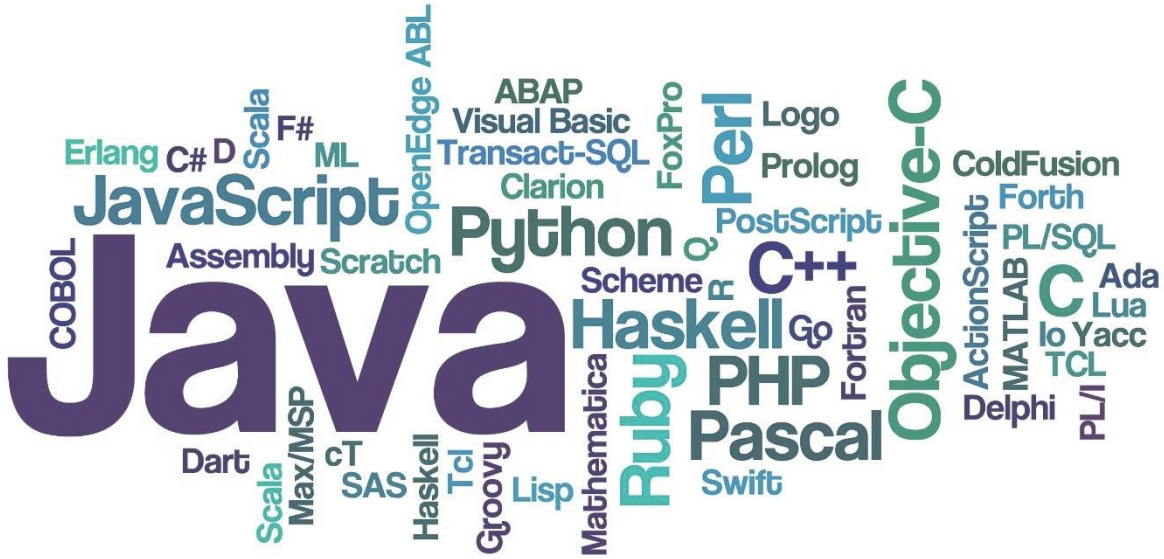


LES LIMITES DES LANGAGES INTERPRÉTÉS

ET LES MOYENS DE LES SURMONTER

QUELLES SONT LES LIMITATIONS ACTUELLES DES LANGAGES INTERPRÉTÉS, ET COMMENT LES SURMONTER ?

Guillaume VILLEREZ, Architecture des logiciels

Maître de mémoire : Frédéric SANANES

5A 2015/2016

RÉSUMÉ

Dans le monde de l'informatique, les langages interprétés ont presque toujours été opposés aux langages compilés traditionnels. Possédant des qualités indéniables, leur utilisation s'est considérablement accrue pour aujourd'hui largement dépasser leurs homologues compilés. Ainsi, 8 des 10 langages de programmation les plus utilisés sont interprétés.

Pourtant cette famille de langages n'est pas exempte de défauts, qui peuvent être inhérents à leur implémentation ou même simplement à la notion même d'interprétation. De nombreuses techniques sont mises en place par les différents interpréteurs et machines virtuelles, avec un but commun : s'affranchir des limitations actuelles, qu'elles soient sécuritaires, d'efficience ou tout simplement de facilité de développement.

La montée en puissance récente du monde du web propose également de nouveaux défis à ces langages en pleine période transitoire ; les anciennes technologies meurent ou s'adaptent afin de gommer toujours plus les limites qui les séparent des langages traditionnels, interprétés ou non.

Ce mémoire tentera traiter avec le plus de recul possible les limitations actuelles des différents langages interprétés, et les différentes réponses possibles à ces problématiques (qu'elles soient déjà utilisées ou simplement à l'état de projet).

MOTS CLEFS

Langage interprété, langage compilé, compilation, compilation ahead of time, compilation just in time, limitation langage interprété, Java, JavaScript, V8, JRE, Web Assembly

ABSTRACT

In the computer world, interpreted languages have almost always been opposed to traditional compiled languages. Possessing undeniable qualities, their use has increased significantly over the last decades, and exceed their compiled counterparts. Thus, 8 of the 10 most used programming languages are interpreted.

Yet this language family is not free of defects, which may be related to their implementation or even just to the notion of interpretation. Many techniques are implemented by the various interpreters and virtual machines, with a common goal: to overcome the current limitations, may they be efficiency, security or even ease of development.

The recent rise in power of the web world also offers new challenges to these languages, which are in full transitional period; old technologies die or changes to adapt themselves for the future, reducing the differences between languages, compiled or not.

This memoir attempt to deal with the current limitations of the different interpreted languages, and the various possible responses to these issues (whether actually used or simply in draft form).

KEYWORDS

Interpreted language, compiled language, compilation, ahead of time compilation, just in time compilation, Interpreted language limitations, Java, JavaScript, V8, JRE, Web Assembly

TABLE DES MATIÈRES

INTRODUCTION

Un **langage interprété** désigne un langage de programmation ne nécessitant pas de phase de compilation préalable avant d'être exécuté. Par extension, les langages compilés en langages plus bas niveau eux même interprétés sont communément considérés comme eux même interprétés (c'est par exemple le cas pour le **C#** ou le **Java**).

Nous considérerons ici que tout langage n'étant pas compilé en code machine natif peut être considéré comme étant « interprété », qu'il s'agisse d'une interprétation directe ou nécessitant préalablement une phase de compilation ou de transpilation. Il convient également de noter qu'il est possible pour un langage d'être à la fois compilé et interprété, puisque le langage n'est en soi qu'une abstraction permettant de définir des instructions. Ici, seules les limites des interpréteurs seront discutées.

On trouve leurs premières origines dans le *scripting*, auxquels ceux-ci répondent parfaitement : ici le besoin de performance est secondaire, contrairement au temps de réalisation et de déploiement qui doit être rapide. Les fonctionnalités sont généralement plus haut niveau et servent à manipuler des commandes ou des bindings dans des langages compilés.

Les langages interprétés ont rapidement été des concurrents sérieux aux autres langages traditionnellement compilés, proposant des couches d'abstractions plus élevées permettant de développer de manière plus simple, rapide et sécurisée des applications. Cependant ils ne sont pas exempts de problèmes, parfois partagés avec leurs homologues compilés, ou même inédits.

POURQUOI LES LANGAGES INTERPRÉTÉS ?

Les **langages interprétés** cherchent principalement à répondre aux problèmes mis en place par leurs homologues compilés, ou bien à des problématiques nouvelles. Là où les langages compilés tendent traditionnellement à être plus bas niveau et plus complexes, afin de chercher la performance, les langages interprétés cherchent au contraire à se défaire de ces contraintes et faciliter la mise en place de la logique de l'application en plaçant le développeur dans une couche d'abstraction plus haute.

Ainsi, les langages interprétés offrent une batterie d'avantages non négligeables :

- Indépendance à la plateforme (ne nécessitant pas de compilation spécifique par architecture), permettant de déployer les mêmes sources quelle que soit la plateforme cible,
- Niveau d'abstraction plus élevé, permettant de se séparer des contraintes d'allocation mémoire ou de manipulations bas niveau,
- Sécurité et contrôle de l'exécution, principalement dans le cadre du web

Ces différents avantages permettent de faciliter la vie du développeur en lui donnant des outils permettant de réaliser plus rapidement et facilement des applications.

Mais ces différents avantages s'accompagnent de limites qui ne leurs permettent pas encore de se substituer partout aux langages dis natifs. En effet, la mise en place de cette couche d'abstraction supplémentaire n'est pas absente de coûts, ou même de problèmes nouveaux apportés par l'interprétation.

PRÉSENTATION DES GRANDS REPRÉSENTANTS

Parmi les grands représentants actuels des langages interprétés, 4 seront activement mentionnés au travers de ce document.

Nous avons choisi :

- **Java**, car il s'agit sans contestation possible d'un des langages de programmation les plus utilisés, se plaçant en 2015 premier au classement TIOBE[1] ; en plus de proposer une machine virtuelle profitant de nombreuses optimisations poussées,
- **C#**, car tout en proposant une solution similaire au Java, il y répond de manière différente. Sa large adoption sur les plateformes Microsoft et la récente volonté de celui-ci de l'ouvrir en font un acteur incontournable,
- **JavaScript**, car il s'agit d'un des seuls représentant du monde du web. L'arrivée d'HTML 5 et sa démocratisation au travers de **NodeJS** ont achevé d'en faire un incontournable des langages interprétés,
- **Python**, car il s'agit d'un langage polyvalent largement utilisé, proposant de nombreuses fonctionnalités et paradigmes.

JAVA & JVM

Le **Java,** et sa **JVM,** lui fournissant à la fois son interpréteur (une fois compilé en bytecode) et son environnement d'exécution, semblent être un choix obligatoire lorsqu'on parle de langages interprétés. En effet **Java** est le langage de programmation le plus utilisé actuellement pour les projets informatiques en 2015, en plus d'être utilisé sur de nombreux systèmes embarqués ou mobiles aux architectures disparates.

Java a été lancé comme en 1991 projet (portant alors le nom « Oak », soit chêne en français) par James Gosling, Mike Sheridan, et Patrick Naughton, alors tous ingénieurs chez **Sun Microsystems**.

Il s'agissait à l'origine de créer un langage et son environnement d'exécution, prévu pour de petits systèmes embarqués, principalement des dispositifs de boîtiers décodeurs à destination du câble. Le langage devait reprendre la syntaxe du **C/C++** afin de ne pas perdre les développeurs existants. Trop en avance sur son temps, le projet changea de cap pour être utilisé dans un navigateur proposant des fonctionnalités novatrices en son temps (1994) avec des animations dans les pages web[2].

La première version de Java sort en 1995 avec le slogan « Écrire une fois, exécuter partout » (« Write Once, Run Anywhere » en original), permettant de déployer du code quel que soit la plateforme, et ce de manière sécurisée en contrôlant les accès disque et réseau. Plus tard, dans sa seconde version, les

[1] TIOBE Index for December 2015 : http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[2] History of Java : http://gcc.uni-paderborn.de/www/WI/WI2/wi2_lit.nsf/64ae864837b22662c12573e70058bbb4/abf8d70f07c12eb3c1256de900638899/$FILE/Java%20Technology%20-%20An%20early%20history.pdf

déclinaisons **Java EE**, **ME** et **SE** apparaissent, permettant de mettre en place de nombreux types de logiciels avec la même plateforme Java.

En 2007 la **JVM** devient GPL, et son code source est librement accessible.

Java propose différentes éditions, regroupant différents framework permettant des spécialisations dans les applications lourdes et les applications serveurs ; ayant chacune leurs spécificités.

```java
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Say hello world
    }
}
```

Exemple de « hello world » en Java.

Résolument haut niveau, le **Java** propose un typage statique fort, une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exception. Le langage présente une philosophie extrêmement liée au paradigme objet, présentant un système de classes, d'héritage et d'interface au cœur de son fonctionnement. Le langage est souvent réputé comme verbeux, affichant volontairement sa volonté de rester simple et de ne pas omettre des informations dans son code.

Il convient de noter que dans sa dernière version **Java** propose de nouveaux outils inspirés du monde de la programmation fonctionnelle, montrant que le langage cherche encore aujourd'hui à s'améliorer et à suivre les tendances actuelles.

PYTHON

Python est un langage portable, dynamique, extensible et orienté objet qui a été développé en 1989 par Guido van Rossum à l'Institut national de recherches mathématiques et informatique de Hollande (Le CWI). Le nom « Python » est un hommage à la troupe de comiques les « Monty Python ».

En 1991 sort la première version publique du langage, qui sera par la suite largement utilisée et gagnera en popularité. En 2003, **Python** 3.0 sort et casse la compatibilité avec les anciennes versions, et ce afin de nettoyer la bibliothèque standard et de corriger certains concepts du langage. Malgré tout, la version 2 reste encore aujourd'hui largement utilisée et est encore maintenue et développée en parallèle de la version 3.

```python
# Python2
print "Hello, World!"

# Python 3
print("Hello, World!")
```

Exemple de « hello world » en Python 2 et 3.

Il s'agit d'un langage doté d'un typage dynamique fort, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exception. Même si orienté objet, **Python** est multi-paradigme favorisant la programmation impérative, fonctionnelle et bien sûr orienté objet. Il propose en conséquence de nombreux outils et facilités d'écriture permettant de facilement embrasser ces paradigmes. Il se démarque par sa syntaxe utilisant les tabulations et espaces comme éléments structurants, par opposition aux accolades popularisées par le **langage C**.

Le langage **Python** utilise également beaucoup le binding avec le **langage C**, utilisé pour écrire son interpréteur. Il permet ainsi de facilement réutiliser des traitements ou routines bas niveau écris en **C** (ou dans d'autres langages), ce qui a contribué à sa popularité actuelle.

Python peut être utilisé comme langage de scripting ou pour des applications plus avancées. On l'utilise souvent pour des démonstrations de prototype de par sa rapidité de développement. Il est également très utilisé dans le monde scientifique et est depuis 2013 enseigné en France dans les classes préparatoires scientifiques au côté de **Scilab**[3].

Il est souvent recommandé pour l'apprentissage du développement, à cause de sa syntaxe claire et de son niveau d'abstraction permettant de se défaire des mécanismes bas niveau et de se concentrer uniquement sur l'apprentissage des concepts de base de la programmation.

JAVASCRIPT

JavaScript est historiquement un langage de programmation de scripts. Il a été créé en 10 jours en mai 1995 par Brendan Eich pour la société **Netscape Communications Corporation**.

Le but premier du langage était de proposer un langage côté client, afin de rendre dynamique les sites web. Le nom du langage est une référence au **Java**, alors gagnant en popularité, afin que les deux langages profitent de la popularité de l'un et de l'autre bien qu'ils ne soient pas réellement liés.

La première standardisation du **JavaScript** date de la sortie d'**ECMAScript** en décembre 1995 par **Ecma International**, fixant le langage. **ECMAScript** est un langage de script qui forme la base de **JavaScript**, qui en est une implémentation. On trouve ainsi d'autres implémentations comme l'**ActionScript** de **Flash** implémentant également **ECMAScript**.

Le **JavaScript** est un langage orienté objet à prototype (les bases du langage et ses principales interfaces sont fournies par des objets qui ne sont pas des instances de classes, mais qui sont chacun équipés de constructeurs permettant de créer leurs propriétés), proposant des outils haut niveau comme une gestion automatique de la mémoire ou un ramasse-miette. Il propose également un fonctionnement asynchrone par évènement, différents de ce que proposent généralement les autres langages comme le **Java** ou le **Python**.

[3] Sujets de Supélec contenant du Python depuis 2013 : http://www.concours-centrale-supelec.fr/CentraleSupelec

```
console.log("Hello world");
```

Exemple de « hello world » en JavaScript.

Il s'agit principalement d'un langage web côté client dont l'exécution prend place dans le navigateur en lui-même. On trouve également de plus en plus d'autres applications aux JavaScript, nettement avec la popularisation de **NodeJS** permettant de créer des serveurs ou même des applications graphiques uniquement en JavaScript. Enfin, l'arrivée d'**HTML 5** et la disparition progressive de **Flash** et des applets **Java** en ont fait un langage incontournable.

On peut citer 3 grands moteurs JavaScript :

- **SpiderMonkey** est le premier moteur JavaScript, il a été créé par l'informaticien américain Brendan Eich pour le navigateur Netscape Navigator, programmé entièrement en **langage C**. Il est maintenant maintenu par la Fondation Mozilla. Il comprend compilateur, interpréteur, ramasse-miettes, ainsi que des classes standards. Celui-ci passe par un *bytecode* intermédiaire afin d'exécuter les instructions.
- **V8** a été développé par **Google** au Danemark. Il est principalement utilisé par les navigateurs **Chromnium** et **Google Chrome**, ainsi que sur **Node.js**.
- **Charkra** a été développé par Microsoft pour Internet Explorer 9. Il équipe depuis tous les navigateurs **Microsoft** (IE 9, 10, 11 et Edge), et a été publié sous licence MIT sur **GitHub** en 2016.

C#

Le **C#** est dévoilé en 2000 par **Microsoft** avec l'implication de Abders Hejlsberg, le créateur du **Delphi**. Celui-ci sera commercialisé en 2002 et est destiné à être développé sur la plateforme **Microsoft .NET**.

C# est un des différents langages destinés à être exécuté dans le **Common Langage Infrastructure** (ou **CLI**) de **Microsoft**, qui peut être comparé à la **JVM** d'**Oracle** pour le **Java**, servant d'interpréteur, de compilateur et d'environnement d'exécution sécurisé. De nombreux autres langages de l'écosystème **Microsoft** fonctionnent également sur le **CLI**, on peut par exemple citer le **VB.Net** ou encore le **F#**.

Il est dérivé du **C++** et très proche du **Java** dont il reprend de nombreux concepts, en y ajoutant des notions plus bas niveau ou emprunté à d'autres paradigmes. Ainsi il permet par exemple de manipuler des pointeurs de mémoire de manière non-sécurisée, chose impossible directement en Java.

Il propose les grandes fonctionnalités habituelles des langages haut niveau, avec une gestion automatique de la mémoire, un ramasse miettes et une bibliothèque de base couvrant de nombreuses fonctionnalités.

