : Le mode vérifié et le mode non vérifié ou le mode « enveloppant ».

Le mode vérifié par défaut détecte les dépassements et provoque l'échec de l'assertion. Vous pouvez désactiver cette vérification en utilisant `unchecked { ... }`, ce qui aura pour effet d'ignorer le débordement en silence. Le code ci-dessus renverrait `0` s'il était enveloppé dans `unchecked { ... }`.

Même en mode vérifié, ne pensez pas que vous êtes protégé des bogues de débordement. Dans ce mode, les débordements se retourneront toujours. S'il n'est pas possible d'éviter le débordement, cela peut conduire à ce qu'un contrat intelligent soit bloqué dans un certain état.

En général, il faut lire les limites de la représentation par complément à deux, qui présente même des cas limites plus spéciaux pour les nombres signés.

Essayez d'utiliser `require` pour limiter la taille des entrées à un intervalle raisonnable et utilisez la fonction *SMT checker* pour trouver les débordements potentiels.

Effacement des mappages

Le type Solidity `mapping` (voir *Type Mapping*) est une structure de données de type clé-valeur qui ne garde pas la trace des clés auxquelles qui ont reçu une valeur non nulle. Pour cette raison, le nettoyage d'un mappage sans informations supplémentaires sur les clés écrites n'est pas possible. Si un `mapping` est utilisé comme type de base d'un tableau de stockage dynamique, la suppression ou l'éclatement du tableau n'aura aucun effet sur les éléments du `mapping`. Il en va de même, par exemple, si un `mapping` est utilisé comme type d'un champ d'une structure qui est le type de base d'un tableau de stockage dynamique. Le site `mapping` est également ignoré dans les affectations de structs ou de tableaux contenant un `mapping`.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Map {
    mapping (uint => uint)[] array;

    function allocate(uint _newMaps) public {
        for (uint i = 0; i < _newMaps; i++)
            array.push();
    }

    function writeMap(uint _map, uint _key, uint _value) public {
        array[_map][_key] = _value;
    }

    function readMap(uint _map, uint _key) public view returns (uint) {
        return array[_map][_key];
    }

    function eraseMaps() public {
        delete array;
    }
}
```

Considérons l'exemple ci-dessus et la séquence d'appels suivante : `allocate(10)`, `writeMap(4, 128, 256)`. À ce stade, l'appel à `readMap(4, 128)` renvoie 256. Si on appelle `eraseMaps`, la longueur de la variable d'état `array` est remise à zéro, mais comme ses éléments `mapping` ne peuvent être mis à zéro, leurs informations restent vivantes dans le stockage du contrat. Après avoir supprimé `array`, l'appel à `allocate(5)` nous permet d'accéder à `array[4]` à nouveau, et l'appel à `readMap(4, 128)` renvoie 256 même sans un autre appel à `writeMap`.

Si vos informations de `mapping` doivent être effacées, envisagez d'utiliser une bibliothèque similaire à `iterable mapping`, vous permettant de parcourir les clés et de supprimer leurs valeurs dans le `mapping` approprié.

Détails mineurs

- Les types qui n'occupent pas la totalité des 32 octets peuvent contenir des « bits d'ordre supérieur sales ». Ceci est particulièrement important si vous accédez à `msg.data` - cela pose un risque de malléabilité : Vous pouvez créer des transactions qui appellent une fonction `f(uint8 x)` avec un argument brut de 32 octets de `0xff000001` et avec `0x00000001`. Les deux sont envoyés au contrat et les deux ressemblent au nombre 1 en ce qui concerne `x`, mais `msg.data` sera différente, donc si vous utilisez `keccak256(msg.data)` pour quoi que ce soit, vous obtiendrez des résultats différents.

3.28.2 Recommandations

Prenez les avertissements au sérieux

Si le compilateur vous avertit de quelque chose, vous devez le modifier. Même si vous ne pensez pas que cet avertissement particulier a des implications de sécurité, il peut y avoir un autre problème caché. Tout avertissement du compilateur que nous émettons peut être réduit au silence par de légères modifications du code.

Utilisez toujours la dernière version du compilateur pour être informé de tous les avertissements récemment introduits.

Les messages de type `info` émis par le compilateur ne sont pas dangereux, et représentent simplement des suggestions supplémentaires et des informations optionnelles que le compilateur pense pourrait être utile à l'utilisateur.

Limiter la quantité d'éther

Restreindre la quantité d'Ether (ou d'autres jetons) qui peut être stockée dans un contrat intelligent. Si votre code source, le compilateur ou la plateforme a un bug, ces fonds peuvent être perdus. Si vous voulez limiter vos pertes, limitez la quantité d'Ether.

Restez petit et modulaire

Gardez vos contrats petits et facilement compréhensibles. Isolez les fonctionnalités sans rapport dans d'autres contrats ou dans des bibliothèques. Les recommandations générales sur la qualité du code source s'appliquent bien sûr : Limitez la quantité de variables locales, la longueur des fonctions et ainsi de suite. Documentez vos fonctions afin que les autres puissent voir quelle était votre intention et si elle est différente de ce que fait le code.

Utiliser le modèle Verifications-Effects-Interactions

La plupart des fonctions vont d'abord effectuer quelques vérifications (qui a appelé la fonction, les arguments sont-ils à portée, ont-ils envoyé assez d'Ether, la personne a-t-elle des jetons, etc.) Ces vérifications doivent être effectuées en premier.

Dans un second temps, si toutes les vérifications sont passées, les effets sur les variables d'état du contrat en cours. L'interaction avec d'autres contrats doit être la toute dernière étape de toute fonction.

Les premiers contrats retardaient certains effets et attendaient que les appels de fonctions externes reviennent dans un état de non-erreur. C'est souvent une grave erreur à cause du problème de ré-entrance expliqué ci-dessus.

Notez également que les appels à des contrats connus peuvent à leur tour provoquer des appels à des contrats inconnus, il est donc probablement préférable de toujours appliquer ce modèle.

Inclure un mode de sécurité intégrée

Bien que le fait de rendre votre système entièrement décentralisé supprime tout intermédiaire, ce serait une bonne idée, surtout pour un nouveau code, d'inclure une sorte de mécanisme de sécurité :

Vous pouvez ajouter une fonction dans votre contrat intelligent qui effectue quelques des auto-vérifications comme « Y a-t-il eu une fuite d'Ether ? », « La somme des jetons est-elle égale au solde du contrat ? » ou des choses similaires. Gardez à l'esprit que vous ne pouvez pas utiliser trop d'essence pour cela, donc de l'aide par des calculs hors-chaîne peut être nécessaire.

Si l'auto-vérification échoue, le contrat passe automatiquement dans une sorte de mode « failsafe », qui, par exemple, désactive la plupart des fonctions, remet le contrôle à un tiers fixe et de confiance ou simplement convertir le contrat en un simple contrat « rendez-moi mon argent ».

Demandez un examen par les pairs

Plus il y a de personnes qui examinent un morceau de code, plus on découvre de problèmes. Demander à des personnes d'examiner votre code permet également de vérifier par recoupement si votre code est facile à comprendre - un critère très important pour les bons contrats intelligents.

3.29 SMTChecker et vérification formelle

En utilisant la vérification formelle, il est possible d'effectuer une preuve mathématique automatisée que votre code source répond à une certaine spécification formelle. La spécification est toujours formelle (tout comme le code source), mais généralement beaucoup plus simple.

Notez que la vérification formelle elle-même ne peut vous aider qu'à comprendre la différence entre ce que vous avez fait (la spécification) et la manière dont vous l'avez fait (l'implémentation réelle). Vous devez toujours vérifier si la spécification correspond à ce que vous vouliez et que vous n'avez pas manqué d'effets involontaires.

Solidity met en œuvre une approche de vérification formelle basée sur [SMT \(Satisfiability Modulo Theories\)](#) et la [Horn](#) de résolution. Le module SMTChecker essaie automatiquement de prouver que le code satisfait à la spécification donnée par les déclarations `require` et `assert`. C'est-à-dire qu'il considère les déclarations `require` comme des hypothèses et essaie de prouver que les conditions contenues dans les déclarations `assert` sont toujours vraies. Si un échec d'assertion est trouvé, un contre-exemple peut être donné à l'utilisateur montrant comment l'assertion peut être violée. Si aucun avertissement n'est donné par le SMTChecker pour une propriété, cela signifie que la propriété est sûre.

Les autres cibles de vérification que le SMTChecker vérifie au moment de la compilation sont :

- Les débordements et les sous-écoulements arithmétiques.
- La division par zéro.
- Conditions triviales et code inaccessible.
- Extraction d'un tableau vide.
- Accès à un index hors limites.
- Fonds insuffisants pour un transfert.

Toutes les cibles ci-dessus sont automatiquement vérifiées par défaut si tous les moteurs sont activés, sauf `underflow` et `overflow` pour Solidity $\geq 0.8.7$.

Les avertissements potentiels que le SMTChecker rapporte sont :

- `<failing property> happens here..` Cela signifie que le SMTChecker a prouvé qu'une certaine propriété est défaillante. Un contre-exemple peut être donné, cependant dans des situations complexes, il peut aussi ne pas montrer de contre-exemple. Ce résultat peut aussi être un faux positif dans certains cas, lorsque l'encodage SMT ajoute des abstractions pour le code Solidity qui est difficile ou impossible à exprimer.

- `<failing property> might happen here`. Cela signifie que le solveur n'a pas pu prouver l'un ou l'autre cas dans le délai imparti. Comme le résultat est inconnu, le SMTChecker rapporte l'échec potentiel pour la solidité. Cela peut être résolu en augmentant le délai d'interrogation, mais le problème peut aussi être simplement trop difficile à résoudre pour le moteur.

Pour activer le SMTChecker, vous devez sélectionner *quel moteur doit fonctionner*, où la valeur par défaut est aucun moteur. La sélection du moteur active le SMTChecker sur tous les fichiers.

Note : Avant Solidity 0.8.4, la manière par défaut d'activer le SMTChecker était via `pragma experimental SMTChecker`; et seuls les contrats contenant le pragma seraient analysés. Ce pragma a été déprécié, et bien qu'il active toujours le qu'il active toujours le SMTChecker pour une compatibilité ascendante, il sera supprimé dans Solidity 0.9.0. Notez également que maintenant l'utilisation du pragma même dans un seul fichier active le SMTChecker pour tous les fichiers.

Note : L'absence d'avertissement pour une cible de vérification représente une preuve mathématique incontestable de l'exactitude, en supposant l'absence de bogues dans le SMTChecker et le solveur sous-jacent. Gardez à l'esprit que ces problèmes sont *très difficiles* et parfois *impossibles* à résoudre automatiquement dans le cas général. Par conséquent, plusieurs propriétés pourraient ne pas être résolues ou pourraient conduire à des faux positifs pour les grands contrats. Chaque propriété prouvée doit être considérée comme une réalisation importante. Pour les utilisateurs avancés, voir *SMTChecker Tuning* pour apprendre quelques options qui pourraient aider à prouver des propriétés complexes.

3.29.1 Tutoriel

Débordement

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Overflow {
    uint immutable x;
    uint immutable y;

    function add(uint _x, uint _y) internal pure returns (uint) {
        return _x + _y;
    }

    constructor(uint _x, uint _y) {
        (x, y) = (_x, _y);
    }

    function stateAdd() public view returns (uint) {
        return add(x, y);
    }
}
```

Le contrat ci-dessus montre un exemple de vérification de débordement (overflow). Le SMTChecker ne vérifie pas l'underflow et l'overflow par défaut pour Solidity `>=0.8.7`, donc nous devons utiliser l'option de ligne de commande `--model-checker-targets "underflow,overflow"` ou l'option JSON `settings.modelChecker.targets = ["underflow", "overflow"]`. Voir *cette section pour la configuration des cibles*. Ici, il signale ce qui suit :

```

Warning: CHC: Overflow (resulting value larger than 2**256 - 1) happens here.
Counterexample:
x = 1, y = 115792089237316195423570985008687907853269984665640564039457584007913129639935
= 0

Transaction trace:
Overflow.constructor(1,
↳ 115792089237316195423570985008687907853269984665640564039457584007913129639935)
State: x = 1, y =
↳ 115792089237316195423570985008687907853269984665640564039457584007913129639935
Overflow.stateAdd()
  Overflow.add(1,
↳ 115792089237316195423570985008687907853269984665640564039457584007913129639935) --
↳ internal call
--> o.sol:9:20:
|
|
9 |         return _x + _y;
|           ^^^^^^^

```

Si nous ajoutons des déclarations `require` qui filtrent les cas de débordement, le SMTChecker prouve qu'aucun débordement n'est atteignable (en ne signalant pas d'avertissement) :

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Overflow {
    uint immutable x;
    uint immutable y;

    function add(uint _x, uint _y) internal pure returns (uint) {
        return _x + _y;
    }

    constructor(uint _x, uint _y) {
        (x, y) = (_x, _y);
    }

    function stateAdd() public view returns (uint) {
        require(x < type(uint128).max);
        require(y < type(uint128).max);
        return add(x, y);
    }
}

```

Affirmer

Une assertion représente un invariant dans votre code : une propriété qui doit être vraie *pour toutes les opérations, y compris toutes les valeurs d'entrée et de stockage*, sinon il y a un bug.

Le code ci-dessous définit une fonction `f` qui garantit l'absence de débordement. La fonction `inv` définit la spécification que `f` est monotone et croissante : pour chaque paire possible `(_a, _b)`, si `_b > _a` alors `f(_b) > f(_a)`. Puisque `f` est effectivement monotone et croissante, le SMTChecker prouve que notre propriété est correcte. Nous vous encourageons à jouer avec la propriété et la définition de la fonction pour voir les résultats qui en découlent !

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Monotonic {
    function f(uint _x) internal pure returns (uint) {
        require(_x < type(uint128).max);
        return _x * 42;
    }

    function inv(uint _a, uint _b) public pure {
        require(_b > _a);
        assert(f(_b) > f(_a));
    }
}
```

Nous pouvons également ajouter des assertions à l'intérieur des boucles pour vérifier des propriétés plus complexes. Le code suivant recherche l'élément maximum d'un tableau non restreint de nombres, et affirme la propriété selon laquelle l'élément trouvé doit être supérieur ou égal à chaque élément du tableau.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Max {
    function max(uint[] memory _a) public pure returns (uint) {
        uint m = 0;
        for (uint i = 0; i < _a.length; ++i)
            if (_a[i] > m)
                m = _a[i];

        for (uint i = 0; i < _a.length; ++i)
            assert(m >= _a[i]);

        return m;
    }
}
```

Notez que dans cet exemple, le SMTChecker va automatiquement essayer de prouver trois propriétés :

1. `++i` dans la première boucle ne déborde pas.
2. `++i` dans la deuxième boucle ne déborde pas.
3. L'assertion est toujours vraie.

Note : Les propriétés impliquent des boucles, ce qui rend l'exercice *beaucoup plus difficile* que les exemples précédents, alors faites attention aux boucles !

Toutes les propriétés sont correctement prouvées sûres. N'hésitez pas à modifier et/ou d'ajouter des restrictions sur le tableau pour obtenir des résultats différents. Par exemple, en changeant le code en

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Max {
    function max(uint[] memory _a) public pure returns (uint) {
        require(_a.length >= 5);
        uint m = 0;
        for (uint i = 0; i < _a.length; ++i)
            if (_a[i] > m)
                m = _a[i];

        for (uint i = 0; i < _a.length; ++i)
            assert(m > _a[i]);

        return m;
    }
}
```

nous donne :

```
Warning: CHC: Assertion violation happens here.
Counterexample:
```

```
_a = [0, 0, 0, 0, 0]
= 0
```

Transaction trace:

```
Test.constructor()
```

```
Test.max([0, 0, 0, 0, 0])
```

```
--> max.sol:14:4:
```

```
|
14 |         assert(m > _a[i]);
```

Propriétés de l'État

Jusqu'à présent, les exemples ont seulement démontré l'utilisation du SMTChecker sur du code pur, prouvant des propriétés sur des opérations ou des algorithmes spécifiques. Un type commun de propriétés dans les contrats intelligents sont les propriétés qui impliquent l'état du contrat. Plusieurs transactions peuvent être nécessaires pour faire échouer pour une telle propriété.

À titre d'exemple, considérons une grille 2D où les deux axes ont des coordonnées dans la plage $(-2^{128}, 2^{128} - 1)$. Plaçons un robot à la position $(0, 0)$. Le robot ne peut se déplacer qu'en diagonale, un pas à la fois, et ne peut pas se déplacer en dehors de la grille. La machine à états du robot peut être représentée par le contrat intelligent ci-dessous.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Robot {
    int x = 0;
    int y = 0;
```

(suite sur la page suivante)

(suite de la page précédente)

```

modifier wall {
    require(x > type(int128).min && x < type(int128).max);
    require(y > type(int128).min && y < type(int128).max);
    -;
}

function moveLeftUp() wall public {
    --x;
    ++y;
}

function moveLeftDown() wall public {
    --x;
    --y;
}

function moveRightUp() wall public {
    ++x;
    ++y;
}

function moveRightDown() wall public {
    ++x;
    --y;
}

function inv() public view {
    assert((x + y) % 2 == 0);
}

```

La fonction `inv` représente un invariant de la machine à états selon lequel $x + y$ doit être pair. Le SMTChecker parvient à prouver que quelque soit le nombre de commandes que l'on donne au robot, même s'ils sont infinis, l'invariant ne peut *jamais* échouer. Le lecteur intéressé peut vouloir prouver ce fait manuellement aussi. Indice : cet invariant est inductif.

Nous pouvons aussi tromper le SMTChecker pour qu'il nous donne un chemin vers une position que nous pensons être atteignable. Nous pouvons ajouter la propriété que $(2, 4)$ est *non* accessible, en ajoutant la fonction suivante.

```

function reach_2_4() public view {
    assert(!(x == 2 && y == 4));
}

```

Cette propriété est fausse, et tout en prouvant que la propriété est fausse, le SMTChecker nous dit exactement *comment* atteindre $(2, 4)$:

```

Warning: CHC: Assertion violation happens here.
Counterexample:
x = 2, y = 4

Transaction trace:
Robot.constructor()
State: x = 0, y = 0

```

(suite sur la page suivante)

```

Robot.moveLeftUp()
State: x = (- 1), y = 1
Robot.moveRightUp()
State: x = 0, y = 2
Robot.moveRightUp()
State: x = 1, y = 3
Robot.moveRightUp()
State: x = 2, y = 4
Robot.reach_2_4()
--> r.sol:35:4:
   |
35 |         assert(!(x == 2 && y == 4));
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

```

Notez que le chemin ci-dessus n'est pas nécessairement déterministe, car il y a d'autres chemins qui pourraient atteindre (2, 4). Le choix du chemin affiché peut changer en fonction du solveur utilisé, de sa version, ou simplement au hasard.

Appels externes et réentrance

Chaque appel externe est traité comme un appel à un code inconnu par le SMTChecker. Le raisonnement derrière cela est que même si le code du contrat appelé est disponible au moment de la compilation, il n'y a aucune garantie que le contrat déployé sera bien le même que le contrat d'où provient l'interface au moment de la compilation.

Dans certains cas, il est possible de déduire automatiquement des propriétés sur les variables d'état qui restent vraies même si le code appelé de l'extérieur peut faire n'importe quoi, y compris réintroduire le contrat de l'appelant.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

interface Unknown {
    function run() external;
}

contract Mutex {
    uint x;
    bool lock;

    Unknown immutable unknown;

    constructor(Unknown _u) {
        require(address(_u) != address(0));
        unknown = _u;
    }

    modifier mutex {
        require(!lock);
        lock = true;
        _;
        lock = false;
    }

    function set(uint _x) mutex public {

```

(suite sur la page suivante)

(suite de la page précédente)

```

    x = _x;
}

function run() mutex public {
    uint xPre = x;
    unknown.run();
    assert(xPre == x);
}
}

```

L'exemple ci-dessus montre un contrat qui utilise un drapeau mutex pour interdire la réentrance. Le solveur est capable de déduire que lorsque `unknown.run()` est appelé, le contrat est déjà « verrouillé », donc il ne serait pas possible de changer la valeur de `x`, indépendamment de ce que fait le code appelé inconnu.

Si nous « oublions » d'utiliser le modificateur `mutex` sur la fonction `set`, le SMTChecker est capable de synthétiser le comportement du code appelé de manière externe que l'assertion échoue :

```

Warning: CHC: Assertion violation happens here.
Counterexample:
x = 1, lock = true, unknown = 1

Transaction trace:
Mutex.constructor(1)
State: x = 0, lock = false, unknown = 1
Mutex.run()
  unknown.run() -- untrusted external call, synthesized as:
    Mutex.set(1) -- reentrant call
--> m.sol:32:3:
|
|
32 |               assert(xPre == x);
|               ^^^^^^^^^^^^^^^^^^^^^

```

3.29.2 Options et réglages de SMTChecker

Délai d'attente

Le SMTChecker utilise une limite de ressource codée en dur (`rlimit`) choisie par solveur, qui n'est pas précisément liée au temps. Nous avons choisi l'option `rlimit` comme défaut car elle donne plus de garanties de déterminisme que le temps à l'intérieur du solveur.

Cette option se traduit approximativement par « un délai de quelques secondes » par requête. Bien sûr de nombreuses propriétés sont très complexes et nécessitent beaucoup de temps pour être résolus, où le déterminisme n'a pas d'importance. Si le SMTChecker ne parvient pas à résoudre les propriétés du contrat avec le `rlimit` par défaut, un `timeout` peut être donné en millisecondes via l'option CLI `--model-checker-timeout <time>` ou l'option JSON `settings.modelChecker.timeout=<time>`, où 0 signifie pas de délai d'attente.

Objectifs de vérification

Les types de cibles de vérification créées par le SMTChecker peuvent aussi être personnalisés via l'option CLI `--model-checker-target <targets>` ou l'option JSON `settings.modelChecker.targets=<targets>`. Dans le cas de l'interface CLI, `<targets>` est une liste non séparée par des virgules d'une ou plusieurs cibles de vérification, et un tableau d'une ou plusieurs cibles comme l'entrée JSON. Les mots-clés qui représentent les cibles sont :

- Assertions : `assert`.
- Débordement arithmétique : `underflow`.
- Débordement arithmétique : `overflow`.
- La division par zéro : `divByZero`.
- Conditions triviales et code inaccessible : `constantCondition`.
- Extraire un tableau vide : `popEmptyArray`.
- Accès hors limites aux tableaux et aux index d'octets fixes : `outOfBounds`.
- Fonds insuffisants pour un transfert : `balance`.
- Tous ces éléments : défaut (CLI uniquement).

Un sous-ensemble commun de cibles pourrait être, par exemple : `--model-checker-targets assert,overflow`.

Toutes les cibles sont vérifiées par défaut, sauf `underflow` et `overflow` pour Solidity $\geq 0.8.7$.

Il n'y a pas d'heuristique précise sur comment et quand diviser les cibles de vérification, mais cela peut être utile, surtout lorsqu'il s'agit de grands contrats.

Cibles non vérifiées

S'il existe des cibles non vérifiées, le SMTChecker émet un avertissement indiquant combien de cibles non vérifiées il y a. Si l'utilisateur souhaite voir toutes les cibles non corrigées, l'option CLI `--model-checker-show-unproved` et l'option JSON `settings.modelChecker.showUnproved = true` peuvent être utilisées.

Contrats vérifiés

Par défaut, tous les contrats déployables dans les sources données sont analysés séparément en tant que celui qui sera déployé. Cela signifie que si un contrat a de nombreux parents d'héritage direct et indirect, ils seront tous analysés séparément, même si seul le plus dérivé sera accessible directement sur la blockchain. Cela entraîne une charge inutile pour le SMTChecker et le solveur. Pour aider les cas comme celui-ci, les utilisateurs peuvent spécifier quels contrats doivent être analysés comme le déployé. Les contrats parents sont bien sûr toujours analysés, mais seulement dans le contexte du contrat le plus dérivé, ce qui réduit la complexité de l'encodage et des requêtes générées. Notez que les contrats abstraits ne sont par défaut pas analysés comme les plus dérivés par le SMTChecker.

Les contrats choisis peuvent être donnés via une liste séparée par des virgules (les espaces blancs ne sont pas autorisés) de paires `<source> :<contrat>` dans le CLI : `--model-checker-contracts "<source1.sol:contract1>,<source2.sol:contract2>,<source2.sol:contract3>"`, et via l'objet `settings.modelChecker.contracts` dans le *JSON input*, qui a la forme suivante :

```
"contracts": {
  "source1.sol": ["contract1"],
  "source2.sol": ["contract2", "contract3"]
}
```

Invariants inductifs rapportés et inférés

Pour les propriétés qui ont été prouvées sûres avec le moteur CHC, le SMTChecker peut récupérer les invariants inductifs qui ont été inférés par le solveur de Horn dans le cadre de la preuve. Actuellement, deux types d'invariants peuvent être rapportés à l'utilisateur :

- Invariants de contrat : ce sont des propriétés sur les variables d'état du contrat qui sont vraies avant et après chaque transaction possible que le contrat peut exécuter. Par exemple, $x \geq y$, où x et y sont les variables d'état d'un contrat.
- Propriétés de réentrainement : elles représentent le comportement du contrat en présence d'appels externes à du code inconnu. Ces propriétés peuvent exprimer une relation entre la valeur des variables d'état avant et après l'appel externe, où l'appel externe est libre de faire n'importe quoi, y compris d'effectuer des appels réentrants au contrat analysé. Les variables amorcées représentent les valeurs des variables d'état après ledit appel externe. Exemple : `lock -> x = x'`.

L'utilisateur peut choisir le type d'invariants à rapporter en utilisant l'option CLI `--model-checker-invariants "contract, reentrancy"` ou comme un tableau dans le champ `settings.modelChecker.invariants` dans l'entrée *JSON*. Par défaut, le SMTChecker ne rapporte pas les invariants.

Division et modulo avec des variables muettes

Spacer, le solveur de Corne par défaut utilisé par le SMTChecker, n'aime souvent pas les opérations de division et de modulo dans les règles de Horn. Pour cette raison, par défaut, les opérations de division et de modulo de Solidity sont codées en utilisant la contrainte suivante $a = b * d + m$ où $d = a / b$ et $m = a \% b$. Cependant, d'autres solveurs, comme Eldarica, préfèrent les opérations syntaxiquement précises. L'indicateur de ligne de commande `--model-checker-div-mod-no-slacks` et l'option JSON `settings.modelChecker.divModNoSlacks` peuvent être utilisés pour basculer le codage en fonction des préférences du solveur utilisé.

Abstraction des fonctions Natspec

Certaines fonctions, y compris les méthodes mathématiques courantes telles que `pow` et `sqrt` peuvent être trop complexes pour être analysées de manière entièrement automatisée. Ces fonctions peuvent être annotées avec des balises Natspec qui indiquent au contrôleur SMTChecker que ces fonctions doivent être abstraites. Cela signifie que de la fonction n'est pas utilisé et que, lorsqu'elle est appelée, la fonction :

- retournera une valeur non déterministe, et soit gardera les variables d'état inchangées si la fonction abstraite est `view/pure`, soit fixera également les variables d'état à des valeurs non déterministes dans le cas contraire. Ceci peut être utilisé via l'annotation `/// @custom:smtchecker abstract-function-nondet`.
- Agir comme une fonction non interprétée. Cela signifie que la sémantique de la fonction (donnée par le corps) est ignorée, et que la seule propriété de cette fonction est que, pour une même entrée, elle garantit la même sortie. Ceci est actuellement en cours de développement et sera disponible via l'annotation `/// @custom:smtchecker abstract-function-uf`.

Moteurs de vérification de modèles réduits

Le module SMTChecker implémente deux moteurs de raisonnement différents, un Bounded Model Checker (BMC) et un système de Clauses de Corne Contraintes (CHC). Les deux moteurs sont actuellement en cours de développement, et ont des caractéristiques différentes. Les moteurs sont indépendants et chaque avertissement de propriété indique de quel moteur il provient. Notez que tous les exemples ci-dessus avec des contre-exemples ont été rapportés par CHC, le moteur le plus puissant.

Par défaut, les deux moteurs sont utilisés, CHC s'exécute en premier, et chaque propriété qui n'a pas été prouvée est transmise à BMC. Vous pouvez choisir un moteur spécifique via l'interface CLI `--model-checker-engine {all, bmc, chc, none}` ou l'option JSON `settings.modelChecker.engine={all, bmc, chc, none}`.

Contrôleur de modèles délimités (BMC)

Le moteur BMC analyse les fonctions de manière isolée, c'est-à-dire qu'il ne prend pas en compte le comportement global du contrat sur plusieurs transactions lorsqu'il analyse chaque fonction. Les boucles sont également ignorées dans ce moteur pour le moment. Les appels de fonctions internes sont inlined tant qu'ils ne sont pas récursifs, directement ou indirectement. Les appels de fonctions externes sont inlined si possible. Connaissance qui est potentiellement affectée par la réentrance est effacée.

Les caractéristiques ci-dessus font que la BMC est susceptible de signaler des faux positifs, mais il est également léger et devrait être capable de trouver rapidement de petits bogues locaux.