C# est comme donc **Java** un langage généraliste orienté objet, permettant de développer des applications web (avec ASP.NET), des applications graphiques, ou même des applications mobiles.

```
using System;
namespace HelloWorld {
    class Hello {
        static void Main() {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Exemple de « hello world » en C#.

Contrairement aux autres langages présentés ici, il n'est pas destiné au multiplateforme (uniquement l'architecture peut changer), **Microsoft** ne proposant que des implémentations fonctionnant sur ses propres systèmes d'exploitation

Cependant il existe des initiatives libres comme **Mono,** visant à permettre de l'utiliser sur d'autre plateformes en proposant une implémentation libre et multiplateforme du framework **.Net** (comprenant le **CLI** permettant son exécution).

LA COMPILATION ET L'INTERPRÉTATION

La compilation désigne le passage d'un langage haut niveau à un langage de plus bas niveau. D'une manière générale cette opération est effectuée en plusieurs tâches, ayant chacune un rôle précis :

- **Analyse lexicale**, qui analyse les différents éléments du code source (*token*).
- **Analyse syntaxique**, qui vérifie que les différents éléments du code source (*tokens* lus à l'étape d'analyse lexicale) sont correctement agencés entre eux.
- **Analyse sémantique**, qui vérifie que les différents éléments du code source ont un sens. C'est à cette étape que le résultat de l'analyse syntaxique est utilisé pour former un arbre syntaxique du langage.
- **Génération de code intermédiaire**, où l'arbre syntaxique est transformé dans un langage plus bas niveau généralement indépendant du langage source.
- **Optimisation du code intermédiaire**, qui permet de réaliser toutes les tâches liées à l'analyse statique. Cette étape peut être soit absente, soit extrêmement poussée comme c'est le cas avec les langages compilés traditionnels comme le **C** ou le **C++**.
- **Génération de code**, où le code intermédiaire est transformé dans le langage cible du compilateur (généralement un bytecode ou directement de l'assembleur). À noter que dans le cas d'interpréteurs le code est directement interprété, et ce parfois sans génération de code plus bas niveau.

Il convient également de noter que ces étapes peuvent être itératives (analyse lexicale, puis syntaxique, etc…) ; soit « ligne par ligne » comme c'est le cas pour les consoles interactives qu'on peut trouver en **Python** ou en **JavaScript** (auquel cas, le procédé est répété ligne par ligne).

Il existe deux grands types de compilation. L'une **prévisionnelle**, et l'autre « **juste à temps** ». La première se réalise avant que l'utilisateur final ai besoin de l'application, et la seconde durant le fonctionnement même de l'application (dans son interpréteur ou sa machine virtuelle) où les opérations de compilation sont réalisées à la volée et uniquement sur des portions de l'application.

COMPILATION

COMPILATION AHEAD-OF-TIME (AOT)

La **compilation anticipée** (**Ahead of time**, ou **AOT**, en anglais) désigne les mécanismes de compilation prévisionnelle, réalisés avant l'utilisation du code en lui-même. La cible de compilation peut être dépendante du système, comme c'est généralement le cas des compilateurs **C/C++**, ou indépendant, comme c'est le cas du **Java** transformé en *bytecode* ou du **C#** transformé en **CLI**.

TRANSPILATION

On peut également parler de **compilation anticipée** dans le cas où un langage est simplement optimisé avec pour cible le même langage utilisé en source, ou bien vers un autre langage également de haut niveau. On parle alors de « transpilation », ou de compilation source à source. On trouve par exemple **TypeScript** de **Microsoft**[4] (*subset* plus strict de **JavaScript**, possédant un typage statique), compilable

[4] Typescript site web : http://www.typescriptlang.org/

en **JavaScript**. Ce mode de fonctionnement permet par exemple de réutiliser des interpréteurs ou compilateurs existants (en profitant de leurs avantages), ou d'assurer une plus grande compatibilité. Ainsi, ce mécanisme est par exemple utilisé avec les code en **JavaScript ECMAScript 6**[5], permettant de les faire exécuter sur des interpréteurs conçus pour la version **5**, et ce en transpilant les sources vers l'ancienne version.

Il est également possible d'effectuer des passes de minifications ou d'optimisations ayant pour cible le même langage source. Ainsi, **Google** a mis en ligne un **Closure Compiler** permettant d'effectuer des optimisations statiques au **JavaScript**. Ces techniques concernent principalement les langages ne bénéficiant pas de « réelle » compilation **AOT**, comme c'est le cas pour le **JavaScript**. Il ne s'agît ici pas à proprement parler de la compilation, mais plus d'une passe supplémentaire d'optimisation du code avant la mise en production. L'ensemble de ces mécanismes seront abordés dans leurs parties respectives.

CONSÉQUENCES D'UNE COMPILATION AOT

Dans le cas où la phase de compilation anticipée produit du code natif, le binaire sera généralement exécutable directement sur la plateforme cible. La compilation dans un code intermédiaire nécessitera quant-à-elle soit une nouvelle phase de compilation (produisant du code machine), soit un logiciel externe permettant l'interprétation des instructions (typiquement une machine virtuelle). On retrouve ce mécanisme par exemple pour le **Java** ou le **C#**.

D'une manière générale, la compilation **AOT** permet d'effectuer des optimisations statiques relativement poussées, puisque la consommation de ressources liée à la compilation n'est pas un facteur limitant comme c'est le cas dans les mécanismes de compilation « juste à temps » où ce coût de compilation est un facteur clef. Les optimisations les plus « agressives » sont généralement proposées par les langages compilés en code natif.

Une autre conséquence d'une compilation **AOT** (hors transpilation) est que son résultat est soit lié à une architecture précise (cas du **C/C++**), soit nécessitant une nouvelle phase de compilation ou une machine virtuelle pour être exécuté.

COMPILATION JUST-IN-TIME (JIT)

La **compilation juste à temps** (**Just In Time**, ou **JIT**, en anglais) regroupe les mécanismes de compilation mis en place uniquement quand ceux-ci sont nécessaires ou présentant un intérêt au déroulement du programme. Ainsi, à la différence de la compilation **AOT**, ce mécanisme se déroule généralement juste avant le lancement, ou durant l'exécution même du code, et ce directement dans la machine virtuelle ou l'interpréteur. Une compilation **JIT** peut donc intervenir à deux moments : au lancement du programme, ou durant son fonctionnement. À noter que les deux (**AOT** et **JIT**) peuvent prendre place pour un même langage.

[5] Babel JS : https://babeljs.io/

AU LANCEMENT DU PROGRAMME

Le mécanisme de compilation **JIT** peut prendre place au lancement du programme, où le code (intermédiaire ou non) est compilé en code natif sur la machine cible, permettant d'avoir (dans la théorie) des performances semblables à une application compilée nativement du type **C/C++**, minus le temps de compilation en lui-même.

Le **C#** (sur Windows, avec le **CLR**) peut ainsi d'être utilisé dans un mode (compilation **pre-JIT** utilisant **NGEN**) qui force l'application à être entièrement compilée en code natif à son premier lancement[6]. Ainsi, le surcoût du premier lancement se rattrape sur tous les lancements suivants (plus rapides, puisque natifs et n'ayant pas de phase de compilation préalable) et l'utilisation de l'application en elle-même (puisqu'elle est déjà compilée).

DURANT LE FONCTIONNEMENT DU PROGRAMME

D'une manière différente, la compilation **JIT** durant le fonctionnement du programme permet de ne compiler que ce qui est actuellement nécessaire ou rentable au programme. Ce mécanisme peut entrer en action soit à l'appel d'une partie qui n'a pas encore été traitée (comme c'est le pour le **CLR** en **C#**, dans son mode normal ou économique)[7], ou bien si le coût de compilation devient rentable (comme c'est le cas pour la **JVM** d'Oracle)[8], c'est-à-dire si le coût d'une compilation additionné à celui du temps de fonctionnement est inférieur à celui du fonctionnement sans compilation.

Des opérations d'optimisation peuvent donc également prendre place durant le *runtime*, et même mettre en place ce qu'on appelle de l'optimisation dynamique qui se base sur des profils de fonctionnement du logiciel. Ainsi, il est possible pour la machine virtuelle de « deviner » la manière dont seront utilisées certaines méthodes ou parties de code et de les optimiser en conséquence lors de ces phases de compilation.

MISE EN CACHE

Le résultat de ces compilations **JIT** peut être mis en cache pour des lancements ultérieurs, permettant de ne pas avoir à recompiler les parties de code ayant déjà été compilées. Ce fonctionnement permet ainsi techniquement de n'avoir à compiler qu'une seule fois le programme ; soit d'une traite (dans le cas des compilations **JIT** au lancement du programme), soit en autant de fois que nécessaire (dans le cas des compilations **JIT** prenant place au *runtime* de l'application).

Cette technique de mise en cache était par exemple utilisée par **Google** sur les versions d'**Android** antérieures à 4.4 : le code des applications était alors interprété dans une machine virtuelle, et compilé quand nécessaire. Ces parties compilées étaient mises en cache et utilisées aux lancements suivants, permettant de profiter des différentes optimisations déjà utilisées au lancement précédant.[9]

---

[6] Article sur NGEN : https://msdn.microsoft.com/fr-fr/library/6t9t5wcf(v=vs.110).aspx

[7] Understanding .NET Just-In-Time Compilation : http://www.telerik.com/blogs/understanding-net-just-in-time-compilation

[8] Understanding Java JIT Compilation : http://www.oracle.com/technetwork/articles/java/architect-evans-pt1-2266278.html

[9] ART and Dalvik : https://source.android.com/devices/tech/dalvik/index.html

CONSÉQUENCES D'UNE COMPILATION JIT

La principale conséquence d'une compilation **JIT** est une amélioration des mécanismes classiques d'interprétation. Ainsi, cet ensemble de techniques essai de concilier l'avantage d'une compilation en code natif avec les avantages de l'interprétation. Elle peut donc prendre place sur n'importe quelle architecture possédant une machine virtuelle/interpréteur compatible, et permet théoriquement d'améliorer les performances sans opération de la part du développeur ou de l'utilisateur (puisque la phase de compilation se déroule dans l'interpréteur ou la machine virtuelle).

Elle permet également de mettre en place des mécanismes d'optimisations dynamiques, permettant au code compilé ou recompilé dynamiquement de s'adapter au profil d'utilisation de l'application.

INTERPRÉTATION

L'interprétation désigne d'une manière générale l'acte pour un logiciel de lire un fichier contenant des instructions (généralement d'un langage informatique « Turing-complet »), et d'exécuter celles-ci. Ces instructions peuvent être directement écrites par un développeur, ou bien provenir d'une génération automatique (transpilation, compilation ou génération de code).

L'interprétation prend généralement place au sein d'un environnement d'exécution, contenant soit un interpréteur ou une machine virtuelle.

CARACTÉRISTIQUES

COUCHE D'ABSTRACTION ÉLEVÉE

L'interprétation permet d'opérer généralement sur une couche d'abstraction plus élevée, puisque l'exécution se passe en elle-même dans un logiciel et non pas directement sur le matériel physique. Ainsi, les opérations n'étant pas utiles à la logique de l'application sont généralement simplifiées (allocation mémoire automatique, accès aux fichiers, accès réseaux, etc…), permettant de proposer des interfaces simplifiées aux développeurs.

En plus de théoriquement faciliter et accélérer le développement, cette couche d'abstraction permet d'éviter les erreurs humaines et de sécuriser ces interactions.

INDÉPENDANCE À LA PLATEFORME

Offrant un environnement indépendant de la plateforme d'exécution, puisque virtuellement exécuté dans l'interpréteur ou la machine virtuelle, les langages interprétés n'ont pas de dépendance vis-à-vis de l'architecture et du système d'exploitation où leur exécution prend place.

C'est donc ici le rôle de l'environnement d'exécution de faire la liaison entre le système hôte et l'interface unifiée proposée par le langage. Ainsi un code identique, même si utilisant des composants dépendants du système (opération sur des fichiers, opération réseau, etc…), doit pouvoir rester le même quel que soit le système hôte. Cela permet par exemple d'économiser des coûts de développement et de réaliser plus facilement des applications compatibles sur différentes plateformes ou système d'exploitation.

Seul l'interpréteur ou la machine virtuelle, dans son implémentation, dépend de la plateforme ou du système d'exploitation cible.

SÉCURITÉ

L'interpréteur ou la machine virtuelle propose également un environnement d'exécution sécurisé, permettant d'exécuter des instructions sans risque de comportements indéfinis en cas d'erreur du développeur ou d'action malveillante, comme cela peut être le cas dans des langages plus bas niveau.

Ainsi, toutes les opérations de gestion de la mémoire sont directement effectuées par la machine virtuelle ou son environnement d'exécution, évitant les causes les plus communes de failles de sécurité ou de « plantage » d'application (type *buffer overflow*). On trouve également parfois des systèmes de contrôles sur les instructions exécutées elles-mêmes, afin par exemple d'éviter d'autoriser des accès à des méthodes qui serait volontairement inaccessibles ou plus simplement des plantages de la machine virtuelle elle-même.

On trouve ainsi par exemple en **Java** des opérations de vérification sur chaque instruction de *bytecode* exécutée, vérifiant par exemple qu'une instruction de saut (*jump*) ne peut être effectuée qu'a l'intérieur même d'une méthode[10], ou bien des vérifications d'accès aux méthodes privées et ce durant le fonctionnement même du programme.

Cette couche de sécurité peut aussi se traduire par du « sandboxing », qu'on pourrait traduire par exécution en environnement restreint. Ce *sandboxing* sert généralement à exécuter en toute sécurité du code de provenance inconnue, et ce en ne lui donnant qu'un accès limité ou contrôlé aux ressources locales de la machine.

On retrouve beaucoup ce comportement dans le monde du web. C'est par exemple le cas de **Silverlight**, des **Applet Java** ou même du **JavaScript**. Étant exécuté depuis un navigateur internet prenant en entrée des sources non contrôlées, ces mécanismes d'interprétation doivent « cloisonner » de manière efficace le code exécuté pour empêcher toute action malveillante.

Il existe aussi parfois d'autres mécanismes internes aux langages permettant de mettre en place des politiques de sécurité pour différents modules utilisés par une application, et dont la provenance ne serait pas certifiée. Il est par exemple possible en **Java**, par l'intermédiaire d'un fichier « policy », de définir des règles de comportements pour une partie d'une application[11].

OPTIMISATION ET COMPILATION À LA VOLÉE

Ne possédant pas de phase de compilation à proprement parler, un interpréteur ou une machine virtuelle ne peut pas effectuer ses optimisations avant le lancement de l'application. Celui-ci doit donc effectuer ses tâches d'optimisations directement durant le déroulement du code, ou juste avant son fonctionnement (à l'intérieur de l'interpréteur/machine virtuelle en lui-même, une fois que celui-ci est lancé). D'une manière générale ces optimisations n'existent que durant le fonctionnement de

[10] Instruction bytecode Ifnull : http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html#jvms-6.5.ifnull

[11] Oracle Java Tutorial : https://docs.oracle.com/javase/tutorial/security/tour1/

l'application, et disparaissent une fois celle-ci terminée ; il reste néanmoins possible de les mettre en cache afin d'un profiter à un prochain lancement.

Il convient également de noter la mise en place d'optimisations dynamiques qui n'auraient pas été possibles dans le cadre d'une compilation prévisionnelle, puisque tirant partie des informations recueillis pendant le fonctionnement de l'application.

LES DIFFÉRENTS TYPES D'INTERPRÉTATION

On peut différencier les machines virtuelles des interpréteurs. Bien que les deux possèdent les points communs évoqués précédemment, leurs fonctionnements internes ainsi que leurs fonctionnalités ne sont pas exactement les mêmes.

MACHINE VIRTUELLE

CARACTÉRISTIQUES

Une **machine virtuelle** est un environnement d'exécution virtuel contenant un ensemble défini d'opérations atomiques indépendantes de la machine hôte (et de son architecture interne). Ce jeu d'instruction est analogue à celui d'un processeur, de par son niveau d'exécution et son fonctionnement. Ainsi, une instruction dépend uniquement du contexte courant de la machine virtuelle (pile ou registre) et non pas des précédentes instructions. Ces instructions sont généralement des suites binaires semblables à de l'assembleur, et sont souvent mentionnées comme *bytecode*, ou code binaire. Elles sont néanmoins de plus haut niveau que de l'assembleur et peuvent contenir des informations supplémentaires (permettant par exemple de la réflexivité).

Dans la théorie, l'unique différence entre du bytecode et de l'assembleur réside dans le fait que pour chaque opération l'un exécute une « routine » dans le programme de la machine virtuelle alors que l'autre « l'envoi » au processeur. En pratique, de nombreuses différences existent et les langages des machines virtuelles sont généralement beaucoup plus haut niveau, permettant des facilités que l'assembleur ne permet généralement pas.

De par son jeu d'instruction extrêmement basique, une **machine virtuelle** est indépendante des différents langages de programmation. En effet, si un compilateur permet de générer un *bytecode* compatible avec une machine virtuelle, celui-ci est en théorie directement utilisable par la machine virtuelle, et ce quel que soit le langage source.

Par exemple, la **Java Virtual Machine** (ou **JVM**) est, contrairement à ce que son nom indique, techniquement compatible avec n'importe quel langage qui serait compilé dans son jeu d'instructions spécifiques. On trouve ainsi d'autres langages comme **Scala**[12] ou **Clojure**[13] (Tous deux des langages fonctionnels), ou même encore des implémentations d'autres langages existants (comme le **JavaScript**, le **Python**, le **PHP**, etc…) ayant des compilateurs permettant de générer du bytecode compatible avec la **JVM**.

[12] Scala - Seamless Java Interop : http://www.scala-lang.org/what-is-scala.html#seamless_java_interop

[13] Clojure – Site web : http://clojure.org/

On trouve également un comportement semblable avec le **C#/F#/VB**, compilés en **CIL** (ou **Common Intermediate Language**), et ensuite utilisés sur le **Common Language Runtime** (**CLR**) de **Microsoft**. Il ne s'agit ni plus ni moins que d'une machine virtuelle, semblable sur le principe à la **JVM** et à son *bytecode*, même si **Microsoft** préfère l'appellation « environnement d'exécution » pour se différencier des concurrents.[14]

Cette inter-compatibilité permet ainsi de dissocier un langage de sa machine virtuelle. Cette dissociation permet en théorie de profiter des bibliothèques d'un langage dans un autre langage[15], ou encore d'écrire un programme en plusieurs langages sans avoir besoin d'un mécanisme quelconque pour faire communiquer les différents modules. Elle permet également de réutiliser des infrastructures logicielles déjà en place et largement déployées, comme pour le **JRE** et sa **JVM**.

DIFFÉRENCE ENTRE STATIQUEMENT TYPÉE ET DYNAMIQUEMENT TYPÉE

Il convient de faire la différenciation entre une **machine virtuelle statiquement typée** et **dynamiquement typée**. L'une connait les différents types utilisés dans le *bytecode*, et seul le choix de l'implémentation pour un type donné constitue une inconnue, alors que l'autre ne connait pas à l'avance les types utilisés (et peut au mieux les deviner).

MACHINE VIRTUELLE STATIQUEMENT TYPÉE

Une **machine virtuelle statiquement typée** connaît à l'avance les tailles de ses types primitifs. Il devient donc plus facile de générer du *bytecode* proche d'un résultat d'une compilation « classique » en assembleur, et d'en approcher virtuellement les performances. C'est par exemple le cas de la **JVM**, qui utilise les types primitifs qu'on retrouve dans le langage **Java** (*double*, *integer*, *float*, *byte*, etc…). Pour le cas des objets, leurs types sont connus à l'avance et seul l'implémentation à utiliser reste à trouver au *runtime* (principe de substitution de **Liskov**, permettant d'avoir plusieurs implémentations redéfinissant des méthodes).

Il est également techniquement « simple » de convertir ce *bytecode* en code natif pour une plateforme donnée, de par ses nombreuses similitudes avec de l'assembleur.

Cette technique a par exemple été adoptée par **Google** à partir d'**Android 4.4** avec le **Android Runtime** (ou **ART**) qui compile en code natif[16] (propre à l'architecture de la machine, généralement *64bit* ou *ARM*) le *bytecode* des applications **Java** ajoutées au téléphone, utilisant un *garbage-collector* et une allocation mémoire en code natif, et non plus par l'intermédiaire d'une machine virtuelle. Ici, le *bytecode* de la machine virtuelle sert uniquement de langage intermédiaire permettant d'assurer l'indépendance à la plateforme.

MACHINE VIRTUELLE DYNAMIQUEMENT TYPÉE

D'une manière différente **une machine virtuelle dynamiquement typée** ne connaît pas à l'avance les différents types d'une valeur, et pire, son type peut même changer en cours de fonctionnement. Ce

---

[14] Is the CLR a Virtual Machine? : http://blogs.msdn.com/b/brada/archive/2005/01/12/351958.aspx

[15] Conversions between Java and Scala collections : http://www.scala-lang.org/docu/files/collections-api/collections_46.html

[16] Comparaison entre Dalvik et ART : https://source.android.com/devices/tech/dalvik/index.html

type de machine virtuelle opère donc à un niveau généralement plus élevé que leurs équivalents statiquement typés, et le *bytecode* s'éloigne donc de l'analogie des jeux d'instructions processeurs. Ce niveau de liberté supplémentaire laissé au développeur (qui peut dynamiquement typer ses valeurs, et se libérer des contraintes des langages « classiques » comme le **C/C++** ou le **Java**) se paie au prix de performances généralement plus faibles, du fait des tâches de vérifications supplémentaires nécessaires.

Par exemple **CPython**, l'implémentation de référence du langage **Python** (réalisé en **C**), utilise une représentation binaire où chaque instruction représente un équivalent de fonction **C**. Il s'agit d'un type de *bytecode* spécifique et de plus haut niveau (ses différentes instructions s'éloignent beaucoup de celle d'un processeur) permettant de gérer le typage dynamique du langage[17].

Il reste néanmoins possible de compiler un langage dynamiquement typé pour fonctionner sur une machine virtuelle statiquement typée en utilisant des opérations de boxing (ajout de métadonnée sur les valeurs) afin de contrôler de manière logicielle les différents changements de valeur des types.

INTERPRÉTEUR

CARACTÉRISTIQUES

Un **interpréteur** fonctionne différemment d'une **machine virtuelle**, et à un plus haut niveau, car celui-ci doit gérer un flux textuel représentant les instructions, dans un langage spécifique, à exécuter. Ce mode de fonctionnement l'empêche d'utiliser le même type de comportement qu'une machine virtuelle, car chaque instruction n'est pas indépendante des autres instructions. Ainsi, il est nécessaire de maintenir un état beaucoup plus complexe lors de l'exécution (portée des variables, déclarations de méthodes et classes, etc…). Il demande aussi à l'**interpréteur** de gérer une étape d'analyse syntaxique où les instructions textuelles sont traduites en opérations compréhensibles par l'**interpréteur**. Un **interpréteur** endosse ainsi généralement la double casquette de compilateur et de machine virtuelle, bien que celui-ci ne respecte par toutes les étapes d'une compilation classique, et n'ai pas le niveau d'abstraction d'une machine virtuelle.

Un **interpréteur** est ainsi fortement lié à son langage et à sa syntaxe, et n'est (généralement) pas utilisable pour un autre langage, contrairement à une machine virtuelle. Il reste néanmoins possible de l'utiliser avec un autre langage si celui-ci est « compilé » à l'aide d'un **transpileur** (voir partie correspondante) dans le langage de l'interpréteur.

Cette dépendance n'apporte cependant pas que des désavantages, et permet par exemple de mettre en place des optimisations poussées propre à son langage.

Par exemple, le **V8** de **Google** est un moteur **JavaScript** extrêmement puissant utilisant des mécanismes avancés de compilation **Just In Time** (ou **JIT**, juste à temps) lui permettant d'avoir des temps d'exécution très faible. Malgré tout, le logiciel ne produit ou ne lit aucun code binaire d'aucune sorte, ne consomme pas de version compilée du langage, et n'utilise aucune représentation interne en

[17] Python internal VM tutorial : https://cs263-technology-tutorial.readthedocs.org/en/latest/

langage intermédiaire. Il s'agit donc ici d'un interpréteur « classique », puisque celui-ci est spécialisé dans un langage cible et procède à toutes ses optimisations durant le *runtime*.

Enfin, il convient de noter que certains langages fonctionnent parfois comme des langages interprétés d'un point de vue externe tout en adoptant un fonctionnement interne de machines virtuelles en transformant du code source en bytecode avant de l'exécuter. Ainsi la machine virtuelle **CPython** est souvent mentionnée comme étant un interpréteur puisque celle-ci en reprend les caractéristiques externes, mais possède comme mentionné précédemment un fonctionnement interne de machine virtuelle.

ENVIRONNEMENT D'EXÉCUTION

On parle souvent d'environnement d'exécution, ou « runtime environment » en anglais, que ce soit pour les langages interprétés ou natifs. Il s'agit simplement du contexte dans lequel le programme est exécuté, regroupant par exemple :

- Les différentes bibliothèques utilisées, soit par le programme, soit directement par le langage pour sa propre exécution. On peut par exemple citer les bibliothèques *built-in* de la **JVM** d'**Oracle**, ou bien la bibliothèque standard du **C++**.
- Les mécanismes d'exécution du langage en lui-même.
- L'ensemble d'outils qui peuvent graviter autour de cette exécution. On pourrait citer l'ensemble du framework **.NET** de **Microsoft** dans le cas du **C#** ou du **VB.NET**, ou du **JRE** d'**Oracle** pour le **Java** (contenant la **JVM**).

Cet environnement d'exécution va toujours contenir une machine virtuelle (ou équivalent) dans le cas de langage interprété. Dans le cas où cet environnement d'exécution exécute du code compilé nativement, par exemple à l'issue d'une compilation **JIT**, l'environnement est toujours présent et assure le même rôle que la machine virtuelle (allocation, sécurité, etc…).

Ainsi le terme « runtime environment » est parfois utilisé comme synonyme de « machine virtuelle », même si en réalité l'un fait partie de l'autre.

LIMITES ACTUELLES

SÉCURITÉ

PROPRIÉTÉ INTELLECTUELLE & PROTECTION DU CODE

Contrairement à ce qu'on pourrait penser, les limites actuelles des langages interprétés ne dépendent pas uniquement de leurs performances ou de leur occupation mémoire, mais également de la propriété intellectuelle du code. Ainsi dans les cas où il est préférable pour une personne ou une organisation de protéger son code et les algorithmes qu'il contient (afin par exemple de garantir la pérennité d'une recette, de justifier des coûts de développement du logiciel en lui-même, ou simplement par soucis de propriété intellectuelle), Il est nécessaire que le logiciel en question (ou du moins ses parties les plus sensibles) ne puisse pas être facilement accessible.

Bien qu'il ne soit pas possible de masquer complétement la logique interne d'un logiciel, la rendre suffisamment complexe et sécurisée permet de lui garantir une certaine pérennité.

Dans le cas des langages interprétés, une version en langage intermédiaire (**Java** ou **C#**), soit une version directement dans le langage source (**Python** ou **JavaScript**), est presque toujours envoyée chez le client final. Ainsi, celui-ci pourrait techniquement décompiler (ou récupérer directement) les sources du logiciel, entraînant des problématiques de sécurité :

- Contournement de mécanismes de vérification de licence du logiciel,
- Création de générateur de code de licence,
- Réutilisation de mécanismes utilisés dans le logiciel, comme des appels à des web-services,
- *Hook* de fonctions à des fins malicieuses,
- Récupération ou réutilisation d'algorithme spécifique, etc…

Les mêmes actions restent atteignables dans des langages compilés, mais demandent plus de ressources et plus de temps, les rendant moins accessibles. Ainsi, un programme réalisé dans un langage compilé aura plus de facilité à « cacher » ses sources.

RÉTRO-INGÉNIERIE

On parle de **retro-engineering** (ou **rétro-ingénierie** en français) quand on cherche à récréer un code source (ou à comprendre un algorithme) en étudiant uniquement son fonctionnement. Ce procédé est généralement utilisé pour des langages compilés en code natif (par exemple, du **C** ou du **C++**), et ce de manière manuelle. Ainsi, on suit généralement le flux d'instruction (généralement assembleur, ou *bytecode*) en essayant de comprendre la logique mise en place.

Ainsi, la retro-ingénierie va permettre à partir d'un code compilé de reconstituer les sources et d'en extraire la logique, afin de la comprendre et le plus souvent de la contourner ou de le réutiliser.

DÉCOMPILATION

On parle de **décompilation** quand le procédé de récréer des sources est effectué de manière automatique par un logiciel tiers. Il s'agit de l'inverse de la compilation : on part d'un langage bas niveau (typiquement *bytecode* ou assembleur) pour arriver à un langage haut niveau. Ce procédé fonctionne très bien avec les langages compilés dans un *bytecode* (comme le **Java** et son *bytecode*, ou

le **C#** et son **CLI**), mais assez mal avec ceux bas niveau : le résultat est souvent peu exploitable et ne reflète pas la logique du code original, permettant au mieux de réutiliser le code généré comme « black box » dans son propre projet.

La décompilation est par exemple si facile, rapide et performante en **Java**, que certains IDE [18] permettent de décompiler automatiquement les fichiers **JAR** de bibliothèques utilisées auxquelles aucune source n'aurait été attachée. Ici, l'unique but est de combler un manque de documentation, permettant au développeur de regarder directement dans le code source reconstitué.

Pour les raisons évoquées précédemment, cette problématique peu devenir très préoccupante avec un logiciel commercial.

LÉGALITÉ

Il est à noter que des procédés comme la **décompilation** ou la **rétro-ingénierie** sont généralement légaux en eux-mêmes tant qu'ils se limitent à un cadre privé et à des buts généralement cadrés[19]. Ainsi, si la décompilation ou la rétro-ingénierie se limitent à de l'apprentissage, leur pratique reste légale.

Malgré tout, ces pratiques sont généralement contraires aux conditions générales d'utilisation du logiciel.

VULNÉRABILITÉ, BUGS ET IMPLÉMENTATIONS

Le fait de passer par un logiciel tiers pour l'exécution de son code peut être une sérieuse limitation et cause de problèmes. Ainsi, confier l'implémentation de certains comportements natifs au système (ou bien libraires livrées avec les « langages » comme c'est le cas en **.NET** ou **Java**) laisse une certaine zone d'incertitude sur l'implémentation qui sera réellement utilisée au moment où l'utilisateur lancera l'application. Cela est particulièrement vrai dans le cas où on utiliserait des fonctionnalités dépendantes du système sur un langage visant du multiplateforme.

Cette zone d'ombre peut apporter des problématiques supplémentaires de sécurité ou de fonctionnement au logiciel, et ce de manière extérieure et indépendante de la manière dont celui-ci aurait été développé.

Ces problématiques ne sont malheureusement pas du seul fait des langages interprétés, puisque les langages compilés dépendent eux aussi de facteurs liés à l'environnement (compilateur, version de celui-ci, librairies extérieures utilisées, etc…) pouvant potentiellement modifier leur comportement, d'une manière semblable à celle évoquée pour les langages interprétés.

---

[18] IntelliJ IDEA 14 EAP 138.1029 is out with a built-in Java decompiler : http://blog.jetbrains.com/idea/2014/07/intellij-idea-14-eap-138-1029-is-out/

[19] Code de la propriété intellectuelle - Article L122-6-1 : http://www.legifrance.gouv.fr/affichCodeArticle.do?cidTexte=LEGITEXT000006069414&idArticle=LEGIARTI000006278920&dateTexte=&categorieLien=cid

IMPLÉMENTATION

MODIFICATION DU COMPORTEMENT

Si l'implémentation en question possède un comportement différent ou erroné par rapport à celui spécifié, les causes peuvent aller des simples différences de performance (à machine égale) jusqu'à l'inclusion de bugs inexistants lors du développement ou des phases de tests.

On trouve par exemple en **Java** des différences minimes (mais existantes) de modification de comportement entre les différentes versions. Ainsi, la montée de version entre **6** et **7** de **Java** s'est accompagnée de différences pouvant causer des cas de code inatteignable ou des erreurs de compilation[20].

EXEMPLE JAVA

Soit le code suivant (provenant du site d'**Oracle**) :