Clauses de corne contraintes (CHC)

Le graphique de flux de contrôle (CFG) d'un contrat est modélisé comme un système de clauses de Horn, où le cycle de vie du contrat est représenté par une boucle qui peut visiter chaque fonction publique/externe de manière non-déterministe. De cette façon, le comportement de l'ensemble du contrat sur un nombre illimité de transactions est pris en compte lors de l'analyse de toute fonction. Les boucles sont entièrement prises en charge par ce moteur. Les appels de fonctions internes sont pris en charge, et les appels de fonctions externes supposent que le code appelé est inconnu et peut faire n'importe quoi.

Le moteur CHC est beaucoup plus puissant que BMC en termes de ce qu'il peut prouver, et peut nécessiter plus de ressources informatiques.

Solveurs SMT et Horn

Les deux moteurs détaillés ci-dessus utilisent des prouveurs de théorèmes automatisés comme leur logique. BMC utilise un solveur SMT, tandis que CHC utilise un solveur de Horn. Souvent le même outil peut agir comme les deux, comme on le voit dans `z3`, qui est principalement un solveur SMT et qui rend `Spacer` disponible comme solveur de Horn, et `Eldarica` qui fait les deux.

L'utilisateur peut choisir quels solveurs doivent être utilisés, s'ils sont disponibles, via l'option CLI `--model-checker-solvers {all,cvc4,smtlib2,z3}` ou l'option JSON `settings.modelChecker.solvers=[smtlib2,z3]`, où :

- `cvc4` n'est disponible que si le binaire `solc` est compilé avec. Seul BMC utilise `cvc4`.
- `smtlib2` produit des requêtes SMT/Horn dans le format `smtlib2`. Celles-ci peuvent être utilisées avec le [mécanisme de rappel du compilateur](#) de sorte que tout solveur binaire du système peut être employé pour renvoyer de manière synchrone les résultats des requêtes au compilateur. C'est actuellement la seule façon d'utiliser `Eldarica`, par exemple, puisqu'il ne dispose pas d'une API C++. Cela peut être utilisé à la fois par BMC et CHC, selon les solveurs appelés.
- `z3` est disponible
 - si `solc` est compilé avec lui ;
 - si une bibliothèque dynamique `z3` de version 4.8.x est installée dans un système Linux (à partir de Solidity 0.7.6) ;
 - statiquement dans `soljson.js` (à partir de Solidity 0.6.9), c'est-à-dire le binaire Javascript du compilateur.

Étant donné que BMC et CHC utilisent tous deux `z3`, et que `z3` est disponible dans une plus grande variété d'environnements, y compris dans le navigateur, la plupart des utilisateurs n'auront presque jamais à se préoccuper de cette option. Les utilisateurs plus avancés peuvent utiliser cette option pour essayer des solveurs alternatifs sur des problèmes plus complexes.

Veuillez noter que certaines combinaisons de moteur et de solveur choisis conduiront à ce que `SMTChecker` ne fera rien, par exemple choisir CHC et `cvc4`.

3.29.3 Abstraction et faux positifs

Le SMTChecker implémente les abstractions d'une manière incomplète et saine : Si un bogue est signalé, il peut s'agir d'un faux positif introduit par les abstractions (dû à l'effacement de connaissances ou l'utilisation d'un type non précis). S'il détermine qu'une cible de vérification est sûre, elle est effectivement sûre, c'est-à-dire qu'il n'y a pas de faux négatifs (à moins qu'il y ait un bug dans le SMTChecker).

Si une cible ne peut pas être prouvée, vous pouvez essayer d'aider le solveur en utilisant les options de réglage dans la section précédente. Si vous êtes sûr d'un faux positif, ajouter des déclarations `require` dans le code avec plus d'informations peut également donner plus de puissance au solveur.

Encodage et types SMT

L'encodage SMTChecker essaye d'être aussi précis que possible, en faisant correspondre les types et expressions Solidity à leur représentation `SMT-LIB` la plus proche, comme le montre le tableau ci-dessous.

Type Solidity	Triage SMT	Théories
Booléen	Bool	Bool
intN, uintN, address, bytesN, enum, contract	Integer	LIA, NIA
array, mapping, bytes, string	Tuple (Array elements, Integer length)	Datatypes, Arrays, LIA
struct	Tuple	Datatypes
autres types	Integer	LIA

Les types qui ne sont pas encore pris en charge sont abstraits par un seul entier non signé de 256 bits, où leurs opérations non supportées sont ignorées.

Pour plus de détails sur la façon dont l'encodage SMT fonctionne en interne, voir l'article [Vérification basée sur SMT des contrats intelligents Solidity](#).

Appels de fonction

Dans le moteur BMC, les appels de fonctions vers le même contrat (ou contrats de base) sont inlined lorsque cela est possible, c'est-à-dire lorsque leur implémentation est disponible. Les appels de fonctions dans d'autres contrats ne sont pas inlined même si leur code est disponible, car nous ne pouvons pas garantir que le code déployé est le même.

Le moteur CHC crée des clauses Horn non linéaires qui utilisent des résumés des fonctions appelées pour prendre en charge les appels de fonctions internes. Les appels de fonctions externes sont traités comme des appels à du code inconnu, y compris les appels réentrants potentiels.

Les fonctions pures complexes sont abstraites par une fonction non interprétée (UF) sur les arguments.

Fonctions	Comportement BMC/CHC
<code>assert</code>	Objectif de vérification.
<code>require</code>	Assomption.
appel interne	BMC : Appel de fonction en ligne. CHC : Résumés des fonctions.
appel externe à un code connu	BMC : Appel de fonction en ligne ou L'appel de fonction en ligne ou l'effacement des connaissances sur les variables d'état et des références de stockage local. CHC : Supposer que le code appelé est inconnu. Essayer de déduire les invariants qui tiennent après le retour de l'appel.
Réseau de stockage push/pop	Supporté précisément. Vérifie s'il s'agit de faire sauter un tableau vide.
Fonctions ABI	Abstracted with UF.
<code>addmod</code> , <code>mulmod</code>	Supported precisely.
<code>gasleft</code> , <code>blockhash</code> , <code>keccak256</code> , <code>ecrecover</code> <code>ripemd160</code>	Abstracted with UF.
Fonctions pures sans implémentation (externe ou complexe)	Abstraction avec UF
fonctions externes sans mise en œuvre	BMC : Effacer les connaissances de l'État et assumer Le résultat est indéterminé. CHC : Résumé non déterministe. Essayez d'inférer des invariants qui tiennent après le retour de l'appel.
<code>transfert</code>	BMC : Vérifie si le solde du contrat est suffisant. CHC : n'effectue pas encore le contrôle.
autres	Actuellement non pris en charge

L'utilisation de l'abstraction signifie la perte de connaissances précises, mais dans de nombreux cas, elle ne signifie pas une perte de puissance de preuve.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Recover
{
    function f(
        bytes32 hash,
        uint8 _v1, uint8 _v2,
        bytes32 _r1, bytes32 _r2,
        bytes32 _s1, bytes32 _s2
    ) public pure returns (address) {
        address a1 = ecrecover(hash, _v1, _r1, _s1);
        require(_v1 == _v2);
        require(_r1 == _r2);
        require(_s1 == _s2);
        address a2 = ecrecover(hash, _v2, _r2, _s2);
        assert(a1 == a2);
        return a1;
    }
}
```

Dans l'exemple ci-dessus, le SMTChecker n'est pas assez expressif pour calculer réellement « ecrecover », mais en modélisant les appels de fonctions comme des fonctions non interprétées, nous savons que la valeur de retour est la

même lorsqu'elle est appelée avec des paramètres équivalents. Ceci est suffisant pour prouver que l'assertion ci-dessus est toujours vraie.

L'abstraction d'un appel de fonction avec un UF peut être faite pour des fonctions connues pour être déterministes, et peut être facilement réalisée pour les fonctions pures. Il est cependant difficile de le faire avec des fonctions externes générales, puisqu'elles peuvent de variables d'état.

Types de référence et alias

Solidity implémente l'aliasing pour les types de référence avec le même *data emplacement*. Cela signifie qu'une variable peut être modifiée à travers une référence à la même données. Le SMTChecker ne garde pas trace des références qui font référence aux mêmes données. Cela implique que chaque fois qu'une référence locale ou une variable d'état de type référence est assignée, toutes les connaissances concernant les variables de même type et de même emplacement données est effacée. Si le type est imbriqué, la suppression de la connaissance inclut également tous les types.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Aliasing
{
    uint[] array1;
    uint[][] array2;
    function f(
        uint[] memory a,
        uint[] memory b,
        uint[][] memory c,
        uint[] storage d
    ) internal {
        array1[0] = 42;
        a[0] = 2;
        c[0][0] = 2;
        b[0] = 1;
        // Effacer les connaissances sur les références mémoire ne devrait pas
        // effacer les connaissances sur les variables d'état.
        assert(array1[0] == 42);
        // Cependant, une affectation à une référence de stockage effacera
        // la connaissance du stockage en conséquence.
        d[0] = 2;
        // Échoue en tant que faux positif à cause de l'affectation ci-dessus.
        assert(array1[0] == 42);
        // Échoue car `a == b` est possible.
        assert(a[0] == 2);
        // Échoue car `c[i] == b` est possible.
        assert(c[0][0] == 2);
        assert(d[0] == 2);
        assert(b[0] == 1);
    }
    function g(
        uint[] memory a,
        uint[] memory b,
        uint[][] memory c,
        uint x
    ) public {
```

(suite sur la page suivante)

```
    f(a, b, c, array2[x]);  
  }  
}
```

Après l'affectation à `b[0]`, nous devons effacer la connaissance de `a`, puisqu'il a le même type (`uint[]`) et le même emplacement de données (mémoire). Nous devons également effacer les connaissances sur `c`, puisque son type de base est également un `uint[]` situé dans la mémoire. Cela implique qu'un `c[i]` pourrait faire référence aux mêmes données que `b` ou `a`.

Remarquez que nous n'avons pas de connaissances claires sur `array` et `d`, parce qu'ils sont situés dans le stockage, même s'ils ont aussi le type `uint[]`. Cependant, si `d` était assigné, nous devrions effacer la connaissance sur `array` et vice-versa.

Bilan des contrats

Un contrat peut être déployé avec des fonds qui lui sont envoyés, si `msg.value > 0` dans la transaction de déploiement. Cependant, l'adresse du contrat peut déjà avoir des fonds avant le déploiement, qui sont conservés par le contrat. Par conséquent, le SMTChecker suppose que `address(this).balance >= msg.value` dans le constructeur afin d'être cohérent avec les règles EVM. Le solde du contrat peut également augmenter sans déclencher d'appel au contrat, si :

- `selfdestruct` est exécuté par un autre contrat avec le contrat analysé comme cible des fonds restants,
- le contrat est la base de données de pièces de monnaie (i.e., `block.coinbase`) d'un bloc.

Pour modéliser cela correctement, le SMTChecker suppose qu'à chaque nouvelle transaction le solde du contrat peut augmenter d'au moins `msg.value`.

3.29.4 Hypothèses du monde réel

Certains scénarios peuvent être exprimés dans Solidity et dans l'EVM, mais on s'attend à ce qu'ils ne se produisent jamais se produire dans la pratique. L'un de ces cas est la longueur d'un tableau de stockage dynamique qui déborde pendant un processus de poussée : Si l'opération `push` est appliquée à un tableau de longueur $2^{256} - 1$, sa longueur déborde silencieusement. Cependant, il est peu probable que cela se produise dans la pratique, car les opérations nécessaires pour faire croître le tableau à ce point prendraient des milliards d'années à être exécutées. Une autre hypothèse similaire prise par le SMTChecker est que le solde d'une adresse ne peut jamais déborder.

Une idée similaire a été présentée dans [EIP-1985](#).

3.30 Ressources

3.30.1 Ressources générales

- [Ethereum.org Developer Portal](#)
- [Ethereum StackExchange](#)
- [Solidity Portal](#)
- [Solidity Changelog](#)
- [Solidity Source Code sur GitHub](#)
- [Solidity Chat des utilisateurs de langues](#)
- [Solidity Chat des développeurs de compilateurs](#)
- [Awesome Solidity](#)
- [Solidity par Example](#)

3.30.2 Environnements de développement intégrés (Ethereum)

- **Brownie** Cadre de développement et de test basé sur Python pour les contrats intelligents ciblant la machine virtuelle Ethereum.
- **Dapp** Outil pour construire, tester et déployer des contrats intelligents à partir de la ligne de commande.
- **Embark** Plateforme de développeurs pour la création et le déploiement d'applications décentralisées.
- **Hardhat** Environnement de développement Ethereum avec réseau Ethereum local, fonctions de débogage et écosystème de plugins.
- **Remix** IDE basé sur un navigateur avec compilateur intégré et environnement d'exécution Solidity sans composants côté serveur.
- **Scaffold-ETH** Pile de développement Ethereum axée sur des itérations rapides du produit.
- **Truffle** Cadre de développement Ethereum.

3.30.3 Intégrations de l'éditeur

- Atom
 - **Etheratom** Plugin pour l'éditeur Atom qui propose la coloration syntaxique, la compilation et un environnement d'exécution (compatible avec les nœuds Backend et VM).
 - **Atom Solidity Linter** Plugin pour l'éditeur Atom qui fournit le linting Solidity.
 - **Atom Solium Linter** Linter Solidity configurable pour Atom utilisant Solium (maintenant Ethlint) comme base.
- Emacs
 - **Emacs Solidity** Plugin pour l'éditeur Emacs fournissant la coloration syntaxique et le signalement des erreurs de compilation.
- IntelliJ
 - **IntelliJ IDEA plugin** Plugin Solidity pour IntelliJ IDEA (et tous les autres IDE de JetBrains)
- Sublime
 - **Package for SublimeText - Solidity language syntax** Coloration syntaxique Solidity pour l'éditeur SublimeText.
- Vim
 - **Vim Solidity** Plugin pour l'éditeur Vim fournissant une coloration syntaxique.
 - **Vim Syntastic** Plugin pour l'éditeur Vim permettant de vérifier la compilation.
- Visual Studio Code
 - **Visual Studio Code extension** Plugin Solidity pour Microsoft Visual Studio Code qui comprend la coloration syntaxique et le compilateur Solidity.

3.30.4 Outils Solidity

- **ABI to Solidity interface converter** Un script pour générer des interfaces de contrat à partir de l'ABI d'un contrat intelligent.
- **abi-to-sol** Outil permettant de générer une source d'interface Solidity à partir d'un JSON ABI donné.
- **Doxity** Générateur de documentation pour Solidity.
- **Ethlint** Linter pour identifier et corriger les problèmes de style et de sécurité dans Solidity.
- **evmdis** EVM Disassembler qui effectue une analyse statique sur le bytecode pour fournir un niveau d'abstraction plus élevé que les opérations EVM brutes.
- **EVM Lab** Ensemble d'outils riches pour interagir avec l'EVM. Comprend une VM, une API Etherchain et un visualiseur de traces avec affichage du coût du gaz.
- **hevm** Débogueur EVM et moteur d'exécution symbolique.
- **leaffleth** Un générateur de documentation pour les smart-contracts de Solidity.
- **PIET** Un outil pour développer, auditer et utiliser les contrats intelligents Solidity à travers une interface graphique simple.

- **sol2uml** Générateur de diagrammes de classe en langage de modélisation unifié (UML) pour les contrats Solidity.
- **solc-select** A script to quickly switch between Solidity compiler versions.
- **Solidity prettier plugin** Un plugin Prettier pour Solidity.
- **Solidity REPL** Essayez Solidity instantanément avec une console Solidity en ligne de commande.
- **solgraph** Visualisez le flux de contrôle Solidity et mettez en évidence les vulnérabilités potentielles en matière de sécurité.
- **Solhint** Linter Solidity qui fournit la sécurité, un guide de style et des règles de bonnes pratiques pour la validation des contrats intelligents.
- **Sūrya** Outil utilitaire pour les systèmes de contrats intelligents, offrant un certain nombre de sorties visuelles et des informations sur la structure des contrats. Il permet également d'interroger le graphe des appels de fonction.
- **Universal Mutator** Un outil pour la génération de mutations, avec des règles configurables et le support de Solidity et Vyper.

3.30.5 Analyseurs et grammaires Solidity tiers

- **Solidity Parser for JavaScript** Un analyseur Solidity pour JS construit à partir d'une grammaire ANTLR4 robuste.

3.31 Résolution du chemin d'importation

Afin de pouvoir supporter des constructions reproductibles sur toutes les plateformes, le compilateur Solidity doit faire abstraction des détails du système de fichiers où sont stockés les fichiers sources. Les chemins utilisés dans les importations doivent fonctionner de la même manière partout, tandis que l'interface de la ligne de commande doit être capable de travailler avec des chemins spécifiques à la plate-forme pour fournir une bonne expérience utilisateur. Cette section vise à expliquer en détail comment Solidity concilie ces exigences.

3.31.1 Système de fichiers virtuel

Le compilateur maintient une base de données interne (*système de fichiers virtuel* ou *VFS* en abrégé) dans laquelle chaque unité source se voit attribuer un *nom d'unité source* unique qui est un identifiant opaque et non structuré. Lorsque vous utilisez l'instruction *import*, vous spécifiez un *chemin d'accès à l'importation* qui fait référence à un nom d'unité source.

Rappel d'importation

Le VFS n'est initialement peuplé que de fichiers que le compilateur a reçus en entrée. Des fichiers supplémentaires peuvent être chargés pendant la compilation en utilisant un *import callback*, qui est différent selon le type de compilateur que vous utilisez (voir ci-dessous). Si le compilateur ne trouve pas de nom d'unité source correspondant au chemin d'importation dans le VFS, il invoque le callback, qui est chargé d'obtenir le code source à placer sous ce nom. Un callback d'importation est libre d'interpréter les noms d'unité source d'une manière arbitraire, pas seulement comme des chemins. S'il n'y a pas de callback disponible lorsqu'on en a besoin ou s'il ne parvient pas à localiser le code source, la compilation échoue.

Le compilateur en ligne de commande fournit le *Host Filesystem Loader* - un rappel rudimentaire qui interprète un nom d'unité source comme un chemin dans le système de fichiers local. L'interface *JavaScript* n'en fournit pas par défaut, mais un peut être fourni par l'utilisateur. Ce mécanisme peut être utilisé pour obtenir du code source à partir d'emplacements autres que le système de fichiers local (qui peut même ne pas être accessible, par exemple lorsque le compilateur est exécuté dans un navigateur). Par exemple l'IDE *Remix* fournit un callback polyvalent qui vous permet

d'importer des fichiers à partir d'URL HTTP, IPFS et Swarm ou de vous référer directement à des paquets dans le registre NPM.

Note : La recherche de fichiers du Host Filesystem Loader dépend de la plate-forme. Par exemple, les barres obliques inverses dans le nom d'une unité source peuvent être interprétées comme des séparateurs de répertoire ou non, et la recherche peut être sensible à la casse ou non, selon la plate-forme sous-jacente.

Pour des raisons de portabilité, il est recommandé d'éviter d'utiliser des chemins d'importation qui ne fonctionnent correctement qu'avec avec une fonction d'appel d'importation spécifique ou uniquement sur une plate-forme. Par exemple, vous devriez toujours utiliser des slashes avant car ils fonctionnent comme des séparateurs de chemin également sur plateformes qui prennent en charge les barres obliques inversées.

Contenu initial du système de fichiers virtuel

Le contenu initial du VFS dépend de la façon dont vous invoquez le compilateur :

1. solc / command-line interface

Lorsque vous compilez un fichier à l'aide de l'interface de ligne de commande du compilateur, vous fournissez un ou plusieurs chemins d'accès à des fichiers contenant du code Solidity :

```
solc contract.sol /usr/local/dapp-bin/token.sol
```

Le nom de l'unité source d'un fichier chargé de cette façon est construit en convertissant son chemin d'accès à une forme canonique et, si possible, en le rendant relatif au chemin de base ou à l'un des chemins d'inclusion. Reportez-vous à *CLI Path Normalization and Stripping* pour une description détaillée de ce processus.

2. Standard JSON

Le nom de l'unité source d'un fichier chargé de cette façon est construit en convertissant son chemin d'accès à une forme canonique et, si possible, en le rendant relatif au chemin de base ou à l'un des chemins d'inclusion. Reportez-vous à *CLI Path Normalization and Stripping* pour une description détaillée de ce processus.

```
{
  "language": "Solidity",
  "sources": {
    "contract.sol": {
      "content": "import \"./util.sol\";\ncontract C {}"
    },
    "util.sol": {
      "content": "library Util {}"
    },
    "/usr/local/dapp-bin/token.sol": {
      "content": "contract Token {}"
    }
  },
  "settings": {"outputSelection": {"*": { "*": ["metadata", "evm.bytecode"]}}}
}
```

Le dictionnaire sources devient le contenu initial du système de fichiers virtuel et ses clés sont utilisées comme noms d'unités sources.

3. Standard JSON (via import callback)

Avec Standard JSON, il est également possible d'indiquer au compilateur d'utiliser le callback d'importation pour obtenir le code source :

```

{
  "language": "Solidity",
  "sources": {
    "/usr/local/dapp-bin/token.sol": {
      "urls": [
        "/projects/mytoken.sol",
        "https://example.com/projects/mytoken.sol"
      ]
    }
  },
  "settings": {"outputSelection": {"*": { "*": ["metadata", "evm.bytecode"]}}}
}

```

Si un import callback est disponible, le compilateur lui donnera les chaînes spécifiées dans `urls` une par une, jusqu'à ce qu'une soit chargée avec succès ou que la fin de la liste soit atteinte.

Les noms des unités de sources sont déterminés de la même manière que lors de l'utilisation de `content` - ce sont des clés du dictionnaire `sources` et le contenu de `urls` ne les affecte en aucune façon.

4. Entrée standard

En ligne de commande, il est également possible de fournir la source en l'envoyant à l'entrée standard du compilateur :

```
echo 'import "./util.sol"; contract C {}' | solc -
```

- utilisé comme l'un des arguments indique au compilateur de placer le contenu de l'entrée standard dans le système de fichiers virtuel sous un nom d'unité source spécial : `<stdin>`.

Une fois le VFS initialisé, des fichiers supplémentaires ne peuvent y être ajoutés que par le biais de la fonction `import` pour y ajouter des fichiers.

3.31.2 Imports

L'instruction d'importation spécifie un *chemin d'importation*. En fonction de la façon dont le chemin d'importation est spécifié, nous pouvons diviser les importations en deux catégories :

- *Imports direct*, où vous spécifiez directement le nom complet de l'unité source.
- *Relative imports*, où vous spécifiez un chemin commençant par `./` ou `../` à combiner avec le nom de l'unité source du fichier d'importation.

Code source 1 – contracts/contract.sol

```
import "../math/math.sol";
import "contracts/tokens/token.sol";
```

Dans l'exemple ci-dessus, `../math/math.sol` et `contracts/tokens/token.sol` sont des chemins d'importation alors que les noms d'unités sources vers lesquels ils sont traduits sont respectivement `contracts/math/math.sol` et `contracts/tokens/token.sol`.

Importations directes

Une importation qui ne commence pas par `./` ou `../` est une *importation directe*.

```
import "/project/lib/util.sol";           // nom de l'unité source: /project/lib/util.sol
import "lib/util.sol";                   // nom de l'unité source: lib/util.sol
import "@openzeppelin/address.sol";      // nom de l'unité source: @openzeppelin/address.
↳ sol
import "https://example.com/token.sol";  // nom de l'unité source: https://example.com/
↳ token.sol
```

Après avoir appliqué tout *import remappings*, le chemin d'importation devient simplement le nom de l'unité source.

Note : Le nom d'une unité source n'est qu'un identifiant et même si sa valeur ressemble à un chemin, il n'est pas soumis aux règles de normalisation que l'on peut attendre d'un shell. Tous les segments `./` ou `../` ou les séquences de barres obliques multiples en font toujours partie. Lorsque la source est fournie via une interface JSON standard, il est tout à fait possible d'associer différents contenus à des noms d'unités de source qui feraient référence au même fichier sur le disque.

Lorsque la source n'est pas disponible dans le système de fichiers virtuel, le compilateur transmet le nom de l'unité source à l'import callback. Le Host Filesystem Loader tentera de l'utiliser comme chemin et de rechercher le fichier sur le disque. À ce stade, les règles de normalisation spécifiques à la plate-forme entrent en jeu et les noms qui étaient considérés comme différents dans le VFS peuvent en fait aboutir au chargement du même fichier. Par exemple, `/projet/lib/math.sol` et `/projet/lib/./lib//math.sol` sont considérés comme complètement différents dans le VFS même s'ils font référence au même fichier sur le disque.

Note : Même si un callback d'importation finit par charger du code source pour deux noms d'unité source différents à partir du même fichier sur le disque, le compilateur les verra toujours comme des unités sources distinctes. C'est le nom de l'unité source qui importe, pas l'emplacement physique du code.

Importations relatives

Une importation commençant par `./` ou `../` est une importation *relative*. Ces importations spécifient un chemin relatif au nom de l'unité source de l'unité source importatrice :

Code source 2 – /project/lib/math.sol

```
import "../util.sol" as util;           // nom de l'unité source: /project/lib/util.sol
import "../token.sol" as token;        // nom de l'unité source: /project/token.sol
```

Code source 3 – lib/math.sol

```
import "./util.sol" as util; // nom de l'unité source: lib/util.sol
import "./token.sol" as token; // nom de l'unité source: token.sol
```

Note : Les importations relatives commencent toujours par ./ ou ../. `import "./util.sol"`, est une importation directe. Alors que les deux chemins seraient considérés comme relatifs dans le système de fichiers hôte, `util.sol` est en fait absolu dans le VFS.

Définissons un *segment de chemin* comme toute partie non vide du chemin qui ne contient pas de séparateur et qui est délimitée par deux séparateurs de chemin. Un séparateur est un slash avant ou le début/la fin de la chaîne. Par exemple, dans `./abc/././`, il y a trois segments de chemin : `.`, `abc` et `..`.

Le compilateur calcule un nom d'unité source à partir du chemin d'importation de la manière suivante :

1. Un préfixe est d'abord calculé
 - Le préfixe est initialisé avec le nom de l'unité source de l'unité source importatrice.
 - Le dernier segment de chemin avec les barres obliques précédentes est supprimé du préfixe.
 - Ensuite, la partie avant du chemin d'importation normalisé, composée uniquement de caractères / et ., est prise en compte. Pour chaque segment .. trouvé dans cette partie, le dernier segment de chemin avec les barres obliques précédant est supprimé du préfixe.
2. Ensuite, le préfixe est ajouté au chemin d'importation normalisé. Si le préfixe n'est pas vide, une seule barre oblique est insérée entre lui et le chemin d'importation.

L'élimination du dernier segment de chemin avec les barres obliques précédentes fonctionne comme suit :

1. Tout ce qui dépasse la dernière barre oblique est supprimé (c'est-à-dire que `a/b//c.sol` devient `a/b//`).
2. Toutes les barres obliques de fin de ligne sont supprimées (par exemple, `a/b//` devient `a/b`).

Les règles de normalisation sont les mêmes que pour les chemins UNIX, à savoir :

- Tous les segments internes . sont supprimés.
- Chaque segment interne .. remonte d'un niveau dans la hiérarchie.
- Les slashes multiples sont écrasés en un seul.

Notez que la normalisation est effectuée uniquement sur le chemin d'importation. Le nom de l'unité source du module d'importation qui est utilisé pour le préfixe n'est pas normalisé. Cela garantit que la partie `protocol://` ne se transforme pas en `protocol:/` si le fichier d'importation est identifié par une URL.

Si vos chemins d'importation sont déjà normalisés, vous pouvez vous attendre à ce que l'algorithme ci-dessus produise des résultats très intuitifs. Voici quelques exemples de ce que vous pouvez attendre s'ils ne le sont pas :

Code source 4 – lib/src/./contract.sol

```
import "./util/./util.sol"; // nom de l'unité source: lib/src/./util/util.sol
import "./util//util.sol"; // nom de l'unité source: lib/src/./util/util.sol
import "../util/./array/util.sol"; // nom de l'unité source: lib/src/array/util.sol
import ".././.././util.sol"; // nom de l'unité source: util.sol
import ".././.././../util.sol"; // nom de l'unité source: util.sol
```

Note : L'utilisation d'importations relatives contenant des segments .. en tête n'est pas recommandée. Le même effet peut être obtenu de manière plus fiable en utilisant des importations directes avec `base path` et `include path`.

3.31.3 Chemin de base et chemins d'inclusion

Le chemin de base et les chemins d'inclusion représentent les répertoires à partir desquels le Host Filesystem Loader chargera les fichiers. Lorsqu'un nom d'unité source est transmis au chargeur, il y ajoute en préambule le chemin de base et effectue une recherche dans le système de fichiers. Si la recherche n'aboutit pas, la même chose est faite avec tous les répertoires de la liste des chemins d'inclusion.

Il est recommandé de définir le chemin de base au répertoire racine de votre projet et d'utiliser les chemins d'inclusion pour spécifier des emplacements supplémentaires qui peuvent contenir des bibliothèques dont dépend votre projet. Cela vous permet d'importer à partir de ces bibliothèques d'une manière uniforme, peu importe où elles sont situées dans le système de fichiers par rapport à votre projet. Par exemple, si vous utilisez npm pour installer des paquets et que votre contrat importe `@openzeppelin/contracts/utils/Strings.sol`, vous pouvez utiliser ces options pour indiquer au compilateur que la bibliothèque peut être trouvée dans l'un des répertoires de paquets npm :

```
solc contract.sol \
  --base-path . \
  --include-path node_modules/ \
  --include-path /usr/local/lib/node_modules/
```

Votre contrat sera compilé (avec les mêmes métadonnées exactes), peu importe que vous installiez la bibliothèque dans le répertoire du paquetage local ou global ou même directement sous la racine de votre projet.

Par défaut, le chemin de base est vide, ce qui laisse le nom de l'unité source inchangé. Lorsque le nom de l'unité source est un chemin relatif, cela a pour conséquence que le fichier est recherché dans le répertoire à partir duquel le compilateur a été invoqué. C'est aussi la seule valeur qui permet d'interpréter les chemins absolus dans les noms d'unités sources interprétés comme des chemins absolus sur le disque. Si le chemin de base est lui-même relatif, il est interprété comme relatif au répertoire de travail actuel du compilateur.

Note : Les chemins d'inclusion ne peuvent pas avoir de valeurs vides et doivent être utilisés avec un chemin de base non vide.

Note : Les chemins d'inclusion et de base peuvent se chevaucher tant que cela ne rend pas la résolution des importations ambiguë. Par exemple, vous pouvez spécifier un répertoire à l'intérieur du chemin de base comme un répertoire d'inclusion ou avoir un répertoire d'inclusion qui est un sous-répertoire d'un autre répertoire include. Le compilateur n'émettra une erreur que si le nom de l'unité source transmis au Host Filesystem Loader représente un chemin existant lorsqu'il est combiné avec plusieurs chemins d'inclusion ou un chemin d'inclusion et un chemin de base.

Normalisation et suppression des chemins CLI

Sur la ligne de commande, le compilateur se comporte comme vous le feriez avec n'importe quel autre programme : Il accepte les chemins dans un format natif de la plate-forme et les chemins relatifs sont relatifs au répertoire de travail actuel. Les noms d'unités sources attribués aux fichiers dont les chemins sont spécifiés sur la ligne de commande, cependant, ne doivent pas changer simplement parce que le projet est compilé sur une plate-forme différente ou parce que le compilateur a été invoqué à partir d'un répertoire différent. Pour cela, les chemins des fichiers sources provenant de la ligne de commande doivent être convertis en une forme canonique et, si possible, rendus relatifs au chemin de base ou à l'un des chemins d'inclusion.

Les règles de normalisation sont les suivantes :

- Si un chemin est relatif, il est rendu absolu en y ajoutant le répertoire de travail actuel.
- Les segments internes `.` et `..` sont réduits.
- Les séparateurs de chemin spécifiques à la plate-forme sont remplacés par des barres obliques.

- Les séquences de plusieurs séparateurs de chemin consécutifs sont écrasées en un seul séparateur (à moins qu’il s’agisse des barres obliques de tête d’un chemin [UNC](#)).
- Si le chemin comprend un nom de racine (par exemple une lettre de lecteur sous Windows) et que la racine est la même que la racine du répertoire de travail actuel, la racine est remplacée par /.
- Les liens symboliques dans le chemin ne sont **pas** résolus.
 - La seule exception est le chemin d’accès au répertoire de travail actuel ajouté aux chemins relatifs dans le but de les rendre absolus. Sur certaines plateformes, le répertoire de travail est toujours signalé avec les liens symboliques résolus, donc pour des raisons de cohérence, le compilateur les résout partout.
- La casse originale du chemin est préservée même si le système de fichiers est insensible à la casse mais [case-preserving](#) et que la casse réelle sur le disque est différent.

Note : Il existe des situations où les chemins ne peuvent pas être rendus indépendants de la plate-forme. Par exemple, sous Windows, le compilateur peut éviter d’utiliser les lettres de lecteur en se référant au répertoire racine du lecteur actuel comme / mais les lettres de lecteur sont toujours nécessaires pour les chemins menant à d’autres lecteurs. Vous pouvez éviter de telles situations en vous assurant que tous les fichiers sont disponibles dans une seule arborescence de répertoire sur le même lecteur.

Après la normalisation, le compilateur essaie de rendre le chemin du fichier source relatif. Il essaie d’abord le chemin de base, puis les chemins d’inclusion dans l’ordre où ils ont été donnés. Si le chemin de base est vide ou non spécifié, il est traité comme s’il était égal au chemin du répertoire de travail actuel (avec tous les liens symboliques résolus). Le résultat est accepté seulement si le chemin du répertoire normalisé est le préfixe exact du chemin du fichier normalisé. Sinon, le chemin du fichier reste absolu. Cela rend la conversion non ambiguë et assure que le chemin relatif ne commence pas par ../. Le chemin de fichier résultant devient le nom de l’unité source.

Note : Le chemin relatif produit par le dépouillement doit rester unique dans le chemin de base et les chemins d’inclusion. Par exemple, le compilateur émettra une erreur pour la commande suivante si à la fois /projet/contract.sol et /lib/contract.sol existent :

```
solc /projet/contract.sol --base-path /project --include-path /lib
```

Note : Avant la version 0.8.8, la suppression des chemins d’accès de l’interface CLI n’était pas effectuée et la seule normalisation appliquée était la conversion des séparateurs de chemin. Lorsque vous travaillez avec des versions plus anciennes du compilateur, il est recommandé d’invoquer le compilateur à partir du chemin de base et de n’utiliser que des chemins relatifs sur la ligne de commande.

3.31.4 Chemins autorisés

Par mesure de sécurité, le Host Filesystem Loader refusera de charger des fichiers en dehors de quelques emplacements qui sont considérés comme sûrs par défaut :

- En dehors du mode JSON standard :
 - Les répertoires contenant les fichiers d’entrée listés sur la ligne de commande.
 - Les répertoires utilisés comme cibles [remapping](#). Si la cible n’est pas un répertoire (c’est-à-dire ne se termine pas par /, /. ou /. .), le répertoire contenant la cible est utilisé à la place.
 - Chemin de base et chemins d’inclusion.
- En mode JSON standard :
 - Le chemin de base et les chemins d’inclusion.

Des répertoires supplémentaires peuvent être mis sur une liste blanche en utilisant l’option `--allow-paths`. L’option accepte une liste de chemins séparés par des virgules :

```
cd /home/user/project/
solc token/contract.sol \
  lib/util.sol=libs/util.sol \
  --base-path=token/ \
  --include-path=/lib/ \
  --allow-paths=../utils/,/tmp/libraries
```

Lorsque le compilateur est invoqué avec la commande indiquée ci-dessus, le Host Filesystem Loader permet d'importer des fichiers depuis les répertoires suivants :

- `/home/user/project/token/` (parce que `token/` contient le fichier d'entrée et aussi parce qu'il s'agit du chemin de base),
- `/lib/` (parce que `/lib/` est un des chemins d'inclusion),
- `/home/user/project/libs/`` (parce que `libs/`` est un répertoire contenant une cible de remappage),
- `/home/user/utils/` (à cause de `../utils/`` passé à `--allow-paths ``),
- `/tmp/libraries/` (à cause de `/tmp/libraries` passé dans `--allow-paths ``),

Note : Le répertoire de travail du compilateur est l'un des chemins autorisés par défaut uniquement s'il se trouve être le chemin de base (ou le chemin de base n'est pas spécifié ou a une valeur vide).

Note : Le compilateur ne vérifie pas si les chemins autorisés existent réellement et s'ils sont des répertoires. Les chemins inexistantes ou vides sont simplement ignorés. Si un chemin autorisé correspond à un fichier plutôt qu'à un répertoire, le fichier est également considéré comme étant sur la liste blanche.

Note : Les chemins autorisés sont sensibles à la casse, même si le système de fichiers ne l'est pas. La casse doit correspondre exactement à celle utilisée dans vos importations. Par exemple, `--allow-paths tokens` ne correspondra pas à `import "Tokens/IERC20.sol"`.

Avertissement : Les fichiers et répertoires accessibles uniquement par des liens symboliques à partir de répertoires autorisés ne sont pas automatiquement sur la liste blanche. Par exemple, si `token/contract.sol` dans l'exemple ci-dessus était en fait un lien symbolique pointant sur `/etc/passwd`, le compilateur refuserait de le charger à moins que `/etc/` ne fasse aussi partie des chemins autorisés.

3.31.5 Remappage des importations

Le remappage des importations vous permet de rediriger les importations vers un emplacement différent dans le système de fichiers virtuel. Le mécanisme fonctionne en modifiant la traduction entre les chemins d'importation et les noms d'unités sources. Par exemple, vous pouvez configurer un remappage de sorte que toute importation à partir du répertoire virtuel `github.com/ethereum/dapp-bin/library/` soit considérée comme une importation depuis `dapp-bin/library/`.

Vous pouvez limiter la portée d'un remappage en spécifiant un *contexte*. Cela permet de créer des remappages qui ne s'appliquent qu'aux importations situées dans une bibliothèque spécifique ou un fichier spécifique. Sans contexte, un remappage est appliqué à chaque `import` correspondant dans tous les fichiers du système de fichiers virtuel.

Les remappages d'importation ont la forme de `context:prefix=target` :

- `context` doit correspondre au début du nom de l'unité source du fichier contenant l'importation.
- `prefix` doit correspondre au début du nom de l'unité source résultant de l'importation.
- `target` est la valeur avec laquelle le préfixe est remplacé.

Par exemple, si vous clonez <https://github.com/ethereum/dapp-bin/> localement dans `/projet/dapp-bin` et que vous exécutez le compilateur avec :

```
solc github.com/ethereum/dapp-bin/=dapp-bin/ --base-path /project source.sol
```

vous pouvez utiliser ce qui suit dans votre fichier source :

```
import "github.com/ethereum/dapp-bin/library/math.sol"; // source unit name: dapp-bin/  
↳ library/math.sol
```

Le compilateur cherchera le fichier dans le VFS sous `dapp-bin/library/math.sol`. Si le fichier n'est pas disponible à cet endroit, le nom de l'unité source sera transmis au Host Filesystem Loader, qui cherchera alors dans `/project/dapp-bin/library/iterable_mapping.sol`.

Avertissement : Les informations sur les remappages sont stockées dans les métadonnées du contrat. Comme le binaire produit par le compilateur contient un hachage des métadonnées, toute modification des réaffectations se traduira par un bytecode différent.

C'est pourquoi vous devez veiller à ne pas inclure d'informations locales dans les cibles de remappage. Par exemple, si votre bibliothèque est située dans le répertoire `/home/user/packages/mymath/math.sol`, un remappage comme `@math/=/home/user/packages/mymath/` aurait pour conséquence d'inclure votre répertoire personnel dans les métadonnées. Pour être en mesure de reproduire le même bytecode avec un tel remappage sur une autre machine, vous devrez recréer des parties de votre structure de répertoire locale dans le VFS et (si vous utilisez le Host Filesystem Loader) également dans le système de fichiers de l'hôte.

Pour éviter que votre structure de répertoire locale ne soit intégrée dans les métadonnées, il est recommandé de désigner les répertoires contenant les bibliothèques comme des *chemins d'inclusion*. Par exemple, dans l'exemple ci-dessus, `--include-path /home/user/packages/` vous permettrait d'utiliser les importations commençant par `mymath/`. Contrairement au remappage, l'option seule ne fera pas apparaître `mymath` comme `@math`, mais cela peut être réalisé en créant un lien symbolique ou en renommant le sous-répertoire du paquetage.

Pour un exemple plus complexe, supposons que vous dépendez d'un module qui utilise une ancienne version de `dapp-bin` que vous avez extraite vers `/project/dapp-bin_old`, alors vous pouvez exécuter :

```
solc module1:github.com/ethereum/dapp-bin/=dapp-bin/ \  
  module2:github.com/ethereum/dapp-bin/=dapp-bin_old/ \  
  --base-path /project \  
  source.sol
```

Cela signifie que tous les imports de `module2` pointent vers l'ancienne version mais que les imports de `module1` pointent vers la nouvelle version.

Voici les règles détaillées qui régissent le comportement des remappages :

1. **Les remappages n'affectent que la traduction entre les chemins d'importation et les noms d'unités sources.**

Les noms d'unités sources ajoutés au VFS de toute autre manière ne peuvent pas être remappés. Par exemple, les chemins que vous spécifiez sur la ligne de commande et ceux qui se trouvent dans `sources.urls` en JSON standard ne sont pas affectés.

```
solc /project/=/contracts/ /project/contract.sol # source unit name: /project/  
↳ contract.sol
```

Dans l'exemple ci-dessus, le compilateur chargera le code source à partir de `/project/contract.sol` et le placera sous ce nom exact d'unité source dans le VFS, et non sous `/contract/contract.sol`.

2. Le contexte et le préfixe doivent correspondre aux noms des unités sources, et non aux chemins d'importation.

- Cela signifie que vous ne pouvez pas remapper `./` ou `./` directement puisqu'ils sont remplacés pendant la traduction en nom d'unité source, mais vous pouvez remapper la partie du nom par laquelle ils sont remplacés avec :

```
solc ./=a/ /project/=b/ /project/contract.sol # source unit name: /project/
↳ contract.sol
```

Code source 5 – /project/contract.sol

```
import "./util.sol" as util; // source unit name: b/util.sol
```

- Vous ne pouvez pas remapper le chemin de base ou toute autre partie du chemin qui est seulement ajouté en interne par un rappel d'importation :

```
solc /project/=/contracts/ /project/contract.sol --base-path /project # source_
↳ unit name: contract.sol
```

Code source 6 – /project/contract.sol

```
import "util.sol" as util; // source unit name: util.sol
```

3. La cible est insérée directement dans le nom de l'unité source et ne doit pas nécessairement être un chemin d'accès valide.

- Il peut s'agir de n'importe quoi tant que le callback d'importation peut le gérer. Dans le cas du Host File-system Loader, cela inclut également les chemins relatifs. Lorsque vous utilisez l'interface JavaScript, vous pouvez même utiliser des URL et des identifiants abstraits si votre callback peut les gérer.
- Le remappage se produit après que les importations relatives aient déjà été résolues en noms d'unités sources. Cela signifie que les cibles commençant par `./` et `./` n'ont pas de signification particulière et sont relatives au chemin de base plutôt qu'à l'emplacement du fichier source.
- Les cibles de remappage ne sont pas normalisées, donc `@root/=./a/b//` remappera `@root/contract.sol` en `./a/b/.` vers `./a/b//contract.sol` et non `a/b/contract.sol`.
- Si la cible ne se termine pas par un slash, le compilateur ne l'ajoutera pas automatiquement :

```
solc /project/=/contracts /project/contract.sol # source unit name: /project/
↳ contract.sol
```

Code source 7 – /project/contract.sol

```
import "/project/util.sol" as util; // source unit name: /contractsutil.sol
```

4. Le contexte et le préfixe sont des modèles et les correspondances doivent être exactes.

- `a//b=c` ne correspondra pas à `a/b``.
- Les noms des unités sources ne sont pas normalisés, donc `a/b=c` ne correspondra pas non plus à `a//b`.
- Les parties des noms de fichiers et de répertoires peuvent également correspondre. `/newProject/con:/new=old` correspondra à `/newProject/contract.sol` et le remappera à `oldProject/contrat.sol`.

5. Un remappage au maximum est appliqué à une seule importation.

- Si plusieurs réaffectations correspondent au même nom d'unité source, celle dont le préfixe est le plus long est choisi.
- Si les préfixes sont identiques, celui qui est spécifié en dernier l'emporte.
- Les réaffectations ne fonctionnent pas sur d'autres réaffectations. Par exemple, `a=b b=c c=d` n'aura pas pour résultat de transformer `a`` en `d`.

6. Le préfixe ne peut être vide, mais le contexte et la cible sont facultatifs.

- Si `target` est une chaîne vide, `prefix` est simplement supprimé des chemins d'importation.

- Un context vide signifie que le remappage s'applique à toutes les importations dans toutes les unités sources.

3.31.6 Utilisation des URLs dans les importations

La plupart des préfixes d'URL tels que `https://` ou `data://` n'ont pas de signification particulière dans les chemins d'importation. La seule exception est `file://` qui est supprimé des noms d'unités sources par le Host Filesystem Loader.

Lorsque vous compilez localement, vous pouvez utiliser le remappage d'importation pour remplacer la partie protocole et domaine par une partie chemin local :

```
solc :https://github.com/ethereum/dapp-bin=/usr/local/dapp-bin contract.sol
```

Notez le premier `:`, qui est nécessaire lorsque le contexte de remappage est vide. Sinon, la partie `https:` serait interprétée par le compilateur comme le contexte.

3.32 Yul

Yul (précédemment aussi appelé JULIA ou IULIA) est un langage intermédiaire qui peut être compilé en bytecode pour différents backends.

Le support d'EVM 1.0, EVM 1.5 et Ewasm est prévu, et il est conçu pour être un dénominateur commun utilisable pour ces trois plateformes. Il peut déjà être utilisé en mode autonome et pour « l'assemblage en ligne » dans Solidity et il existe une implémentation expérimentale du compilateur Solidity qui utilise Yul comme langage intermédiaire. Le Yul est une bonne cible pour étapes d'optimisation de haut niveau qui peuvent bénéficier à toutes les plates-formes cibles de manière égale.

3.32.1 Motivation et description de haut niveau

La conception de Yul vise à atteindre plusieurs objectifs :

1. Les programmes écrits en Yul doivent être lisibles, même si le code est généré par un compilateur de Solidity ou d'un autre langage de haut niveau.
2. Le flux de contrôle doit être facile à comprendre pour faciliter l'inspection manuelle, la vérification formelle et l'optimisation.
3. La traduction de Yul en bytecode doit être aussi simple que possible.
4. Yul doit être adapté à l'optimisation de l'ensemble du programme.

Afin d'atteindre le premier et le second objectif, Yul fournit des constructions de haut niveau comme les boucles `for`, les instructions `if` et `switch` et les appels de fonctions. Ces éléments devraient être suffisantes pour représenter adéquatement le flux de contrôle des programmes assembleurs. Par conséquent, il n'y a pas d'instructions explicites pour `SWAP`, `DUP`, `JUMPDEST`, `JUMP` et `JUMPI` sont fournis, parce que les deux premiers obscurcissent le flux de données et les deux derniers obfusquent le flux de contrôle. De plus, les instructions fonctionnelles de la forme `mul (add(x, y), 7)` sont préférées aux instructions opcode pures telles que `7 y x add mul` car dans la première forme, il est beaucoup plus facile de voir quel opérande est utilisé pour quel opcode.

Même s'il a été conçu pour les machines à pile, Yul n'expose pas la complexité de la pile elle-même. Le programmeur ou l'auditeur ne devrait pas avoir à se soucier de la pile.

Le troisième objectif est atteint en compilant les constructions de niveau supérieur en bytecode de manière très régulière. La seule opération non-locale effectuée par l'assembleur est la recherche de noms d'identifiants définis par l'utilisateur (fonctions, variables, ...) et le nettoyage des variables locales de la pile.

Pour éviter les confusions entre des concepts comme les valeurs et les références, Yul est typée statiquement. En même temps, il existe un type par défaut (généralement le mot entier de la machine cible) qui peut toujours être omis pour faciliter la lisibilité.

Pour garder le langage simple et flexible, Yul n'a pas d'opérations, de fonctions ou de types intégrés dans sa forme pure. Ceux-ci sont ajoutés avec leur sémantique lors de la spécification d'un dialecte de Yul, ce qui permet de spécialiser Yul pour répondre aux exigences de différentes plateformes et ensembles de fonctionnalités cibles.

Actuellement, il n'existe qu'un seul dialecte spécifié de Yul. Ce dialecte utilise les opcodes EVM en tant que fonctions intégrées (voir ci-dessous) et ne définit que le type `u256`, qui est le type natif 256-bit de l'EVM. Pour cette raison, nous ne fournissons pas de types dans les exemples ci-dessous.

3.32.2 Exemple simple

Le programme d'exemple suivant est écrit dans le dialecte EVM et calcule l'exponentiation. Il peut être compilé en utilisant `solc --strict-assembly`. Les fonctions intégrées `mul` et `div` calculent le produit et la division, respectivement.

```
{
  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) }
    }
  }
}
```

Le programme d'exemple suivant est écrit dans le dialecte EVM et calcule l'exponentiation. Il peut être compilé en utilisant `solc --strict-assembly`. Les fonctions intégrées `mul` et `div` calculent le produit et la division, respectivement.

```
{
  function power(base, exponent) -> result
  {
    result := 1
    for { let i := 0 } lt(i, exponent) { i := add(i, 1) }
    {
      result := mul(result, base)
    }
  }
}
```

À la fin de la section, une implémentation complète du standard de la norme ERC-20 peut être trouvée.

3.32.3 Utilisation autonome

Vous pouvez utiliser Yul sous sa forme autonome dans le dialecte EVM en utilisant le compilateur Solidity. Il utilisera la notation d'objet *Yul* afin qu'il soit possible de se référer au code comme à des données pour déployer des contrats. Ce mode Yul est disponible pour le compilateur en ligne de commande (utilisez `--strict-assembly`) et pour l'interface *standard-json* :

```
{
  "language": "Yul",
  "sources": { "input.yul": { "content": "{ sstore(0, 1) }" } },
  "settings": {
    "outputSelection": { "**": { "**": ["*"], "" : [ "*" ] } },
    "optimizer": { "enabled": true, "details": { "yul": true } }
  }
}
```

Avvertissement : Yul est en cours de développement actif et la génération de bytecode n'est entièrement implémentée que pour le dialecte EVM de Yul avec EVM 1.0 comme cible.

3.32.4 Description informelle de Yul

Dans ce qui suit, nous allons parler de chaque aspect individuel du langage Yul. Dans les exemples, nous utiliserons le dialecte EVM par défaut.

Syntaxe

Yul analyse les commentaires, les littéraux et les identifiants de la même manière que Solidity, donc vous pouvez par exemple utiliser `//` et `/* */` pour désigner des commentaires. Il y a une exception : Les identificateurs dans Yul peuvent contenir des points : ..

Yul peut spécifier des « objets » qui se composent de code, de données et de sous-objets. Veuillez consulter *Yul Objects* ci-dessous pour plus de détails à ce sujet. Dans cette section, nous ne sommes concernés que par la partie code d'un tel objet. Cette partie code consiste toujours en un bloc délimité par des accolades. La plupart des outils supportent la spécification d'un seul bloc de code où un objet est attendu.

Inside a code block, the following elements can be used (see the later sections for more details) :

- des littéraux, par exemple `0x123, 42` ou `"abc"` (chaînes de caractères jusqu'à 32 caractères)
- les appels à des fonctions intégrées, par exemple `add(1, mload(0))`
- les déclarations de variables, par exemple `let x := 7`, `« let x := add(y, 3) »` ou `let x` (la valeur initiale de 0 est attribuée)
- des identificateurs (variables), par exemple `add(3, x)`
- des affectations, par exemple `x := add(y, 3)`
- les blocs à l'intérieur desquels les variables locales ont une portée, par exemple `{ let x := 3 { let y := add(x, 1) } }`
- les instructions if, par exemple `if lt(a, b) { sstore(0, 1) }`
- les instructions switch, par exemple `: switch mload(0) case 0 { revert() } default { mstore(0, 1) }`
- Boucles for, par exemple `: for { let i := 0 } lt(i, 10) { i := add(i, 1) } { mstore(i, 7) }`
- des définitions de fonctions, par exemple `: fonction f(a, b) -> c { c := add(a, b) }`

Plusieurs éléments syntaxiques peuvent se succéder en étant simplement séparés par un espace, c'est-à-dire qu'il n'est pas nécessaire de mettre un ; ou un saut de ligne à la fin.

Littéraux

En tant que littéraux, vous pouvez utiliser :

- Des constantes entières en notation décimale ou hexadécimale.
- Des chaînes ASCII (par exemple, "abc"), qui peuvent contenir des échappatoires hexagonales `xNN` et des échappatoires Unicode `uNNNN` où `N` sont des chiffres hexadécimaux.
- Chaînes hexadécimales (par exemple, `hex "616263"`).

Dans le dialecte EVM de Yul, les littéraux représentent des mots de 256 bits comme suit :

- Les constantes décimales ou hexadécimales doivent être inférieures à 2^{256} . Elles représentent le mot de 256 bits avec cette valeur comme un entier non signé en codage big endian.
- Une chaîne de caractères ASCII est d'abord vue comme une séquence d'octets, en voyant un caractère ASCII non échappé comme un seul octet dont la valeur est le code ASCII, un caractère d'échappement `\xNN` comme un octet unique ayant cette valeur, et un échappement `uNNNN` comme la séquence d'octets UTF-8 pour ce point de code. La séquence d'octets ne doit pas dépasser 32 octets. La séquence d'octets est complétée par des zéros sur la droite pour atteindre une longueur de 32 octets ; En d'autres termes, la chaîne est stockée alignée à gauche. La séquence d'octets remplie représente un mot de 256 bits dont les 8 bits les plus significatifs sont les uns du premier octet, c'est-à-dire que les octets sont interprétés sous la forme big endian.
- Une chaîne hexadécimale est d'abord considérée comme une séquence d'octets, en regardant chaque paire de chiffres hexadécimaux contigus comme un octet. La séquence d'octets ne doit pas dépasser 32 octets (c'est-à-dire 64 chiffres hexadécimaux) et est traitée comme ci-dessus.

Lors de la compilation pour l'EVM, ceci sera traduit en une instruction `PUSHi` appropriée. Dans l'exemple suivant, 3 et 2 sont additionnés, ce qui donne 5. avec la chaîne « abc » est calculée. La valeur finale est affectée à une variable locale appelée `x`.

La limite de 32 octets ci-dessus ne s'applique pas aux chaînes de caractères passées aux fonctions intégrées qui requièrent des arguments littéraux (par exemple, `setimmutable` ou `loadimmutable`). Ces chaînes de caractères ne se retrouvent jamais dans le bytecode généré.

```
let x := and("abc", add(3, 2))
```

À moins qu'il ne s'agisse du type par défaut, le type d'un littéral doit être spécifié après un deux-points :

```
// Cela ne compilera pas (les types u32 et u256 ne sont pas encore implémentés).
let x := and("abc":u32, add(3:u256, 2:u256))
```

Appels de fonction

Les fonctions intégrées et les fonctions définies par l'utilisateur (voir ci-dessous) peuvent être appelées de la même manière que dans l'exemple précédent. Si la fonction renvoie une seule valeur, elle peut être directement utilisée à l'intérieur d'une expression. Si elle renvoie plusieurs valeurs, elles doivent être assignées à des variables locales.

```
function f(x, y) -> a, b { /* ... */ }
mstore(0x80, add(mload(0x80), 3))
// Ici, la fonction définie par l'utilisateur `f` renvoie deux valeurs.
let x, y := f(1, mload(0))
```

Pour les fonctions intégrées de l'EVM, les expressions fonctionnelles peuvent être directement traduites en un flux d'opcodes : Il suffit de lire l'expression de droite à gauche pour obtenir les opcodes. Dans le cas de la première ligne de l'exemple, il s'agit de `PUSH1 3 PUSH1 0x80 MLOAD ADD PUSH1 0x80 MSTORE`.

Pour les appels aux fonctions définies par l'utilisateur, les arguments sont également placés sur la pile de droite à gauche et c'est dans cet ordre dans lequel les listes d'arguments sont évaluées. Les valeurs de retour, par contre, sont attendues sur la pile de gauche à droite, c'est-à-dire que dans cet exemple, `y` est en haut de la pile et `x` est en dessous.

Déclarations de variables

Vous pouvez utiliser le mot-clé `let` pour déclarer des variables. Une variable n'est visible qu'à l'intérieur du bloc `{ . . }` dans lequel elle a été définie. Lors de la compilation vers l'EVM, un nouvel emplacement de pile est créé, qui est réservé pour la variable et est automatiquement supprimé lorsque la fin du bloc est atteinte. Vous pouvez fournir une valeur initiale pour la variable. Si vous ne fournissez pas de valeur, la variable sera initialisée à zéro.

Comme les variables sont stockées sur la pile, elles n'ont pas d'influence directe sur la mémoire ou le stockage, mais elles peuvent être utilisées comme pointeurs vers des emplacements de mémoire ou de stockage dans les fonctions intégrées `mstore`, `mload`, `sstore` et `sload`. De futurs dialectes pourraient introduire des types spécifiques pour ces pointeurs.