```
class Main {
    public static void main(String[] args) {
      try {
         throw new DaughterOfFoo();
      } catch (final Foo exception) {
         try {
            throw exception;
         } catch (SonOfFoo anotherException) {
            // Reachable in JAVA 6
         }
      }
   }
}

class Foo extends Exception {}
```

Celui-ci est compatible en **Java 6**, mais provoquera des erreurs de compilation et un comportement différent en **Java 7**. Ainsi, l'instruction « *throw exception*» ne lancera plus une exception de type « *Foo* », mais bien de type « *DaughterOfFoo* » comme on pourrait s'y attendre. Enfin le second « *catch* » sera compilé en **Java 6**, mais provoquera une erreur en **Java 7** indiquant qu'aucun objet de type « *SonOfFoo* » n'est lancé dans le bloc « *try* ».

On trouvera également en **Java** des incompatibilités entre version avec toutes les méthodes définies dans les packages « sun.* », puisqu'il s'agit de méthodes généralement internes à la machine virtuelle **Java** et propre à l'implémentation de **Sun Microsystems**. Ces fonctionnalités sont ainsi à utiliser « à vos propres risques »[21], et leur pérennité n'est pas garantie entre les différentes versions.

[20] Incompatibilities between Java SE 7 and Java SE 6 : http://www.oracle.com/technetwork/java/javase/compatibility-417013.html#incompatibilities

[21] Why Developers Should Not Write Programs That Call 'sun' Packages : http://www.oracle.com/technetwork/java/faq-sun-packages-142232.html

INTER-COMPATIBILITÉ

L'inter-compatibilité entre les différents interpréteurs ou machine virtuelle peut également être une source de problème. En effet, même si d'une manière générale les langages sont définis par des normes strictes, certaines implémentations ne respectent pas forcément à la lettre ces spécifications ou se permettent de fournir des éléments supplémentaires au langage en ajoutant de nouvelles fonctionnalités non spécifiées ou dont la spécification n'est pas encore totalement fixée.

Ces différences d'implémentations, si elles sont trop nombreuses, peuvent conduire à devoir développer des versions spécifiques. On retrouve beaucoup cette problématique dans le monde du web, ou le **JavaScript** n'est malheureusement pas encore supporté de la même manière sur tous les supports, forçant à vérifier directement dans le code si les implémentations sont disponibles, ou bien à passer par des bibliothèques extérieures assurant elles-mêmes ces vérifications.

Ces inter-compatibilités créent des problématiques supplémentaires que les langages interprétés étaient justement censés éviter, avec en quelque sorte une « dépendance à la plateforme » comme on peut le trouver dans le monde des langages compilés.

Ces inter-compatibilités sont souvent liées aux histoires même des langages, et proviennent de différences historiques. On trouve par exemple pour la **JavaScript** la première différence entre le **JScript** de **Microsoft** et le **JavaScript** de **Netscape**, qui n'a que bien plus tard abouti à l'**ECMAScript**[22] que suit le **JavaScript** moderne.

De nos jours encore **JavaScript** demeure légèrement différent d'un navigateur à l'autre, tous n'implémentant pas les mêmes fonctionnalités[23] ou certains en proposant des non standards[24].

VULNÉRABILITÉS

Le fait de passer par un logiciel externe peut aussi causer des problèmes de sécurité totalement externes au développement de l'application en elle-même, puisque présente dans la plateforme qui se charge de l'exécution.

Ces vulnérabilités peuvent prendre place à différents points du fonctionnement des interpréteurs et machines virtuelles, ou de l'implémentation de frameworks que ceux-ci utiliseraient. Elles constituent des vecteurs possibles d'attaque des machines.

Celles-ci sont très préoccupantes pour les entreprises, qui sont de grandes utilisatrices du **Java** pour leurs applications professionnelles, et semblent ne pas suivre leur rythme de mise à jour comme elles le devraient. Ainsi, en 2013 une étude de **Websense Security Lab** montrait que 93% des entreprises

[22] A Short History of JavaScript : https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript

[23] ECMAScript 5 compatibility table : http://kangax.github.io/compat-table/es5/

[24] ECMAScript 5 non-standard compatibility table : http://kangax.github.io/compat-table/non-standard/

testées étaient vulnérables à des attaques utilisant des failles connues du Java, et que presque la moitié de ces entreprises utilisaient des versions obsolètes de plus de 2 ans de **Java**[25].

On peut également citer des politiques de mise à jour qui ne sont pas forcément optimales, comme c'est le cas pour **Oracle** et son application **Java SE** qui s'est récemment vu épinglé sur sa politique de mise à jour, la **Federal Trade Commission** jugeant qu'**Oracle** induisait en erreur ses consommateurs[26].

VULNÉRABILITÉ DE L'INTERPRÉTEUR OU DE LA MACHINE VIRTUELLE

L'interpréteur ou la machine virtuelle se porte en garant de la sécurité de l'exécution du logiciel, que ce soit au niveau du fonctionnement ou du comportement du logiciel. Celui-ci se porte également garant des différents mécanismes de sécurité en place, qu'ils soient un isolement total ou uniquement des restrictions d'accès à certains composants logiciel ou matériel.

Une faille d'implémentation dans un interpréteur ou une machine virtuelle se traduit donc généralement par un isolement imparfait de celui-ci. Dans les cas les moins graves il est simplement possible de causer une interruption inattendue de l'interpréteur ou de la machine virtuelle (chose qui normalement ne doit pas arriver, la machine virtuelle devant garder le contrôle du code) ; mais dans les cas plus grave peut conduire à l'exécution de code arbitraire à l'intérieur même de la machine virtuelle. Ce code arbitraire peut donc ensuite effectuer des actions à l'extérieur de l'environnement protégé et restreint qu'avait mis en place la machine virtuelle.

Ces erreurs proviennent généralement d'erreurs humaines d'implémentation ou de design internes aux machines virtuelles, et ne dépendent d'aucunes librairies extérieures (dans le cas où une librairie est elle-même fautive, l'erreur n'est pas imputable au mécanisme d'interprétation en lui-même)

On trouve ainsi des exemples en **Java** de vulnérabilité permettant d'exécuter du code arbitraire directement la machine virtuelle[27], contournant tous les mécanismes de sécurité qui seraient mis en place.

Ici, le seul moyen de se protéger efficacement est de maintenir à jour l'environnement d'exécution du langage, qu'il soit une application à part entière ou bien faisant partie d'un logiciel (cas des navigateurs). En effet, seul l'application de correctifs sur des failles existantes permet de corriger ces problèmes. Il également à noter que ces mises à jour concernent le moteur d'interprétation en lui-même, et n'ont donc généralement pas d'impact sur le code, facilitant leur déploiement.

MAUVAISE IMPLÉMENTATION DE BIBLIOTHÈQUE

Les erreurs d'implémentation couvrent un large panel de type de vulnérabilités, qui dépendent de la nature et de l'emplacement de celles-ci. Elles diffèrent des vulnérabilités d'interpréteurs et machines virtuelles puisqu'elles reposent sur des erreurs présentes dans des bibliothèques et non pas sur des mécanismes d'interprétations. Elles peuvent soit provenir de bibliothèques accompagnants

[25] New Java and Flash Research Shows a Dangerous Update Gap : http://community.websense.com/blogs/securitylabs/archive/2013/09/05/new-java-and-flash-research-shows-a-dangerous-update-gap.aspx

[26] FTC Complaint : https://www.ftc.gov/system/files/documents/cases/151221oraclecmpt.pdf

[27] Vulnerability Details : CVE-2008-1189 https://www.cvedetails.com/cve/CVE-2008-1189/

directement les langages, ou d'autres bibliothèques réalisées par des tiers. Enfin, ces bibliothèques tierces peuvent aussi présenter un risque similaire à celui des interpréteurs et machines virtuelles si celles-ci sont liées avec du code natif, donnant potentiellement un accès externe à l'environnement d'exécution en lui-même.

On peut par exemple trouver en **Java** (dans son implémentation **Open JDK**) des erreurs d'implémentation dans des mécanismes de vérification de certificats conduisant par exemple à valider des certificats invalides ou bien à des chiffrements imparfaits laissant transparaître en clair des éléments chiffrés[28].

Ces anomalies sont d'autant plus graves que même si l'environnement d'exécution est maintenu à jour, l'utilisation de bibliothèques tierces (et donc qui ne seront pas mises à jour avec le mécanisme d'interprétation en lui-même) possédant des vulnérabilités connues compromet l'application en question. Ici le seul moyen est de maintenir l'application afin de suivre les différentes évolutions et correctifs des différentes bibliothèques utilisées.

Comme dans le cas des erreurs liées aux vulnérabilités des interpréteurs et des machines virtuelles, il convient donc de garder à jour les différentes bibliothèques utilisées par le logiciel en question. Malgré tout cette opération est plus délicate, car dépendant de la maintenance de l'application et non pas de l'utilisateur. Ainsi, une application qui n'est plus activement maintenue pourra présenter de plus en plus de vulnérabilités qui ne seront malheureusement jamais corrigées.

Il convient également de noter que des changements de versions majeurs peuvent apporter des modifications de comportement nécessitant un travail supplémentaire d'adaptation du code.

SANDBOX ESCAPE

On parle de *sandbox espace* quand le logiciel exécuté essaie (et parvient) à sortir de son « environnement protégé ». On retrouve généralement cette problématique dans le monde du web, où les navigateurs chargent et exécutent en permanence du code d'origine non contrôlée. Le but est ici d'avoir accès à l'ordinateur hôte.

Ces failles concernent tous les langages utilisés dans le monde web, on trouve ainsi des exemples de vulnérabilité en **Flash** et sa machine virtuelle **ActionScript**[29] ou en **Java** avec sa **JVM**. On trouve également des exemples internes aux implémentations des navigateurs. Ainsi, en 2014 une faille touchant le navigateur **Chrome** permettait potentiellement d'exécuter du code arbitraire, simplement en renvoyant des données **JSON** à un format incorrect[30].

La multiplication de ce type de faille et la difficulté de déploiement massif de correctifs, notamment avec **Java** et **Flash**, a par exemple conduit certains navigateurs à automatiquement bloquer les

[28] Open JDK security update : http://www.itsecdb.com/oval/definition/oval/com.redhat.rhsa/def/20151228/RHSA-2015-1228-java-1.8.0-openjdk-security-update-Importan.html

[29] List of Flash Player vulnerabilities : https://www.cvedetails.com/vulnerability-list/vendor_id-53/product_id-6761/Adobe-Flash-Player.html

[30] Vulnerability Details : CVE-2014-3188 https://www.cvedetails.com/cve/CVE-2014-3188/

contenus flash[31], ou même à ne simplement pas les gérer[32], et ce afin d'assurer la meilleure expérience utilisateur possible et de limiter les vecteurs de risques. Il est à noter que le **JavaScript** n'est pas concerné (même si celui-ci n'est pas exempt de risques), en effet celui-ci étant massivement utilisé sa désactivation nuirait gravement à l'expérience utilisateur, là où par exemple **Flash** tend à de plus en plus disparaître et est donc de moins en moins utilisé[33].

MAUVAISE UTILISATION

Enfin, il convient également de parler de la mauvaise utilisation d'outils mise en place par les différents langages, et conduisant à des vulnérabilités qui peuvent être plus imputables au développeur plus qu'au langage. Il s'agit généralement de méthodes de traitement d'entrée utilisateur (qu'elles soient textuelles, fichiers ou autre) qui ne mettent pas en place suffisamment de vérifications.

L'exemple certainement le plus parlant et le plus connu demeure la fonction « eval ». Cette fonction est présente dans différents langages (**JavaScript**, **Python** et **PHP** par exemple) et permet d'évaluer des chaînes de caractères comme s'il s'agissait de code source. Comme on pourrait le penser, cette méthode peut être un important vecteur de risques si elle est utilisée sur du code dont la provenance n'est pas assurée d'être fiable.

Ainsi si l'utilisateur peut modifier des données utilisées dans un appel à « eval » il devient alors possible pour lui de réaliser des « eval injection », lui permettant d'injecter son propre code directement dans celui exécuté par l'application, ouvrant une porte d'accès dans l'application.

On par exemple ce type de problématique en **PHP** [34] dans une application qui utiliserait des informations contenues dans les URLS (et donc accessibles aux utilisateurs de l'application) dans des méthodes « eval », conduisant à rendre l'application vulnérable aux « eval injection ». En **JavaScript** côté navigateur, son utilisation peut conduire à des failles de type « cross-site scripting » permettant d'injecter du code dans les différentes pages vues par les utilisateurs.

[31] Adobe Flash bloqué par le navigateur Mozilla : http://www.leparisien.fr/actualite-people-medias/internet-adobe-flash-bloque-par-le-navigateur-mozilla-et-combattu-par-facebook-14-07-2015-4944421.php

[32] Thoughts on Flash (Apple) : https://www.apple.com/hotnews/thoughts-on-flash/

[33] Flash is dying a death by 1,000 cuts, and that's a good thing : http://www.theguardian.com/technology/2015/aug/24/adobe-flash-dying-amazon-google-chrome

[34] Eval() Vulnerability & Exploitation (PHP) : https://www.exploit-db.com/papers/13694/

PERFORMANCE

BENCHMARK COMPARATIF

Il existe de nombreux benchmark sur internet permettant de classer les différents langages suivant leurs performances. D'une manière générale, ces mesures s'appuient sur des résolutions d'algorithmes connus, comme des opérations d'arbres, des calculs de fractales ou simplement de calculer un certain nombre de décimales de pi.

On trouve par exemple le site « The Computer Language Benchmarks Game » regroupant une importante collection de langages et de tests, et fournissant des résultats pour un nombre important de langages.[35]

On peut ainsi par exemple voir que le **Java** est en moyenne 3 à 5 fois plus lent que du C (compilé avec GCC), alors que le **Python** est plus de 100 fois plus lent.

Concernant le **JavaScript**, on trouve par exemple le site « ARE WE FAST YET » (« est-ce qu'on est rapide ? » en français) proposant des benchmarks des différentes builds de **Chrome** et de **Firefox**, et compare leurs résultats pour mieux voir l'évolution des performances dans le temps.[36]

D'une manière générale, le **JavaScript** est considéré sur ces benchmark comme 2.5 fois plus lent que du C++.

LIMITES DES MESURES

Bien que fournissant des informations intéressantes, ces benchmarks ne représentent le comportement de ces différents langages que sur des portions relativement retreintes de code, et peuvent présenter des différences significatives sur des cas « réels » d'utilisation.

Ainsi, même si sur cas restreint de certains tests le **JavaScript V8** se présente comme 2.5 fois plus lent que du **C++**, celui-ci le sera bien plus sur une application beaucoup plus conséquente. En effet, celle-ci sera moins simple à optimiser pour le moteur **V8**, et pourra par exemple rencontrer des contraintes de consommation mémoire que son homologue **C++** n'aura pas.

Enfin, il convient de rappeler que des programmes écris en **C/C++** ou **Python** n'ont pas forcément la même vocation ni les mêmes domaines d'utilisations.

[35] The Computer Language Benchmarks Game : http://benchmarksgame.alioth.debian.org/u64q/which-programs-are-fastest.html

[36] ARE WE FAST YET : https://arewefastyet.com/

TEMPS PROCESSEUR

L'interprétation, de manière générale, consommera toujours plus de temps processeur qu'un code natif. Ainsi, la mise en place d'un environnement d'exécution nécessitera toujours des opérations processeur, créant dès lors une différence inaltérable avec le code natif. Cette mise en place initiale est toujours nécessaire, même si le code ensuite exécuté est lui-même déjà près-compilé nativement (comme c'est par exemple le cas en **C#** où le code déjà compilé en **JIT** est placé en cache).

Les différents mécanismes internes à l'environnement d'exécution, et plus particulièrement relatifs à l'interprétation, qu'ils prennent place dans une machine virtuelle ou un interpréteur constituent un des plus importants des postes de dépense.

Dans le cadre des logiciels dis « bas niveau » certaines opérations sont gérées directement par le code, comme l'allocation mémoire, ou même directement de manière matérielle dans le cas des opérations assembleurs présentes dans les sources compilées. Dans le cas d'un langage interprété, ces opérations sont toutes gérées par la machine virtuelle ou l'interpréteur, qui sert d'intermédiaire entre le matériel et le logiciel qu'il exécute. Certaines opérations sont également gérées de manière transparente et sécurisée, comme par exemple l'allocation mémoire, nécessitant donc des vérifications supplémentaires.

L'interprétation propose également plus de vérifications, qu'elles soient inhérentes aux langages de plus haut niveau (on peut par exemple citer les langages avec un typage dynamique qui doivent tester à chaque appel les différentes valeurs, ou les systèmes de réflexivité sur les classes et les types) ou bien directement sur le comportement du programme lui-même, et ce afin de proposer une meilleure gestion de la sécurité.

L'ensemble de ces contrôles, en plus du coût inhérent à l'interprétation en elle-même, conduisent généralement à proposer des performances plus faibles (à cause du surcoût inhérent de consommation processeur).

Enfin, on peut également citer les mécanismes de compilations **JIT**, nécessitant généralement des threads et du temps processeur pour effectuer leurs optimisations. Même si leurs optimisations peuvent être considérées comme rentable pour le logiciel, celle-ci ont un coût non-négligeable.

CONSOMMATION DE MÉMOIRE

De même que la consommation de temps processeur, l'interprétation consommera toujours plus de mémoire qu'un code natif. Néanmoins ce surcoût initial, étant statique, peut être considéré comme négligeable sur des logiciels suffisamment importants. Par exemple le coût initial de la mise en place de la **JVM** d'Oracle est de l'ordre de 40Mo, et ce même si le logiciel lancé en fait lui-même moins.

On peut citer comme précédemment les mêmes postes de dépenses liés aux différentes opérations de vérification et de sécurité que mettent en places les interpréteurs et les machines virtuelles, généralement indépendantes du développeur ou de l'utilisateur.

De plus la gestion de la mémoire, confiée à la charge d'un allocateur et d'un *garbage collector*, entraîne une consommation de mémoire supplémentaire là où celle-ci est directement gérée dans le programme par l'utilisateur dans des langages plus bas niveau. En effet le *garbage collector* aura

généralement tendance à pré-allouer des blocs de mémoire, et donc de consommer plus que ce dont le logicielle nécessite réellement (même si ce comportement évite le fractionnement de la mémoire et permet de limiter les opérations d'allocations, souvent coûteuses en temps).

Au niveau des autres postes de dépenses incompressibles on peut citer le *boxing* des valeurs. Ainsi, par exemple en **Python** (dans son implémentation de référence **CPython**), toutes les valeurs (y compris les valeurs « naturelles » comme les entiers et les flottants) sont en réalité des objets. Ces « métadonnées » comprises dans le *boxing* définissant la valeur, ses informations et ses méthodes. Celles-ci créent une consommation en mémoire plus importante qu'une valeur naturelle. On retrouve ce mécanisme de *boxing* automatique également dans d'autres langages comme le **C#** (qui propose même des méthodes directement sur ces valeurs). Enfin, certains langages différencient clairement la valeur naturelle de la valeur « objet » comme en **Java** où pour chaque type naturel il existe une classe équivalente dans le framework **Java**. Ici, la différenciation se repose sur le choix du développeur là où celle-ci est automatique en **C#** (point abordé dans les différentes optimisations mises en place).

Ces différents éléments constituent une différence avec les langages plus bas niveau et compilés, où le code est directement exécuté sans entraîner de surcoût indépendant, comme c'est le cas en **C/C++**. Ainsi dans le cadre d'un logiciel **Java**, une partie de la consommation mémoire du logiciel est directement imputable à la mise en place de l'environnement d'exécution, là où un programme **C/C++** n'entraîne pas automatiquement la mise en place d'un environnement d'exécution.

Ainsi, d'après une étude de Robert Hundt (**Google**) sur le **Java** et le **Scala**, le **Java** consommerait en moyenne 3.7 fois plus de mémoire qu'un programme équivalent écrit en **C++**[37].

TAILLES DES SOURCES

LE CAS DU WEB

Le monde du web apporte de nouvelles problématiques liées au volume des données d'une page web. Ainsi, même s'il ne s'agit généralement pas d'un point critique, des sources plus petites sont toujours préférables puisqu'elles permettent d'économiser le volume de données à transférer pour afficher une page[38].

Pourtant le monde du web utilise des langages (**JavaScript**, **HTML**, **CSS**) ne passant pas par un bytecode et donc généralement plus volumineux (Si on omet les sites utilisant des technologies comme du **Java** ou du **Flash**), posant une problématique propre au monde du web, qui cherche d'un côté à concilier ces langages haut niveau et des temps de chargement rapides.

[37] Loop Recognition in C++/Java/Go/Scala : https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf

[38] Google Developers – Performance guidelines : https://developers.google.com/web/fundamentals/performance/?hl=en

FRACTIONNEMENT DES LANGAGES

Le fractionnement des langages désigne la multiplication des langages de programmation informatique, certains couvrant inévitablement les mêmes buts et possédant des caractéristiques semblables.

Ainsi, la page Wikipédia listant les différents langages cite à elle seul près de 500 langages de programmations ou de scripting différents. Même si certains langages sont des évolutions d'anciens ou uniquement des subsets, ceux-ci définissent de nouvelles notions non triviales nécessitant un temps d'adaptation. Enfin, on peut différencier que les *domains specific language* (langage dédié en français) qui concernent généralement des domaines restreints et spécifiques, par opposition aux langages généralistes (type **C/C++** et **Java**) pouvant être utilisés de manière générale quelle que soit la tâche à effectuer.

Cette profusion de langage de programmation est souvent citée comme critique lors de l'arrivée de nouveaux langages. En effet, même si semblables, chacun de ceux-ci demande généralement des connaissances spécifiques, créant dès lors un fractionnement entre eux. La critique peut également porter sur le fait que de nombreux langages partagent généralement un but semblable, voir une syntaxe relativement proche.

Cette profusion trouve généralement son origine dans la volonté d'équipes ou de personnes de créer un langage spécifique possédant les fonctionnalités qu'ils attendent d'un langage, ou bien par soucis de maîtriser totalement l'implémentation et le fonctionnement de celui-ci.

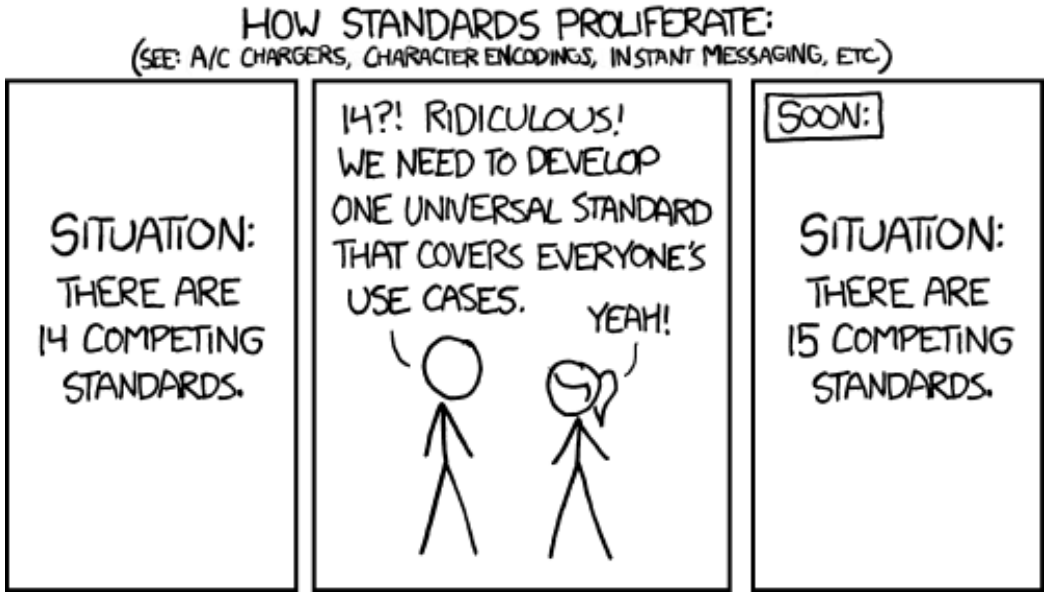


Illustration de xkcd sur la prolifération des standards

SOLUTIONS EXISTANTES

PERFORMANCE

COMPILATION

La compilation, qu'elle soit de type **AOT** ou **JIT**, constitue en elle-même un excellent moyen de solution à de nombreuses problématiques de performance et d'occupation mémoire qu'apportent les langages interprétés, et ce sans pour autant leur retirer leurs avantages de langage haut-niveaux. Comme mentionnée précédemment la phase de compilation permet la mise en place de différentes optimisations

COMPILATION AOT

La compilation **AOT** nécessite, que ce soit pour des langages interprétés ou natifs, une première phase de compilation avant de pouvoir être exécuté. Libérant des problématiques de temps et de puissance que peut avoir une compilation **JIT** (car celle-ci se déroule au *runtime,* où la gestion des ressources est critique), ce type de compilation peut être relativement poussé au niveau des optimisations statiques mises en place.

Le résultat de ce type de compilation, dans le cas des langages interprétés, sera presque toujours un *bytecode* utilisable avec une machine virtuelle. Ce passage par un *bytecode* intermédiaire permet de réduire le poids des sources et de réduire le temps d'analyse syntaxique et lexical ; en plus des optimisations apportées à la compilation.

Enfin, il est possible d'utiliser à la fois une compilation **AOT** puis une compilation **JIT** à l'exécution, comme c'est le cas du **Java** et du **C#**. Ainsi, la compilation **AOT** permet d'effectuer une première passe d'optimisations statiques sur le code avant son exécution.

COMPILATION JIT

Là où la compilation **AOT** se déroulait de manière prévisionnelle, la compilation **JIT** se déroule durant l'exécution même du programme. La mise en place de ce type de compilation ne nécessite généralement pas ou peu de changement du côté des habitudes de développement déjà en place.

Ce type de compilation, contrairement aux compilations prévisionnelles se concentre uniquement sur des portions critiques du code qui sont rentables à optimiser et donc à compiler. Ces compilations sont également généralement adaptatives, et s'adaptent ainsi au profil d'utilisation courant de l'application, permettant théoriquement de surpasser des langages compilés nativement qui ne peuvent profiter de ce type d'optimisations dynamiques.

Durant ces phases de compilation, de nombreuses optimisations peuvent prendre place. Elles peuvent soit être homologues aux optimisations statiques classiques, ou bien uniquement possible dans le cadre d'optimisation dynamique puisque tirant partie d'informatique de profilage sur l'application.

Ce type de compilation peut être exécuté sur du code interprété déjà issu d'une compilation **AOT**, comme c'est le cas du **Java** ou du **C#** dans toutes leurs implémentations, ou bien directement comme la majorité des moteurs **JavaScript**.

TRANSPILATION

La transpilation, sans pour autant être réellement considérée réellement comme une phase de compilation (le langage en sortie reste de haut niveau), permet de combler de nombreux problèmes qu'apportent les langages interprétés n'ayant aucune phase de compilation (comme c'est par exemple le cas du **JavaScript**).

D'une part, même si l'interpréteur accepte directement du code source, une transpilation permet d'apporter une couche supplémentaire d'analyse statique. Celle-ci permet, entre autre, de supprimer le code mort, d'empêcher des erreurs syntaxiquement correctes mais incorrectes d'un point de vue logique, et d'effectuer certaines des optimisations suscitées.

Une autre utilisation de la transpilation est de permettre d'utiliser certains interpréteurs ou compilateurs existants avec un nouveau langage, comme le **TypeScript** de M**icrosoft**, ou encore de profiter des nouvelles implémentations de langages tout en gardant une compatibilité avec les anciennes versions des interpréteurs.

CLOJURE COMPILER

On trouve par exemple de **Closure compiler**[39] de **Google**, qui permet de transpiler du **JavaScript** vers du **JavaScript**, mais en apportant des fonctionnalités d'analyse statique très poussées lui permettant d'effectuer des optimisations dans le code donné en entrée. Celui-ci a pour but avoué de proposer du code **JavaScript** plus rapidement téléchargeable (puisque plus petit) et plus rapide (puisque statiquement analysé). Il convient de noter que les exemples suivant n'utilisent pas l'option de minification, et ce afin de laisser le code lisible.

Ainsi, pour ce code d'exemple relativement simple, possédant 2 fonctions et une boucle :