Quand une variable est référencée, sa valeur actuelle est copiée. Pour l'EVM, cela se traduit par une instruction `DUP`.

```
{
  let zero := 0
  let v := calldataload(zero)
  {
    let y := add(sload(v), 1)
    v := y
  } // y est "désalloué" ici
  sstore(v, zero)
} // v et zéro sont "désalloués" ici
```

Si la variable déclarée doit avoir un type différent du type par défaut, vous l'indiquez en suivant les deux points. Vous pouvez également déclarer plusieurs variables dans une déclaration lorsque vous effectuez une assignation à partir d'un appel de fonction qui renvoie plusieurs valeurs.

```
// Cela ne compilera pas (les types u32 et u256 ne sont pas encore implémentés).
{
  let zero:u32 := 0:u32
  let v:u256, t:u32 := f()
  let x, y := g()
}
```

Selon les paramètres de l'optimiseur, le compilateur peut libérer les emplacements de pile déjà après que la variable ait été utilisée pour la dernière fois, même si elle est encore dans la portée.

Affectations

Les variables peuvent être assignées après leur définition en utilisant l'opérateur `:=`. Il est possible d'affecter plusieurs variables en même temps. Pour cela, le nombre et le type des valeurs doivent correspondre. Si vous voulez affecter les valeurs renvoyées par une fonction qui a plusieurs paramètres de retour, vous devez fournir plusieurs variables. La même variable ne peut pas apparaître plusieurs fois dans la partie gauche d'une affectation, par exemple `x, x := f()` n'est pas valide.

```
let v := 0
// réassignation de v
v := 2
let t := add(v, 2)
function f() -> a, b { }
// assigner des valeurs multiples
v, t := f()
```

If

L'instruction `if` peut être utilisée pour exécuter du code de manière conditionnelle. Aucun bloc « `else` » ne peut être défini. Envisagez d'utiliser « `switch` » à la place (voir ci-dessous) si vous avez besoin de plusieurs alternatives.

```
if lt(calldatasize(), 4) { revert(0, 0) }
```

Les accolades pour le corps sont nécessaires.

Interrupteur

Vous pouvez utiliser une instruction `switch` comme une version étendue de l'instruction `if`. Elle prend la valeur d'une expression et la compare à plusieurs constantes littérales. La branche correspondant à la constante correspondante est prise. Contrairement aux autres langages de programmation, le flux de contrôle ne se poursuit pas d'un cas à l'autre. Il peut y avoir un cas de repli ou par défaut appelé `default` qui est pris si aucune des constantes littérales ne correspond.

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

La liste des cas n'est pas entourée d'accolades, mais le corps d'un cas en a besoin.

Boucles

Yul supporte les boucles `for` qui consistent en un en-tête contenant une partie d'initialisation, une condition, une partie de post-itération et un corps. La condition doit être une expression, tandis que les trois autres sont des blocs. Si la partie d'initialisation déclare des variables au niveau supérieur, la portée de ces variables s'étend à toutes les autres parties de la boucle.

Les instructions `break` et `continue` peuvent être utilisées dans le corps de la boucle pour en sortir ou passer à la partie suivante, respectivement.

L'exemple suivant calcule la somme d'une zone en mémoire.

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

Les boucles `for` peuvent également être utilisées en remplacement des boucles `while` : Il suffit de laisser les parties d'initialisation et de post-itération vides.

```

{
  let x := 0
  let i := 0
  for { } lt(i, 0x100) { } { // while(i < 0x100)
    x := add(x, mload(i))
    i := add(i, 0x20)
  }
}

```

Déclarations de fonctions

Yul permet de définir des fonctions. Celles-ci ne doivent pas être confondues avec les fonctions dans Solidity, car elles ne font jamais partie d'une interface externe d'un contrat et font partie d'un espace de noms distinct de celui des fonctions Solidity.

Pour l'EVM, les fonctions Yul prennent leurs arguments (et un PC de retour) de la pile et mettent également les résultats sur la pile. Les fonctions définies par l'utilisateur et les fonctions intégrées sont appelées exactement de la même manière.

Les fonctions peuvent être définies n'importe où et sont visibles dans le bloc dans lequel elles sont déclarées. À l'intérieur d'une fonction, vous ne pouvez pas accéder aux variables locales définies en dehors de cette fonction.

Les fonctions déclarent des paramètres et renvoient des variables, comme dans Solidity. Pour retourner une valeur, vous l'affectez à la ou aux variables de retour.

Si vous appelez une fonction qui renvoie plusieurs valeurs, vous devez les affecter à plusieurs variables en utilisant `a, b := f(x)` ou `let a, b := f(x)`.

L'instruction `leave` peut être utilisée pour quitter la fonction en cours. Elle fonctionne comme l'instruction `return` dans d'autres langages, mais elle ne prend pas de valeur à retourner, elle quitte juste la fonction et la fonction retournera les valeurs qui sont actuellement assignées à la ou aux variables de retour.

Notez que le dialecte EVM a une fonction intégrée appelée `return` qui quitte le contexte d'exécution complet (appel de message interne) et non pas seulement la fonction yul courante.

L'exemple suivant implémente la fonction puissance par carré et multiplication.

```

{
  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) }
    }
  }
}

```

3.32.5 Spécification de Yul

Ce chapitre décrit le code Yul de manière formelle. Le code Yul est généralement placé à l'intérieur d'objets Yul, qui sont expliqués dans leur propre chapitre.

```

Block = '{' Statement* '}'
Statement =
    Block |
    FunctionDefinition |
    VariableDeclaration |
    Assignment |
    If |
    Expression |
    Switch |
    ForLoop |
    BreakContinue |
    Leave
FunctionDefinition =
    'function' Identifier '(' TypedIdentifierList? ')'
    ( '->' TypedIdentifierList )? Block
VariableDeclaration =
    'let' TypedIdentifierList ( ':' Expression )?
Assignment =
    IdentifierList ':' Expression
Expression =
    FunctionCall | Identifier | Literal
If =
    'if' Expression Block
Switch =
    'switch' Expression ( Case+ Default? | Default )
Case =
    'case' Literal Block
Default =
    'default' Block
ForLoop =
    'for' Block Expression Block Block
BreakContinue =
    'break' | 'continue'
Leave = 'leave'
FunctionCall =
    Identifier '(' ( Expression ( ',' Expression )* )? ')'
Identifier = [a-zA-Z_] [a-zA-Z_0-9.]*
IdentifierList = Identifier ( ',' Identifier)*
TypeName = Identifier
TypedIdentifierList = Identifier ( ':' TypeName )? ( ',' Identifier ( ':' TypeName )? )*
Literal =
    (NumberLiteral | StringLiteral | TrueLiteral | FalseLiteral) ( ':' TypeName )?
NumberLiteral = HexNumber | DecimalNumber
StringLiteral = '"' ([^"\\r\n\\] | '\\'.)* '"'
TrueLiteral = 'true'
FalseLiteral = 'false'
HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+

```

Restrictions sur la grammaire

En dehors de celles qui sont directement imposées par la grammaire, les restrictions suivantes s'appliquent :

Les commutateurs doivent avoir au moins un cas (y compris le cas par défaut). Toutes les valeurs de cas doivent avoir le même type et des valeurs distinctes. Si toutes les valeurs possibles du type d'expression sont couvertes, un cas par défaut n'est pas autorisé (par exemple, un commutateur avec une expression `bool` qui a à la fois un cas vrai et un cas faux ne permet pas de cas par défaut).

Chaque expression est évaluée à zéro ou plusieurs valeurs. Identificateurs et littéraux évaluent à exactement une valeur et les appels de fonction sont évalués à un nombre de valeurs égal au nombre de variables de retour de la fonction appelée.

Dans les déclarations de variables et les affectations, l'expression de droite (si elle est présente) doit être évaluée sur un nombre de valeurs égal au nombre de variables du côté gauche. C'est la seule situation dans laquelle une expression évaluant à plus d'une valeur est autorisée. Le même nom de variable ne peut pas apparaître plus d'une fois dans la partie gauche d'une affectation ou d'une déclaration de variable.

Les expressions qui sont également des instructions (c'est-à-dire au niveau du bloc) doivent être évaluées à des valeurs nulles.

Dans toutes les autres situations, les expressions doivent être évaluées à une seule valeur.

Une instruction `continue` ou `break` ne peut être utilisée que dans le corps d'une boucle `for`, comme suit. Considérez la boucle la plus interne qui contient l'instruction. La boucle et l'instruction doivent être dans la même fonction, ou les deux doivent être au niveau supérieur. L'instruction doit se trouver dans le bloc de corps de la boucle ; elle ne peut pas se trouver dans le bloc d'initialisation ou le bloc de mise à jour de la boucle. Il est important de souligner que cette restriction ne s'applique que à la boucle la plus interne qui contient l'instruction `continue` ou `break` : cette boucle la plus interne, et donc l'instruction `continue` ou `break`, peut apparaître n'importe où dans une boucle externe, éventuellement dans le bloc d'initialisation ou le bloc de mise à jour d'une boucle externe. Par exemple, ce qui suit est légal, car l'instruction `break` apparaît dans le bloc `body` de la boucle interne, bien qu'elle apparaisse également dans le bloc de mise à jour de la boucle externe :

```
for {} true { for {} true {} { break } }  
{  
}
```

La partie condition de la boucle `for` doit être évaluée à une seule valeur.

L'instruction `leave` ne peut être utilisée qu'à l'intérieur d'une fonction.

Les fonctions ne peuvent pas être définies n'importe où dans les blocs d'init de la boucle `for`.

Les littéraux ne peuvent pas être plus grands que leur type. Le plus grand type défini est d'une largeur de 256 bits.

Pendant les affectations et les appels de fonction, les types des valeurs respectives doivent correspondre. Il n'y a pas de conversion de type implicite. La conversion de type en général ne peut être réalisée que si le dialecte fournit une fonction intégrée appropriée qui prend une valeur d'un type et retourne une valeur d'un type différent.

Règles de scoping

Dans Yul, les champs d'application sont liés aux blocs (à l'exception des fonctions et de la boucle `for` comme expliqué ci-dessous) et toutes les déclarations (`FunctionDefinition`, `VariableDeclaration`) introduisent de nouveaux identifiants dans ces champs d'application.

Les identificateurs sont visibles dans le bloc dans lequel ils sont définis (y compris tous les sous-noeuds et sous-blocs) : Les fonctions sont visibles dans tout le bloc (même avant leurs définitions) alors que les variables ne sont visibles qu'à partir de la déclaration qui suit la `VariableDeclaration`.

En particulier, variables ne peuvent pas être référencées dans la partie droite de leur propre déclaration de variable. Les fonctions peuvent être référencées dès avant leur déclaration (si elles sont visibles).

En tant qu'exception à la règle générale de délimitation, la portée de la partie « `init` » de la boucle `for` (le premier bloc) s'étend à toutes les autres parties de la boucle `for`. Cela signifie que les variables (et les fonctions) déclarées dans la partie `init` (mais pas dans un bloc à l'intérieur de la partie `init`) sont visibles dans toutes les autres parties de la boucle `for`.

Les identificateurs déclarés dans les autres parties de la boucle `for` respectent les règles syntaxiques de scoping.

Cela signifie qu'une boucle `for` de la forme `for { I... } C { P... } { B... }` est équivalent à `I... for { } C { P... } { B... }`.

Les paramètres et les paramètres de retour des fonctions sont visibles dans le corps de la fonction et leurs noms doivent être distincts.

À l'intérieur des fonctions, il n'est pas possible de référencer une variable qui a été déclarée en dehors de cette fonction.

L'ombrage est interdit, c'est-à-dire que vous ne pouvez pas déclarer un identificateur à un endroit où un autre identificateur portant le même nom est également visible, même s'il n'est pas possible de le référencer parce qu'il a été déclaré en dehors de la fonction courante.

Spécification formelle

Nous spécifions formellement Yul en fournissant une fonction d'évaluation `E` surchargée sur les différents noeuds de l'AST. Comme les fonctions intégrées peuvent avoir des effets secondaires, `E` prend deux objets d'état et le noeud AST et retourne deux nouveaux objets d'état et un nombre variable d'autres valeurs. Les deux objets d'état sont l'objet d'état global (qui, dans le contexte de l'EVM, est la mémoire, le stockage et l'état de la blockchain) et l'objet d'état local (l'état des variables locales, c'est-à-dire un segment de la pile dans l'EVM).

Si le noeud AST est une déclaration, `E` retourne les deux objets d'état et un « mode », qui est utilisé pour les instructions `break`, `continue` `` et `` `leave`. Si le noeud de l'AST est une expression, `E` retourne les deux objets d'état et autant de valeurs que l'expression en évalue.

La nature exacte de l'état global n'est pas spécifiée dans cette description de haut niveau. L'état local `L` est une correspondance entre les identifiants `i` et les valeurs `v`, noté `L[i] = v`.

Pour un identifiant `v`, on note `$v` le nom de l'identifiant.

Nous utiliserons une notation de déstructuration pour les noeuds de l'AST.

```
E(G, L, <{St1, ..., Stn}>: Block) =
  let G1, L1, mode = E(G, L, St1, ..., Stn)
  let L2 be a restriction of L1 to the identifiers of L
  G1, L2, mode
E(G, L, St1, ..., Stn: Statement) =
  if n is zero:
    G, L, regular
  else:
```

(suite sur la page suivante)

```

    let G1, L1, mode = E(G, L, St1)
    if mode is regular then
        E(G1, L1, St2, ..., Stn)
    otherwise
        G1, L1, mode
E(G, L, FunctionDefinition) =
    G, L, regular
E(G, L, <let var_1, ..., var_n := rhs>: VariableDeclaration) =
    E(G, L, <var_1, ..., var_n := rhs>: Assignment)
E(G, L, <let var_1, ..., var_n>: VariableDeclaration) =
    let L1 be a copy of L where L1[$var_i] = 0 for i = 1, ..., n
    G, L1, regular
E(G, L, <var_1, ..., var_n := rhs>: Assignment) =
    let G1, L1, v1, ..., vn = E(G, L, rhs)
    let L2 be a copy of L1 where L2[$var_i] = vi for i = 1, ..., n
    G, L2, regular
E(G, L, <for { i1, ..., in } condition post body>: ForLoop) =
    if n >= 1:
        let G1, L, mode = E(G, L, i1, ..., in)
        // le mode doit être régulier ou congé en raison des restrictions syntaxiques
        if mode is leave then
            G1, L1 restricted to variables of L, leave
        otherwise
            let G2, L2, mode = E(G1, L1, for {} condition post body)
            G2, L2 restricted to variables of L, mode
    else:
        let G1, L1, v = E(G, L, condition)
        if v is false:
            G1, L1, regular
        else:
            let G2, L2, mode = E(G1, L, body)
            if mode is break:
                G2, L2, regular
            otherwise if mode is leave:
                G2, L2, leave
            else:
                G3, L3, mode = E(G2, L2, post)
                if mode is leave:
                    G2, L3, leave
                otherwise
                    E(G3, L3, for {} condition post body)
E(G, L, break: BreakContinue) =
    G, L, break
E(G, L, continue: BreakContinue) =
    G, L, continue
E(G, L, leave: Leave) =
    G, L, leave
E(G, L, <if condition body>: If) =
    let G0, L0, v = E(G, L, condition)
    if v is true:
        E(G0, L0, body)
    else:

```

(suite de la page précédente)

```

    G0, L0, regular
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn>: Switch) =
    E(G, L, switch condition case l1:t1 st1 ... case ln:tn stn default {})
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn default st'>: Switch) =
    let G0, L0, v = E(G, L, condition)
    // i = 1 .. n
    // Evaluer les littéraux, le contexte n'a pas d'importance.
    let _, _, v1 = E(G0, L0, l1)
    ...
    let _, _, vn = E(G0, L0, ln)
    if there exists smallest i such that vi = v:
        E(G0, L0, sti)
    else:
        E(G0, L0, st')

E(G, L, <name>: Identifïer) =
    G, L, L[$name]
E(G, L, <fname(arg1, ..., argn)>: FunctionCall) =
    G1, L1, vn = E(G, L, argn)
    ...
    Gn-1, Ln-1, v2 = E(Gn-2, Ln-2, arg2)
    Gn, Ln, v1 = E(Gn-1, Ln-1, arg1)
    Let <function fname (param1, ..., paramn) -> ret1, ..., retm block>
    be the function of name $fname visible at the point of the call.
    Let L' be a new local state such that
    L'[$parami] = vi and L'[$reti] = 0 for all i.
    Let G'', L'', mode = E(Gn, Ln, block)
    G'', Ln, L''[$ret1], ..., L''[$retm]
E(G, L, l: StringLiteral) = G, L, str(l),
    where str is the string evaluation function,
    which for the EVM dialect is defined in the section 'Literals' above
E(G, L, n: HexNumber) = G, L, hex(n)
    where hex is the hexadecimal evaluation function,
    which turns a sequence of hexadecimal digits into their big endian value
E(G, L, n: DecimalNumber) = G, L, dec(n),
    where dec is the decimal evaluation function,
    which turns a sequence of decimal digits into their big endian value

```

Dialecte EVM

Le dialecte par défaut de Yul est actuellement le dialecte EVM avec une version de l'EVM. Le seul type disponible dans ce dialecte est `u256`, le type natif 256 bits de la machine virtuelle Ethereum. Comme il s'agit du type par défaut de ce dialecte, il peut être omis.

Le tableau suivant liste toutes les fonctions intégrées (selon la version de la machine virtuelle Ethereum) et fournit une brève description de la sémantique de la fonction / opcode. Ce document ne veut pas être une description complète de la machine virtuelle Ethereum. Veuillez vous référer à un autre document si vous êtes intéressé par la sémantique précise.

Les opcodes marqués avec - ne retournent pas de résultat et tous les autres retournent exactement une valeur. Les opcodes marqués par F, H, B, C, I et L sont présents depuis Frontier, Homestead, Byzance, Constantinople, Istanbul ou Londres respectivement.

Dans ce qui suit, `mem[a..b]` signifie les octets de mémoire commençant à la position `a` et allant jusqu'à mais sans inclure la position `b` et `storage[p]` signifie le contenu de la mémoire à l'emplacement `p`.

Puisque Yul gère les variables locales et le flux de contrôle, les opcodes qui interfèrent avec ces fonctionnalités ne sont pas disponibles. Ceci inclut les instructions `dup` et `swap` ainsi que les instructions `jump`, les labels et les instructions `push`.

Instruction			Explication
<code>stop()</code>	-	F	arrête l'exécution, identique à <code>return(0, 0)</code>
<code>add(x, y)</code>		F	$x + y$
<code>sub(x, y)</code>		F	$x - y$
<code>mul(x, y)</code>		F	$x * y$
<code>div(x, y)</code>		F	x / y ou 0 if $y == 0$
<code>sdiv(x, y)</code>		F	x / y , pour les nombres signés en complément à deux, 0 if $y == 0$
<code>mod(x, y)</code>		F	$x \% y$, 0 if $y == 0$
<code>smod(x, y)</code>		F	$x \% y$, pour les nombres signés en complément à deux, 0 if $y == 0$
<code>exp(x, y)</code>		F	x au pouvoir de y
<code>not(x)</code>		F	bitwise « not » of x (chaque bit de x est annulé)
<code>lt(x, y)</code>		F	1 if $x < y$, 0 sinon
<code>gt(x, y)</code>		F	1 if $x > y$, 0 sinon
<code>slt(x, y)</code>		F	1 if $x < y$, 0 sinon, pour les nombres signés en complément à deux
<code>sgt(x, y)</code>		F	1 if $x > y$, 0 sinon, pour les nombres signés en complément à deux
<code>eq(x, y)</code>		F	1 if $x == y$, 0 sinon
<code>iszero(x)</code>		F	1 if $x == 0$, 0 sinon
<code>and(x, y)</code>		F	par bit « and » of x et y
<code>or(x, y)</code>		F	par bit « or » of x et y
<code>xor(x, y)</code>		F	par bit « xor » of x et y
<code>byte(n, x)</code>		F	le n ème octet de x , où l'octet le plus significatif est le 0ème octet
<code>shl(x, y)</code>		C	décalage logique à gauche de y par x bits
<code>shr(x, y)</code>		C	décalage logique vers la droite de y par x bits
<code>sar(x, y)</code>		C	décalage arithmétique signé vers la droite de y par x bits
<code>addmod(x, y, m)</code>		F	$(x + y) \% m$ avec une précision arithmétique arbitraire, 0 if $m == 0$
<code>mulmod(x, y, m)</code>		F	$(x * y) \% m$ avec une précision arithmétique arbitraire, 0 if $m == 0$
<code>signextend(i, x)</code>		F	le signe s'étend du $(i*8+7)$ ème bit en comptant à partir du moins significatif
<code>keccak256(p, n)</code>		F	<code>keccak(mem[p..(p+n)])</code>
<code>pc()</code>		F	position actuelle dans le code
<code>pop(x)</code>	-	F	valeur de rejet x
<code>mload(p)</code>		F	<code>mem[p..(p+32))</code>
<code>mstore(p, v)</code>	-	F	<code>mem[p..(p+32)) := v</code>
<code>mstore8(p, v)</code>	-	F	<code>mem[p] := v & 0xff</code> (ne modifie qu'un seul octet)
<code>sload(p)</code>		F	<code>storage[p]</code>
<code>sstore(p, v)</code>	-	F	<code>storage[p] := v</code>
<code>msize()</code>		F	taille de la mémoire, c.à.d l'indice de mémoire le plus important auquel on accède
<code>gas()</code>		F	gaz encore disponible pour l'exécution
<code>address()</code>		F	adresse du contrat actuel / contexte d'exécution
<code>balance(a)</code>		F	wei balance à l'adresse a
<code>selfbalance()</code>		I	équivalent à <code>balance(address())</code> , mais moins cher
<code>caller()</code>		F	expéditeur de l'appel (à l'exclusion de « <code>delegatecall</code> »)
<code>callvalue()</code>		F	wei envoyé avec l'appel en cours
<code>calldata(p)</code>		F	données d'appel à partir de la position p (32 octets)
<code>calldatasize()</code>		F	taille des données d'appel en octets

Instruction			Explication
calldatacopy(t, f, s)	-	F	copier s octets de calldata à la position f vers mem à la position t
codesize()		F	taille du code du contrat / contexte d'exécution actuel
codecopy(t, f, s)	-	F	copier s octets du code à la position f vers la mémoire à la position t
extcodesize(a)		F	taille du code à l'adresse a
extcodecopy(a, t, f, s)	-	F	comme codecopy(t, f, s) mais prendre le code à l'adresse a
returndatasize()		B	taille de la dernière donnée retournée
returndatacopy(t, f, s)	-	B	copier s octets de returndata à la position f vers mem à la position t
extcodehash(a)		C	code de hachage de l'adresse a
create(v, p, n)		F	créer un nouveau contrat avec le code mem[p...(p+n)] et envoyer v wei et renvoie
create2(v, p, n, s)		C	créer un nouveau contrat avec le code mem[p...(p+n)] à l'adresse keccak256(0xff
call(g, a, v, in, insize, out, outsize)		F	appeler le contrat à l'adresse a avec l'entrée mem[in...(in+insize)] fournir g gaz e
callcode(g, a, v, in, insize, out, outsize)		F	identique à call mais n'utilise que le code de a et reste dans le contexte du contra
delegatecall(g, a, in, insize, out, outsize)		H	identique à callcode mais conserve aussi caller. et callvalue <i>Voir plus</i>
staticcall(g, a, in, insize, out, outsize)		B	identique à call(g, a, 0, in, insize, out, outsize) mais font ne pas a
return(p, s)	-	F	fin de l'exécution, retour des données mem[p...(p+s)]
revert(p, s)	-	B	terminer l'exécution, annuler les changements d'état, retourner les données mem[
selfdestruct(a)	-	F	mettre fin à l'exécution, détruire le contrat en cours et envoyer les fonds à un organ
invalid()	-	F	terminer l'exécution avec une instruction invalide
log0(p, s)	-	F	journal sans sujets et données mem[p...(p+s)]
log1(p, s, t1)	-	F	journal avec sujet t1 et données mem[p...(p+s)]
log2(p, s, t1, t2)	-	F	journal avec les sujets t1, t2 et les données mem[p...(p+s)]
log3(p, s, t1, t2, t3)	-	F	journal avec les sujets t1, t2, t3 et les données mem[p...(p+s)]
log4(p, s, t1, t2, t3, t4)	-	F	journal avec les sujets t1, t2, t3, t4 et les données mem[p...(p+s)]
chainid()		I	ID de la chaîne d'exécution (EIP-1344)
basefee()		L	les frais de base du bloc actuel (EIP-3198 et EIP-1559)
origin()		F	émetteur de la transaction
gasprice()		F	prix du gaz de la transaction
blockhash(b)		F	hash du bloc nr b - uniquement pour les 256 derniers blocs, à l'exclusion du bloc a
coinbase()		F	bénéficiaire actuel de l'exploitation minière
timestamp()		F	Horodatage du bloc actuel en secondes depuis l'époque.
number()		F	numéro du bloc actuel
difficulty()		F	difficulté du bloc actuel
gaslimit()		F	limite de gaz du bloc en cours

Note : Les instructions call* utilisent les paramètres out et outsize pour définir une zone de mémoire où les données de retour ou d'échec sont placées. Cette zone est écrite en fonction du nombre d'octets que le contrat appelé renvoie. S'il retourne plus de données, seuls les premiers octets outsize sont écrits. Vous pouvez accéder au reste des données en utilisant l'opcode returndatacopy. S'il retourne moins de données, les octets restants ne sont pas touchés du tout. Vous devez utiliser l'opcode ``returndatasize`` pour vérifier quelle partie de cette zone mémoire contient les données retournées. Les autres octets conserveront leurs valeurs d'avant l'appel.

Dans certains dialectes internes, il existe des fonctions supplémentaires :

datasize, dataoffset, datacopy

Les fonctions `datasize(x)`, `dataoffset(x)` et `datacopy(t, f, l)` sont utilisées pour accéder à d'autres parties d'un objet Yul.

`datasize` et `dataoffset` ne peuvent prendre que des chaînes de caractères (les noms d'autres objets) comme arguments et renvoient respectivement la taille et le décalage dans la zone de données. Pour l'EVM, la fonction `datacopy` est équivalente à `codecopy`.

setimmutable, loadimmutable

Les fonctions `setimmutable(offset, "name", value)` et `loadimmutable("name")` sont utilisées pour le mécanisme d'immuabilité de Solidity et ne sont pas adaptées à Yul. L'appel à `setimmutable(offset, "name", value)` suppose que le code d'exécution du contrat contenant l'immuable donné a été copié en mémoire à l'offset `offset` et écrira `value` à toutes les positions en mémoire (par rapport à `offset` `) qui contiennent le placeholder généré pour les appels à ``loadimmutable("name")` dans le code d'exécution.

linkersymbol

La fonction `linkersymbol("library_id")` est un espace réservé pour un littéral d'adresse à substituer par l'éditeur de liens. Son premier et seul argument doit être une chaîne de caractères et représente de manière unique l'adresse à insérer. Les identifiants peuvent être arbitraires mais lorsque le compilateur produit du code Yul à partir de sources Solidity, il utilise un nom de bibliothèque qualifié avec le nom de l'unité source qui définit cette bibliothèque. Pour lier le code avec une adresse de bibliothèque particulière, le même identifiant doit être fourni à la commande `--libraries` sur la ligne de commande.

Par exemple, ce code

```
let a := linkersymbol("file.sol:Math")
```

est équivalent à

```
let a := 0x1234567890123456789012345678901234567890
```

lorsque le linker est invoqué avec l'option `--libraries "file.sol:Math=0x1234567890123456789012345678901234567890"`.

Voir *Utilisation du compilateur en ligne de commande* pour plus de détails sur l'éditeur de liens Solidity.

memoryguard

Cette fonction est disponible dans le dialecte EVM avec des objets. L'appelant de `let ptr := memoryguard(size)` (où `size` doit être un nombre littéral) promet qu'il n'utilisera la mémoire que dans l'intervalle `[0, size)` ou dans l'intervalle non borné commençant à `ptr`.

Puisque la présence d'un appel `memoryguard` indique que tous les accès à la mémoire adhère à cette restriction, il permet à l'optimiseur d'effectuer des étapes d'optimisation supplémentaires, par exemple l'évasion de la limite de la pile, qui tente de déplacer les variables de la pile qui seraient autrement inaccessibles à la mémoire.

L'optimiseur Yul promet de n'utiliser que la plage de mémoire `[size, ptr)` pour ses besoins. Si l'optimiseur n'a pas besoin de réserver de la mémoire, il considère que `ptr == size`.

`memoryguard` peut être appelé plusieurs fois, mais doit avoir le même littéral comme argument dans un seul sous-objet Yul. Si au moins un appel `memoryguard` est trouvé dans un sous-objet, les étapes supplémentaires d'optimisation seront exécutées sur lui.

verbatim

L'ensemble des fonctions intégrées `verbatim`. . . vous permet de créer du bytecode pour des opcodes qui ne sont pas connus du compilateur Yul. Il vous permet également de créer séquences de bytecode qui ne seront pas modifiées par l'optimiseur.

Les fonctions sont `verbatim_<n>i_<m>o("<data>", . . .)`, où

- `n` est une valeur décimale comprise entre 0 et 99 qui spécifie le nombre d'emplacements de pile / variables d'entrée
- `m` est une décimale entre 0 et 99 qui spécifie le nombre d'emplacements de pile / variables de sortie
- `data` est une chaîne littérale qui contient la séquence d'octets.

Si vous voulez, par exemple, définir une fonction qui multiplie par deux, sans que l'optimiseur ne touche à la constante deux, vous pouvez utiliser

```
let x := calldataload(0)
let double := verbatim_1i_1o(hex"600202", x)
```

Ce code résultera en un opcode `dup1` pour récupérer `x`. (l'optimiseur pourrait réutiliser directement le résultat de l'opcode `calldataload`, cependant) directement suivi de `600202`. Le code est supposé consommer la valeur copiée de `x` et de produire le résultat en haut de la pile. Le compilateur génère alors du code pour allouer un slot de pile pour `double` et `y` stocker le résultat.

Comme avec tous les opcodes, les arguments sont disposés sur la pile avec l'argument le plus à gauche en haut, tandis que les valeurs de retour sont supposées être disposées de telle sorte que la variable la plus à droite se trouve en haut de la pile.

Puisque `verbatim` peut être utilisé pour générer des opcodes arbitraires ou même des opcodes inconnus du compilateur Solidity, il faut être prudent lorsqu'on utilise `verbatim` avec l'optimiseur. Même lorsque l'optimiseur est désactivé, le générateur de code doit déterminer la disposition de la pile, ce qui signifie que, par exemple, l'utilisation de `verbatim` pour modifier la hauteur de la pile peut conduire à un comportement non défini.

La liste suivante est une liste non exhaustive des restrictions sur le bytecode `verbatim` qui ne sont pas vérifiées par le compilateur. La violation de ces restrictions peut entraîner un comportement non défini.

- Le flux de contrôle ne doit pas sauter dans ou hors des blocs `verbatim`, mais il peut sauter à l'intérieur d'un même bloc `verbatim`
- Le contenu des piles, hormis les paramètres d'entrée et de sortie ne doit pas être accessible
- La différence de hauteur de la pile doit être exactement `m - n` (emplacements de sortie moins emplacements d'entrée)
- Le bytecode `verbatim` ne peut pas faire d'hypothèses sur le bytecode environnant. Tous les paramètres requis doivent être passés en tant que variables de pile

L'optimiseur n'analyse pas le bytecode `verbatim` et suppose toujours qu'il modifie tous les aspects de l'état et peut donc seulement faire que très peu d'optimisations à travers les appels de fonction `verbatim`.

L'optimiseur traite le bytecode `verbatim` comme un bloc de code opaque. Il ne le divise pas, mais peut le déplacer, le dupliquer ou le combiner avec des blocs de bytecode `verbatim` identiques. Si un bloc de bytecode `verbatim` est inaccessible par le flux de contrôle, il peut être supprimé.

Avertissement : Pendant les discussions sur le fait que les améliorations de l'EVM ne risquent pas de casser les contrats intelligents existants, les caractéristiques de `verbatim` ne peuvent pas recevoir la même considération que celles utilisées par le compilateur Solidity lui-même.

Note : Pour éviter toute confusion, tous les identificateurs commençant par la chaîne `verbatim` sont réservés et ne peuvent pas être utilisés pour des identificateurs définis par l'utilisateur.

3.32.6 Spécification de l'objet Yul

Les objets Yul sont utilisés pour regrouper des sections de code et de données nommées. Les fonctions `datasize`, `dataoffset` et `datacopy` peuvent être utilisées pour accéder à ces sections à partir du code. Les chaînes hexadécimales peuvent être utilisées pour spécifier des données en codage hexadécimal, les chaînes régulières en codage natif. Pour le code, `datacopy` accédera à sa représentation binaire assemblée.

```
Object = 'object' StringLiteral '{' Code ( Object | Data )* '}'
Code = 'code' Block
Data = 'data' StringLiteral ( HexLiteral | StringLiteral )
HexLiteral = 'hex' ( '"' ([0-9a-fA-F]{2})* '"' | '\' ' ([0-9a-fA-F]{2})* '\' )
StringLiteral = '"' ([^"r\n\\] | '\\\' .)* '"'
```

Ci-dessus, `Block` fait référence à `Block` dans la grammaire de code Yul expliquée dans le chapitre précédent.

Note : Les objets de données ou les sous-objets dont le nom contient un `.` peuvent être définis mais il n'est pas possible d'y accéder via `datasize`, `dataoffset` ou `datacopy` parce que `.` est utilisé comme un séparateur pour accéder à des objets à l'intérieur d'un autre objet.

Note : L'objet de données appelé `".metadata"` a une signification particulière : Il n'est pas accessible depuis le code et il est toujours ajouté à la toute fin du bytecode, quelle que soit sa position dans l'objet.

D'autres objets de données avec une signification particulière pourraient être ajoutés dans le futur, mais leurs noms commenceront toujours par un `.`

Un exemple d'objet Yul est présenté ci-dessous :

```
// Un contrat consiste en un objet unique avec des sous-objets représentant
// le code à déployer ou d'autres contrats qu'il peut créer.
// Le noeud unique "code" est le code exécutable de l'objet.
// Chaque (autre) objet nommé ou section de données est sérialisé et // rendu
// accessible aux fonctions spéciales intégrées datacopy / dataoffset / datasize.
// L'objet actuel, les sous-objets et les éléments de données à l'intérieur de l
↪'objet actuel
// sont dans le champ d'application.
object "Contract1" {
    // C'est le code du constructeur du contrat.
    code {
        function allocate(size) -> ptr {
            ptr := mload(0x40)
            if iszero(ptr) { ptr := 0x60 }
            mstore(0x40, add(ptr, size))
        }

        // créer d'abord "Contract2"
        let size := datasize("Contract2")
        let offset := allocate(size)
        // Ceci se transformera en codecopie pour EVM
        datacopy(offset, dataoffset("Contract2"), size)
        // le paramètre du constructeur est un seul nombre 0x1234
        mstore(add(offset, size), 0x1234)
```

(suite sur la page suivante)

(suite de la page précédente)

```

    pop(create(offset, add(size, 32), 0))

    // retourne maintenant l'objet d'exécution (le code
    // actuellement exécuté est le code du constructeur)
    size := datasize("runtime")
    offset := allocate(size)
    // Cela se transformera en une copie mémoire->mémoire pour Ewasm et
    // une codecopie pour EVM
    datacopy(offset, dataoffset("runtime"), size)
    return(offset, size)
}

data "Table2" hex"4123"

object "runtime" {
    code {
        function allocate(size) -> ptr {
            ptr := mload(0x40)
            if iszero(ptr) { ptr := 0x60 }
            mstore(0x40, add(ptr, size))
        }

        // code d'exécution

        mstore(0, "Hello, World!")
        return(0, 0x20)
    }
}

// Objet embarqué. Le cas d'utilisation est que l'extérieur est un contrat d
↪'usine,
// et Contract2 est le code à créer par la fabrique
object "Contract2" {
    code {
        // code ici ...
    }

    object "runtime" {
        code {
            // code ici ...
        }
    }

    data "Table1" hex"4123"
}
}

```

3.32.7 Optimiseur de Yul

L'optimiseur Yul fonctionne sur du code Yul et utilise le même langage pour l'entrée, la sortie et les états intermédiaires. Cela permet de faciliter le débogage et la vérification de l'optimiseur.

Veillez vous référer à la documentation générale *optimizer* pour plus de détails sur les différentes étapes d'optimisation et l'utilisation de l'optimiseur.

Si vous voulez utiliser Solidity en mode autonome Yul, vous activez l'optimiseur en utilisant `--optimize` et spécifiez éventuellement le *nombre attendu d'exécutions de contrats* avec `--optimize-runs` :

```
solc --strict-assembly --optimize --optimize-runs 200
```

En mode Solidity, l'optimiseur Yul est activé en même temps que l'optimiseur normal.

Séquence des étapes d'optimisation

Par défaut, l'optimiseur Yul applique sa séquence prédéfinie d'étapes d'optimisation à l'assemblage généré. Vous pouvez remplacer cette séquence et fournir la vôtre en utilisant l'option `--yul-optimizations` :

```
solc --optimize --ir-optimized --yul-optimizations 'dhfoD[xarrscLMcCTU]uljmul'
```

L'ordre des étapes est significatif et affecte la qualité du résultat. De plus, l'application d'une étape peut révéler de nouvelles possibilités d'optimisation pour d'autres qui ont déjà été appliquées. La répétition des étapes est donc souvent bénéfique. En plaçant une partie de la séquence entre crochets ([]), vous indiquez à l'optimiseur d'appliquer cette partie jusqu'à ce qu'elle n'améliore plus la taille de l'assemblage résultant. Vous pouvez utiliser les crochets plusieurs fois dans une même séquence mais ils ne peuvent pas être imbriqués.

Les étapes d'optimisation suivantes sont disponibles :

Abréviation	Nom complet
f	BlockFlattener
l	CircularReferencesPruner
c	CommonSubexpressionEliminator
C	ConditionalSimplifier
U	ConditionalUnsimplifier
n	ControlFlowSimplifier
D	DeadCodeEliminator
v	EquivalentFunctionCombiner
e	ExpressionInliner
j	ExpressionJoiner
s	ExpressionSimplifier
x	ExpressionSplitter
I	ForLoopConditionIntoBody
O	ForLoopConditionOutOfBody
o	ForLoopInitRewriter
i	FullInliner
g	FunctionGrouper
h	FunctionHoister
F	FunctionSpecializer
T	LiteralRematerialiser
L	LoadResolver
M	LoopInvariantCodeMotion

suite sur la page suivante

Tableau 2 – suite de la page précédente

Abréviation	Nom complet
r	RedundantAssignEliminator
R	ReasoningBasedSimplifier - highly experimental
m	Rematerialiser
V	SSAReverser
a	SSATransform
t	StructuralSimplifier
u	UnusedPruner
p	UnusedFunctionParameterPruner
d	VarDeclInitializer

Certaines étapes dépendent de propriétés assurées par `BlockFlattener`, `FunctionGrouper`, `ForLoopInitRewriter`. Pour cette raison, l'optimiseur Yul les applique toujours avant d'appliquer les étapes fournies par l'utilisateur.

Le `ReasoningBasedSimplifier` est une étape de l'optimiseur qui n'est actuellement pas activée dans le jeu d'étapes par défaut. Elle utilise un solveur SMT pour simplifier les expressions arithmétiques et les conditions booléennes. Il n'a pas encore été testé ou validé de manière approfondie et peut produire des résultats non reproductibles, veuillez donc l'utiliser avec précaution !

3.32.8 Exemple complet d'ERC20

```
object "Token" {
  code {
    // Enregistrez le créateur dans l'emplacement zéro.
    sstore(0, caller())

    // Déployer le contrat
    datacopy(0, dataoffset("runtime"), datasize("runtime"))
    return(0, datasize("runtime"))
  }
  object "runtime" {
    code {
      // Protection contre l'envoi d'Ether
      require(iszero(callvalue()))

      // Distributeur
      switch selector()
      case 0x70a08231 /* "balanceOf(address)" */ {
        returnUint(balanceOf(decodeAsAddress(0)))
      }
      case 0x18160ddd /* "totalSupply()" */ {
        returnUint(totalSupply())
      }
      case 0xa9059cbb /* "transfer(address,uint256)" */ {
        transfer(decodeAsAddress(0), decodeAsUint(1))
        returnTrue()
      }
      case 0x23b872dd /* "transferFrom(address,address,uint256)" */ {
        transferFrom(decodeAsAddress(0), decodeAsAddress(1), decodeAsUint(2))
        returnTrue()
      }
    }
  }
}
```

(suite sur la page suivante)

```

}
case 0x095ea7b3 /* "approve(address,uint256)" */ {
    approve(decodeAsAddress(0), decodeAsUint(1))
    returnTrue()
}
case 0xdd62ed3e /* "allowance(address,address)" */ {
    returnUint(allowance(decodeAsAddress(0), decodeAsAddress(1)))
}
case 0x40c10f19 /* "mint(address,uint256)" */ {
    mint(decodeAsAddress(0), decodeAsUint(1))
    returnTrue()
}
default {
    revert(0, 0)
}

function mint(account, amount) {
    require(calledByOwner())

    mintTokens(amount)
    addToBalance(account, amount)
    emitTransfer(0, account, amount)
}
function transfer(to, amount) {
    executeTransfer(caller(), to, amount)
}
function approve(spender, amount) {
    revertIfZeroAddress(spender)
    setAllowance(caller(), spender, amount)
    emitApproval(caller(), spender, amount)
}
function transferFrom(from, to, amount) {
    decreaseAllowanceBy(from, caller(), amount)
    executeTransfer(from, to, amount)
}

function executeTransfer(from, to, amount) {
    revertIfZeroAddress(to)
    deductFromBalance(from, amount)
    addToBalance(to, amount)
    emitTransfer(from, to, amount)
}

/* ----- fonctions de décodage des données d'appel ----- */
function selector() -> s {
    s := div(calldataLoad(0),
↳ 0x1000000000000000000000000000000000000000000000000000000000000000)
}

function decodeAsAddress(offset) -> v {
    v := decodeAsUint(offset)

```

(suite de la page précédente)

```

        if iszero(iszero(and(v,
↳not(0xffffffffffffffffffffffffffffffff))) {
            revert(0, 0)
        }
    }
    function decodeAsUint(offset) -> v {
        let pos := add(4, mul(offset, 0x20))
        if lt(calldatasize(), add(pos, 0x20)) {
            revert(0, 0)
        }
        v := calldataload(pos)
    }
    /* ----- fonctions d'encodage des données d'appel ----- */
    function returnUint(v) {
        mstore(0, v)
        return(0, 0x20)
    }
    function returnTrue() {
        returnUint(1)
    }

    /* ----- événements ----- */
    function emitTransfer(from, to, amount) {
        let signatureHash :=
↳0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef
        emitEvent(signatureHash, from, to, amount)
    }
    function emitApproval(from, spender, amount) {
        let signatureHash :=
↳0x8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b925
        emitEvent(signatureHash, from, spender, amount)
    }
    function emitEvent(signatureHash, indexed1, indexed2, nonIndexed) {
        mstore(0, nonIndexed)
        log3(0, 0x20, signatureHash, indexed1, indexed2)
    }

    /* ----- schéma de stockage ----- */
    function ownerPos() -> p { p := 0 }
    function totalSupplyPos() -> p { p := 1 }
    function accountToStorageOffset(account) -> offset {
        offset := add(0x1000, account)
    }
    function allowanceStorageOffset(account, spender) -> offset {
        offset := accountToStorageOffset(account)
        mstore(0, offset)
        mstore(0x20, spender)
        offset := keccak256(0, 0x40)
    }

    /* ----- accès au stockage ----- */
    function owner() -> o {

```

(suite sur la page suivante)

```

    o := sload(ownerPos())
}
function totalSupply() -> supply {
    supply := sload(totalSupplyPos())
}
function mintTokens(amount) {
    sstore(totalSupplyPos(), safeAdd(totalSupply(), amount))
}
function balanceOf(account) -> bal {
    bal := sload(accountToStorageOffset(account))
}
function addToBalance(account, amount) {
    let offset := accountToStorageOffset(account)
    sstore(offset, safeAdd(sload(offset), amount))
}
function deductFromBalance(account, amount) {
    let offset := accountToStorageOffset(account)
    let bal := sload(offset)
    require(lte(amount, bal))
    sstore(offset, sub(bal, amount))
}
function allowance(account, spender) -> amount {
    amount := sload(allowanceStorageOffset(account, spender))
}
function setAllowance(account, spender, amount) {
    sstore(allowanceStorageOffset(account, spender), amount)
}
function decreaseAllowanceBy(account, spender, amount) {
    let offset := allowanceStorageOffset(account, spender)
    let currentAllowance := sload(offset)
    require(lte(amount, currentAllowance))
    sstore(offset, sub(currentAllowance, amount))
}

/* ----- fonctions d'utilité ----- */
function lte(a, b) -> r {
    r := iszero(gt(a, b))
}
function gte(a, b) -> r {
    r := iszero(lt(a, b))
}
function safeAdd(a, b) -> r {
    r := add(a, b)
    if or(lt(r, a), lt(r, b)) { revert(0, 0) }
}
function calledByOwner() -> cbo {
    cbo := eq(owner(), caller())
}
function revertIfZeroAddress(addr) {
    require(addr)
}
function require(condition) {

```

(suite sur la page suivante)

(suite de la page précédente)

```
        if iszero(condition) { revert(0, 0) }
    }
}
```

3.33 Guide de style

3.33.1 Introduction

Ce guide est destiné à fournir des conventions de codage pour l'écriture du code Solidity. Ce guide doit être considéré comme un document évolutif qui changera au fur et à mesure que des conventions utiles seront trouvées et que les anciennes conventions seront rendues obsolètes.

De nombreux projets mettront en place leurs propres guides de style. En cas de conflits, les guides de style spécifiques au projet sont prioritaires.

La structure et un grand nombre de recommandations de ce guide de style ont été tirées du guide de style de python [pep8 style guide](#).

Le but de ce guide n'est *pas* d'être la bonne ou la meilleure façon d'écrire du code Solidity. Le but de ce guide est la *consistance*. Une citation de python [pep8](#) résume bien ce concept.

Note : Un guide de style est une question de cohérence. La cohérence avec ce guide de style est importante. La cohérence au sein d'un module ou d'une fonction est la plus importante.

Mais le plus important : **savoir quand être incohérent** - parfois le guide de style ne s'applique tout simplement pas. En cas de doute, utilisez votre meilleur jugement. Regardez d'autres exemples et décidez de ce qui vous semble le mieux. Et n'hésitez pas à demander !

3.33.2 Présentation du code

Indentation

Utilisez 4 espaces par niveau d'indentation.

Tabs ou Espaces

Les espaces sont la méthode d'indentation préférée.

Il faut éviter de mélanger les tabulations et les espaces.

Lignes vierges

Entourer les déclarations de haut niveau dans le code source de solidity de deux lignes vides.

Oui :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract A {
    // ...
}

contract B {
    // ...
}

contract C {
    // ...
}
```

Non :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract A {
    // ...
}
contract B {
    // ...
}

contract C {
    // ...
}
```

Dans un contrat, les déclarations de fonctions sont entourées d'une seule ligne vierge.

Les lignes vides peuvent être omises entre des groupes de déclarations d'une seule ligne (comme les fonctions de base d'un contrat abstrait).

Oui :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract A {
    function spam() public virtual pure;
    function ham() public virtual pure;
}
```

(suite sur la page suivante)

(suite de la page précédente)

```

contract B is A {
    function spam() public pure override {
        // ...
    }

    function ham() public pure override {
        // ...
    }
}

```

Non :

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract A {
    function spam() virtual pure public;
    function ham() public virtual pure;
}

contract B is A {
    function spam() public pure override {
        // ...
    }
    function ham() public pure override {
        // ...
    }
}

```

Longueur maximale de la ligne

Garder les lignes sous la recommandation [PEP 8](#) à un maximum de 79 (ou 99) caractères aide les lecteurs à analyser facilement le code.

Les lignes enveloppées doivent se conformer aux directives suivantes.

1. Le premier argument ne doit pas être attaché à la parenthèse ouvrante.
2. Une, et une seule, indentation doit être utilisée.
3. Chaque argument doit être placé sur sa propre ligne.
4. L'élément de terminaison, `);`, doit être placé seul sur la dernière ligne.

Appels de fonction

Oui :

```

thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3
);

```

Non :

```
thisFunctionCallIsReallyLong(longArgument1,
                             longArgument2,
                             longArgument3
);

thisFunctionCallIsReallyLong(longArgument1,
                             longArgument2,
                             longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1, longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
longArgument1,
longArgument2,
longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3);
```

Déclarations d'affectation

Oui :

```
thisIsALongNestedMapping[being][set][to_some_value] = someFunction(
    argument1,
    argument2,
    argument3,
    argument4
);
```

Non :

```
thisIsALongNestedMapping[being][set][to_some_value] = someFunction(argument1,
                                                                    argument2,
                                                                    argument3,
                                                                    argument4);
```

Définitions d'événements et émetteurs d'événements

Oui :

```
event LongAndLotsOfArgs(
    address sender,
    address recipient,
    uint256 publicKey,
    uint256 amount,
    bytes32[] options
```

(suite sur la page suivante)

(suite de la page précédente)

```
);

LongAndLotsOfArgs(
    sender,
    recipient,
    publicKey,
    amount,
    options
);
```

Non «

```
event LongAndLotsOfArgs(address sender,
                        address recipient,
                        uint256 publicKey,
                        uint256 amount,
                        bytes32[] options);

LongAndLotsOfArgs(sender,
                  recipient,
                  publicKey,
                  amount,
                  options);
```

Codage du fichier source

L'encodage UTF-8 ou ASCII est préféré.

Importations

Les déclarations d'importation doivent toujours être placées en haut du fichier.

Oui :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

import "./Owned.sol";

contract A {
    // ...
}

contract B is Owned {
    // ...
}
```

Non :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;
```

(suite sur la page suivante)

```
contract A {
    // ...
}

import "./Owned.sol";

contract B is Owned {
    // ...
}
```

Ordre des fonctions

L'ordre aide les lecteurs à identifier les fonctions qu'ils peuvent appeler et à trouver plus facilement les définitions des constructeurs et des fonctions de repli.

Les fonctions doivent être regroupées en fonction de leur visibilité et ordonnées :

- constructor
- receive function (si elle existe)
- fallback function (si elle existe)
- external
- public
- internal
- private

Dans un regroupement, placez les fonctions `view` et `pure` en dernier.

Oui :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract A {
    constructor() {
        // ...
    }

    receive() external payable {
        // ...
    }

    fallback() external {
        // ...
    }

    // Fonctions externes
    // ...

    // Fonctions externes qui sont view
    // ...

    // Fonctions externes qui sont pure
```

(suite sur la page suivante)

(suite de la page précédente)

```
// ...  
  
// Fonctions publiques  
// ...  
  
// Fonctions internes  
// ...  
  
// Fonctions privées  
// ...  
}
```

Non :

```
// SPDX-License-Identifier: GPL-3.0  
pragma solidity >=0.7.0 <0.9.0;  
contract A {  
  
    // External functions  
    // ...  
  
    fallback() external {  
        // ...  
    }  
    receive() external payable {  
        // ...  
    }  
  
    // Fonctions privées  
    // ...  
  
    // Fonctions publiques  
    // ...  
  
    constructor() {  
        // ...  
    }  
  
    // Fonctions internes  
    // ...  
}
```

Espaces blancs dans les expressions

Évitez les espaces blancs superflus dans les situations suivantes :

Immédiatement à l'intérieur des parenthèses, des crochets ou des accolades, à l'exception des déclarations de fonctions sur une seule ligne.

Oui :

```
spam(ham[1], Coin({name: "ham"}));
```

Non :

```
spam( ham[ 1 ], Coin( { name: "ham" } ) );
```

Exception :

```
function singleLine() public { spam(); }
```

Immédiatement avant une virgule, un point-virgule :

Oui :

```
function spam(uint i, Coin coin) public;
```

Non;

```
function spam(uint i , Coin coin) public ;
```

More than one space around an assignment or other operator to align with another :

Yes :

```
x = 1;
y = 2;
long_variable = 3;
```

Non :

```
x          = 1;
y          = 2;
long_variable = 3;
```

Ne pas inclure d'espace dans les fonctions de réception et de repli :

Oui :

```
receive() external payable {
    ...
}

fallback() external {
    ...
}
```

Non :

```

receive () external payable {
    ...
}

fallback () external {
    ...
}

```

Structures de contrôle

Les accolades désignant le corps d'un contrat, d'une bibliothèque, de fonctions et de structs doivent :

- s'ouvrir sur la même ligne que la déclaration
- se fermer sur leur propre ligne au même niveau d'indentation que le début de la déclaration.
- L'accolade d'ouverture doit être précédée d'un espace.

Oui :

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Coin {
    struct Bank {
        address owner;
        uint balance;
    }
}

```

Non :

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Coin
{
    struct Bank {
        address owner;
        uint balance;
    }
}

```

Les mêmes recommandations s'appliquent aux structures de contrôle `if`, `else`, `while`, et `for`.

En outre, les structures de contrôle suivantes doivent être séparées par un espace unique `if`, `while` et `for` et le bloc entre parenthèses représentant le conditionnel, ainsi qu'un espace entre le bloc parenthétique conditionnel et l'accolade ouvrante.

Oui :

```

if (...) {
    ...
}

for (...) {

```

(suite sur la page suivante)

(suite de la page précédente)

```
    ...  
}
```

Non :

```
if (...)  
{  
    ...  
}  
  
while(...) {  
}  
  
for (...) {  
    ...;  
}
```

Pour les structures de contrôle dont le corps contient une seule déclaration, l'omission des accolades est acceptable *si* la déclaration est contenue sur une seule ligne.

Oui :

```
if (x < 10)  
    x += 1;
```

Non :

```
if (x < 10)  
    someArray.push(Coin({  
        name: 'spam',  
        value: 42  
    }));
```

Pour les blocs `if` qui ont une clause `else` ou `else if`, la clause `else` doit être placée sur la même ligne que l'accolade fermant le bloc `if`. Il s'agit d'une exception par rapport aux règles des autres structures de type bloc.

Oui :

```
if (x < 3) {  
    x += 1;  
} else if (x > 7) {  
    x -= 1;  
} else {  
    x = 5;  
}  
  
if (x < 3)  
    x += 1;  
else  
    x -= 1;
```

Non :

```

if (x < 3) {
    x += 1;
}
else {
    x -= 1;
}

```

Déclaration de fonction

Pour les déclarations de fonction courtes, il est recommandé de garder l'accolade d'ouverture du corps de la fonction sur la même ligne que la déclaration de la fonction.

L'accolade fermante doit être au même niveau d'indentation que la déclaration de fonction. de la fonction.

L'accolade ouvrante doit être précédée d'un seul espace.

Oui :

```

function increment(uint x) public pure returns (uint) {
    return x + 1;
}

function increment(uint x) public pure onlyOwner returns (uint) {
    return x + 1;
}

```

Non :

```

function increment(uint x) public pure returns (uint)
{
    return x + 1;
}

function increment(uint x) public pure returns (uint){
    return x + 1;
}

function increment(uint x) public pure returns (uint) {
    return x + 1;
}

function increment(uint x) public pure returns (uint) {
    return x + 1;}

```

L'ordre des modificateurs pour une fonction doit être :

1. Visibilité
2. Mutabilité
3. Virtuel
4. Remplacer
5. Modificateurs personnalisés

Oui :

```
function balance(uint from) public view override returns (uint) {
    return balanceOf[from];
}

function shutdown() public onlyOwner {
    selfdestruct(owner);
}
```

Non :

```
function balance(uint from) public override view returns (uint) {
    return balanceOf[from];
}

function shutdown() onlyOwner public {
    selfdestruct(owner);
}
```

Pour les longues déclarations de fonctions, il est recommandé de déposer chaque argument sur sa propre ligne au même niveau d'indentation que le corps de la fonction. La parenthèse fermante et la parenthèse ouvrante doivent être placées sur leur propre ligne au même niveau d'indentation que la déclaration de fonction.

Oui :

```
function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f
)
public
{
    doSomething();
}
```

Non :

```
function thisFunctionHasLotsOfArguments(address a, address b, address c,
    address d, address e, address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(address a,
                                        address b,
                                        address c,
                                        address d,
                                        address e,
                                        address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(
```

(suite sur la page suivante)

(suite de la page précédente)

```

address a,
address b,
address c,
address d,
address e,
address f) public {
doSomething();
}

```

Si une longue déclaration de fonction comporte des modificateurs, chaque modificateur doit être déposé sur sa propre ligne.

Oui :

```

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();
}

```

Non :

```

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)

```

(suite sur la page suivante)

```
public
onlyOwner
priced
returns (address) {
doSomething();
}
```

Les paramètres de sortie et les instructions de retour multilignes doivent suivre le même style que celui recommandé pour l'habillage des longues lignes dans la section Longueur de ligne maximale.

Oui :

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
public
returns (
    address someAddressName,
    uint256 LongArgument,
    uint256 Argument
)
{
doSomething()

return (
    veryLongReturnArg1,
    veryLongReturnArg2,
    veryLongReturnArg3
);
}
```

Non :

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
public
returns (address someAddressName,
    uint256 LongArgument,
    uint256 Argument)
{
doSomething()

return (veryLongReturnArg1,
    veryLongReturnArg1,
    veryLongReturnArg1);
}
```

Pour les fonctions constructrices sur les contrats hérités dont les bases nécessitent des arguments, il est recommandé de déposer les constructeurs de base sur de nouvelles lignes de la même manière que les modificateurs si la déclaration

de la fonction est longue ou difficile à lire.

Oui :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// Contrats de base juste pour que cela compile
contract B {
    constructor(uint) {
    }
}
contract C {
    constructor(uint, uint) {
    }
}
contract D {
    constructor(uint) {
    }
}

contract A is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4)
    {
        // do something with param5
        x = param5;
    }
}
```

Non :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// Contrats de base juste pour que cela compile
contract B {
    constructor(uint) {
    }
}

contract C {
    constructor(uint, uint) {
    }
}

contract D {
    constructor(uint) {
    }
}
```

(suite sur la page suivante)