```
// Add a and b
function add(a, b) { return a + b; }
// Substract b to a
function minus(a, b) { return a - b; }
// Do some operations
var a = 1, b = 99;
for(var i = 0; i < 2; i++) {
  a = minus(b, add(a, 2));
}
// Print 0
console.log(a - 1);
```

En mode simple, le résultat reste relativement proche de celui écris. Comme prévu les commentaires ont tous été supprimés, mais on peut noter que les variable **a** et **b** ont été déplacées et le sens de la boucle inversé :

```
function add(c, d) { return c + d; }
function minus(c, d) { return c - d; }
for (var a = 1, b = 99, i = 0;2 > i;i++) {
  a = minus(b, add(a, 2));
}
console.log(a - 1);
```

[39] Closure Compiler : https://developers.google.com/closure/compiler/

Mais une fois poussé en « advanced », le code change radicalement puisque les méthodes sont mises en ligne et disparaissent donc :

```
for (var a = 1, b = 0;2 > b;b++) {
  a = 99 - (a + 2);
}
console.log(a - 1);
```

Ainsi, cette phase permet d'ajouter une étape d'optimisation à un langage n'en possédant pas avant son exécution, lui apportant toute une batterie d'optimisations statiques.

BABEL JS

Babel JS est un transpileur **JavaScript**, permettant de transpiler du **JavaScript** utilisant la dernière norme **ECMAScript 6** vers une version plus ancienne. Cela permet d'utiliser la dernière norme du langage sans attendre les différentes mises à jour de compatibilité des différents navigateurs web.

Ainsi, le **JavaScript ES6** gérant nativement les classes et les propriétés de classes, il est possible d'écrire ceci :

```
class MyClass {
  constructor(value) {
    this.value = value;
  }
  get value() {
    return this.value;
  }
  set value(value) {
    this.value = value;
  }
}
```

Malheureusement ce code est encore incompatible avec de nombreux navigateurs. Cependant **Babel JS** permet de le transpiler, donnant le résultat suivant :

```
var MyClass = (function () {
  function MyClass(value) {
    _classCallCheck(this, MyClass);
    this.value = value;
  }
  _createClass(MyClass, [{
    key: "value",
    get: function get() { return this.value; },
    set: function set(value) { this.value = value; }
  }]);
  return MyClass;
})();
```

On peut voir clairement ici que le **JavaScript** généré utilise des techniques **ES5** pour simuler des comportements de classes et de propriétés (« classCallCheck » et « createClass » étant des routines de manipulation d'objet s'assurant que le comportement des « classes » soit cohérent avec leurs spécifications **ES6**).

ASM.JS

ASM.js est un subset strict du **JavaScript** mis en place par Mozilla, visant à être utilisé comme une cible de « compilation » bas niveau de langages n'ayant pas de contrôle de mémoire[40]. Ce subset se veut donc notamment axé sur la performance. Celui-ci met en avant le fait que même si les navigateurs actuels ne permettent d'interpréter que du **JavaScript**, il reste possible d'écrire en n'importe quel autre langage pourvu que celui-ci soit transpilé en **JavaScript**. Ainsi, il devient possible de profiter des caractéristiques de certains langages et absents du **JavaScript** (typage statique par exemple), ou même d'outils spécifiques aux autres langages.

ASM.js se veut aussi comme plus rapide que le **JavaScript** écrit manuellement, et ce grâce à des techniques de code JavaScript qui serait difficilement lisibles et maintenables par des humains.

Au niveau de ses différentes implémentations, on peut citer **Emscripten**[41], proposant de compiler du **C/C++** en **JavaScript**, notamment en utilisant **LLVM** (une infrastructure de compilateur proposant des routines déjà existantes d'optimisations durant les différentes phases de compilation), permettant la mise en place d'optimisations plus agressives et poussées que dans les mécanismes habituels de transpilation.

Emscripten utilise généralement des techniques permettant aux différents interpréteurs **JavaScript** d'inférer les types de variables plus facilement que dans une utilisation normale, et ce afin de permettre de gagner du temps sur ces opérations (le type étant déjà connu à l'avance sur des langages statiquement typés comme le **C** ou le **C++**). On trouve par exemple des conversions explicites de valeur en entier (au travers d'opération arithmétiques « neutre »), ou bien l'utilisation de types optimisés déjà proposé par le **JavaScript** (tel quel le type Uint32Array[42], permettant de définir des tableaux d'entiers 32bit). On trouve également les mécanismes habituels de compilation **AOT**, puisque mis en place sur du code **C/C++** dont le résultat sera ensuite lui-même transpilé en **JavaScript**. Ce fonctionnement permet de mettre en place des optimisations que le **Clojure Compiler** (vu précédemment) ne peut déployer, puisque celui-ci n'a pas de certitude sur les types de variables (puisque prenant du **JavaScript** en entrée).

[40] Asm.js specifications : http://asmjs.org/spec/latest/

[41] Emscripten : http://kripken.github.io/emscripten-site/

[42] Uint32Array : https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Uint32Array

Cet exemple, tiré de la présentation d'**ASM.js** de **Alon Zakai**[43] permet de mettre en avant quelques une des optimisations fondamentales que **Emscripten** met en place lors de ses phases de compilation :

```
function compiledCalculation() {
        var x = f()|0;  // x is a 32-bit value (infered from static type)
        var y = g()|0;  // so is y
        return (x+y)|0; // 32-bit addition, no type or overflow checks
}

var MEM8  = new Uint8Array(1024*1024);
var MEM32 = new Uint32Array(MEM8.buffer); // alias MEM8's data

function compiledMemoryAccess(x) {
        MEM8[x] = MEM8[x+10]; // read from x+10, write to x
        MEM32[(x+16)>>2] = 100;
}
```

On retrouve également l'utilisateur de buffers en lieux et place de pointeurs sur la mémoire, et ce afin de facilement transposer du code **C/C++** à du **JavaScript**, tout en gardant le même fonctionnement mais dans un environnement d'exécution sécurisé.

Ainsi, les premiers tests montrent des performances grossièrement deux fois plus lentes que leurs équivalents **C/C++** une fois compilés, semblable à ce que propose le **Java** et le **C#**[44].

PROFILAGE DE CODE

Le profilage de code entre en jeux dans les différentes mises en place d'optimisations dynamiques. Cette appellation désigne généralement l'ensemble des mécanismes permettant de mesurer les différentes performances d'une portion de code ou d'une application et d'en produire des statistiques (utilisation processeur, utilisation mémoire, nombres d'appels de méthode, etc…) permettant d'avoir une vue générale et synthétique des différents postes de dépenses.

Le profilage de code peut prendre place de manière totalement extérieure de l'application, en analysant via un outil extérieur le comportement du code en lui-même. Il sert alors d'outil de rapport permettant de faire ressortir les postes de dépense les plus important de l'application, et donc de potentiellement les corriger ou les améliorer. Cette mise en place peut être automatique dans le cadre de l'intégration continue d'un logiciel, ou bien de manière ponctuelle afin de résoudre des problèmes de performances.

Il est également possible que ce mécanisme de profilage entre en action à l'insu de l'utilisateur, directement dans le mécanisme d'interprétation, qu'il s'agisse d'une machine virtuelle ou d'un interpréteur. Dans ce cas-ci, l'analyse va permettre à l'environnement d'exécution d'optimiser le code pour l'adapter au mieux à son profil d'utilisation actuel, et ce en tirant partie des différentes informations de profilage recueillies. L'analyse dynamique du code permet donc de mettre en place ce qu'on appelle des optimisations adaptives, contenant par exemples des recompilations dynamiques

[43] BIG WEB APP? COMPILE IT! : http://kripken.github.io/mloc_emscripten_talk/

[44] REALISTIC/LARGE BENCHMARKS : http://kripken.github.io/mloc_emscripten_talk/#/28

visant à adapter des composants déjà compilés (depuis une compilation **JIT** ou même **AOT**) au profil d'utilisation actuel.

Ainsi, la même méthode pourrait avoir deux versions optimisées compilées différentes pourvu qu'elle soit utilisée différemment à deux endroits du même programme.

On trouve par exemple en **Java** (dans l'implémentation d'**Oracle**) différents mécanismes d'optimisation dynamiques à l'intérieur du **JRE** (utilisant **HotSpot**, depuis **Java 1.3**) : l'un rapide permettant un lancement rapide de l'application et un autre plus lourd permettant une optimisation plus agressive. Depuis **Java 7**, ces deux mécanismes prennent place au sein des applications serveurs : le premier se charge d'un démarrage rapide, et après un certain « temps de chauffe » de l'application la méthode plus agressive prend le pas, permettant de proposer des performances supérieures dans le temps (les applications type serveur étant supposées tourner longtemps)[45].

OPTIMISATIONS

Les optimisations regroupent ici les différentes techniques permettant à la machine virtuelle ou l'interpréteur de réduire la consommation des différentes ressources (principalement temps processeur et mémoire) utilisées par un logiciel.

Les optimisations peuvent entrer en action à plusieurs points cruciaux de la vie d'un programme interprété. On les retrouve ainsi à la compilation (compilation **AOT**) ou à la transpilation, au déroulement du programme en lui-même, et durant des phases de compilation ou recompilation **JIT**.

Les différentes optimisations présentées ici peuvent parfois être réalisées de manière statique, soit de manière dynamique durant le déroulement d'un programme. Certaines comme le monomorphisme sont ainsi uniquement possibles durant le « runtime » (puisque dépendant de donnée de profilage du code), alors que l'inlining est par exemple possible durant toutes les phases.

INLINING

L'***inlining*** (qui pourrait se traduire par « mise en ligne ») est une méthode visant à remplacer un appel de méthode directement par le code de la méthode en lui-même ; économisant ainsi l'appel et les opérations de manipulation de la pile nécessaire. Cette technique est également très utilisée sur les mécanismes classiques de compilation **AOT**, en utilisant exactement le même principe.

On peut également parler de « mise en ligne native » quand il s'agit de transformer des instructions non pas par le code appelé, mais directement par du code machine. Cette pratique, utilisée conjointement avec le monomorphisme (permettant de présupposer l'implémentation utilisée) et l'affinage de type (permettant de spécialiser les opérations pour un type défini), permet de proposer des séries d'instructions aussi rapides que si elles avaient été compilées.

[45] Java HotSpot Virtual Machine Performance Enhancements : http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html

Par exemple, pour le **Java**, si une méthode contient moins de 35 octets d'instructions, la **JVM** essayera d'***inliner*** ses différents appels si ceux-ci sont assez nombreux pour que l'opération soit jugée rentable.[46]

EXEMPLE

Soit une méthode « add » et « add5 » :