```

}

contract A is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
    B(param1)
    C(param2, param3)
    D(param4) {
        x = param5;
    }
}

contract X is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
    B(param1)
    C(param2, param3)
    D(param4) {
        x = param5;
    }
}

```

Lorsque vous déclarez des fonctions courtes avec une seule déclaration, il est permis de le faire sur une seule ligne.

C'est autorisé :

```
function shortFunction() public { doSomething(); }
```

Ces directives pour les déclarations de fonctions sont destinées à améliorer la lisibilité. Les auteurs doivent faire preuve de discernement car ce guide ne prétend pas couvrir toutes les permutations possibles pour les déclarations de fonctions.

Mappages

Dans les déclarations de variables, ne séparez pas le mot-clé `mapping` de son type par un espace. Ne séparez pas un mot-clé `mapping` imbriqué de son type par un espace.

Oui :

```

mapping(uint => uint) map;
mapping(address => bool) registeredAddresses;
mapping(uint => mapping(bool => Data[])) public data;
mapping(uint => mapping(uint => s)) data;

```

Non :

```

mapping (uint => uint) map;
mapping( address => bool ) registeredAddresses;
mapping (uint => mapping (bool => Data[])) public data;
mapping(uint => mapping (uint => s)) data;

```

Déclarations de variables

Les déclarations de variables de tableau ne doivent pas comporter d'espace entre le type et les parenthèses.

Oui :

```
uint[] x;
```

Non :

```
uint [] x;
```

Autres recommandations

- Les chaînes de caractères devraient être citées avec des guillemets doubles au lieu de guillemets simples.

Oui :

```
str = "foo";
str = "Hamlet dit : 'Être ou ne pas être...'";
```

Non :

```
str = 'bar';
str = "'Soyez vous-même ; tous les autres sont déjà pris.' -Oscar Wilde';
```

- Entourer les opérateurs d'un espace unique de chaque côté.

Oui :

```
x = 3;
x = 100 / 10;
x += 3 + 4;
x |= y && z;
```

Non :

```
x=3;
x = 100/10;
x += 3+4;
x |= y&&z;
```

- Les opérateurs ayant une priorité plus élevée que les autres peuvent exclure les espaces afin d'indiquer la pré-séance. Ceci a pour but de permettre d'améliorer la lisibilité d'une déclaration complexe. Vous devez toujours utiliser la même quantité d'espaces blancs de part et d'autre d'un opérateur :

Oui :

```
x = 2**3 + 5;
x = 2*y + 3*z;
x = (a+b) * (a-b);
```

Non :

```
x = 2** 3 + 5;  
x = y+z;  
x +=1;
```

3.33.3 Ordre de mise en page

Disposez les éléments du contrat dans l'ordre suivant :

1. Déclarations de pragmatisme
2. Instructions d'importation
3. Interfaces
4. Bibliothèques
5. Contrats

À l'intérieur de chaque contrat, bibliothèque ou interface, utilisez l'ordre suivant :

1. Les déclarations de type
2. Variables d'état
3. Événements
4. Fonctions

Note : Il peut être plus clair de déclarer les types à proximité de leur utilisation dans les événements ou les variables d'état.

3.33.4 Conventions d'appellation

Les conventions de dénomination sont puissantes lorsqu'elles sont adoptées et utilisées à grande échelle. L'utilisation de différentes conventions peut véhiculer des informations *méta* significatives qui, autrement, ne seraient pas immédiatement disponibles.

Les recommandations de nommage données ici sont destinées à améliorer la lisibilité, et ne sont donc pas des règles, mais plutôt des lignes directrices pour essayer d'aider à transmettre le plus d'informations à travers les noms des choses.

Enfin, la cohérence au sein d'une base de code devrait toujours prévaloir sur les conventions décrites dans ce document.

Styles de dénomination

Pour éviter toute confusion, les noms suivants seront utilisés pour faire référence à différents styles d'appellation.

- b (lettre minuscule simple)
- B (lettre majuscule simple)
- lettresminuscules
- minuscule_avec_underscores
- MAJUSCULE
- MAJUSCULE_AVEC_UNDERSCORES
- MotsEnMajuscule (ou MotsEnMaj)
- casMixe (diffère des CapitalizedWords par le caractère minuscule initial !)
- Mots_Capitalisés_Avec_Underscores

Note : Lorsque vous utilisez des sigles dans CapWords, mettez toutes les lettres des sigles en majuscules. Ainsi, HTTP-ServerError est préférable à HttpServerError. Lors de l'utilisation d'initiales en mixedCase, mettez toutes les lettres des initiales en majuscules, mais gardez la première en minuscule si elle est le début du nom. Ainsi, xmlHTTPRequest est préférable à XMLHTTPRequest.

Noms à éviter

- 1 - Lettre minuscule el
- 0 - Lettre majuscule oh
- I - Lettre majuscule eye

N'utilisez jamais l'un de ces noms pour des noms de variables à une seule lettre. Elles sont souvent impossibles à distinguer des chiffres un et zéro.

Noms de contrats et de bibliothèques

- Les contrats et les bibliothèques doivent être nommés en utilisant le style CapWords. Exemples : SimpleToken, SmartBank, CertificateHashRepository, Player, Congress, Owned.
- Les noms des contrats et des bibliothèques doivent également correspondre à leurs noms de fichiers.
- Si un fichier de contrat comprend plusieurs contrats et/ou bibliothèques, alors le nom du fichier doit correspondre au *contrat principal*. Cela n'est cependant pas recommandé si cela peut être évité.

Comme le montre l'exemple ci-dessous, si le nom du contrat est Congress et celui de la bibliothèque Owned, les noms de fichiers associés doivent être Congress.sol et Owned.sol.

Oui :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// Owned.sol
contract Owned {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

et dans Congress.sol :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;
```

(suite sur la page suivante)

```
import "./Owned.sol";

contract Congress is Owned, TokenRecipient {
    //...
}
```

Non :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// owned.sol
contract owned {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

et dans Congress.sol :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.7.0;

import "./owned.sol";

contract Congress is owned, tokenRecipient {
    //...
}
```

Noms de structures

Les structures doivent être nommées en utilisant le style CapWords. Exemples : `MonCoin`, `Position`, `PositionXY`.

Noms d'événements

Les événements doivent être nommés en utilisant le style CapWords. Exemples : `Dépôt`, `Transfert`, `Approbation`, `AvantTransfert`, `AprèsTransfert`.

Noms des fonctions

Les fonctions doivent utiliser la casse mixte. Exemples : `getBalance`, `transfer`, `verifyOwner`, `addMember`, `changeOwner`.

Noms des arguments de la fonction

Les arguments des fonctions doivent utiliser des majuscules et des minuscules. Exemples : `initialSupply`, `account`, `recipientAddress`, `senderAddress`, `newOwner`.

Lorsque vous écrivez des fonctions de bibliothèque qui opèrent sur un struct personnalisé, le struct doit être le premier argument et doit toujours être nommée `self`.

Noms des variables locales et des variables d'état

Utilisez la casse mixte. Exemples : `totalSupply`, `remainingSupply`, `balancesOf`, `creatorAddress`, `isPreSale`, `tokenExchangeRate`.

Constantes

Les constantes doivent être nommées avec des lettres majuscules et des caractères de soulignement pour séparer les mots. Exemples : `MAX_BLOCKS`, `TOKEN_NAME`, `TOKEN_TICKER`, `CONTRACT_VERSION`.

Noms des modificateurs

Utilisez la casse mixte. Exemples : `onlyBy`, `onlyAfter`, `onlyDuringThePreSale`.

Enums

Les Enums, dans le style des déclarations de type simples, doivent être nommés en utilisant le style CapWords. Exemples : `TokenGroup`, `Frame`, `HashStyle`, `CharacterLocation`.

Éviter les collisions de noms

— `single_trailing_underscore_`

Cette convention est suggérée lorsque le nom souhaité entre en collision avec celui d'un nom intégré ou autrement réservé.

3.33.5 NatSpec

Les contrats Solidity peuvent également contenir des commentaires NatSpec. Ils sont écrits avec une triple barre oblique (`///`) ou un double astérisque (`/** ... */`). Ils doivent être utilisés directement au-dessus des déclarations de fonctions ou des instructions.

Par exemple, le contrat de *un smart contract simple* avec les commentaires ajoutés, ressemble à celui ci-dessous :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

/// @author L'équipe Solidity
/// @title Un exemple simple de stockage
contract SimpleStorage {
    uint storedData;

    /// Stocke `x`.
    /// @param x la nouvelle valeur à stocker
    /// @dev stocke le nombre dans la variable d'état `storedData`.
    function set(uint x) public {
        storedData = x;
    }

    /// Retourner la valeur stockée.
    /// @dev récupère la valeur de la variable d'état `storedData`.
    /// @retourne la valeur stockée
    function get() public view returns (uint) {
        return storedData;
    }
}
```

Il est recommandé que les contrats Solidity soient entièrement annotés en utilisant *NatSpec* pour toutes les interfaces publiques (tout ce qui se trouve dans l'ABI).

Veuillez consulter la section sur *NatSpec* pour une explication détaillée.

3.34 Modèles communs

3.34.1 Retrait des contrats

La méthode recommandée pour envoyer des fonds après un effet est d'utiliser le modèle de retrait. Bien que la méthode la plus intuitive la méthode la plus intuitive pour envoyer de l'Ether, suite à un effet, est un appel direct de « transfert », ce n'est pas recommandé car il introduit un car elle introduit un risque potentiel de sécurité. Vous pouvez lire plus d'informations à ce sujet sur la page *Considérations de sécurité*.

Voici un exemple du schéma de retrait en pratique dans un contrat où l'objectif est d'envoyer le plus d'argent vers le contrat afin de devenir le plus « riche », inspiré de *King of the Ether*.

Dans le contrat suivant, si vous n'êtes plus le plus riche, vous recevez les fonds de la personne qui est maintenant la plus riche.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    mapping (address => uint) pendingWithdrawals;

    /// La quantité d'Ether envoyé n'était pas supérieur au
    /// montant le plus élevé actuellement.
    error NotEnoughEther();

    constructor() payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() public payable {
        if (msg.value <= mostSent) revert NotEnoughEther();
        pendingWithdrawals[richest] += msg.value;
        richest = msg.sender;
        mostSent = msg.value;
    }

    function withdraw() public {
        uint amount = pendingWithdrawals[msg.sender];
        /// N'oubliez pas de mettre à zéro le remboursement en attente avant
        /// l'envoi pour éviter les attaques de ré-entrance
        pendingWithdrawals[msg.sender] = 0;
        payable(msg.sender).transfer(amount);
    }
}
```

Cela s'oppose au modèle d'envoi plus intuitif :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract SendContract {
    address payable public richest;
    uint public mostSent;

    /// La quantité d'Ether envoyée n'était pas plus élevée que
    /// le montant le plus élevé actuellement.
    error NotEnoughEther();

    constructor() payable {
```

(suite sur la page suivante)

```

    richest = payable(msg.sender);
    mostSent = msg.value;
}

function becomeRichest() public payable {
    if (msg.value <= mostSent) revert NotEnoughEther();
    // Cette ligne peut causer des problèmes (expliqués ci-dessous).
    richest.transfer(msg.value);
    richest = payable(msg.sender);
    mostSent = msg.value;
}
}

```

Remarquez que, dans cet exemple, un attaquant pourrait piéger le contrat dans un état inutilisable en faisant en sorte que `richest` soit l'adresse d'un contrat qui possède une fonction de réception ou de repli qui échoue (par exemple en utilisant `revert()` ou simplement en consommant plus que l'allocation de 2300 gaz qui leur a été transférée). De cette façon, chaque fois que `transfer` est appelé pour livrer des fonds au contrat « empoisonné », il échouera et donc aussi `becomeRichest` échouera aussi, et le contrat sera bloqué pour toujours.

En revanche, si vous utilisez le motif « `withdraw` » du premier exemple, l'attaquant ne peut faire échouer que son propre retrait, et pas le reste le reste du fonctionnement du contrat.

3.34.2 Restriction de l'accès

La restriction de l'accès est un modèle courant pour les contrats. Notez que vous ne pouvez jamais empêcher un humain ou un ordinateur de lire le contenu de vos transactions ou l'état de votre contrat. Vous pouvez rendre les choses un peu plus difficiles en utilisant le cryptage, mais si votre contrat est supposé lire les données, tout le monde le fera aussi.

Vous pouvez restreindre l'accès en lecture à l'état de votre contrat par **d'autres contrats**. C'est en fait le cas par défaut sauf si vous déclarez vos variables d'état `public`.

De plus, vous pouvez restreindre les personnes qui peuvent apporter des modifications l'état de votre contrat ou appeler les fonctions de votre contrat. fonctions de votre contrat et c'est ce dont il est question dans cette section.

L'utilisation de **modificateurs de fonction** permet de rendre ces restrictions très lisibles.

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract AccessRestriction {
    // Ils seront attribués lors de la construction
    // phase de construction, où `msg.sender` est le compte
    // qui crée ce contrat.
    address public owner = msg.sender;
    uint public createTime = block.timestamp;

    // Suit maintenant une liste d'erreurs que
    // ce contrat peut générer ainsi que
    // avec une explication textuelle dans des
    // commentaires spéciaux.

    /// L'expéditeur n'est pas autorisé pour cette
    /// opération.

```

(suite de la page précédente)

```

error Unauthorized();

/// La fonction est appelée trop tôt.
error TooEarly();

/// Pas assez d'Ether envoyé avec l'appel de fonction.
error NotEnoughEther();

// Les modificateurs peuvent être utilisés pour changer
// le corps d'une fonction.
// Si ce modificateur est utilisé, il
// ajoutera une vérification qui ne se passe
// que si la fonction est appelée depuis
// une certaine adresse.
modifier onlyBy(address _account)
{
    if (msg.sender != _account)
        revert Unauthorized();
    // N'oubliez pas le "_;"! Il sera
    // remplacé par le corps de la fonction
    // réelle lorsque le modificateur est utilisé.
    -;
}

/// Faire de `_newOwner` le nouveau propriétaire de ce
/// contrat.
function changeOwner(address _newOwner)
    public
    onlyBy(owner)
{
    owner = _newOwner;
}

modifier onlyAfter(uint _time) {
    if (block.timestamp < _time)
        revert TooEarly();
    -;
}

/// Effacer les informations sur la propriété.
/// Ne peut être appelé que 6 semaines après
/// que le contrat ait été créé.
function disown()
    public
    onlyBy(owner)
    onlyAfter(creationTime + 6 weeks)
{
    delete owner;
}

// Ce modificateur exige qu'un certain
// frais étant associé à un appel de fonction.

```

(suite sur la page suivante)

```
// Si l'appelant a envoyé trop de frais, il ou elle est
// remboursé, mais seulement après le corps de la fonction.
// Ceci était dangereux avant la version 0.4.0 de Solidity,
// où il était possible de sauter la partie après `;`.
modifier costs(uint _amount) {
    if (msg.value < _amount)
        revert NotEnoughEther();

    -;
    if (msg.value > _amount)
        payable(msg.sender).transfer(msg.value - _amount);
}

function forceOwnerChange(address _newOwner)
    public
    payable
    costs(200 ether)
{
    owner = _newOwner;
    // juste quelques exemples de conditions
    if (uint160(owner) & 0 == 1)
        // Cela n'a pas remboursé pour Solidity
        // avant la version 0.4.0.
        return;
    // rembourser les frais payés en trop
}
}
```

Une manière plus spécialisée de restreindre l'accès aux appels peut être restreint, sera abordée dans l'exemple suivant.

3.34.3 Machine à états

Les contrats se comportent souvent comme une machine à états, ce qui signifie qu'ils ont certaines **étapes** dans lesquelles ils se comportent différemment ou dans lesquelles différentes fonctions peuvent être appelées. Un appel de fonction termine souvent une étape et fait passer le contrat à l'étape suivante (surtout si le contrat modélise une **interaction**). Il est également courant que certaines étapes soient automatiquement à un certain moment dans le **temps**.

Par exemple, un contrat d'enchères à l'aveugle qui commence à l'étape « accepter des offres à l'aveugle », puis qui passe ensuite à l'étape « révéler les offres » et qui se termine par « déterminer le résultat de l'enchère ».

Les modificateurs de fonction peuvent être utilisés dans cette situation pour modéliser les états et se prémunir contre l'utilisation incorrecte du contrat.

Exemple

Dans l'exemple suivant, le modificateur `atStage` assure que la fonction ne peut être appelée qu'à un certain stade.

Les transitions automatiques temporisées sont gérées par le modificateur `timedTransitions`, devrait être utilisé pour toutes les fonctions.

Note : L'ordre des modificateurs est important. Si `atStage` est combiné avec `timedTransitions`, assurez-vous que vous le mentionnez après cette dernière, afin que la nouvelle étape soit prise en compte.

Enfin, le modificateur `transitionNext` peut être utilisé pour passer automatiquement à l'étape suivante lorsque la fonction se termine.

Note : Le Modificateur Peut Être Ignoré. Ceci s'applique uniquement à Solidity avant la version 0.4.0 : Puisque les modificateurs sont appliqués en remplaçant simplement code et non en utilisant un appel de fonction, le code dans le modificateur `transitionNext` peut être ignoré si la fonction elle-même utilise `return`. Si vous voulez faire cela, assurez-vous d'appeler `nextStage` manuellement à partir de ces fonctions. À partir de la version 0.4.0, le code du modificateur sera exécuté même si la fonction retourne explicitement.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract StateMachine {
    enum Stages {
        AcceptingBlindedBids,
        RevealBids,
        AnotherStage,
        AreWeDoneYet,
        Finished
    }
    /// La fonction ne peut pas être appelée pour le moment.
    error FunctionInvalidAtThisStage();

    // Il s'agit de l'étape actuelle.
    Stages public stage = Stages.AcceptingBlindedBids;

    uint public creationTime = block.timestamp;

    modifier atStage(Stages _stage) {
        if (stage != _stage)
            revert FunctionInvalidAtThisStage();
        -;
    }

    function nextStage() internal {
        stage = Stages(uint(stage) + 1);
    }

    // Effectuez des transitions chronométrées. Veillez à mentionner
    // ce modificateur en premier, sinon les gardes
    // ne tiendront pas compte de la nouvelle étape.
```

(suite sur la page suivante)

```
modifier timedTransitions() {
    if (stage == Stages.AcceptingBlindedBids &&
        block.timestamp >= creationTime + 10 days)
        nextStage();
    if (stage == Stages.RevealBids &&
        block.timestamp >= creationTime + 12 days)
        nextStage();
    // Les autres étapes se déroulent par transaction
    -;
}

// L'ordre des modificateurs est important ici !
function bid()
    public
    payable
    timedTransitions
    atStage(Stages.AcceptingBlindedBids)
{
    // Nous n'implémenterons pas cela ici
}

function reveal()
    public
    timedTransitions
    atStage(Stages.RevealBids)
{
}

// Ce modificateur passe à l'étape suivante
// après que la fonction soit terminée.
modifier transitionNext()
{
    -;
    nextStage();
}

function g()
    public
    timedTransitions
    atStage(Stages.AnotherStage)
    transitionNext
{
}

function h()
    public
    timedTransitions
    atStage(Stages.AreWeDoneYet)
    transitionNext
{
}
```

(suite de la page précédente)

```

function i()
  public
  timedTransitions
  atStage(Stages.Finished)
{
}
}

```

3.35 Liste des bogues connus

Ci-dessous, vous trouverez une liste, formatée en JSON, de certains des bogues connus relatifs à la sécurité dans le compilateur Solidity. Le fichier lui-même est hébergé dans le [dépositaire Github](#). La liste remonte jusqu'à la version 0.3.0, les bogues connus pour être présents uniquement dans les versions précédentes ne sont pas listés.

Il existe un autre fichier appelé `bugs_by_version.json`, qui peut être utilisé pour vérifier quels bugs affectent une version spécifique du compilateur.

Les outils de vérification des sources des contrats et aussi les autres outils interagissant avec les contrats doivent consulter cette liste selon les critères suivants :

- Il est légèrement suspect qu'un contrat ait été compilé avec une version nocturne du compilateur au lieu d'une version publiée. Cette liste ne garde pas des versions non publiées ou des versions nocturnes.
- Il est également légèrement suspect qu'un contrat ait été compilé avec une version qui n'était pas la plus récente au moment où le contrat a été établi. Pour les contrats créés à partir d'autres contrats, vous devez suivre la chaîne de création jusqu'à une transaction et utiliser la date de cette transaction comme date de création.
- Il est très suspect qu'un contrat ait été compilé à l'aide d'un compilateur qui contient un bogue connu et que le contrat a été créé à un moment où une version plus récente du compilateur contenant un correctif était déjà disponible.

Le fichier JSON des bogues connus ci-dessous est un tableau d'objets, un pour chaque bogue, avec les clés suivantes :

uid Identifiant unique donné au bogue sous la forme SOL-`<year>-<number>`. Il est possible que plusieurs entrées existent avec le même uid. Cela signifie que que plusieurs gammes de versions sont affectées par le même bogue.

name Nom unique donné au bogue

summary Brève description du bogue

description Description détaillée du bogue

link URL d'un site web contenant des informations plus détaillées, facultatif

introduced La première version du compilateur publiée qui contenait le bogue, facultatif

fixed La première version du compilateur publiée qui ne contenait plus le bogue

publish La date à laquelle le bogue a été connu publiquement, facultative.

severity Gravité du bug : très faible, faible, moyenne, élevée. Prend en compte la possibilité de découverte dans les tests contractuels, la probabilité d'occurrence et les dommages potentiels par des exploits.

conditions Les conditions qui doivent être remplies pour déclencher le bug. Les touches suivantes peuvent être utilisées : `optimizer`, valeur booléenne qui signifie que l'optimiseur booléen qui signifie que l'optimiseur doit être activé pour activer le bogue. `evmVersion`, une chaîne qui indique quelle version de EVM les paramètres de compilation déclenche le bogue. La chaîne peut contenir des opérateurs de comparaison. Par exemple, `">=constantinople"` signifie que le bogue est présent lorsque la version de l'EVM est définie sur `constantinople` ou plus. Si aucune condition n'est donnée, on suppose que le bogue est présent.

check Ce champ contient différentes vérifications qui indiquent si le contrat intelligent contient ou non le bogue. Le premier type de vérification est constitué d'expressions régulières Javascript qui doivent être comparées au code source (« source-regex ») si le bogue est présent. S'il n'y a pas de correspondance, alors le bogue est

très probablement pas présent. S'il y a une correspondance, le bogue pourrait être présent. Pour une meilleure précision, les vérifications doivent être appliquées au code source après avoir enlevé les commentaires. commentaires. Le deuxième type de vérification concerne les motifs à vérifier sur l'AST compact du programme le programme Solidity (« ast-compact-json-path »). La requête de recherche spécifiée est une expression `JsonPath`. Si au moins un chemin de l'AST Solidity correspond à la requête, le bogue est probablement présent.

```
[
  {
    "uid": "SOL-2021-4",
    "name": "UserDefinedValueTypesBug",
    "summary": "User defined value types with underlying type shorter than 32 bytes.
↳ used incorrect storage layout and wasted storage",
    "description": "The compiler did not correctly compute the storage layout of.
↳ user defined value types based on types that are shorter than 32 bytes. It would.
↳ always use a full storage slot for these types, even if the underlying type was.
↳ shorter. This was wasteful and might have problems with tooling or contract upgrades.",
    "link": "https://blog.soliditylang.org/2021/09/29/user-defined-value-types-bug/",
    "introduced": "0.8.8",
    "fixed": "0.8.9",
    "severity": "very low"
  },
  {
    "uid": "SOL-2021-3",
    "name": "SignedImmutables",
    "summary": "Immutable variables of signed integer type shorter than 256 bits can.
↳ lead to values with invalid higher order bits if inline assembly is used.",
    "description": "When immutable variables of signed integer type shorter than 256.
↳ bits are read, their higher order bits were unconditionally set to zero. The correct.
↳ operation would be to sign-extend the value, i.e. set the higher order bits to one if.
↳ the sign bit is one. This sign-extension is performed by Solidity just prior to when.
↳ it matters, i.e. when a value is stored in memory, when it is compared or when a.
↳ division is performed. Because of that, to our knowledge, the only way to access the.
↳ value in its unclean state is by reading it through inline assembly.",
    "link": "https://blog.soliditylang.org/2021/09/29/signed-immutables-bug/",
    "introduced": "0.6.5",
    "fixed": "0.8.9",
    "severity": "very low"
  },
  {
    "uid": "SOL-2021-2",
    "name": "ABIDecodeTwoDimensionalArrayMemory",
    "summary": "If used on memory byte arrays, result of the function ``abi.decode``.
↳ can depend on the contents of memory outside of the actual byte array that is decoded.
↳ ",
    "description": "The ABI specification uses pointers to data areas for everything.
↳ that is dynamically-sized. When decoding data from memory (instead of calldata), the.
↳ ABI decoder did not properly validate some of these pointers. More specifically, it.
↳ was possible to use large values for the pointers inside arrays such that computing.
↳ the offset resulted in an undetected overflow. This could lead to these pointers.
↳ targeting areas in memory outside of the actual area to be decoded. This way, it was.
↳ possible for ``abi.decode`` to return different values for the same encoded byte array.
↳ ",
  },

```

(suite sur la page suivante)

(suite de la page précédente)

```

    "link": "https://blog.soliditylang.org/2021/04/21/decoding-from-memory-bug/",
    "introduced": "0.4.16",
    "fixed": "0.8.4",
    "conditions": {
      "ABIEncoderV2": true
    },
    "severity": "very low"
  },
  {
    "uid": "SOL-2021-1",
    "name": "KeccakCaching",
    "summary": "The bytecode optimizer incorrectly re-used previously evaluated
↪Keccak-256 hashes. You are unlikely to be affected if you do not compute Keccak-256
↪hashes in inline assembly.",
    "description": "Solidity's bytecode optimizer has a step that can compute Keccak-
↪256 hashes, if the contents of the memory are known during compilation time. This step
↪also has a mechanism to determine that two Keccak-256 hashes are equal even if the
↪values in memory are not known during compile time. This mechanism had a bug where
↪Keccak-256 of the same memory content, but different sizes were considered equal. More
↪specifically, ``keccak256(mpos1, length1)`` and ``keccak256(mpos2, length2)`` in some
↪cases were considered equal if ``length1`` and ``length2``, when rounded up to nearest
↪multiple of 32 were the same, and when the memory contents at ``mpos1`` and ``mpos2``
↪can be deduced to be equal. You maybe affected if you compute multiple Keccak-256
↪hashes of the same content, but with different lengths inside inline assembly. You are
↪unaffected if your code uses ``keccak256`` with a length that is not a compile-time
↪constant or if it is always a multiple of 32.",
    "link": "https://blog.soliditylang.org/2021/03/23/keccak-optimizer-bug/",
    "fixed": "0.8.3",
    "conditions": {
      "optimizer": true
    },
    "severity": "medium"
  },
  {
    "uid": "SOL-2020-11",
    "name": "EmptyByteArrayCopy",
    "summary": "Copying an empty byte array (or string) from memory or calldata to
↪storage can result in data corruption if the target array's length is increased
↪subsequently without storing new data.",
    "description": "The routine that copies byte arrays from memory or calldata to
↪storage stores unrelated data from after the source array in the storage slot if the
↪source array is empty. If the storage array's length is subsequently increased either
↪by using ``.push()`` or by assigning to its ``.length`` attribute (only before 0.6.0),
↪the newly created byte array elements will not be zero-initialized, but contain the
↪unrelated data. You are not affected if you do not assign to ``.length`` and do not
↪use ``.push()`` on byte arrays, or only use ``.push(<arg>`` or manually initialize
↪the new elements.",
    "link": "https://blog.soliditylang.org/2020/10/19/empty-byte-array-copy-bug/",
    "fixed": "0.7.4",
    "severity": "medium"
  },
  },
  {

```

(suite sur la page suivante)

```

    "uid": "SOL-2020-10",
    "name": "DynamicArrayCleanup",
    "summary": "When assigning a dynamically-sized array with types of size at most
↪16 bytes in storage causing the assigned array to shrink, some parts of deleted slots
↪were not zeroed out.",
    "description": "Consider a dynamically-sized array in storage whose base-type is
↪small enough such that multiple values can be packed into a single slot, such as
↪`uint128[]`. Let us define its length to be `l`. When this array gets assigned from
↪another array with a smaller length, say `m`, the slots between elements `m` and `l`
↪have to be cleaned by zeroing them out. However, this cleaning was not performed
↪properly. Specifically, after the slot corresponding to `m`, only the first packed
↪value was cleaned up. If this array gets resized to a length larger than `m`, the
↪indices corresponding to the unclean parts of the slot contained the original value,
↪instead of 0. The resizing here is performed by assigning to the array `length`, by a
↪`push()` or via inline assembly. You are not affected if you are only using `push(
↪<arg>` or if you assign a value (even zero) to the new elements after increasing the
↪length of the array.",
    "link": "https://blog.soliditylang.org/2020/10/07/solidity-dynamic-array-cleanup-
↪bug/",
    "fixed": "0.7.3",
    "severity": "medium"
  },
  {
    "uid": "SOL-2020-9",
    "name": "FreeFunctionRedefinition",
    "summary": "The compiler does not flag an error when two or more free functions
↪with the same name and parameter types are defined in a source unit or when an
↪imported free function alias shadows another free function with a different name but
↪identical parameter types.",
    "description": "In contrast to functions defined inside contracts, free
↪functions with identical names and parameter types did not create an error. Both
↪definition of free functions with identical name and parameter types and an imported
↪free function with an alias that shadows another function with a different name but
↪identical parameter types were permitted due to which a call to either the multiply
↪defined free function or the imported free function alias within a contract led to the
↪execution of that free function which was defined first within the source unit.
↪Subsequently defined identical free function definitions were silently ignored and
↪their code generation was skipped.",
    "introduced": "0.7.1",
    "fixed": "0.7.2",
    "severity": "low"
  },
  {
    "uid": "SOL-2020-8",
    "name": "UsingForCalldata",
    "summary": "Function calls to internal library functions with calldata
↪parameters called via ``using for`` can result in invalid data being read.",
    "description": "Function calls to internal library functions using the ``using
↪for`` mechanism copied all calldata parameters to memory first and passed them on like
↪that, regardless of whether it was an internal or an external call. Due to that, the
↪called function would receive a memory pointer that is interpreted as a calldata
↪pointer. Since dynamically sized arrays are passed using two stack slots for calldata,
↪but only one for memory, this can lead to stack corruption. An affected library call
↪will consider the JUMPDEST to which it is supposed to return as part of its arguments
↪and will instead jump out to whatever was on the stack before the call.",

```

(suite de la page précédente)

```

    "introduced": "0.6.9",
    "fixed": "0.6.10",
    "severity": "very low"
  },
  {
    "uid": "SOL-2020-7",
    "name": "MissingEscapingInFormatting",
    "summary": "String literals containing double backslash characters passed
↳ directly to external or encoding function calls can lead to a different string being
↳ used when ABIEncoderV2 is enabled.",
    "description": "When ABIEncoderV2 is enabled, string literals passed directly to
↳ encoding functions or external function calls are stored as strings in the intermediate
↳ code. Characters outside the printable range are handled correctly, but backslashes
↳ are not escaped in this procedure. This leads to double backslashes being reduced to
↳ single backslashes and consequently re-interpreted as escapes potentially resulting in
↳ a different string being encoded.",
    "introduced": "0.5.14",
    "fixed": "0.6.8",
    "severity": "very low",
    "conditions": {
      "ABIEncoderV2": true
    }
  },
  {
    "uid": "SOL-2020-6",
    "name": "ArraySliceDynamicallyEncodedBaseType",
    "summary": "Accessing array slices of arrays with dynamically encoded base types
↳ (e.g. multi-dimensional arrays) can result in invalid data being read.",
    "description": "For arrays with dynamically sized base types, index range
↳ accesses that use a start expression that is non-zero will result in invalid array
↳ slices. Any index access to such array slices will result in data being read from
↳ incorrect calldata offsets. Array slices are only supported for dynamic calldata types
↳ and all problematic type require ABIEncoderV2 to be enabled.",
    "introduced": "0.6.0",
    "fixed": "0.6.8",
    "severity": "very low",
    "conditions": {
      "ABIEncoderV2": true
    }
  },
  {
    "uid": "SOL-2020-5",
    "name": "ImplicitConstructorCallvalueCheck",
    "summary": "The creation code of a contract that does not define a constructor
↳ but has a base that does define a constructor did not revert for calls with non-zero
↳ value.",
    "description": "Starting from Solidity 0.4.5 the creation code of contracts
↳ without explicit payable constructor is supposed to contain a callvalue check that
↳ results in contract creation reverting, if non-zero value is passed. However, this
↳ check was missing in case no explicit constructor was defined in a contract at all,
↳ but the contract has a base that does define a constructor. In these cases it is
↳ possible to send value in a contract creation transaction or using inline assembly
↳ without revert, even though the creation code is supposed to be non-payable."
  }
}

```

(suite sur la page suivante)

```

    "introduced": "0.4.5",
    "fixed": "0.6.8",
    "severity": "very low"
  },
  {
    "uid": "SOL-2020-4",
    "name": "TupleAssignmentMultiStackSlotComponents",
    "summary": "Tuple assignments with components that occupy several stack slots, i.
↪e. nested tuples, pointers to external functions or references to dynamically sized
↪calldata arrays, can result in invalid values.",
    "description": "Tuple assignments did not correctly account for tuple components
↪that occupy multiple stack slots in case the number of stack slots differs between
↪left-hand-side and right-hand-side. This can either happen in the presence of nested
↪tuples or if the right-hand-side contains external function pointers or references to
↪dynamic calldata arrays, while the left-hand-side contains an omission.",
    "introduced": "0.1.6",
    "fixed": "0.6.6",
    "severity": "very low"
  },
  {
    "uid": "SOL-2020-3",
    "name": "MemoryArrayCreationOverflow",
    "summary": "The creation of very large memory arrays can result in overlapping
↪memory regions and thus memory corruption.",
    "description": "No runtime overflow checks were performed for the length of
↪memory arrays during creation. In cases for which the memory size of an array in bytes,
↪i.e. the array length times 32, is larger than 2^256-1, the memory allocation will
↪overflow, potentially resulting in overlapping memory areas. The length of the array
↪is still stored correctly, so copying or iterating over such an array will result in
↪out-of-gas.",
    "link": "https://blog.soliditylang.org/2020/04/06/memory-creation-overflow-bug/",
    "introduced": "0.2.0",
    "fixed": "0.6.5",
    "severity": "low"
  },
  {
    "uid": "SOL-2020-1",
    "name": "YulOptimizerRedundantAssignmentBreakContinue",
    "summary": "The Yul optimizer can remove essential assignments to variables
↪declared inside for loops when Yul's continue or break statement is used. You are
↪unlikely to be affected if you do not use inline assembly with for loops and continue
↪and break statements.",
    "description": "The Yul optimizer has a stage that removes assignments to
↪variables that are overwritten again or are not used in all following control-flow
↪branches. This logic incorrectly removes such assignments to variables declared inside
↪a for loop if they can be removed in a control-flow branch that ends with ``break`` or
↪``continue`` even though they cannot be removed in other control-flow branches.
↪Variables declared outside of the respective for loop are not affected.",
    "introduced": "0.6.0",
    "fixed": "0.6.1",
    "severity": "medium",
    "conditions": {

```

(suite sur la page suivante)

(suite de la page précédente)

```

        "yulOptimizer": true
    }
},
{
    "uid": "SOL-2020-2",
    "name": "privateCanBeOverridden",
    "summary": "Private methods can be overridden by inheriting contracts.",
    "description": "While private methods of base contracts are not visible and
↳ cannot be called directly from the derived contract, it is still possible to declare a
↳ function of the same name and type and thus change the behaviour of the base contract
↳'s function.",
    "introduced": "0.3.0",
    "fixed": "0.5.17",
    "severity": "low"
},
{
    "uid": "SOL-2020-1",
    "name": "YulOptimizerRedundantAssignmentBreakContinue0.5",
    "summary": "The Yul optimizer can remove essential assignments to variables
↳ declared inside for loops when Yul's continue or break statement is used. You are
↳ unlikely to be affected if you do not use inline assembly with for loops and continue
↳ and break statements.",
    "description": "The Yul optimizer has a stage that removes assignments to
↳ variables that are overwritten again or are not used in all following control-flow
↳ branches. This logic incorrectly removes such assignments to variables declared inside
↳ a for loop if they can be removed in a control-flow branch that ends with ``break`` or
↳ ``continue`` even though they cannot be removed in other control-flow branches.
↳ Variables declared outside of the respective for loop are not affected.",
    "introduced": "0.5.8",
    "fixed": "0.5.16",
    "severity": "low",
    "conditions": {
        "yulOptimizer": true
    }
},
{
    "uid": "SOL-2019-10",
    "name": "ABIEncoderV2LoopYulOptimizer",
    "summary": "If both the experimental ABIEncoderV2 and the experimental Yul
↳ optimizer are activated, one component of the Yul optimizer may reuse data in memory
↳ that has been changed in the meantime.",
    "description": "The Yul optimizer incorrectly replaces ``mload`` and ``sload``
↳ calls with values that have been previously written to the load location (and
↳ potentially changed in the meantime) if all of the following conditions are met: (1)
↳ there is a matching ``mstore`` or ``sstore`` call before; (2) the contents of memory
↳ or storage is only changed in a function that is called (directly or indirectly) in
↳ between the first store and the load call; (3) called function contains a for loop
↳ where the same memory location is changed in the condition or the post or body block.
↳ When used in Solidity mode, this can only happen if the experimental ABIEncoderV2 is
↳ activated and the experimental Yul optimizer has been activated manually in addition
↳ to the regular optimizer in the compiler settings.",
    "introduced": "0.5.14",

```

(suite sur la page suivante)

```

    "fixed": "0.5.15",
    "severity": "low",
    "conditions": {
      "ABIEncoderV2": true,
      "optimizer": true,
      "yulOptimizer": true
    }
  },
  {
    "uid": "SOL-2019-9",
    "name":
    ↪ "ABIEncoderV2CalldataStructsWithStaticallySizedAndDynamicallyEncodedMembers",
    "summary": "Reading from calldata structs that contain dynamically encoded, but
    ↪ statically-sized members can result in incorrect values.",
    "description": "When a calldata struct contains a dynamically encoded, but
    ↪ statically-sized member, the offsets for all subsequent struct members are calculated
    ↪ incorrectly. All reads from such members will result in invalid values. Only calldata
    ↪ structs are affected, i.e. this occurs in external functions with such structs as
    ↪ argument. Using affected structs in storage or memory or as arguments to public
    ↪ functions on the other hand works correctly.",
    "introduced": "0.5.6",
    "fixed": "0.5.11",
    "severity": "low",
    "conditions": {
      "ABIEncoderV2": true
    }
  },
  {
    "uid": "SOL-2019-8",
    "name": "SignedArrayStorageCopy",
    "summary": "Assigning an array of signed integers to a storage array of
    ↪ different type can lead to data corruption in that array.",
    "description": "In two's complement, negative integers have their higher order
    ↪ bits set. In order to fit into a shared storage slot, these have to be set to zero.
    ↪ When a conversion is done at the same time, the bits to set to zero were incorrectly
    ↪ determined from the source and not the target type. This means that such copy
    ↪ operations can lead to incorrect values being stored.",
    "link": "https://blog.soliditylang.org/2019/06/25/solidity-storage-array-bugs/",
    "introduced": "0.4.7",
    "fixed": "0.5.10",
    "severity": "low/medium"
  },
  {
    "uid": "SOL-2019-7",
    "name": "ABIEncoderV2StorageArrayWithMultiSlotElement",
    "summary": "Storage arrays containing structs or other statically-sized arrays
    ↪ are not read properly when directly encoded in external function calls or in abi.
    ↪ encode*.",
    "description": "When storage arrays whose elements occupy more than a single
    ↪ storage slot are directly encoded in external function calls or using abi.encode*,
    ↪ their elements are read in an overlapping manner, i.e. the element pointer is not
    ↪ properly advanced between reads. This is not a problem when the storage data is first
    ↪ copied to a memory variable or if the storage array only contains value
    ↪ dynamically-sized arrays.",
  }
}

```

(suite sur la page suivante)

(suite de la page précédente)

```

    "link": "https://blog.soliditylang.org/2019/06/25/solidity-storage-array-bugs/",
    "introduced": "0.4.16",
    "fixed": "0.5.10",
    "severity": "low",
    "conditions": {
      "ABIEncoderV2": true
    }
  },
  {
    "uid": "SOL-2019-6",
    "name": "DynamicConstructorArgumentsClippedABIV2",
    "summary": "A contract's constructor that takes structs or arrays that contain
↳dynamically-sized arrays reverts or decodes to invalid data.",
    "description": "During construction of a contract, constructor parameters are
↳copied from the code section to memory for decoding. The amount of bytes to copy was
↳calculated incorrectly in case all parameters are statically-sized but contain
↳dynamically-sized arrays as struct members or inner arrays. Such types are only
↳available if ABIEncoderV2 is activated.",
    "introduced": "0.4.16",
    "fixed": "0.5.9",
    "severity": "very low",
    "conditions": {
      "ABIEncoderV2": true
    }
  },
  {
    "uid": "SOL-2019-5",
    "name": "UninitializedFunctionPointerInConstructor",
    "summary": "Calling uninitialized internal function pointers created in the
↳constructor does not always revert and can cause unexpected behaviour.",
    "description": "Uninitialized internal function pointers point to a special
↳piece of code that causes a revert when called. Jump target positions are different
↳during construction and after deployment, but the code for setting this special jump
↳target only considered the situation after deployment.",
    "introduced": "0.5.0",
    "fixed": "0.5.8",
    "severity": "very low"
  },
  {
    "uid": "SOL-2019-5",
    "name": "UninitializedFunctionPointerInConstructor_0.4.x",
    "summary": "Calling uninitialized internal function pointers created in the
↳constructor does not always revert and can cause unexpected behaviour.",
    "description": "Uninitialized internal function pointers point to a special
↳piece of code that causes a revert when called. Jump target positions are different
↳during construction and after deployment, but the code for setting this special jump
↳target only considered the situation after deployment.",
    "introduced": "0.4.5",
    "fixed": "0.4.26",
    "severity": "very low"
  },
  {
    {

```

(suite sur la page suivante)

```

    "uid": "SOL-2019-4",
    "name": "IncorrectEventSignatureInLibraries",
    "summary": "Contract types used in events in libraries cause an incorrect event.↳
↳signature hash",
    "description": "Instead of using the type `address` in the hashed signature, the.↳
↳actual contract name was used, leading to a wrong hash in the logs.",
    "introduced": "0.5.0",
    "fixed": "0.5.8",
    "severity": "very low"
  },
  {
    "uid": "SOL-2019-4",
    "name": "IncorrectEventSignatureInLibraries_0.4.x",
    "summary": "Contract types used in events in libraries cause an incorrect event.↳
↳signature hash",
    "description": "Instead of using the type `address` in the hashed signature, the.↳
↳actual contract name was used, leading to a wrong hash in the logs.",
    "introduced": "0.3.0",
    "fixed": "0.4.26",
    "severity": "very low"
  },
  {
    "uid": "SOL-2019-3",
    "name": "ABIEncoderV2PackedStorage",
    "summary": "Storage structs and arrays with types shorter than 32 bytes can.↳
↳cause data corruption if encoded directly from storage using the experimental.↳
↳ABIEncoderV2.",
    "description": "Elements of structs and arrays that are shorter than 32 bytes.↳
↳are not properly decoded from storage when encoded directly (i.e. not via a memory.↳
↳type) using ABIEncoderV2. This can cause corruption in the values themselves but can.↳
↳also overwrite other parts of the encoded data.",
    "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↳abiencoderv2-bug/",
    "introduced": "0.5.0",
    "fixed": "0.5.7",
    "severity": "low",
    "conditions": {
      "ABIEncoderV2": true
    }
  },
  {
    "uid": "SOL-2019-3",
    "name": "ABIEncoderV2PackedStorage_0.4.x",
    "summary": "Storage structs and arrays with types shorter than 32 bytes can.↳
↳cause data corruption if encoded directly from storage using the experimental.↳
↳ABIEncoderV2.",
    "description": "Elements of structs and arrays that are shorter than 32 bytes.↳
↳are not properly decoded from storage when encoded directly (i.e. not via a memory.↳
↳type) using ABIEncoderV2. This can cause corruption in the values themselves but can.↳
↳also overwrite other parts of the encoded data.",
    "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↳abiencoderv2-bug/",

```

(suite sur la page suivante)

(suite de la page précédente)

```

    "introduced": "0.4.19",
    "fixed": "0.4.26",
    "severity": "low",
    "conditions": {
      "ABIEncoderV2": true
    }
  },
  {
    "uid": "SOL-2019-2",
    "name": "IncorrectByteInstructionOptimization",
    "summary": "The optimizer incorrectly handles byte opcodes whose second argument
↪ is 31 or a constant expression that evaluates to 31. This can result in unexpected
↪ values.",
    "description": "The optimizer incorrectly handles byte opcodes that use the
↪ constant 31 as second argument. This can happen when performing index access on
↪ bytesNN types with a compile-time constant value (not index) of 31 or when using the
↪ byte opcode in inline assembly.",
    "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪ abiencoderv2-bug/",
    "introduced": "0.5.5",
    "fixed": "0.5.7",
    "severity": "very low",
    "conditions": {
      "optimizer": true
    }
  },
  {
    "uid": "SOL-2019-1",
    "name": "DoubleShiftSizeOverflow",
    "summary": "Double bitwise shifts by large constants whose sum overflows 256
↪ bits can result in unexpected values.",
    "description": "Nested logical shift operations whose total shift size is 2**256
↪ or more are incorrectly optimized. This only applies to shifts by numbers of bits that
↪ are compile-time constant expressions.",
    "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪ abiencoderv2-bug/",
    "introduced": "0.5.5",
    "fixed": "0.5.6",
    "severity": "low",
    "conditions": {
      "optimizer": true,
      "evmVersion": ">=constantinople"
    }
  },
  {
    "uid": "SOL-2018-4",
    "name": "ExpExponentCleanup",
    "summary": "Using the ** operator with an exponent of type shorter than 256 bits
↪ can result in unexpected values.",
    "description": "Higher order bits in the exponent are not properly cleaned
↪ before the EXP opcode is applied if the type of the exponent expression is smaller
↪ than 256 bits and not smaller than the type of the base. In that case, the result
↪ might be larger than expected if the exponent is assumed to lie within the value range
↪ of the type. Literal numbers as exponents are unaffected as are exponents or bases of
↪ type uint256.",

```

(suite sur la page suivante)

```

    "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
    "fixed": "0.4.25",
    "severity": "medium/high",
    "check": {"regex-source": "[^/]*\\* *[^/0-9 ]"}
  },
  {
    "uid": "SOL-2018-3",
    "name": "EventStructWrongData",
    "summary": "Using structs in events logged wrong data.",
    "description": "If a struct is used in an event, the address of the struct is
↳ logged instead of the actual data.",
    "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
    "introduced": "0.4.17",
    "fixed": "0.4.25",
    "severity": "very low",
    "check": {"ast-compact-json-path": "$..[?(@.nodeType === 'EventDefinition')].?[
↳ (@.nodeType === 'UserDefinedTypeName' && @.typeDescriptions.typeString.startsWith(
↳ 'struct'))]"}
  },
  {
    "uid": "SOL-2018-2",
    "name": "NestedArrayFunctionCallDecoder",
    "summary": "Calling functions that return multi-dimensional fixed-size arrays
↳ can result in memory corruption.",
    "description": "If Solidity code calls a function that returns a multi-
↳ dimensional fixed-size array, array elements are incorrectly interpreted as memory
↳ pointers and thus can cause memory corruption if the return values are accessed.
↳ Calling functions with multi-dimensional fixed-size arrays is unaffected as is
↳ returning fixed-size arrays from function calls. The regular expression only checks if
↳ such functions are present, not if they are called, which is required for the contract
↳ to be affected.",
    "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
    "introduced": "0.1.4",
    "fixed": "0.4.22",
    "severity": "medium",
    "check": {"regex-source": "returns[^;]*\\[[\\s*[^\\] \\t\\r\\n\\v\\f][^\\]]*\\]\\[
↳ s*\\[[\\s*[^\\] \\t\\r\\n\\v\\f][^\\]]*\\][^;]*;"}
  },
  {
    "uid": "SOL-2018-1",
    "name": "OneOfTwoConstructorsSkipped",
    "summary": "If a contract has both a new-style constructor (using the
↳ constructor keyword) and an old-style constructor (a function with the same name as
↳ the contract) at the same time, one of them will be ignored.",
    "description": "If a contract has both a new-style constructor (using the
↳ constructor keyword) and an old-style constructor (a function with the same name as
↳ the contract) at the same time, one of them will be ignored. There will be a compiler
↳ warning about the old-style constructor, so contracts only using new-style
↳ constructors are fine.",
    "introduced": "0.4.22",
    "fixed": "0.4.23",
    "severity": "very low"
  }
}

```

(suite de la page précédente)

```

    },
    {
      "uid": "SOL-2017-5",
      "name": "ZeroFunctionSelector",
      "summary": "It is possible to craft the name of a function such that it is
↳executed instead of the fallback function in very specific circumstances.",
      "description": "If a function has a selector consisting only of zeros, is
↳payable and part of a contract that does not have a fallback function and at most five
↳external functions in total, this function is called instead of the fallback function
↳if Ether is sent to the contract without data.",
      "fixed": "0.4.18",
      "severity": "very low"
    },
    {
      "uid": "SOL-2017-4",
      "name": "DelegateCallReturnValue",
      "summary": "The low-level .delegatecall() does not return the execution outcome,
↳but converts the value returned by the functioned called to a boolean instead.",
      "description": "The return value of the low-level .delegatecall() function is
↳taken from a position in memory, where the call data or the return data resides. This
↳value is interpreted as a boolean and put onto the stack. This means if the called
↳function returns at least 32 zero bytes, .delegatecall() returns false even if the
↳call was successful.",
      "introduced": "0.3.0",
      "fixed": "0.4.15",
      "severity": "low"
    },
    {
      "uid": "SOL-2017-3",
      "name": "EcrecoverMalformedInput",
      "summary": "The ecrecover() builtin can return garbage for malformed input.",
      "description": "The ecrecover precompile does not properly signal failure for
↳malformed input (especially in the 'v' argument) and thus the Solidity function can
↳return data that was previously present in the return area in memory.",
      "fixed": "0.4.14",
      "severity": "medium"
    },
    {
      "uid": "SOL-2017-2",
      "name": "SkipEmptyStringLiteral",
      "summary": "If \"\" is used in a function call, the following function arguments
↳will not be correctly passed to the function.",
      "description": "If the empty string literal \"\" is used as an argument in a
↳function call, it is skipped by the encoder. This has the effect that the encoding of
↳all arguments following this is shifted left by 32 bytes and thus the function call
↳data is corrupted.",
      "fixed": "0.4.12",
      "severity": "low"
    },
    {
      "uid": "SOL-2017-1",
      "name": "ConstantOptimizerSubtraction",

```

(suite sur la page suivante)

```

    "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.soliditylang.org/2017/05/03/solidity-optimizer-bug/",
    "fixed": "0.4.11",
    "severity": "low",
    "conditions": {
      "optimizer": true
    }
  },
  {
    "uid": "SOL-2016-11",
    "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"
  },
  {
    "uid": "SOL-2016-10",
    "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
    }
  },
  {
    "uid": "SOL-2016-9",
    "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.soliditylang.org/2016/11/01/security-alert-solidity-
↳ variables-can-overwritten-storage/",

```

(suite de la page précédente)

```

    "severity": "high",
    "introduced": "0.1.6",
    "fixed": "0.4.4"
  },
  {
    "uid": "SOL-2016-8",
    "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
    }
  },
  {
    "uid": "SOL-2016-7",
    "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"
  },
  {
    "uid": "SOL-2016-6",
    "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"
  },
  {
    "uid": "SOL-2016-5",
    "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.",

```

(suite sur la page suivante)

```

    "severity": "low",
    "fixed": "0.3.6"
  },
  {
    "uid": "SOL-2016-4",
    "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
    }
  },
  {
    "uid": "SOL-2016-3",
    "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"
  },
  {
    "uid": "SOL-2016-2",
    "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"
  },
  {
    "uid": "SOL-2016-1",
    "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"
  }

```

(suite sur la page suivante)

```
}  
]
```

3.36 Contribution

L'aide est toujours la bienvenue et il existe de nombreuses possibilités de contribuer à Solidity.

En particulier, nous apprécions le soutien dans les domaines suivants :

- Signaler les problèmes.
- Corriger et répondre aux problèmes de [Solidity's GitHub issues.](#), en particulier ceux marqués comme « [good first issue](#) » qui sont destinés à servir de problèmes d'introduction pour les contributeurs externes.
- Améliorer la documentation.
- Traduire la documentation dans plus de langues.
- Répondre aux questions des autres utilisateurs sur [StackExchange](#) et le [Solidity Gitter Chat](#).
- S'impliquer dans le processus de conception du langage en proposant des changements de langage ou de nouvelles fonctionnalités sur le forum [Solidity](#) et en fournissant des commentaires.

Pour commencer, vous pouvez essayer *Construire à partir de la source* afin de vous familiariser avec les composants de Solidity et le processus de construction. En outre, il peut être utile de vous familiariser avec l'écriture de contrats intelligents dans Solidity.

Veuillez noter que ce projet est publié avec un [Code de conduite du contributeur](#). En participant à ce projet - dans les problèmes, les demandes de pull, ou les canaux Gitter - vous acceptez de respecter ses termes.

3.36.1 Appels de l'équipe

Si vous avez des problèmes ou des demandes de pull à discuter, ou si vous êtes intéressé à entendre ce sur quoi l'équipe et les contributeurs travaillent, vous pouvez rejoindre nos appels d'équipe publics :

- Les lundis à 15h CET/CEST.
- Les mercredis à 14h CET/CEST.

Les deux appels ont lieu sur [Jitsi](#).

3.36.2 Comment signaler des problèmes

Pour signaler un problème, veuillez utiliser le [GitHub issues tracker](#). Lorsque rapportant des problèmes, veuillez mentionner les détails suivants :

- Version de Solidity.
- Code source (le cas échéant).
- Système d'exploitation.
- Étapes pour reproduire le problème.
- Le comportement réel par rapport au comportement attendu.

Il est toujours très utile de réduire au strict minimum le code source à l'origine du problème. Très utile et permet même parfois de clarifier un malentendu.

3.36.3 Flux de travail pour les demandes de Pull

Pour contribuer, merci de vous détacher de la branche `develop` et d'y faire vos modifications ici. Vos messages de commit doivent détailler *pourquoi* vous avez fait votre changement en plus de *ce que vous avez fait* (*sauf si c'est un changement minuscule*).

Si vous avez besoin de retirer des changements de la branche `develop` après avoir fait votre fork (par exemple, pour résoudre des conflits de fusion potentiels), évitez d'utiliser `git merge` et à la place, `git rebase` votre branche. Cela nous aidera à revoir votre changement plus facilement.

De plus, si vous écrivez une nouvelle fonctionnalité, veuillez vous assurer que vous ajoutez des tests appropriés sous `test/` (voir ci-dessous).

Cependant, si vous effectuez un changement plus important, veuillez consulter le [canal Gitter du développement de Solidity](#) (différent de celui mentionné ci-dessus, celui-ci est axé sur le développement du compilateur et du langage plutôt que sur l'utilisation du langage) en premier lieu.

Les nouvelles fonctionnalités et les corrections de bogues doivent être ajoutées au fichier `Changelog.md` : veuillez suivre le style des entrées précédentes, le cas échéant.

Enfin, veuillez à respecter le ```` style de codage https://github.com/ethereum/solidity/blob/develop/CODING_STYLE.md pour ce projet. De plus, même si nous effectuons des tests CI, veuillez tester votre code et assurez-vous qu'il se construit localement avant de soumettre une demande de pull.

Merci pour votre aide !

3.36.4 Exécution des tests du compilateur

Conditions préalables

Pour exécuter tous les tests du compilateur, vous pouvez vouloir installer facultativement quelques dépendances (`evmone`, `libz3`, et `libhera`).

Sur macOS, certains des scripts de test attendent que GNU coreutils soit installé. Ceci peut être accompli plus facilement en utilisant Homebrew : `brew install coreutils`.

Exécution des tests

Solidity inclut différents types de tests, la plupart d'entre eux étant regroupés dans l'application `Boost C++ Test Framework`. `Boost C++ Test Framework` application `soltest`. Exécuter `build/test/soltest` ou son wrapper `scripts/soltest.sh` est suffisant pour la plupart des modifications.

Le script `./scripts/tests.sh` exécute automatiquement la plupart des tests Solidity, y compris ceux inclus dans le `Boost C++ Test Framework` l'application `soltest` (ou son enveloppe `scripts/soltest.sh`), ainsi que les tests en ligne de commande et les tests de compilation.

Le système de test essaie automatiquement de découvrir l'emplacement du `evmone` pour exécuter les tests sémantiques.

La bibliothèque `evmone` doit être située dans le répertoire `deps` ou `deps/lib` relativement au répertoire de travail actuel, à son parent ou au parent de son parent. Alternativement, un emplacement explicite pour l'objet partagé `evmone` peut être spécifié via la variable d'environnement `ETH_EVMONE`.

`evmone` est principalement nécessaire pour l'exécution de tests sémantiques et de gaz. Si vous ne l'avez pas installé, vous pouvez ignorer ces tests en passant l'option `--no-semantic-tests` à `scripts/soltest.sh`.

L'exécution des tests Ewasm est désactivée par défaut et peut être explicitement activée via `./scripts/soltest.sh --ewasm` et nécessite que `hera` soit trouvé par `soltest.sh`. Pour être trouvé par `soltest`. Le mécanisme de

localisation de la bibliothèque `hera` est le même que pour `evmone`, sauf que la variable permettant de spécifier un emplacement explicite est appelée `ETH_HERA`.