```
void add(int a, int b) { return a + b; }
void add5(int a, int b) {
    int c = 0;
    for(int i = 0; i < 5; i++)
        c += add(a, b);
    return c;
}
```

Comme on peut le voir, la méthode « add » peut être très simplement mise en ligne, ce qui une fois appliqué donnera :

```
void add5(int a, int b) {
    int c = 0;
    for(int i = 0; i < 5; i++)
        c += a + b;
    return c;
}
```

Cette mise en ligne permet ici d'économiser toutes les opérations relatives à l'appel d'une méthode (modification de la pile, etc…).

AFFINAGE DE TYPE

L'**affinage de type**, ou « **type sharpening** » en anglais, désigne l'action pour le compilateur d'affiner les types utilisés dans une méthode afin d'en construire une version spécialisée plus performante. On retrouve généralement ce comportement pour les langages à typage dynamique, puisque les types présents en entrée des méthodes peuvent être différents d'un appel à l'autre. Il diffère du monomorphisme évoqué postérieurement, puisqu'il concerne le type réel d'une valeur et peut prendre place sans que le langage ne gère le polymorphisme.

Le **type sharpening** se base sur l'idée qu'une méthode va souvent être utilisée avec des éléments du même type, même si celle-ci offre la possibilité d'être utilisée avec n'importe quel type de variable. Il est ainsi plus rentable à partir d'un certain point d'en définir une version spécialisée, et ce en trouvant les types réels ayant le plus de chance d'être utilisés. Il peut à la fois se baser sur des analyses statiques d'utilisation d'une méthode, en plus de profils d'utilisation dynamique.

Le moteur **JavaScript** V8 de **Google** utilise par exemple ce principe pour attribuer des équivalents natifs à certaines méthodes souvent utilisées avec les même types[47], qu'ils soient natifs ou des objets. Cette mise en place permet théoriquement d'obtenir des performances équivalentes à des langages natifs sur ces portions de code.

---

[46] Java JIT compiler inlining : https://techblug.wordpress.com/2013/08/19/java-jit-compiler-inlining/

[47] Google io 2012 : http://v8-io12.appspot.com/#61

EXEMPLE - SPÉCIALISATION

Soit une fonction **JavaScript** permettant de déterminer si un nombre est premier ou non :

```
function isPrime(number) {
    var start = 2;
    while (start <= Math.sqrt(number)) {
        if (number % start++ < 1) return false;
    }
    return number > 1;
}
```

Comme on peut l'observer, cette méthode est relativement peu complexe à traiter pour un interpréteur (une boucle et une condition). Pourtant, dans la pratique, un interpréteur naïf testerait à chaque opération effectuée sur la variable « number » ou « start » qu'il s'agit bien d'un nombre, afin de vérifier si la valeur peut être utilisée ou si une conversion implicite doit entrer en compte.

Une première passe d'analyse statique permettrait de ne pas recourir à ces tests pour la variable « start », mais la valeur « number » resterait sujet à vérification. C'est là que les optimisations dynamiques d'affinage de type entrent en jeu. Après avoir été appelé un certain nombre de fois avec uniquement des nombres en entrée, un profileur va déclencher la création d'une version spécialisée de la méthode, uniquement utilisable avec en entrée des nombres. Cette méthode spécialisée, ne dépendant plus d'aucune variable externe, pourra ainsi être entièrement compilée en code natif, proposant des performances équivalentes à un langage compilé.

En pseudocode, le résultat en interne d'un interpréteur pourrait être semblable à celui-ci :

```
function isPrime(number) {
    if(isNumber(number))
      return compiledIsPrime(number);
    else
      return oldIsPrime(number);
}
```

Dans le cas où la méthode est appelée avec en entrée des types ne profitant d'aucune optimisation, la méthode de référence sans spécialisation sera réutilisée.

ROUTAGE MONOMORPHIQUE

En informatique, le polymorphisme repose sur l'idée que plusieurs éléments de types différents peuvent être considérés comme étant du même type. En pratique, cela se traduit par des méthodes acceptant des objets d'un certain type, ou héritant de ce type de base. On retrouve le même comportement avec les interfaces (ou traits, suivant les langages). Il s'agit du principe de **Liskov**, assurant que tout type **S**, sous-type de **T**, pourra être utilisé à la place d'un objet de type **T**.

La première conséquence du polymorphisme pour un interpréteur ou une machine virtuelle est que celui-ci doit vérifier quelle implémentation d'une méthode appelée au *runtime*, et ce à chaque appel (à cause du principe de **Liskov,** permettant de surcharger des méthodes et donc de proposer plusieurs implémentations pour une seule méthode). Ceci cause plus d'opérations là où un langage qui ne supporterait pas le polymorphisme en effectuerait moins (car celui-ci connaitrait à l'avance la bonne implémentation à utiliser).

Ce principe limite aussi l'application d'autres optimisations : puisque l'implémentation n'est pas connue à l'avance, l'interpréteur ne saura pas s'il peut la mettre en ligne ou la remplacer par une version déjà compilée.

Le **routage monomorphique** se base sur l'idée que malgré le polymorphisme d'un langage, l'implémentation appelée est presque toujours la même. On retrouve un principe semblable au « **type sharpening** » évoqué précédemment, mais sur l'implémentation interne des objets manipulés. Cela signifie qu'il est possible de supprimer l'étape de vérification en elle-même et de directement appeler la bonne implémentation. Il est donc possible de créer des chemins de code techniquement plus rapides que leurs équivalents **C++** (où la vérification de l'implémentation à utiliser ne peut pas être aussi simplement éliminée). Dans le cas où un type différent serait utilisé, la machine virtuelle se contente de reprendre la version non-optimisée du chemin, revenant dans le pire des cas aux performances de base. Ce type d'optimisation est notamment mis en place par la **JVM d'Oracle**.[48]

EXEMPLE

Par exemple, soit les classes Java suivantes :

```
class A {
        public foo() { System.out.println("Hello world"); }
}

class B extends A {
        public foo() { System.out.println("Bonjour le monde"); }
}

class Hello {
        public sayHello(A a) { a.foo();}
}
```

Le principe de **Liskov**, évoqué plus haut, ne définit pas quelle implémentation de la méthode « foo » sera réellement appelée. Il peut s'agir de l'implémentation de « A », de « B » ou bien d'une autre implémentation héritant directement ou indirectement de la classe « A ». On doit donc théoriquement à chaque appel aller chercher l'implémentation à utiliser, ce qui empêche par exemple dans ce cas-ci de mettre la méthode en ligne (celle-ci s'y prête particulièrement bien : une seule opération ne dépendant d'aucune variable).

Le monomorphisme permet de supposer que la méthode « sayHello » sera majoritairement appelée avec des classes du même type réel, et permet donc de définir une version spécialisée. Ce choix de spécialisation peut-être le résultat d'un mécanisme interne de profilage de code, qui analyserait qu'une majorité des appels est effectuée avec le même type réel, ou plus simplement d'analyse statique. Cette version spécialisée pourra ainsi contenir des optimisations plus poussées ou être compilée nativement afin d'offrir de meilleures performances.

[48] Understanding Java JIT: http://www.oracle.com/technetwork/articles/java/architect-evans-pt1-2266278.html

OPTIMISATION DE BOUCLE

L'optimisation de boucle regroupe les différents mécanismes ayant pour but commun d'optimiser les opérations d'itérations dans des boucles. Plusieurs de ces optimisations peuvent prendre place sur la même boucle afin d'optimiser au maximum ses performances. On peut citer par exemple :

DÉROULEMENT

Le **déroulement** vise à dérouler une boucle et répéter ses instructions plutôt que de réellement parcourir la boucle en elle-même. Elle permet de diminuer le nombre d'opérations entrant en jeu en supprimant un bloc de code, et en répétant simplement les instructions qu'il contenait.

Cette technique a de fortes chances d'entrer en compte si le compilateur peut connaître à l'avance le nombre d'itérations, comme c'est le cas lorsqu'on définit celui-ci sans passer par des variables.

Il convient également de noter que cette technique n'entre généralement en jeu que si le nombre d'itérations reste relativement faible et que le contenu de la boucle reste lui aussi maitrisé, et ce afin de l'optimisation reste rentable.

EXEMPLE

Soit une fonction **JavaScript** hypothétique permettant de multiplier par 3 une valeur et de la retourner :

```
function add3(number) {
    for(var i = 0; i < 3; i++)
          number += number;
    return number;
}
```

Comme on peut le voir, le nombre d'itération est connu à l'avance et l'interpréteur pourra donc dérouler la boucle, donnant un résultat semblable au suivant :

```
function add3(number) {
    number += number;
    number += number;
    number += number;
    return number;
}
```

Le fait de retirer la boucle permet ici d'économiser la création d'une variable, 3 tests de valeurs, et 3 incrémentations. Cette économie d'instruction se traduit par des sources plus petites en mémoire, en plus de meilleures performances évitant des actions inutiles au déroulement du programme.

INVERSION

L'inversion transforme les boucle *while* (*tant que* en français) en boucle *do/while* (répète tant que, en français) enveloppé dans une condition initiale, et ce afin d'éviter des opérations supplémentaires dans le cas où la boucle est exécutée.

Cette optimisation peut techniquement prendre place sur chaque boucle de type *while* présente dans le code.

DÉPLACEMENT DES INVARIANTS

Si des éléments dans le bloc de code d'une boucle restent les mêmes quelle que soit l'itération, il est plus rentable de les effectuer une seule fois à l'extérieur de la boucle et de simplement les réutiliser à l'intérieur de celle-ci (économisant la répétition des opérations).

EXEMPLE

Soit le code suivant, effectuant une somme avec des valeurs invariantes :

```
var total = 0;

for(var i = 0; i < 10; i++) {
          var a = 12, b = 8;
          total += a + b;
}
```

On peut voir facilement que les valeurs « a » et « b » ne changent pas, mais sont re-déclarée à chacune des 10 itérations de la boucle. Il donc possible de les déplacer à l'extérieur de celle-ci sans affecter le comportement du programme :

```
var total = 0, a = 12, b = 8;

for(var i = 0; i < 10; i++) {
          total += a + b;
}
```

FISSION, OU DISTRIBUTION

Si la boucle peut être divisée en plusieurs sous-boucles ayant le même nombre d'itérations, cela peut améliorer le principe de **localité des références**.

Ce principe défini que les valeurs les plus proches (en termes de blocs d'instructions) ont le plus de chance d'être utilisées un plus grand nombre de fois que leurs homologues plus éloignés. Cela permet par exemple de placer ces valeurs dans des registres ou des caches plus rapides, afin d'accélérer leurs accès. Le fait de scinder la boucle permet ainsi de mieux profiter de ce mécanisme d'optimisation d'accès.

ELIMINATION DE CODE MORT

L'**élimination de code mort** vise à enlever du code les chemins qui ne seront jamais empruntés par le flux d'exécution. On trouve généralement ce type d'optimisation sur des conditions toujours fausses ou sur des blocs de code situés après des instructions de sauts toujours empruntés par le flux du programme.

FONCTION INTRINSÈQUE

Une **fonction intrinsèque** réfère généralement à ce qu'on appelle une fonction « *built-in* » ; c'est-à-dire une méthode codée directement dans l'interpréteur ou la machine virtuelle. Il en résulte que celui-ci en a une connaissance supérieure à ce qu'il pourrait avoir d'une fonction « utilisateur », permettant des optimisations plus poussées. Ces méthodes sont également généralement en code natif (puisque codée directement *dans* l'interpréteur ou la machine virtuelle) et donc plus rapide que des équivalents « utilisateurs ».

ANALYSE STATIQUE

L'**analyse statique** a pour fonction de récupérer le maximum d'informations sur le code source d'un programme, sans avoir à l'exécuter. On trouve généralement un fonctionnement semblable sur tous les logiciels d'analyse, prenant en entrée le code source et produisant en sortie un rapport. Ce rapport contiendra toutes les informations relevées, qui peuvent aller d'informations de « style », comme une indentation incorrecte, jusqu'à des informations d'analyse statique poussées sur l'utilisation de variables ou la présence de fuites mémoires.

On trouve ce type d'outils pour presque tous les langages, qu'ils soient compilés ou interprétés. On peut par exemple citer **Lint**, un analyseur statique historique pour le **langage C**. On retrouve ainsi de nombre déclinaisons de cet outil, adaptés pour les différents langages existants (**Pylint** pour **Python**[49], **Jlint** pour **Java**[50], etc…).

Le but d'un outil d'analyse statique est uniquement d'informer et de produire des rapports pouvant être vus comme des recommandations de développement. Ils se différencient des transpileurs, qui eux vont mettre en place des optimisations et produire en sortie un code optimisé, là où un analyseur se contente de présenter un rapport. Il convient toutefois de noter que les transpileurs, compilateurs et interpréteurs contiennent généralement en interne des analyseurs statiques, permettant de mettre en place les différentes optimisations que ceux-ci permettent de détecter. Il s'agit ici pour le développeur d'effectuer lui-même des optimisations et « corrections » qu'un compilateur ou interpréteur effectuerait, afin de rendre le code plus performant (dans le cas où celui-ci serait directement interprété) et également plus lisible et maintenable.

EXEMPLE

Pylint est une adaptation de **Lint** pour le **Python**, permettant de générer des rapports d'analyse statique pour des sources en **Python**.

Par exemple, soit la sortie suivante de **Pylint** (issue de son aide officielle)[51] :

```
************* Module pylint.checkers.format
W: 50: Too long line (86/80)
W:108: Operator not followed by a space
     print >>sys.stderr, 'Unable to match %r', line
            ^
W:141: Too long line (81/80)
W: 74:searchall: Unreachable code
W:171:FormatChecker.process_tokens: Redefining built-in (type)
W:150:FormatChecker.process_tokens: Too many local variables (20/15)
W:150:FormatChecker.process_tokens: Too many branches (13/12)
```

Celle-ci met à la fois en avant des éléments purement « esthétiques », comme une ligne trop longue à la ligne 50, mais également des informations plus intéressantes comme du code mort (ligne 47) ou l'utilisation de mots clefs réservés (ligne 150).

[49] Site officiel de Pylint : http://www.pylint.org/

[50] Site officiel de Jlint : http://jlint.sourceforge.net/

[51] Aide de Pylint : http://docs.pylint.org/output.html#source-code-analysis-section

Ici, la correction de ces différentes remarques conduit à la création de code plus sûr et plus maintenable, en plus d'éviter de potentiels futurs bugs.

RESPECT DES GUIDELINES

Le respect des bonnes pratiques de développement doit s'appliquer au tout début du développement d'un projet, et n'est pas spécifique aux langages interprétés. L'ensemble de ces recommandations touchent généralement tous les domaines du développement, de la création de documentation à la manière de coder en elle-même. Ainsi, l'intérêt de ces règles n'est pas uniquement de produire un code plus maintenable, mais également d'en produire un plus performant.

Là où un langage compilé en code natif subit de lourdes optimisations lors de sa compilation, un code interprété sans phase de compilation sera quant à lui par exemple directement exécuté. Sans phase préalable d'optimisation, il sera donc beaucoup plus sensible aux « bonnes pratiques » visant à améliorer sa vitesse d'exécution. Il s'agit ici, comme dans le cadre de l'analyse statique, d'un effort supplémentaire pour le développeur là où certaines autres techniques d'optimisations (notamment interne aux mécanismes d'interprétations) ne dépendent pas de lui.

On retrouve ainsi par exemples des guidelines provenant de **Google** sur comment développer de manière optimisée en **JavaScript** pour leur interpréteur **V8**, et ce en créant du code permettant de faciliter les opérations d'optimisation de l'interpréteur en lui donnant différents indices sur le code à exécuter[52].

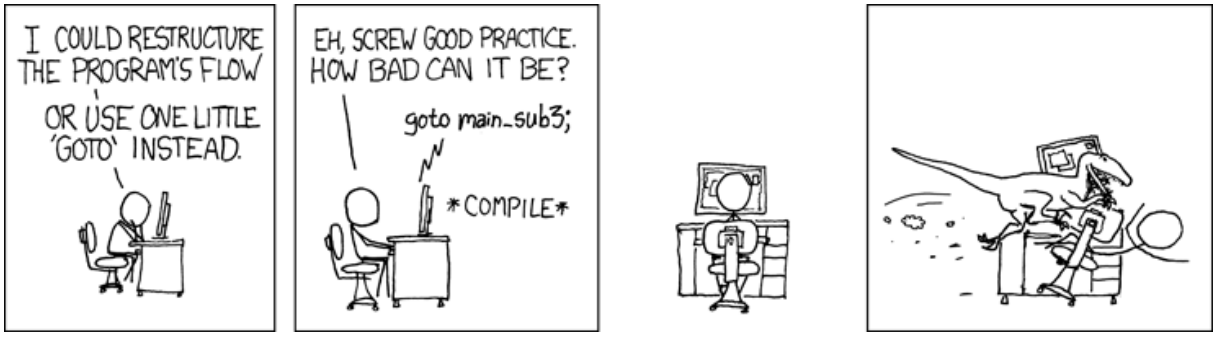


Illustration de xkcd sur le respect des « bonnes pratiques » en développement

BIDING À DES LANGAGES NATIFS

Le **binding** désigne l'action de lier un langage avec un autre. On trouve généralement du binding d'un langage haut niveau vers un langage de plus bas niveau, permettant pour celui-ci de profiter des différents avantages proposés par le bas niveau tout en proposant au développeur une interface haut niveau. Ils se différencient des méthodes dites « *built-in* » car il ne s'agit pas de composants essentiels au langage, et leur implémentation est extérieure au mécanisme d'interprétation.

Ainsi, l'idée est de généralement profiter de fonctionnalités système bas niveau incessible depuis le langage en question (ou difficilement accessible), ou bien d'avoir un contrôle plus grand sur une partie clef du programme. Il convient également de noter que l'implémentation doit quant-à-elle gérer

[52] Google io 2012, slide 25 : http://v8-io12.appspot.com/#25

correctement les différentes plateformes afin de proposer le même comportement. Celle-ci doit également être faite avec plus d'attention car les sécurités mises en place sont moins restrictives sur le langage bas niveau liés (certains éléments comment la gestion de la mémoire peuvent pas exemple devenir dépendantps du développeur).

Il est à noter que le *binding* peut également servir à réutiliser des fonctionnalités ou des libraires déjà existantes et reconnues dont le redéveloppement dans un nouveau langage n'est pas justifiable. Il s'agit alors ici de simplement proposer une interface haut niveau sur ces fonctionnalités, afin de pouvoir les exploiter directement depuis le langage haut niveau. Ce type de philosophie est par exemple très présent en **Python**, et a contribué à son expansion rapide.

Il existe des bibliothèques de *binding* dans tous les langages interprétés, mais ces techniques sont très utilisés avec **NodeJS** et **Python**, qui offrent tous deux des bibliothèques permettant de développer de manière facilité des plugins contenant des *bindings* vers du **C/C++**.

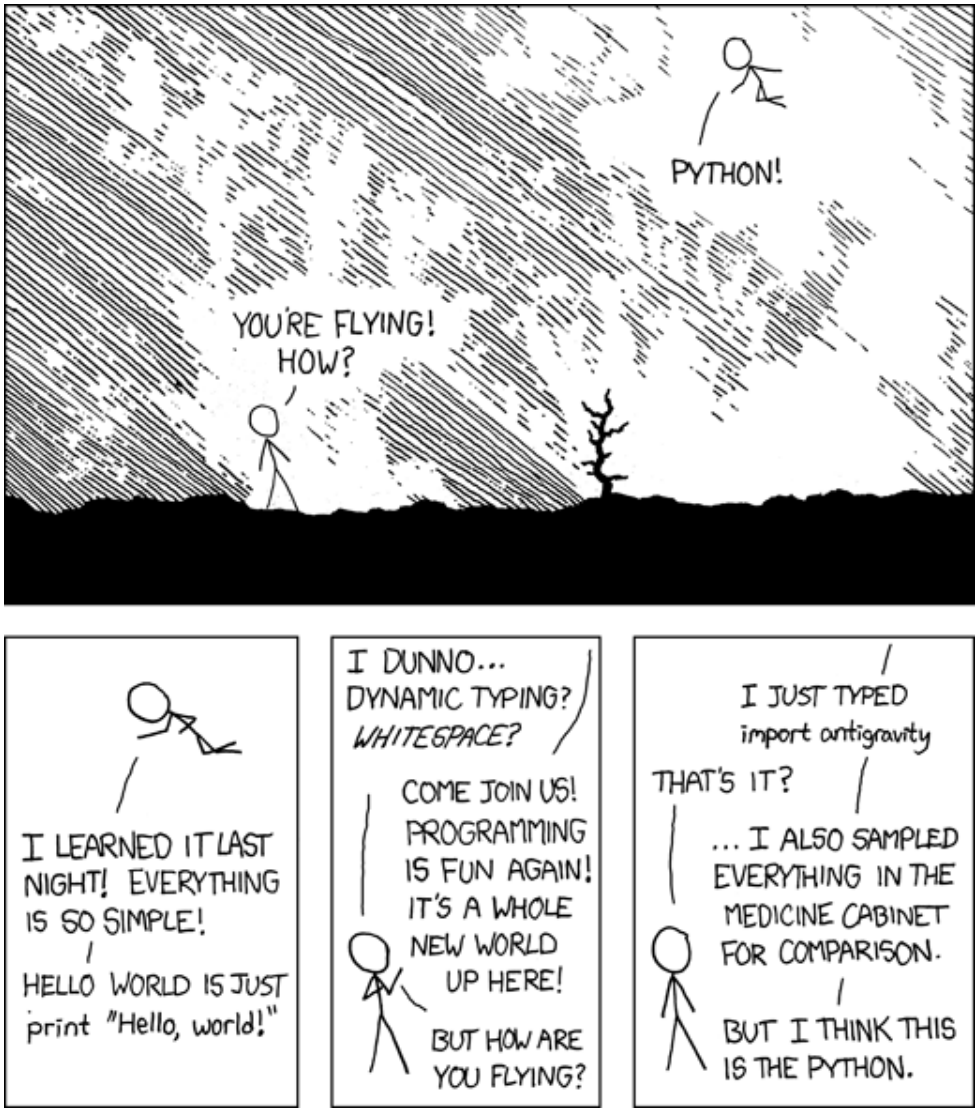


Illustration de xkcd sur les différentes bibliothèques proposées par le Python

PROBLÉMATIQUE DE SÉCURITÉ

OBFUSCATION

Un moyen efficace de répondre aux problématiques de propriété du code est d'effectuer des opérations d'obfuscation avant la publication d'un code. Il s'agit d'une manière générale de cacher les informations sensibles à l'intérieur du code, et ce sans altérer son comportement. Ces données sensibles peuvent être des clefs d'accès, des algorithmes sensibles ou même la logique de l'application ou du module en lui-même.

Il existe différents moyens permettant de mettre en place ces mécanismes, certains étant plus efficaces que d'autres mais plus complexes à mettre en œuvre.

Ces méthodes sont généralement mises en œuvre automatiquement avant les phases de mise en production, afin de ne pas altérer la qualité du code en phase de développement.

LE STYLE DU CODE

Le type d'obfuscation le plus simple à mettre en place concerne le style de l'application. Sans pour autant apporter de changements radicaux, celui-ci peut rendre plus difficilement lisible par un humain l'application sans pour autant réellement altérer les informations sensibles qu'elle pourrait contenir.

Cette opération peut être considérée comme semblable à celle de minification, puisque la suppression des espaces blancs et le remplacement des identifiants locaux permet de raccourcir le code, et ainsi de diminuer la taille des fichiers.

SUPPRESSION DES COMMENTAIRES ET DES CARACTÈRES BLANCS

Il s'agit ici simplement de supprimer les caractères n'ayant pas d'impacts sur le code. Ainsi durant cette phase relativement simple à mettre en place les commentaires et les espaces blanc ou autre saut de lignes (hors chaîne de caractère) sont supprimés.

Il est à noter que ce genre d'opération est inutile sur des langages compilés dans un langage intermédiaire comme le **Java** ou le **C#** où les commentaires disparaissent à la compilation et ne servent qu'à générer de la documentation à partir des sources (en plus de leur rôle de documentation dans le code).

Enfin il convient de noter que les langages utilisant justement des caractères blancs pour leur construction (comme c'est le cas du **Python**) rendent impossible ou plus complexe cette opération.

On retrouve généralement ce type d'opération au centre de la minification, et ce afin de réduire la taille des sources (voir partie sur la minification de code).

TRANSFORMATION DES IDENTIFIANTS LOCAUX

Afin de compliquer la compréhension du code sans altérer aucunement son fonctionnement il est également possible de modifier les identifiants de toutes les valeurs internes aux méthodes. En effet, ces valeurs internes n'entrent généralement pas en jeux dans les mécanismes de réflexivités et sont donc sans effet de bord sur le reste du code.

Il existe plusieurs méthodes pour renommer des identifiants, on peut par exemple nommer :

- **Méthode aléatoire**, où une chaîne d'un certain nombre de caractères est générée aléatoirement puis utilisée pour remplacer le nom de la valeur cible.
- **Méthode par « Overload induction »**, où on part simplement de « a » pour la première valeur, en incrémentant d'une lettre à chaque nouvelle valeur à obfusquer « b », « c », …, « aa », « ab », etc… Le bon point de cette méthode est qu'elle permet de réduire la taille des sources et d'augmenter l'efficacité des langages sans compilations stockant leurs variables dans des dictionnaires.
- **Méthode invisible**, où des caractères spéciaux sont utilisés comme nom de méthode. Cette technique est surtout possible depuis que de nombreux compilateurs possèdent des supports de l'UTF8/16/32, permettant des noms composés de caractères difficilement lisibles. Cette technique est par exemple souvent utilisée en **JavaScript** pour obfusquer du code, celui-ci n'ayant pas de restrictions sur les caractères composant ses noms de variables (hors mots réservés ou commençant par un chiffre).

LES DONNÉES

La transformation des données vise quant à elle à éviter que des données sensibles (valeurs spéciales, chaînes de caractères de validation, clef, etc…) ne puissent être extraites du code. Il est à noter que ces techniques ont généralement un impact sur les performances, et peuvent apporter des effets de bords si les variables en question sont utilisées dans des processus exploitant la réflexivité. On retrouve encore une fois plusieurs techniques :

- **Cryptage de chaîne de caractère**. Afin d'éviter que des chaînes puissent être directement extraites du code, il est possible de les chiffrer. Elles seront ensuite déchiffrées durant l'exécution à l'aide d'une clef noyée quelque part d'autre dans le code ou même directement à l'aide d'un morceau de code ajouté juste avant l'utilisation de la chaîne. Il est à noter qu'en suivant pas à pas (ou simplement en analysant un dump au bon moment) la chaîne finira presque par toujours revenir, mais rendra sa récupération moins aisée. Enfin, il convient de noter que cette méthode ajoute un certain temps, non négligeable d'exécution si de nombreuses chaînes de caractères sont obfusquées.
- **Transtypage**. Le transtypage vise à transformer un type pour complexifier le code. On peut par exemple réaliser des opérations de *boxing/unboxing* inutiles, ou encore décomposer une variable en plusieurs sous variables. Ici encore les performances pourront être dégradées par de trop nombreuses opérations inutiles de transtypage.
- **Modification de la visibilité**. Le principe de localité définie qu'une variable a le plus de chance d'être utilisée souvent dans le bloc où elle est définie. Il s'agit ici d'appliquer le paradigme inverse et d'essayer de passer toutes les variables locales en globales (que ce soit global à l'application, objet ou méthode) et ce afin de ne pas permettre de reconstituer facilement où est utilisée chaque variable.
- **Alias de variable**. Il s'agit ici de simplement créer de nombreux alias inutiles de valeurs, afin de rendre plus difficile de savoir quelle valeur est réellement utilisée à quel endroit. Ceci est notamment possible en utilisant des références.

LA STRUCTURE DU CODE

L'obfuscation de la structure de l'application permet de masquer totalement la logique de l'application. Elle va ainsi toucher aux différents éléments structurants du code, comme les classes et méthodes ou les structures de contrôle. On trouve notamment pour les classes les techniques suivantes :

- **Modification des noms de méthodes, de classes ou d'interfaces**, enlevant toute la logique présente dans celles-ci. On retrouve ainsi les mêmes techniques abordées pour l'obfuscation de variables locales (**aléatoire**, **overload induction** et **invisible**) permettant de définir de nouveaux noms générés par l'obfuscateur.
- **Modification des arbres d'héritage ou d'implémentations**. Il est par exemple possible de scinder en plusieurs classes des héritages ou d'y créer des étapes inutiles.
- **Modification des design patterns** employés, afin de les rendre moins facilement reconnaissables et exploitables.

Une fois ces différentes méthodes appliquées, il devient alors très difficile de réellement comprendre la logique d'héritage ou d'implémentation de l'application, puisque même une fois reconstitué un diagramme de classe n'aura pas de sens exploitable. Il convient également de noter que ces techniques affecteront les performances et ne peuvent être appliquées que si la réflexivité n'est pas utilisée sur les éléments en questions.

On trouve également de l'obfuscation au niveau des différentes structures de contrôle de l'application :

- **Ajout de code mort** dans des conditions non-triviales. Il s'agit ici de créer des blocs de code qui ne seront jamais utilisés, afin de créer des chemins inutiles dans le code et d'augmenter sa complexité cyclomatique.
- **Ajout de conditions ou de boucle**, afin d'encore une fois créer inutilement de la complexité cyclomatique.
- **Ajout de fausses méthodes inutilisées**, afin de créer du « bruit » dans l'application.

L'intérêt principal de ces techniques est de créer du « bruit » dans le code, en augmentant sa complexité et ses chemins possibles, et ce afin de perdre un éventuel logiciel de désobfuscation essayant de reconstituer le code.

SANDBOXING

Le sandboxing, anglicisme signifiant « bac à sable », désigne généralement un environnement contrôlé permettant d'exécuter en toute sécurité du code de provenance inconnue.

L'ensemble des ressources de l'ordinateur où se déroule l'exécution n'est ainsi pas accessible directement par le code, mais uniquement de manière contrôlée ou alors soumis à l'acceptation de l'utilisateur.

On retrouve généralement, dans les cas classiques, ce comportement sur les sites web et les différents langages qui y prennent place. Ainsi, le **JavaScript** (dans les navigateurs) n'a qu'un accès très limité au disque dur et aux ressources de l'ordinateur, et ne peut qu'indirectement y accéder (par exemple, par

l'intermédiaire de l'utilisateur qui lui donne accès à un fichier). Il s'agit ici clairement de limiter les risques encourus par la machine si le code en question est malveillant.

Ce moyen est également utilisé pour les applications « lourdes » web, de type **Flash**, **Java** ou **Silverlight**. Il permet ici de cloisonner l'exécution de ces logiciels tout en leur conférant des performances d'applications lourdes classiques, puisqu'utilisant les mêmes outils.

Malgré tout il convient de noter que comme cité précédemment dans les limites de sécurité, les sandboxing actuels, et notamment **Java** et **Flash**, sont souvent décriés comme vecteur possible d'attaques malveillantes.

MISES À JOUR

Un autre moyen de répondre rapidement et de manière efficace aux différentes vulnérabilités reste le déploiement massif et rapide de correctifs. Ce déploiement permet de rapidement combler les différentes failles existantes et connues, et de minimiser les vecteurs d'attaques possibles.

On trouve ainsi généralement des mécanismes de mise à jour sur les différents interpréteurs et machines virtuelles, qu'ils soient à l'intérieur d'un autre logiciel (cas de navigateurs web) ou bien directement (cas du **JRE** d'**Oracle**, qui possède son propre système de mise à jour)

Malgré tout comme annoncé dans les problématiques de sécurité, ces déploiements ne sont pas toujours correctement mis en œuvre dans les entreprises.

ET LE FUTUR ?

WEB ASSEMBLY

Le **web assembly**, littéralement assembleur web, est un langage de programmation bas niveau à destination des navigateurs web, et propulsé par certains des plus grands acteurs du monde informatique : **Mozilla**, **Google**, **Microsoft** et **Apple**. Il ne vise pas à proposer une alternative plus performante au **JavaScript**, mais à combler le manque présent entre le navigateur et le langage haut niveau qu'est le **JavaScript**. Ainsi, certaines opérations qui ne tirent pas partie du niveau d'abstraction élevé du **JavaScript** (traitement de données, streaming, etc…) pourraient gagner à être réalisées dans un langage plus bas niveau offrant des performances supérieures. Enfin, il est à noter que *web assembly* n'en est qu'a ses tout débuts, et n'est pas encore fixé dans sa forme ou sa syntaxe.

Contrairement à **ASM.js** (Qui est un subset du **JavaScript**), le **web assembly** est bien un nouveau langage qui se veut bas niveau, donnant accès à de nombreuses fonctionnalités absentes (et qui n'auraient pas forcément leur place dans un langage haut niveau) du **JavaScript**. On peut par exemple citer[53] :

- Typage statique des différentes valeurs.
- Typage des prototypes de méthode, permettant de savoir à l'avance quelles valeurs sont acceptées et retournées.
- Mémoire directement adressable, dans la lignée ce qu'il est possible de faire en **C/C++** (dans les limites de la sécurité de l'exécution dans un navigateur)

L'ensemble de ces possibilités vise à donner au développeur une liberté quasi-totale de développement, comme on peut le trouver dans des langages compilés en code natif.

Le **web assembly** se veut également plus « simple » dans son fonctionnement, et ne propose donc aucun système d'héritage ou de prototypage d'objets, ni aucun *garbage collector* ; il s'agit principalement de donner un contrôle total à l'utilisateur et de limiter les éléments sur lequel il n'aurait pas la main.

Même si ce langage propose de nombreux aspects bas niveau, il ne s'agit pas pour autant réellement d'un *bytecode*. Ainsi, celui-ci restera directement lisible avec un éditeur de code classique (pas de passage en binaire, en opposition aux bytecode **Java** et **CLI**), et est composé d'un arbre syntaxique abstrait possédant notamment des instructions de conditions ou de boucles. Ainsi, **web assembly** se veut à la fois comme une cible possible de compilation, mais aussi comme un langage de développement à part entière[54]. Cette double casquette lui permettra donc à la fois d'être utilisé directement dans le développement de modules cherchant la performance, et comme résultat de compilation d'un autre langage plus haut niveau (à la manière d'un *bytecode*).

---

[53] Web Assembly semantic : https://github.com/WebAssembly/design/blob/master/AstSemantics.md#local-and-memory-types

[54] What is WebAssembly? : https://medium.com/javascript-scene/what-is-webassembly-the-dawn-of-a-new-era-61256ec5a8f6

La première implémentation de **web assembly** utilisera **AMS.js** pour fonctionner, et ce afin de tirer parti des différentes implémentations et optimisations déjà existantes. Le projet vise ensuite une implémentation à part entière (lui permettant de fonctionner sans passer par **ASM.js**), mais réutilisant l'interpréteur **JavaScript** déjà en place. Les premières idées visaient à utiliser une implémentation de la **JVM** à l'intérieur des navigateurs, afin de réutiliser des composants existants. Malheureusement cela aurait signifié alourdir les navigateurs et le processus d'interprétation en maintenant deux interpréteurs différents (l'un pour le **web assembly**, l'autre pour le **JavaScript**)

LOI DE MOORE ET ÉVOLUTION DU MATÉRIEL

Gordon Moore, co-fondateur d'Intel a affirmé que le nombre de transistors par circuit allait doubler chaque année, entraînant de facto une évolution exponentielle des performances. Même si cette loi empirique a par la suite été rectifiée et risque de se heurter aux limites de la miniaturisation, elle reste dans son ensemble vérifiable.

Ainsi, au fil du temps les différentes machines ont vu les performances décuplées. Ce qui était impossible pour il y a 10 ans, car dépendant des performances, l'est maintenant. On trouve également plus récemment cette évolution dans le monde du mobile et des objets connectés : des choses auparavant réservées aux ordinateurs de bureau comme la navigation web ou les jeux vidéo en 3D sont maintenant possibles sur des terminaux mobiles ou différents objets connectés. Cette évolution de la puissance disponible profite également aux différents langages interprétés, leur ouvrant des possibilités qui leurs étaient auparavant fermées. Ainsi, le monde du web s'est par exemple désormais étendu sur mobile et des langages interprétés comme le JavaScript ou le Python font peu à peu leur apparition sur de nouveaux types de logiciels (serveurs web, application lourde, et même jeux vidéo).

Même si l'augmentation de la puissance du matériel n'est pas une réponse à l'ensemble des problématiques soulevées par les différents langages interprétés, elle peut néanmoins répondre aux deux principaux facteurs considérés aujourd'hui comme majoritairement limitants : la consommation processeur et mémoire.

Cette augmentation des performances, conjointement avec la recherche d'optimisations toujours plus poussés, permet donc d'ouvrir des nouvelles pistes de développement à ces langages.

LA MONTÉE EN PUISSANCE DU WEB

On ne peut également mentionner le futur des langages sans parler de la prise d'importance relativement récente du monde du web, du cloud et du dématérialisé. Ce changement a entrainé de nombreuses modifications dans nos manières de penser, la place du système d'exploitation diminuant à mesure que de plus en plus d'applications sont directement disponibles dans un navigateur internet.

Ce changement a favorisé la popularisation des différents langages du web, qui sont enfin vu comme des compétiteurs très sérieux aux langages plus lourds comme le **Java** ou le **C#**. Bénéficiant d'une image plus jeune et dynamique (même si du même âge), ceux-ci opèrent généralement sur un couche d'abstraction encore plus élevée.

On peut s'attendre sans surprise à voir de plus en plus d'applications directement déplacées dans le cloud, et une mort lente mais certaines des applications lourdes classiques. Seules celles demandant des accès spécifiques, des fonctionnalités avancées ou des performances très importantes pourront justifier de rester locales. Ce changement du paysage logiciel risque de forcer les langages « traditionnels » que sont le **Java** et le **C#** à se tourner plus vers des applications serveurs ou bas niveau dans le cas du **C#** et de son implication dans le système **Windows**.