Les bibliothèques `evmone` et `hera` doivent toutes deux se terminer par l'extension de fichier avec l'extension `.so` sur Linux, `.dll` sur les systèmes Windows et `.dylib` sur macOS.

Pour exécuter les tests SMT, la bibliothèque `libz3` doit être installée et localisable par `cmake` pendant l'étape de configuration du compilateur.

Si la bibliothèque `libz3` n'est pas installée sur votre système, vous devriez désactiver les tests SMT en exportant `SMT_FLAGS=--no-smt` avant de lancer `./scripts/tests.sh` ou de en exécutant `./scripts/soltest.sh --no-smt`. Ces tests sont `libsolidity/smtCheckerTests` et `libsolidity/smtCheckerTestsJSON`.

Note : Pour obtenir une liste de tous les tests unitaires exécutés par `Soltest`, exécutez `./build/test/soltest --list_content=HRF`.

Pour obtenir des résultats plus rapides, vous pouvez exécuter un sous-ensemble de tests ou des tests spécifiques.

Pour exécuter un sous-ensemble de tests, vous pouvez utiliser des filtres : `./scripts/soltest.sh -t TestSuite/ TestName`, où `TestName` peut être un joker `*`.

Ou, par exemple, pour exécuter tous les tests pour le désambiguïseur `yul` : `./scripts/soltest.sh -t "yulOptimizerTests/disambiguator/*" --no-smt`.

`./build/test/soltest --help` a une aide étendue sur toutes les options disponibles.

Voir en particulier :

- `show_progress (-p)` pour montrer l'achèvement du test,
- `run_test (-t)` pour exécuter des cas de tests spécifiques, et
- `report-level (-r)` donner un rapport plus détaillé.

Note : Ceux qui travaillent dans un environnement Windows et qui veulent exécuter les jeux de base ci-dessus sans `libz3`. En utilisant Git Bash, vous utilisez : `./build/test/Release/soltest.exe -- --no-smt`. Si vous exécutez ceci dans une Invite de Commande simple, utilisez : `./build/test/Release/soltest.exe -- --no-smt`.

Si vous voulez déboguer à l'aide de GDB, assurez-vous que vous construisez différemment de ce qui est « habituel ». Par exemple, vous pouvez exécuter la commande suivante dans votre dossier `build` : .. code-block : : bash

```
cmake -DCMAKE_BUILD_TYPE=Debug .. make
```

Cela crée des symboles de sorte que lorsque vous déboguez un test en utilisant le drapeau `--debug`, vous avez accès aux fonctions et aux variables avec lesquelles vous pouvez casser ou imprimer.

Le CI exécute des tests supplémentaires (y compris `solc-js` et le test de frameworks Solidity tiers) qui nécessitent la compilation de la cible Emscripten.

Écrire et exécuter des tests de syntaxe

Les tests de syntaxe vérifient que le compilateur génère les messages d'erreur corrects pour le code invalide et accepte correctement le code valide. Ils sont stockés dans des fichiers individuels à l'intérieur du dossier `tests/libsolidity/syntaxTests`. Ces fichiers doivent contenir des annotations, indiquant le(s) résultat(s) attendu(s) du test respectif. La suite de tests les compile et les vérifie par rapport aux attentes données.

Par exemple : `./test/libsolidity/syntaxTests/double_stateVariable_declaration.sol`

```

contract test {
    uint256 variable;
    uint128 variable;
}
// ----
// DeclarationError: (36-52): Identifiant déjà déclaré.

```

Un test de syntaxe doit contenir au moins le contrat testé lui-même, suivi du séparateur `// ----`. Les commentaires qui suivent le séparateur sont utilisés pour décrire les erreurs ou les avertissements attendus du compilateur. La fourchette de numéros indique l'emplacement dans le code source où l'erreur s'est produite. Si vous voulez que le contrat compile sans aucune erreur ou avertissement, vous pouvez omettre le séparateur et les commentaires qui le suivent.

Dans l'exemple ci-dessus, la variable d'état `variable` a été déclarée deux fois, ce qui n'est pas autorisé. Il en résulte un `DeclarationError` indiquant que l'identifiant a déjà été déclaré.

L'outil `isoltest` est utilisé pour ces tests et vous pouvez le trouver sous `./build/test/tools/`. C'est un outil interactif qui permet d'éditer les contrats défailants en utilisant votre éditeur de texte préféré. Essayons de casser ce test en supprimant la deuxième déclaration de `variable` :

```

contract test {
    uint256 variable;
}
// ----
// DeclarationError: (36-52): Identifiant déjà déclaré.

```

Lancer `./build/test/tools/isoltest` à nouveau entraîne un échec du test :

```

syntaxTests/double_stateVariable_declaration.sol: FAIL
  Contract:
    contract test {
      uint256 variable;
    }

  Expected result:
    DeclarationError: (36-52): Identifiant déjà déclaré.
  Obtained result:
    Success

```

`isoltest` imprime le résultat attendu à côté du résultat obtenu, et fournit aussi un moyen de modifier, de mettre à jour ou d'ignorer le fichier de contrat actuel, ou de quitter l'application.

Il offre plusieurs options pour les tests qui échouent :

- `edit` : `isoltest` essaie d'ouvrir le contrat dans un éditeur pour que vous puissiez l'ajuster. Il utilise soit l'éditeur donné sur la ligne de commande (comme `isoltest --editor /path/to/editor`), dans la variable d'environnement `EDITOR` ou juste `/usr/bin/editor` (dans cet ordre).
- `update` : Met à jour les attentes pour le contrat en cours de test. Cela met à jour les annotations en supprimant les attentes non satisfaites et en ajoutant les attentes manquantes. Le test est ensuite exécuté à nouveau.
- `skip` : Ignore l'exécution de ce test particulier.
- `quit''` : Quitte `isoltest`.

Toutes ces options s'appliquent au contrat en cours, à l'exception de `quit` qui arrête l'ensemble du processus de test.

La mise à jour automatique du test ci-dessus le change en

```

contract test {
    uint256 variable;
}

```

(suite sur la page suivante)

(suite de la page précédente)

```
}
// ----
```

et relancez le test. Il passe à nouveau :

```
Ré-exécution du cas de test...
syntaxTests/double_stateVariable_declaration.sol: OK
```

Note : Choisissez un nom pour le fichier du contrat qui explique ce qu'il teste, par exemple « double_variable_declaration.sol ». Ne mettez pas plus d'un contrat dans un seul fichier, sauf si vous testez l'héritage ou les appels croisés de contrats. Chaque fichier doit tester un aspect de votre nouvelle fonctionnalité.

3.36.5 Exécution du Fuzzer via AFL

Le fuzzing est une technique qui consiste à exécuter des programmes sur des entrées plus ou moins aléatoires afin de trouver des états d'exécution exceptionnels (défauts de segmentation, exceptions, etc.). Les fuzzers modernes sont intelligents et effectuent une recherche dirigée à l'intérieur de l'entrée. Nous avons un binaire spécialisé appelé `solfuzzer` qui prend le code source comme entrée et échoue chaque fois qu'il rencontre une erreur interne du compilateur, un défaut de segmentation ou similaire. mais n'échoue pas si, par exemple, le code contient une erreur. De cette façon, les outils de fuzzing peuvent trouver des problèmes internes dans le compilateur.

Nous utilisons principalement [AFL](#) pour le fuzzing. Vous devez télécharger et installer les paquets AFL depuis vos dépôts (`afl`, `afl-clang`) ou les construire manuellement. Ensuite, construisez Solidity (ou juste le binaire `solfuzzer`) avec AFL comme compilateur :

```
cd build
# if needed
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-gcc -DCMAKE_CXX_COMPILER=path/to/afl-g++
make solfuzzer
```

À ce stade, vous devriez pouvoir voir un message similaire à celui qui suit :

```
Scanning dependencies of target solfuzzer
[ 98%] Building CXX object test/tools/CMakeFiles/solfuzzer.dir/fuzzer.cpp.o
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 1949 locations (64-bit, non-hardened mode, ratio 100%).
[100%] Linking CXX executable solfuzzer
```

Si les messages d'instrumentation n'apparaissent pas, essayez de changer les drapeaux `cmake` pointant vers les binaires `clang` de l'AFL :

```
# si l'échec précédent
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-clang -DCMAKE_CXX_COMPILER=path/to/afl-clang++
make solfuzzer
```

Sinon, lors de l'exécution, le fuzzer s'arrête avec une erreur disant que le binaire n'est pas instrumenté :

```

afl-fuzz 2.52b by <lcamtuf@google.com>
... (truncated messages)
[*] Validating target binary...

[-] Looks like the target binary is not instrumented! The fuzzer depends on
    compile-time instrumentation to isolate interesting test cases while
    mutating the input data. For more information, and for tips on how to
    instrument binaries, please see /usr/share/doc/afl-doc/docs/README.

    When source code is not available, you may be able to leverage QEMU
    mode support. Consult the README for tips on how to enable this.
    (It is also possible to use afl-fuzz as a traditional, "dumb" fuzzer.
    For that, you can use the -n option - but expect much worse results.)

[-] PROGRAM ABORT : No instrumentation detected
    Location : check_binary(), afl-fuzz.c:6920

```

Ensuite, vous avez besoin de quelques fichiers sources d'exemple. Cela permet au fuzzer de trouver des erreurs plus facilement. Vous pouvez soit copier certains fichiers des tests de syntaxe, soit extraire des fichiers de test de la documentation ou des autres tests :

```

mkdir /tmp/test_cases
cd /tmp/test_cases
# extract from tests:
path/to/solidity/scripts/isolate_tests.py path/to/solidity/test/libsolidity/
↳SolidityEndToEndTest.cpp
# extract from documentation:
path/to/solidity/scripts/isolate_tests.py path/to/solidity/docs

```

La documentation de l'AFL indique que le corpus (les fichiers d'entrée initiaux) ne doit pas être trop volumineux. Les fichiers eux-mêmes ne devraient pas être plus grands que 1 kB et il devrait y avoir au maximum un fichier d'entrée par fonctionnalité, donc mieux vaut commencer avec un petit nombre de fichiers. Il existe également un outil appelé `afl-cmin` qui peut couper les fichiers d'entrée qui ont pour résultat un comportement similaire du binaire.

Maintenant, lancez le fuzzer (le `-m` étend la taille de la mémoire à 60 Mo) :

```

afl-fuzz -m 60 -i /tmp/test_cases -o /tmp/fuzzer_reports -- /path/to/solfuzzer

```

Le fuzzer crée des fichiers sources qui conduisent à des échecs dans `/tmp/fuzzer_reports`. Il trouve souvent de nombreux fichiers sources similaires qui produisent la même erreur. Vous pouvez utiliser l'outil `scripts/uniqueErrors.sh` pour filtrer les erreurs uniques.

3.36.6 Moustaches

Whiskers est un système de modélisation de chaînes de caractères similaire à [Mustache](#). Il est utilisé par le compilateur à divers endroits pour faciliter la lisibilité, et donc la maintenabilité et la vérifiabilité, du code.

La syntaxe présente une différence par rapport à *Mustache*. Les marqueurs de template `{{`` et `}}` sont remplacés par `<` et `>` afin de faciliter l'analyse et d'éviter les conflits avec `yul`. (Les symboles `<`` et `>`` sont invalides dans l'assemblage en ligne, tandis que `{` et `}` sont utilisés pour délimiter les blocs). Une autre limitation est que les listes ne sont résolues qu'à une seule profondeur et qu'elles ne sont pas récursives. Cela peut changer dans le futur.

Une spécification approximative est la suivante :

Toute occurrence de `<name>` est remplacée par la valeur de la variable fournie `name` sans aucun échappement et sans remplacement itératif. Une zone peut être délimitée par `<#name>...</name>`. Elle est remplacée par autant de concaténations de son contenu qu'il y avait d'ensembles de variables fournis au système de modèles, en remplaçant chaque fois les éléments `<inner>` par leur valeur respective. Les variables de haut niveau peuvent également être utilisées à l'intérieur de ces zones.

Il existe également des conditionnels de la forme `<?name>...<!name>...</name>`, où les remplacements de modèles se poursuivent récursivement dans le premier ou le second segment, en fonction de la valeur du paramètre booléen `name`. Si `<?+name>...<!+name>...</+name>` est utilisé, alors la vérification consiste à savoir si le paramètre chaîne de caractères `name` est non vide.

3.36.7 Guide de style de la documentation

Dans la section suivante, vous trouverez des recommandations de style spécifiquement axées sur la documentation des contributions à Solidity.

Langue anglaise

Utilisez l'anglais, avec une préférence pour l'orthographe anglaise britannique, sauf si vous utilisez des noms de projets ou de marques. Essayez de réduire l'utilisation de l'argot et les références locales, en rendant votre langage aussi clair que possible pour tous les lecteurs. Vous trouverez ci-dessous quelques références pour vous aider :

- L'anglais technique simplifié.
- Anglais international
- L'orthographe de l'anglais britannique

Note : Bien que la documentation officielle de Solidity soit écrite en anglais, il existe des traductions contribuées par la communauté dans d'autres langues. dans d'autres langues sont disponibles. Veuillez vous référer au [guide de traduction](#) pour savoir comment contribuer aux traductions de la communauté.

Cas de titre pour les en-têtes

Utilisez la casse des titres <https://titlecase.com> pour les titres. Cela signifie qu'il faut mettre en majuscule tous les mots principaux dans titres, mais pas les articles, les conjonctions et les prépositions, sauf s'ils commencent le titre.

Par exemple, les exemples suivants sont tous corrects :

- Title Case for Headings.
- Pour les titres, utilisez la casse du titre.
- Noms de variables locales et d'État.
- Ordre de mise en page.

Développer les contractions

Utilisez des contractions développées pour les mots, par exemple :

- « Do not » au lieu de « Don't ».
- « Can not » au lieu de « Can't ».

Voix active et passive

La voix active est généralement recommandée pour la documentation de type tutoriel car elle aide le lecteur à comprendre qui ou quoi effectue une tâche. Cependant, comme la documentation de Solidity est un mélange de tutoriels et de contenu de référence, la voix passive est parfois plus appropriée.

En résumé :

- Utilisez la voix passive pour les références techniques, par exemple la définition du langage et les éléments internes de la VM Ethereum.
- Utilisez la voix active pour décrire des recommandations sur la façon d'appliquer un aspect de Solidity.

Par exemple, le texte ci-dessous est à la voix passive car il spécifie un aspect de Solidity :

Les fonctions peuvent être déclarées « pures », auquel cas elles promettent de ne pas lire ou de modifier l'état.

Par exemple, le texte ci-dessous est à la voix active car il traite d'une application de Solidity :

Lorsque vous invoquez le compilateur, vous pouvez spécifier comment découvrir le premier élément d'un chemin, ainsi que les remappages de préfixes de chemin.

Termes courants

- « Paramètres de fonction » et « variables de retour », et non pas paramètres d'entrée et de sortie.

Exemples de code

Un processus CI teste tous les exemples de code formatés en blocs de code qui commencent par `» pragma solidity « , » contrat « , » bibliothèque » ou » interface « . ou » interface » en utilisant le script » ./test/cmdlineTests.sh » lorsque vous créez un PR. Si vous ajoutez de nouveaux exemples de code, assurez-vous qu'ils fonctionnent et passent les tests avant de créer le PR.`

Assurez-vous que tous les exemples de code commencent par une version de `pragma` qui couvre la plus grande partie où le code du contrat est valide. Par exemple, `pragma solidity >=0.4.0 <0.9.0;`

Exécution des Tests de Documentation

Assurez-vous que vos contributions passent nos tests de documentation en exécutant `./scripts/docs.sh` qui installe les dépendances nécessaires à la documentation et vérifie les problèmes éventuels. Nécessaires à la documentation et vérifie l'absence de problèmes tels que des liens brisés ou des problèmes de syntaxe.

3.36.8 Conception du langage Solidity

Pour vous impliquer activement dans le processus de conception du langage et partager vos idées concernant l'avenir de Solidity, veuillez rejoindre le [forum Solidity](#).

Le forum Solidity sert de lieu pour proposer et discuter de nouvelles fonctionnalités du langage et de leur mise en œuvre dans les premiers stades de l'idéation ou des modifications de fonctionnalités existantes.

Dès que les propositions deviennent plus tangibles, leur implémentation sera également discutée dans le dépôt [Solidity GitHub](#) sous la forme de questions.

En plus du forum et des discussions sur les problèmes, nous organisons régulièrement des appels de discussion sur la conception du langage dans lesquels des sujets, questions ou implémentations de fonctionnalités sélectionnés sont débattus en détail. L'invitation à ces appels est partagée via le forum.

Nous partageons également des enquêtes de satisfaction et d'autres contenus pertinents pour la conception des langues sur le forum.

Si vous voulez savoir où en est l'équipe en termes d'implémentation de nouvelles fonctionnalités, vous pouvez suivre le statut de l'implémentation dans le projet [Solidity Github](#). Les questions dans le backlog de conception nécessitent une spécification plus approfondie et seront soit discutées dans un appel de conception de langue ou dans un appel d'équipe régulier. Vous pouvez voir les changements à venir pour la prochaine version de rupture en passant de la branche par défaut (*develop*) à la *breaking branch*.

Pour les cas particuliers et les questions, vous pouvez nous contacter via le canal [Solidity-dev Gitter](#), un chatroom dédié aux conversations autour du compilateur Solidity et du développement du langage.

Nous sommes heureux d'entendre vos réflexions sur la façon dont nous pouvons améliorer le processus de conception du langage pour qu'il soit encore plus collaboratif et transparent.

3.37 Guide de la marque Solidity

Ce guide de la marque contient des informations sur la politique de la marque Solidity et les directives d'utilisation du logo.

3.37.1 La marque Solidity

Le langage de programmation Solidity est un projet communautaire à code source ouvert dirigé par une équipe centrale. L'équipe centrale est parrainée par la [Ethereum Foundation](#).

Ce document a pour objectif de fournir des informations sur la meilleure façon d'utiliser la marque et le logo Solidity.

Nous vous encourageons à lire attentivement ce document avant d'utiliser la marque ou le logo. Votre coopération est très appréciée !

3.37.2 Nom de marque Solidity

Le terme « Solidity » doit être utilisé pour faire référence au langage de programmation Solidity uniquement.

Veillez ne pas utiliser « Solidity » :

- Pour faire référence à tout autre langage de programmation.
- D'une manière qui pourrait induire en erreur ou impliquer l'association de modules, d'outils, de documentation ou d'autres ressources sans rapport avec le langage Solidity.
- D'une manière qui sème la confusion dans la communauté quant à savoir si le langage de programmation Solidity est open-source et libre d'utilisation.

3.37.3 Licence du logo Solidity



Le logo Solidity est distribué et mis sous licence [Creative Commons](#)®. [Attribution 4.0 International License](#).

Il s'agit de la licence Creative Commons la plus permissive, qui autorise la réutilisation et les modifications à toutes fins, et les modifications dans n'importe quel but.

Vous êtes libre de :

- **Partager** - Copier et redistribuer le matériel sur tout support ou format.

- **Adapter** - Remixer, transformer et construire à partir de ce matériel dans n'importe quel but, même commercial. dans n'importe quel but, même commercial.

Aux conditions suivantes :

- **Attribution** - Vous devez donner le crédit approprié, fournir un lien à la licence, et indiquer si des modifications ont été apportées. Vous pouvez le faire de toute manière raisonnable, mais pas d'une manière qui suggère que l'équipe centrale de Solidity vous approuve ou approuve votre utilisation.

Lorsque vous utilisez le logo Solidity, veuillez respecter les directives relatives au logo Solidity.

3.37.4 Directives relatives au logo Solidity

(Cliquez avec le bouton droit de la souris sur le logo pour le télécharger.)

Veuillez ne pas :

- Modifier le ratio du logo (ne pas l'étirer ou le couper).
- Modifier les couleurs du logo, sauf si cela est absolument nécessaire.

3.37.5 Crédits

Ce document a été, en partie, dérivé de la [Python Software Foundation](#) sur l'utilisation des marques déposées et du [Guide des médias de Rust](#).

3.38 Influences de la langue

Solidity est un [langage à virgule flottante](#) qui a été influencé et inspiré par plusieurs langages de programmation bien connus.

Solidity est le plus profondément influencé par le C++, mais a également emprunté des concepts à des langages comme Python, JavaScript, et autres.

L'influence du C++ est visible dans la syntaxe des déclarations de variables, les boucles for, le concept de surcharge des fonctions, les conversions de type implicites et explicites et de nombreux autres détails.

Aux premiers jours du langage, Solidity était en partie influencé par JavaScript. Cela était dû à la détermination de la portée des variables au niveau des fonctions et à l'utilisation du mot-clé « var ». L'influence de JavaScript a été réduite à partir de la version 0.4.0. Maintenant, la principale similitude restante avec JavaScript est que les fonctions sont définies en utilisant le mot-clé `function`. Solidity prend également en charge la syntaxe et la sémantique de l'importation qui sont similaires à celles disponibles en JavaScript. En dehors de ces points, Solidity ressemble à la plupart des autres langages à accolades et n'a plus d'influence majeure de JavaScript.

Python a également influencé Solidity. Les modificateurs de Solidity ont été ajoutés en essayant de modéliser les décorateurs de Python avec plus d'efficacité. Les décorateurs de Python avec une fonctionnalité beaucoup plus restreinte. De plus, l'héritage multiple, la linéarisation C3, et le mot-clé « super » sont tirés de Python, ainsi que la sémantique générale de l'assignation et des types de référence.

Symbols

--allow-paths, 153, **276**
--base-path, 153, 154, **274**
--include-path, 153, **274**
--libraries, **153**
--link, **153**
--standard-json, 154, **155**
<stdin>, 272

A

abi, 86, 87, 212
abstract contract, **135**
access
 restricting, 326
account, **12**
addmod, 88, 149
address, 12, 56, 60
allowed paths, 153, **276**
analyse, 169
anonymous, 151
application binary interface, 212
array, 67, **69**, 115
 allocating, **70**
 length, **72**
 literals, **70**
 pop, **72**
 push, **72**
 slice, **74**
array of strings, 115
asm, **144**, 169, **280**
assembly, **144**, **280**
assert, 87, **100**, 149
assignment, 81, **96**
 destructuring, **96**
auction
 blind, 27
 open, 27

B

balance, 12, 56, 89, 149
ballot, 24
base
 constructor, **133**
base class, **126**
base path, 153, **274**
blind auction, 27
block, **11**, 86, 149
 number, 86, 149
 timestamp, 86, 149
bool, **53**
break, 91
Bugs, 331
byte array, 59
bytes, 62, **69**
bytes members, 87
bytes32, 59
bytes-concat, **69**

C

C3 linearization, **134**
call, 56, 89
callcode, 14, 89, 137
cast, **82**
checked, 98
cleanup, 178
codehash, 89, 149
coding style, 303
coin, 10
coinbase, 86, 149
commandline compiler, **152**
comment, **49**
common subexpression elimination, 188
compile target, 154
compiler
 commandline, 152
compound operators, **81**
constant, **112**, 151

- constant propagation, 188
- constructor, 105, **132**
 - arguments, 105
- continue, 91
- contract, 50, **105**
 - abstract, **135**
 - base, **126**
 - creation, **105**
 - interface, **136**
 - modular, 45
 - precompiled, **15**
- contract creation, 15
- contract type, **58**
- contract verification, 209
- contracts
 - creating, 94
- creationCode, 90
- cryptography, 88, 149
- custom type, 63

D

- data, 86, 149
- days, 85
- deactivate, 15
- declarations, 97
- default value, 97
- delegatecall, 14, 56, 89, 137
- delete, **81**
- deriving, **126**
- difficulty, 86, 149
- direct import, **273**
- dirty bits, 178
- do/while, 91
- dynamic array, 115

E

- ecrecover, 88, 149
- else, 91
- encode, 86
- encoding, 87
- enum, 50, 62
- error, **125**
- errors, **100**
- escrow, 34
- ether, 85
- ethereum virtual machine, **12**
- evaluation order
 - expression, **176**
 - function arguments, **176**
- event, 10, 50, **123**
- evm, **12**
- EVM version, **154**
- evmasm, **144, 280**
- exception, **100**

- experimental, 48
- extern, 151
- external, 107

F

- fallback function, **119**
- false, **53**
- file://, 280
- filesystem path, 49
- finney, 85
- fixed, **56**
- fixed point number, **56**
- for, 91
- function, 50
 - call, 14, **91**
 - external, 91
 - fallback, 119
 - getter, **108**
 - internal, 91
 - modifier, 50, **110, 326, 328**
 - pure, 117
 - receive ! receive, 118
 - view, 116
- function parameter, 91
- function pointers, 178
- function type, **64**
- functions, **114**

G

- gas, **13, 86, 149**
- gas price, **13, 86, 149**
- getter
 - function, **108**
- goto, 91
- gwei, 85

H

- Host Filesystem Loader, **270**
- hours, 85

I

- if, 91
- import, **48**
 - direct, 273
 - path, 49, **272**
 - relative, **273**
 - remapping, **277**
- import callback, 49, **270**
- include paths, 153, **274**
- indexed, 151
- inheritance, **126**
 - multiple, **134**
- inline

- arrays, [70](#)
- installing, [15](#)
- instruction, [13](#)
- int, [53](#)
- integer, [53](#)
- interface contract, [136](#)
- intern, [151](#)
- internal, [107](#)
- iterable mappings, [78](#)
- iulia, [280](#)

J

- julia, [280](#)

K

- keccak256, [88](#), [149](#)

L

- length, [72](#)
- library, [14](#), [137](#), [142](#)
- license, [46](#)
- linearization, [134](#)
- linker, [153](#)
- literal, [60–62](#)
 - address, [60](#)
 - rational, [60](#)
 - string, [61](#)
- location, [67](#)
- log, [14](#)
- lvalue, [81](#)

M

- mapping, [9](#), [76](#), [178](#)
- memory, [13](#), [67](#)
- message call, [14](#)
- metadata, [209](#)
- minutes, [85](#)
- modifiers, [151](#)
- modular contract, [45](#)
- module, [48](#)
- msg, [86](#), [149](#)
- mulmod, [88](#), [149](#)

N

- natspec, [49](#)
- new, [70](#), [94](#)
- number, [86](#), [149](#)

O

- open auction, [27](#)
- operator, [80](#)
- optimiser, [188](#)
- optimizer, [188](#)

- origin, [86](#), [149](#)
- overload, [121](#)
- overriding
 - function, [129](#)
 - modifier, [131](#)

P

- packed, [87](#)
- parameter, [91](#)
 - function, [91](#)
 - input, [91](#)
 - output, [91](#)
- payable, [151](#)
- pop, [72](#)
- pragma, [47](#), [48](#)
- precedence, [149](#)
- precompiled contracts, [15](#)
- precompiles, [15](#)
- private, [107](#), [151](#)
- public, [107](#), [151](#)
- purchase, [34](#)
- pure, [151](#)
- pure function, [117](#)
- push, [72](#)

R

- receive ether function, [118](#)
- reference type, [67](#)
- relative import, [273](#)
- remapping
 - context, [277](#)
 - import, [277](#)
 - prefix, [277](#)
 - target, [276](#), [277](#)
- Remix IDE, [49](#), [280](#)
- remote purchase, [34](#)
- require, [87](#), [100](#), [149](#)
- return, [91](#)
- return array, [115](#)
- return string, [115](#)
- return struct, [115](#)
- return variable, [91](#)
- revert, [87](#), [100](#), [125](#), [149](#)
- ripemd160, [88](#), [149](#)
- runtimeCode, [90](#)

S

- safe math, [98](#)
- safemath, [98](#)
- scoping, [97](#)
- seconds, [85](#)
- selector, [141](#), [212](#)
- self-destruct, [15](#)
- selfdestruct, [15](#), [90](#), [149](#)

- send, 56, 89, 149
- sender, 86, 149
- set, 138
- sha256, 88, 149
- solc, **152**
- source file, 48
- source mappings, 187
- source unit, 48
- source unit name, 49, **270**
- spdx, 46
- stack, **13**
- standard input, 272
- standard JSON, **155**, 271
- state machine, 328
- state variable, 50, 178
- staticcall, 56, 89
- stdin, 272
- storage, 12, **13**, 67, 178
- string, 61, **69**, 115
- struct, 50, 67, **75**, 115
- style, 303
- subcurrency, **8**
- super, 149
- switch, 91
- szabo, 85

T

- this, 90, 149
- throw, **100**
- time, 85
- timestamp, 86, 149
- transaction, 11, **12**
- transfer, 56, 89
- true, **53**
- type, 53, 90
 - contract, **58**
 - conversion, **82**
 - function, **64**
 - reference, **67**
 - struct, **75**
 - value, **53**

U

- ufixed, **56**
- uint, **53**
- unchecked, 98
- user defined value type, **63**
- using for, 138, **142**

V

- value, 86, 149
- value type, **53**
- variable
 - return, 91

- variably sized array, 115
- version, 47
- VFS, **270**
- view, 151
- view function, **116**
- virtual filesystem, 49, **270**
- visibility, **107**, 151
- voting, 24

W

- weeks, 85
- wei, 85
- while, 91
- withdrawal, 324

Y

- years, 85
- yul, **280**