UN LANGAGE POUR LES GOUVERNER TOUS ?

Bien qu'il soit irréaliste de vouloir créer un langage permettant de répondre à tous les besoins imaginables (de par leur multitude et hétérogénéités), il est néanmoins possible d'effectuer la réflexion de ce que devrait contenir (dans son implémentation aussi bien que dans ses spécifications) un langage « parfait ».

FONCTIONNALITÉS

Notre langage devrait tout d'abord proposer les fonctionnalités habituelles haut niveau qu'on retrouve chez tous les langages interprétés :

- Une gestion de la mémoire contrôlée et automatique, en passant par un garbage collector. Ici comme dans tous les langages le but est de proposer une simplification et une sécurisation des opérations d'allocation de mémoire,
- Indépendance de la plateforme,
- Typage dynamique avec sécurité optionnelle des types,
- Outils de réfléxivité permettant au programme de modifier son comportement en fonction de sa propre structure,
- Une bibliothèque « built-in » contenant les outils les plus nécessaires à la construction d'un programme. Exactement comme le **Java** et le **JRE**, ou le **C#** et le **.Net**, un langage se doit de proposer un ensemble de base d'outils

GESTION DES DÉPENDANCES

Il serait aussi un grand plus de posséder un système simple et fonctionnel de gestion de dépendances et de librairie, généralement sous la forme d'un logiciel à part entière livré avec l'interpréteur en lui-même.

On peut par exemple citer **npm** pour **NodeJS**, ou **pip** pour **Python**. Ce passage par un logiciel centralisateur pour la gestion des bibliothèques permet de conserver une certaine clarté dans les bibliothèques disponibles, de favoriser la communauté en plus de rendre plus facile le déploiement d'applications. On peut citer par opposition le **Java** qui ne possède pas d'outils livré permettant de gérer les dépendances et leurs déploiements ; on passe alors souvent par des logiciels externes comme **maven** et **gradle**.

BINDING NATIF

Un langage moderne doit également pouvoir facilement faire appel à du code écris dans d'autres langages, généralement par le biais d'API utilisant des langages bas niveau comme le **C++** ou le **C**. Cette API, sa documentation et sa facilité d'utilisation sont primordiales afin de permettre la réutilisation de code et bibliothèques existantes.

On trouve ainsi en **NodeJS** ou en **Python** des API et des outils pour créer des plugins appelant du code natif, permettant d'enrichir le langage en proposant soit même des extensions.

DÉPLOIMENT, COMPILATION ET INTERPRÉTATION

Il doit également proposer des outils permettant de créer des projets et de les compiler ou déployer de manière simplifiée. A l'instar du **Makefile** du **C**, ces outils doivent permettre à ceux qui le souhaitent de se passer d'IDE pour de micro-projets, sans pour autant restreindre leur utilisation.

Ce langage doit à la fois être compilable dans un bytecode pour son déploiement, et être directement interprétable afin de proposer le « meilleur » des deux mondes pour l'utilisateur.

Il sera ainsi possible de l'exécuter directement comme un simple script, mais également de le compiler pour ensuite le déployer.

SÉCURITÉ

Le langage devrait proposer des mécanismes de sandboxing permettant par exemple de manipuler du code d'origine inconnue sans risque, et ce en limitant ses actions possibles.

Il devrait également être en mesure de n'exposer via réflexivité qu'une partie souhaitée de lui-même, empêchant d'utiliser les autres composants. Cette notion pourrait également permettre de mettre en place des compilations plus agressives pour ces implémentations non exposées, apportant un gain potentiel non négligeable.

Enfin, il convient également de préciser qu'un langage interprété a vocation plutôt générique ne doit pas se soucier d'opération trop bas niveau, ni même y avoir accès. Autoriser ces accès peut causer des problèmes de sécurités, et surtout alourdis le langage en proposant des fonctionnalités qui seront peu utilisées. Il convient également qu'exposer des API bas niveau au travers du langage n'est pas contraire à ce principe, puisque ces appels sont wrappés dans le langage en lui-même. On pourrait citer le contre-exemple du **C#** qui permet de nombreuses opérations très bas niveau, pouvant conduire à des manipulations directes de la mémoire et les conséquences associées.

IMPLÉMENTATION

La meilleure implémentation semble être le passage par un *bytecode*. Celui-ci permet d'effectuer déjà une première phase de compilation et de vérification avant l'exécution du code. Dans le cas où le langage est utilisé comme un script, cette phase de compilation prendra simplement place de manière implicite avant l'exécution du code.

MACHINE VIRTUELLE & ENVIRONNEMENT D'ÉXÉCUTION

Le passage par une machine virtuelle permet de dissocier fortement le langage de son environnement d'exécution, permettant par exemple de réutiliser le langage *bytecode* de cette machine virtuelle comme cible de compilation pour d'autre langage.

Cette machine virtuelle devrait utiliser tout ce qui se fait en matière d'optimisation :

- Compilation Just In Time, en analysant les parties de code étant utilisées le plus souvent et en les compilant suivant leur profil d'utilisation

- Mise en cache de ces parties compilées, conservant pour chaque partie de code un nombre défini de compilations, et ce afin de proposer au prochain lancement de meilleures performances
- Recompilation dynamique. Si le profileur interne détecte qu'une méthode est utilisée d'une manière différente, il doit être en mesure de refaire une nouvelle version compilée pour cette optimisation
- Différentes optimisations, citées dans la partie des solutions au problème des performances. Celles-ci doivent permettre de gagner du temps dans les parties compilées afin de les rendre encore plus efficaces.
- Système de heap dynamique, avec gestion des générations d'éléments.

Ces différentes techniques devraient proposer des performances relativement proches d'un code compilé en natif, et sans pour autant impacter le langage en lui-même.

INSPIRATIONS

Comme nous pouvons le voir ce langage proposerait des fonctionnalités empruntées à tous les langages existants. On peut par exemple citer, dans le désordre :

- **Hack lang**, un subset strict du **PHP** qui propose un typage des variables, tout en conservant le typage dynamique du **PHP** : il devient alors possible de continuer d'exploiter des méthodes sans type d'entrée défini ou des méthodes renvoyant des valeurs de type divers ; mais également de s'assurer de la sécurité de type de variable
- **Java**, langage le plus utilisé au monde, proposant une machine virtuelle aux optimisations impressionnantes
- **JavaScript**, fonctionnant aussi bien à l'intérieur d'un navigateur que comme application. On peut également citer **npm**, ou même la communauté grandissante autour de **nodeJs**
- **C#** qui se veut résolument plus proche de la machine, proposant par exemple d'utiliser du code « unmanaged » permettant une gestion très fine de la mémoire
- **Python**, qui fonctionne comme un langage de script alors qu'il s'agit réellement d'un langage compilé en bytecode et exécuté par une machine virtuelle
- Et bien d'autres…

CONCLUSION

Comme nous pouvons le voir il existe une multitude de langages, ayant tous leurs points forts et leurs faiblesses, et dépendant également des approches proposées à un même problème. Ainsi la philosophie d'un programme en **Java** ne sera pas la même qu'un programme en **JavaScript**, de sa syntaxe jusqu'à l'idée derrière la résolution d'un même problème.

Comme indiqué dans la partie sur la fragmentation des connaissances, la création d'un nouveau langage ne ferait en aucun cas l'unanimité au seins de la communauté des développeurs, créant dans le meilleur des cas un nouveau langage en compétition avec les autres. Il est enfin à noter que les DSL sont moins concernés, puisque construits pour combler un réel manque dans un domaine spécifique.

CONCLUSION

L'INTERPRÉTATION ET LES MACHINES VIRTUELLES À LA MODE

Qu'il s'agisse du monde du web, du monde des serveurs ou même des clients lourds les langages interprétés sont très présents. Ainsi, en 2015 l'indice **TIOBE** classait parmi les 10 langages les plus populaires 8 interprétés[55] : **Java**, **Python**, **C#**, **PHP**, **VB.Net**, **JavaScript**, **Perl** et **Ruby**.

Conjugué avec la prise d'importance relativement récente du monde du web, les langages interprétés sont plus qu'un effet de mode : Ils apportent des atouts non négligeables et des facilités de développement dans des applications réservées auparavant à des langages plus complexes.

Malgré les différentes limites citées dans le document, de nombreuses méthodes permettent de les contourner ou au mieux de les diminuer pour les rendre négligeables dans la majorité des cas d'utilisations possibles de ces langages. Ainsi, il devient pensable de réaliser dans ces langages des choses qui leur étaient auparavant impossibles, augmentant leur champ d'action et contribuant à leur déploiement. On trouve par exemple de nettes améliorations dans les moteurs **JavaScript** des différents navigateurs, la guerre des performances faisant rage entre le **V8** de **Google**, **SpiderMonkey** de **Mozilla** et **Chakra** de **Microsoft**. Du côté des applications lourdes, le **JRE** d'**Oracle** propose des mécanismes d'optimisations toujours plus poussés en plus de maintenir dans l'air du temps le langage **Java** avec sa récente version 8 empruntant au fonctionnel. Le **C#** profite lui aussi de ces évolutions, proposant toujours plus de fonctionnalités et des performances toujours plus proche du **C++**.

Ces limites sont donc de moins en moins présentes, et demeurent uniquement bloquantes dans des applications spécifiques nécessitant par exemple beaucoup de puissance, comme des jeux vidéo (Où le **C++** est majoritaire), ou des applications très bas niveau où les couches d'abstractions élevées desserviraient plus qu'elles ne serviraient.

On peut également mentionner un certain élan communautaire autour de ces langages, **NodeJS** ou **Python** possédant par exemple une communauté très active, offrant de nombreuses pistes d'innovations pour ces technologies.

LES PISTES D'ÉVOLUTION

Le constat actuel semble montrer que le meilleur moyen de conserver une certaine portabilité sans sacrifier les performances revient au passage par une phase de compilation avant l'utilisation. On retrouve ainsi cette phase de compilation dans bon nombre d'implémentations modernes, qu'elle soit à la charge du développeur comme en **Java** ou **C#** ou bien à la charge de l'interpréteur comme en **Python**. Les récentes initiatives du **webAssembly** et d'**ASM.js** semblent aller dans ce sens, prouvant qu'une phase de compilation, ou même simplement de transpilation, permet d'apporter des optimisations conséquentes dans le langage, sans pour autant empêcher une seconde phase d'optimisation dynamique au *runtime.*

Cette seconde phase d'optimisation dynamique semble elle aussi à ses balbutiements, proposant encore des performances qui ne sont pas comparables avec les langages natifs tout y ajoutant un

[55] TIOBE Index for December 2015 : http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

surcoût non négligeable. Ces techniques, comme pour le moteur **V8** de **Google**, empruntent généralement beaucoup aux mécaniques déjà connues des compilations classiques, et ne semblent pas encore profiter clairement de ce que l'optimisation dynamique rend possible. Ces optimisations à la volée s'adaptant au profil d'utilisation du code pourraient à terme permettre de proposer des performances plus intéressantes que leurs homologues compilés n'ayant pas la capacité de s'adapter à leur utilisation.

On peut également citer la compilation lors du déploiement ou du premier lancement de l'application, comme en **C#** où sur **Android** avec **ART**. Cette technique permet logiquement de proposer des performances de code compilé, tout en conservant les différents avantages et sécurités déjà mis en place.

Ces différentes techniques, couplées avec les différentes optimisations et des mécanismes de recompilations dynamiques, assurent des performances de très bonne qualité qui se rapprochent des langages compilés, voir les dépassent sur certaines portions de programmes.

LA PLACE DES LANGAGES INTERPRÉTÉS

Enfin il convient également de noter que les langages interprétés peuvent cohabiter « pacifiquement » avec leurs homologues compilés, chacun couvrant des domaines spécifiques et proposant des outils adaptés à ceux-ci.

Python est l'exemple parfait de cette symbiose entre haut et bas niveau. Ici le langage interprété manipule de manière transparente des implémentations dans son propre langage et des implémentations en **C**, proposant quand il le faut des bindings à des librairies bas niveau assurant de n'avoir que peu de redondance de code, en plus d'assurer la rapidité de l'opération.

Certains autres langages, comme le **C#** fonctionnent aussi très bien avec des modules écris en **C++**, permettant d'interfacer de manière simple pour le développeur des outils déjà existants et fonctionnels, et ce malgré sa volonté de se présenter comme un langage « bas niveau ».

Cette répartition des tâches permet ici de contrer les limites de performances appliquées plus haut : si le besoin de vitesse est crucial, le passage par un langage compilé prend alors tout son sens.

LE FUTUR DES LANGAGES INTERPRÉTÉS

Les langages interprétés ont donc devant eux un futur certain. Comme mentionné dans ce document, la majorité de leurs limites peuvent trouver des solutions et des contournements, ne présentant des points bloquants que dans de rares cas spécifiques. Leurs avantages et leur facilité d'approche en font des langages de choix quelle que soit la plateforme ou le type d'application effectuée. Ainsi leur rôle ne se limite plus aux monde du web et des scripts, et on trouve leurs utilisations dans de nombreuses technologies novatrices et prometteuses comme le **big data** ou le **cloud**.

Les nombreuses initiatives visant à améliorer l'existant promettent une amélioration constante des performances dans les années à venir, ouvrant par exemple des possibilités au monde du web qui lui sont aujourd'hui encore fermées.

Leur popularité en fait aujourd'hui des acteurs incontournables du monde du développement logiciel.

ANNEXES

BIBLIOGRAPHIE ET SOURCES

http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

- Index de **TIOBE**, retraçant chaque année la liste des langages les plus populaires.

http://gcc.uni-paderborn.de/www/WI/WI2/wi2_lit.nsf/64ae864837b22662c12573e70058bbb4/abf8d70f07c12eb3c1256de900638899/$FILE/Java%20Technology%20-%20An%20early%20history.pdf

- PDF retraçant toute l'histoire du **Java** de sa création jusqu'en 1995.

http://www.typescriptlang.org/

- Site officiel du **TypeScript**, un subset du **JavaScript** avec un typage plus fort et des classes. Le **TypeScript** est ensuite transpilé en **JavaScript**.

https://babeljs.io/

- Si officiel de **Babel**, un transpileur **JavaScript** permettant d'écrire du **JavaScript** respectant la norme **ECMAScript 6** tout en restant compatible avec **ECMAScript 5**. Le site propose des exemples et d'utiliser directement en ligne le transpileur.

https://msdn.microsoft.com/fr-fr/library/6t9t5wcf(v=vs.110).aspx

- Article d'information et d'aide de **Microsoft** sur **Ngen.exe**, permettant d'effectuer une tâche de compilation **AOT** au lieu d'utiliser des mécanismes de compilation JIT sur du **langage C#.**

http://www.telerik.com/blogs/understanding-net-just-in-time-compilation

- « Understanding .NET Just-In-Time Compilation » est un article sur le fonctionnement interne des mécanismes de compilation JIT entrant en jeux dans le **CLR** de **Microsoft**.

http://www.oracle.com/technetwork/articles/java/architect-evans-pt1-2266278.html

- « Understanding Java JIT Compilation with JITWatch » est un article sur le fonctionnement interne de la **JVM** d'**Oracle** et ses mécanismes internes de compilation JIT. L'article utilise le logiciel **JITWatch**, permettant d'analyser « à chaud » le contenu de la **JVM**.

https://source.android.com/devices/tech/dalvik/index.html

- Document de **Google** présentant les fonctionnalités d'**ART** par rapport à **Dalvik**.

http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html

- Liste des spécifications de la **JVM** de **Java 6**, permettant de mettre en avant par exemple quelques fonctionnalités haut niveau présentes dans le bytecode Java. Même si **Java 6** n'est plus maintenu, ses spécifications restent une bonne source d'informations.

https://docs.oracle.com/javase/tutorial/security/tour1/

- Tutorial d'**Oracle** sur le fichier « Policy » permettant de définir des droits en **Java**.

http://blogs.msdn.com/b/brada/archive/2005/01/12/351958.aspx

- Billet de blog traitant de la différence entre le **CLR** de **Microsoft**, et une « machine virtuelle » classique. Il conclue qu'il s'agit plus d'une convention de nommage propre à **Microsoft** qu'une véritable différenciation avec les mécanismes classiques.

http://www.scala-lang.org/docu/files/collections-api/collections_46.html

- Utilisation en **Scala** des différentes librairies présentes en **Java**.

https://cs263-technology-tutorial.readthedocs.org/en/latest/

- Tutoriel présentant le fonctionnement interne de l'implémentation **CPython**, interpréteur de référence de **Python**.

http://blog.jetbrains.com/idea/2014/07/intellij-idea-14-eap-138-1029-is-out/

- Article de **Jetbrains** indiquant que leur IDE **Java**, **IntelliJ**, sera désormais livré nativement avec un décompileur **Java** afin d'accéder aux codes de Jars n'ayant pas de sources attachées.

http://www.legifrance.gouv.fr/affichCodeArticle.do?cidTexte=LEGITEXT000006069414&idArticle=LEGIARTI000006278920&dateTexte=&categorieLien=cid

- Code de la propriété intellectuelle définissant les droits relatifs à la retro-ingénierie et à la décompilations de logiciels

http://www.oracle.com/technetwork/java/javase/compatibility-417013.html#incompatibilities

- Incompatibilités entre **Java 6** et **7**, listés sur le site d'**Oracle**.

http://www.oracle.com/technetwork/java/faq-sun-packages-142232.html

- Page d'aide d'**Oracle** indiquant pourquoi un développeur ne doit jamais appeler les packages « sun », car ceux-ci peuvent changer.

https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript

- Histoire rapide du **JavaScript** sur le site du **W3C**.

http://kangax.github.io/compat-table/es5/

- Liste des compatibilités **ECMAScript 5** sur les différents navigateurs et interpréteurs.

http://kangax.github.io/compat-table/non-standard/

- Liste des compatibilités des éléments non standards du **JavaScript**.

http://community.websense.com/blogs/securitylabs/archive/2013/09/05/new-java-and-flash-research-shows-a-dangerous-update-gap.aspx

- Article de 2013 mettant en avant le retard dangereux de mises à jour de sécurité sur **Java** et **Flash**, et ce dans le milieu de l'entreprise.

https://www.ftc.gov/system/files/documents/cases/151221oraclecmpt.pdf

- Plainte de la **FTC** sur le comportement d'**Oracle** sur son système de mises à jour concernant **Java SE**.

https://www.apple.com/hotnews/thoughts-on-flash/

- Commentaire d'**Apple** sur **Flash**, notamment au sujet de leur choix de ne pas l'implémenter pour leur OS mobile.

http://www.leparisien.fr/actualite-people-medias/internet-adobe-flash-bloque-par-le-navigateur-mozilla-et-combattu-par-facebook-14-07-2015-4944421.php

- Article sur comment **Flash** est maintenant bloqué par défaut sur le navigateur web **Mozilla Firefox**.

http://www.theguardian.com/technology/2015/aug/24/adobe-flash-dying-amazon-google-chrome

- Article traitant de la mort proche de la technologie **Flash** sur le web, et pourquoi cette disparition est une bonne chose.

https://www.exploit-db.com/papers/13694/

- Exploit **PHP** utilisant la méthode « Eval », permettant d'illustrer comment cette méthode peut être vectrice de vulnérabilités si mal utilisée.

https://developers.google.com/web/fundamentals/performance/?hl=en

- Performance guideline de **Google** pour le développement web.

https://developers.google.com/closure/compiler/

- **Closure compiler** de **Google**, permettant de faire une transpilation **JavaScript** à **JavaScript**, avec une passe d'analyse statique.

http://asmjs.org/spec/latest/

- Spécifications d'**ASM.js**.

http://kripken.github.io/emscripten-site/

- **Emscripten**, implémentation de **Web Assembly**.

http://kripken.github.io/mloc_emscripten_talk/#/

- Présentation d'**ASM.js** mettant en avant sa raison d'être et ses applications possibles.

http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html

- Article d'**Oracle** sur les gains apportés par la version **HotSpot** de la **JVM**.

http://v8-io12.appspot.com/

- **Google IO** de 2012, présentant notamment le **V8** et les bonnes pratiques de développement associés.

http://www.pylint.org/

- Page de **Pylint**, implémentation de **lint** pour **Python**.

https://github.com/WebAssembly/design/blob/master/AstSemantics.md#local-and-memory-types

- Arbre syntaxique de **Web Assembly**.

https://github.com/WebAssembly/design/blob/master/AstSemantics.md#local-and-memory-types

- Article de blog sur **Web Assembly**

https://fr.wikipedia.org/

- L'encyclopédie libre, principalement en anglais et en français, sur les différents sujets traités

GLOSSAIRE

LANGAGE DE PROGRAMMATION

En informatique, un langage de programmation est une notation conventionnelle destinée à formuler des algorithmes. D'une manière similaire à une langue naturelle, un langage de programmation est composé d'un alphabet, d'un vocabulaire, de règles de grammaire, et de significations. Ceux-ci permettent de décrire à la fois les structures de données manipulées par la machine et les différentes transformations qui leurs seront appliquées.

Un langage de programmation est construit à partir d'une grammaire formelle, qui inclut des symboles et des règles syntaxiques, auxquels on associe des règles sémantiques.

Un langage n'est pas forcément lié à son implémentation de référence (que ce soit un compilateur ou un interpréteur), même si cette implémentation de référence est très souvent mentionnée indirectement par l'intermédiaire du langage de programmation en lui-même.

LANGAGE TURING-COMPLET

Le concept de « Turing-complet » est apparu suite à l'invention de la machine de Turing en 1936 par Alan Turing. Une machine de Turing est une machine abstraite, ou théorique, crée pour servir de modèle idéal lors d'un calcul mathématique. Par extension, tous les ordinateurs modernes sont conçus selon le principe de fonctionnement qu'elle présente.

En pratique, il s'agit d'un simple système de bande divisé en cases, d'une tête de lecture/écriture, un registre d'état qui mémorise l'état en court de la machine, et une table d'actions qui précise les interventions à réaliser pour la tête. Chaque case contient un symbole issu d'un alphabet connu (et contenant un symbole "vide", ou "0"). Cet alphabet se limite généralement à 0 et 1 - pour rester simple - et réalise des traitements binaires.

La machine de Turing servit, lors de sa création, à montrer la faisabilité d'un automate programmable capable de calculer toute fonction calculable. De nos jours, on qualifie généralement un langage de programmation de Turing-complet s'il peut calculer toutes les fonctions calculables. En pratique, il faut que le langage permette d'émuler une machine de Turing, ou une autre machine turing-complète. En général, les langages turing-complets comprennent les possibilités suivantes :

- L'allocation dynamique de mémoire
- La récursivité ou un autre moyen d'exécuter des boucles infinies
- L'exécution infinie (pas de garantie de fin de programme)
- Le lambda-calcul (défini par Alonzo Church en 1930)

Il convient également de noter qu'un langage de programmation n'a pas forcément besoin d'être Turing-complet. C'est par exemple le cas de nombreux langages *domaine specific* comme le SQL qui n'ont pas besoin de toutes les fonctionnalités d'une machine de Turing.

DOMAIN SPECIFIC (LANGAGE DE PROGRAMMATION)

Un **langage dédié**, ou ***domaine specific language*** en anglais, est un langage de programmation dont les spécifications sont conçues pour répondre aux contraintes d'un domaine d'application précis.

Il s'oppose conceptuellement aux langages de programmation classiques (ou généralistes) comme le **Java** ou le **C**, qui tendent à traiter un ensemble de domaines. Ces langages dédiés ne sont pas forcément Turing-complet et proposent des fonctionnalités fortement adaptées à leur utilisation.

Il convient également de noter qu'il n'existe aucun consensus ni définition officielle sur ce qu'est réellement un *domain specific language*.

PARADIGME (LANGAGE DE PROGRAMMATION)

Dans le langage courant un **paradigme** désigne une représentation du monde, une manière spécifique de voir les choses. Il désigne un modèle cohérent de pensée reposant sur une base définie de valeurs.

Appliqué au développement et à la programmation, un **paradigme** définie la manière dont la solution à des problèmes doit être formulé dans un langage informatique. Un paradigme de programmation fournit (et détermine) la vue qu'a le développeur de l'exécution de son programme.

On peut par exemple citer :

- **Programmation impérative**, décrivant les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme. Ce type de programmation est le plus répandu parmi l'ensemble des langages de programmation existants.
- **Programmation orientée objet**, consistant en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique. Il possède une structure interne et un comportement, et il sait interagir avec ses pairs. Il s'agit donc de représenter ces objets et leurs interactions.
- **Programmation déclarative**, consistant à créer des applications sur la base de composants logiciels indépendants du contexte et ne comportant aucun état interne. Autrement dit, l'appel d'un de ces composants avec les mêmes arguments produit exactement le même résultat, quel que soit le moment et le contexte de l'appel. La programmation fonctionnelle est un sous paradigme de la programmation déclarative.
- **Programmation événementielle**, consiste uniquement à répondre à des évènements arrivant dans le programme informatique.
- **Programmation orientée aspect**, qui permet de traiter séparément les préoccupations transversales, qui relèvent souvent de la technique (*aspect* en anglais), des préoccupations métier, qui constituent le cœur d'une application.

Il est à noter que les langages sont souvent multi-paradigmes, et permettent même parfois de les combiner au sein d'un même programme. Certains paradigmes (orienté aspect et événementielles par exemple) peuvent également être directement implémentés dans un langage de programmation, sans dépendre de celui-ci.

LANGAGE TYPÉ STATIQUEMENT

Le **typage statique** est une technique utilisée dans certains langages de programmation impératifs pour associer à un symbole dénotant une variable le type de la valeur dénotée par la variable. Ainsi il sera par la suite impossible d'affecter à une variable une valeur d'un type différent de celle-ci. On retrouve généralement les mêmes contraintes dans les prototypes de fonction.

Cette vérification de cohérence des types est définie durant la phase de compilation du programme.

LANGAGE TYPÉ DYNAMIQUEMENT

Le **typage dynamique** s'oppose au typage statique, puisqu'ici une variable ne contient pas l'information de son type et est donc affectable à tous type de valeurs.

La vérification sera ici effectuée durant le fonctionnement même du programme, et non pas durant la phase de compilation.

COMPILATEUR & COMPILATION

Un **compilateur** désigne de manière générale un logiciel permettant de transformer du code source écris dans le langage source en un autre langage cible.

Ce langage cible peut être natif dans le cas des compilations classiques, du bytecode, ou même un autre langage haut niveau.

La **compilation** désigne par extension l'utilisation du compilateur.

TRANSPILATION

La **transpilation** est une forme de compilation dont le langage cible est considéré haut niveau. On trouve généralement ce type de compilateur sur des langages interprétés, afin de profiter des différentes implémentations déjà existantes pour celui-ci, ou par contraintes techniques. Ainsi le JavaScript est par exemple souvent utilisé comme cible de transpilation.

COMPILATION AHEAD OF TIME

La **compilation prévisionnelle**, ou ***ahead of time*** en anglais, désigne une compilation prenant place avant que celle-ci ne soit nécessaire. On retrouve ce principe avec les mécanismes classiques de compilation en **C/C++** ou encore en **Java**, où le programme doit être compilé avant de pouvoir être exécuté.

COMPILATION JUST IN TIME

La compilation **juste à temps**, ou ***just in time*** en anglais, désigne une compilation « fainéante » ne prenant place que lorsque celle-ci est réellement nécessaire, que soit pour pouvoir être exécuté ou pour apporter un gain de performance. Il est à noter qu'elle est généralement effectuée uniquement sur des portions d'instructions, à la différence de la compilation prévisionnelle qui concernent généralement l'ensemble d'un programme ou d'une bibliothèque.

INTERPRÉTEUR & INTERPRÉTATION

Un **interpréteur** est un logiciel ayant pour tâche d'analyser, de traduire et d'exécuter les programmes écrits dans un langage informatique. On qualifie parfois, et abusivement, les langages dont les programmes sont généralement exécutés par un interpréteur de langages interprétés.

BYTECODE

Le **bytecode**, littéralement **code binaire**, représente des instructions d'un programme informatique écrites en binaire. Au contraire des langages destinés aux humains dont les instructions sont écrites en langue naturelle (ou presque), le bytecode se compose donc uniquement de code binaire.

Ce type de langage est ainsi généralement destiné soit à d'autres programmes informatiques, soit directement à l'ordinateur dans le cas de l'assembleur.

SANDBOXING

Dans le domaine de la sécurité des systèmes informatiques, un **sandbox** (anglicisme signifiant bac à sable) est un mécanisme qui permet l'exécution de logiciels avec moins de risques pour le système d'exploitation.

Le **sandboxing** désigne par extension le fonctionnement d'un logiciel dans un environnement sécurisé, où les actions possibles du logiciel sont soit limitées soit soumises à contrôle.

DÉCOMPILATION

La décompilation représente l'acte inverse de la compilation : il s'agit de partie du langage cible et de générer automatiquement ou manuellement les instructions dans le langage source.

Cette pratique peut être automatisée avec de bon résultats pour certains langages haut niveau (**Java** ou **C#** par exemple) mais devient plus complexe et moins probante avec des langages bas niveau compilés (**C** ou **C++** par exemple).

VULNÉRABILITÉ

Dans le domaine de la sécurité informatique, une vulnérabilité ou faille est une faiblesse dans un système informatique, permettant à un attaquant de porter atteinte à l'intégrité de ce système, c'est-à-dire à son fonctionnement normal, à la confidentialité et l'intégrité des données qu'il contient.

OBFUSCATION

L'obfuscation désigne les moyens mis en œuvre afin de complexifier la compréhension d'un programme informatique, qu'il soit compilé ou non.

FONCTIONNEMENT DES PRINCIPALES IMPLÉMENTATIONS

JAVA ET LA JVM

Malgré un *bytecode* relativement proche d'un code natif, la **Java Virtual Machine** possède une architecture beaucoup plus intéressante et complexe qu'au premier abord. Ainsi, il ne s'agit pas d'une simple machine exécutant instruction après instruction des fichiers binaires au travers de bindings natifs.

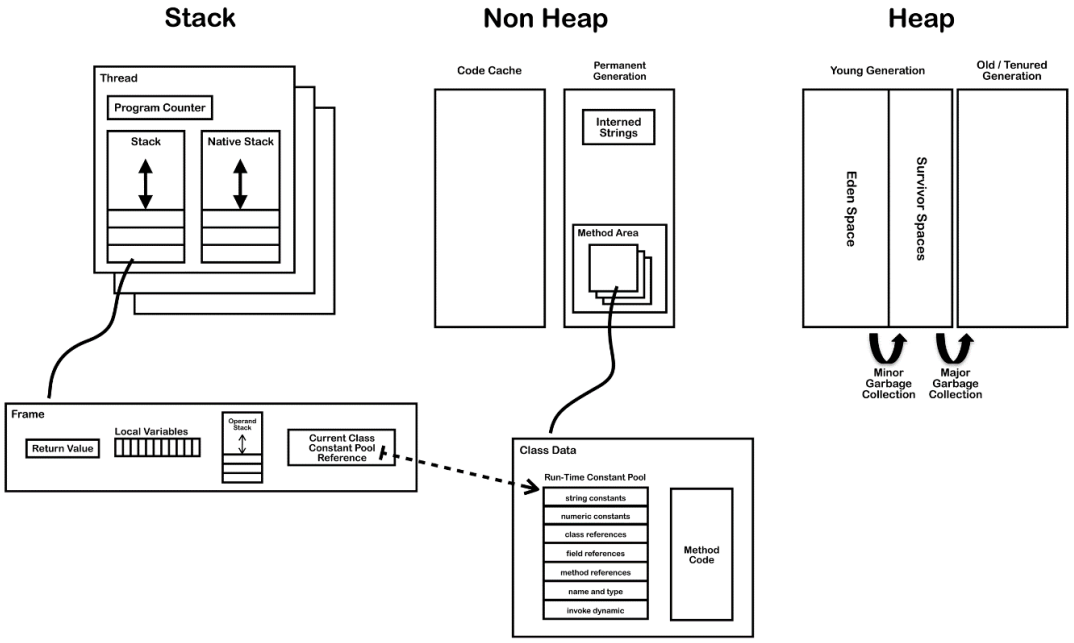
Une **JVM** se compose de plusieurs threads de base à son lancement :

- *VM thread*, qui s'occupe des opérations nécessitant un « safe-point » de la JVM,
- *Periodic task thread*, qui s'occupe des tâches périodiques,
- *GC threads*, qui s'occupent des opérations du *garbage-collector*,
- *Compiler threads*, qui s'occupent des opérations de compilation **JIT** en code natif
- *Signal dispatcher thread*, qui s'occupe d'interfacer les signaux du système avec les méthodes de la **JVM**

En plus de ces threads s'ajoutent ceux définis par l'utilisateur (ou les bibliothèques qu'il utilise) dans son application. Il s'agit des threads d'exécution.

La **JVM** se compose également d'une *heap*, zone d'allocation mémoire destinée aux instances et tableaux, et d'un compilateur **JIT**.

VUE D'ENSEMBLE



La *stack* contient la liste des *frames*, chaque frame étant relative au contexte d'exécution d'une méthode et contenant ses variables naturelles et références vers les instances (instances allouées dans la *heap*). À chaque appel, une nouvelle *frame* est construite et ajoutée à la stack. À la fin de cette méthode, la *stack* la plus récente est supprimée.

Les informations relatives à la classe/méthode elle-même (instructions par exemple) ne sont pas dans la *heap*, mais dans une mémoire *non-heap* contenant les allocations internes à la **JVM**.

THREAD D'EXÉCUTION

Chaque thread d'exécution de la **JVM** possède un ensemble de composants :

PROGRAM COUNTER

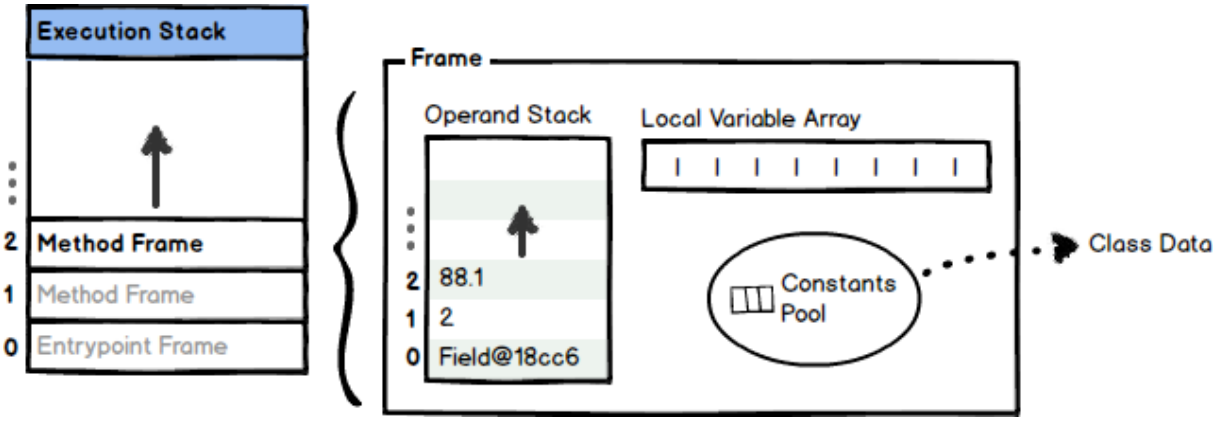
Program Counter, ou *PC*, est une valeur qui indique l'adresse de l'instruction en cours dans le thread. Cette valeur est à nulle si le code en question est natif.

STACK

Le *stack*, ou pile en français, contient une *frame* différente par méthode appelée, la dernière étant au-dessus de la pile, et la première en bas de la pile. En cas de débordement de cette pile, la fameuse erreur de débordement de pile « *StackOverflowError* » est lancée.

Dans le cas où le code est natif, alors la pile est la pile native. Si un appel depuis du code natif est effectué sur du code Java, alors la pile contenant les frames sera sollicitée pour cet appel.

FRAME

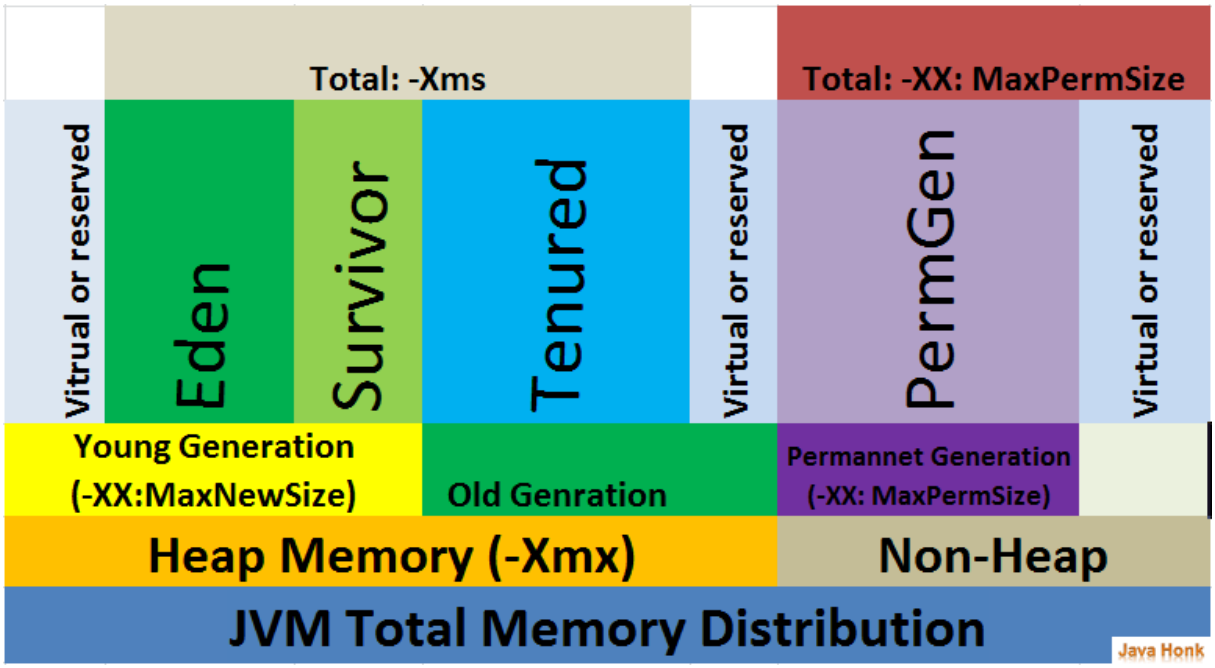


Une *frame* contient l'ensemble des données locales nécessaires à un appel de méthode. Celle-ci va en effet contenir :

- Le tableau des variables locales de la méthode. Ces variables peuvent être des types primitifs, ou bien des références vers des valeurs allouées dans la *heap* de Java. A noter que « *this* » est toujours en première position du tableau local des variables,
- La valeur de retour,
- Pile d'opération, utilisée pour les opérations locales dans la méthode, ils sont analogues aux systèmes de registres présent dans les processeurs et servent à rendre plus efficaces des opérations locales à la méthode,
- Référence au pool constant d'exécution, permettant d'effectuer dynamiquement la résolution des liens présents dans les codes natifs utilisés. Au contraire des mécanismes classiques de compilation où la résolution se fait généralement durant la compilation, en Java la résolution de ces liens est effectuée de manière dynamique uniquement quand nécessaire,

MEMORY MANAGEMENT

La gestion de la mémoire dans la **JVM** est séparée entre la mémoire *heap* et la mémoire *non-heap*. La mémoire *heap* va être utilisée par la **JVM** pour les allocations d'éléments créés par l'utilisateur, alors que la partie *non-heap* est utilisée pour le fonctionnement interne de la **JVM**.



Java utilise un système de compteur de référence pour garder la trace des objets qui sont encore utilisés et détecter ceux pour lequel leur compteur est passé à zéro. La libération n'est pas automatique et n'interviendra que si nécessaire. La taille de la *heap* n'est également pas fixée, et sera agrandie tant que nécessaire dans la limite définie par la **JVM**.

La mémoire Java se décompose en 2 grandes parties, la *heap* et la *non-heap*. D'une manière générale la *heap* contient les objets alloués par le logiciel (soit directement soit via des bibliothèques) et la non-*heap* les éléments relatifs au fonctionnement interne de la **JVM**.

HEAP

Le *heap*, littéralement « tas » en français, est utilisé pour allouer les instances de classe et les tableaux durant le fonctionnement de la **JVM**. Celui-ci est nécessaire car les frames précédemment évoquées ne peuvent pas changer de taille, et ne contiennent que des références vers ces éléments. Ainsi uniquement les valeurs naturelles sont envoyées par valeur entre appels de méthode et directement présent dans la *frame*, et à contrario les instances et tableaux sont quant-à-eux envoyés par références et alloués dans la *heap*.

La *heap* se sépare en trois catégories d'objets, par génération. La « *young generation* » représente des éléments venant d'être alloués ou étant en vie depuis peu de temps. Après une certaine durée ces éléments passent dans la « *old generation* » puis enfin la « *permanent generation* ». Comme vu sur le graphique ci-dessus, ces générations sont elles-mêmes sous divisées en plusieurs zones suivant la durée de vie des éléments.

GARBAGE COLLECTOR

La *garbage collector* agit uniquement sur la *heap*, et cherche périodiquement à libérer les éléments n'étant plus utilisés (dont le logiciel n'a plus aucune références). Il a également pour rôle de réorganiser la *heap* afin d'optimiser son utilisation.

Les phases de *garbage collector* mineures n'opèrent qu'au sein de la « *young generation* » et ne peuvent pas changer de génération des éléments. Ces phases ne requièrent pas d'arrêt complet des threads de la **JVM** et sont donc moins coûteuses que leurs équivalent majeurs.

Les phases majeures s'occupent de changer de génération les éléments ayant survécus assez longtemps aux différentes phases de *garbage collecto*r. Cette phase nécessite généralement pour les threads de la **JVM** d'avoir atteint un *safe point*, et les bloque le temps de l'opération. Les générations « *old* » et « permanent » ne sont collectées que lorsque la *heap* est pleine.

Cette dissociation des objets dans la *heap* suivant leur durée de vie permet de créer des zones d'accès rapides (de nombreuses classes sont créées pour être détruites aussi tôt, créant potentiellement du fractionnement dans la mémoire et de nombreuses opérations d'allocation/désallocation). Ainsi les éléments susceptibles de rester en vie durant toute l'application sont mis de côté afin de ne pas être bloqués ni modifiés.

NON-HEAP

Les éléments relatifs au fonctionnement de la **JVM** ne sont pas alloués dans la *heap*, mais dans une zone mémoire séparée appelée simplement la *non-heap*.

Celle-ci contient la zone de « permanent génération » qui contient les chaines de caractères internes et les informations relatives aux différentes méthodes. On trouve ainsi par exemple :

- Classloader
- Liste des champs d'une classe
    - Pour chaque champ, son nom, type modifieur, etc…
- Liste des méthodes
    - Nom
    - Type de retour
    - Type pris en entrée
- Instruction des méthodes
    - Bytecode et donnée relative à son fonctionnement

La *non-heap* contient également une zone réservée à la mise en cache de code, qui est compilé en code natif par des mécanismes de compilation **JIT**.

SOURCES

http://blog.jamesdbloom.com/JVMInternals.html#threads

http://javahonk.com/how-many-types-memory-areas-allocated-by-jvm/

http://www.maths.lse.ac.uk/Courses/MA407/gcsurvey.pdf

PYTHON ET CPYTHON

CPython est l'implémentation de référence de *Python*, et également la plus répandue. Il existe également des interpréteurs ou compilateurs se basant par exemple sur la **JVM** ou le moteur **JavaScript V8** qui relèvent donc de leurs fonctionnements respectifs.

CPython utilise tout comme Java du *bytecode*, qu'il interprète dans un environnement virtuel. Cependant la compilation n'est pas à la charge du développeur comme en **Java**, mais effectuée automatiquement au premier lancement et ensuite mise en cache.

MEMORY MANAGEMENT

Python propose comme tous les langages haut niveau une allocation automatique et transparente de la mémoire, l'utilisateur n'ayant plus à se soucier de ces contraintes.

Contrairement au *garbage collector* de **Java**, celui de **CPython** se déclenche uniquement suivant un certain seuil d'allocation et de désallocation, et non pas quand la place vient à manquer.

CPython utilise un système de compteur de référence analogue à celui de **Java** pour garder la trace de ses objets alloués et savoir lesquels ne sont plus utilisés.

SOURCES

http://www.digi.com/wiki/developer/index.php/Python_Garbage_Collection

http://arctrix.com/nas/python/gc/

JAVASCRIPT ET V8

Le **V8** se compose de 2 compilateurs, l'un **full-codegen** et l'autre **Crankshaft**.

FULL CODEGEN

Full-codegen est le premier compilateur à entrer en jeu dans l'exécution de code JavaScript. Il ne produit aucune représentation intermédiaire, contrairement à nombre de ses homologues, mais directement du code machine sous forme d'une machine à état, utilisant une syntaxe d'arbre abstrait (ou **abstract syntax tree** (**AST**) en anglais) représentant les différentes optimisations lues. Il se contente de parcourir chacun des nœuds de l'arbre syntaxique, et de générer les opérations correspondantes.

Ce premier compilateur procède de manière fainéante, ne compilant que ce qui est nécessaire pour l'utilisation du programme. Il n'effectue également pas d'optimisation, et produit donc du code « lent ».

Full-codegen est également, de par son lazy loading et son absence d'optimisation est très rapide, permettant au code de commencer son exécution très rapidement.

CRANKSHAFT

Crankshaft prend la main après **full-codegen**. Celui-ci est plus lent, mais permet de produire du code optimisé de manière dynamique en détectant les fonctions « chaudes » dont l'optimisation devient nécessaire.

Ce compilateur se base sur des prévisions en fonction du code déjà exécuté pour prévoir au mieux les futures instructions exécutées, et peut revenir sur la version non-optimisée de **full-codegen** si les conditions de ces optimisations sont violées.

Crankshaft se décompose en plusieurs phases, les deux plus importantes étant **Hydrogen** et **Lithium**. Leurs représentations internes sont différentes et leurs types d'optimisation également différents.

Afin de permettre facilement de passer d'une version optimisée à une version non-optimisée en cas de violation de contraintes, **Crankshaft** doit en permanence garder une simulation de ce que serait la machine à stack à cet instant. En cas de sortie du mode optimisé, la machine à stack est reconstruite dans l'état où elle aurait été en cas d'exécution non optimisée, avant de reprendre le fil d'exécution.

PARSING

Cette phase analyse et transforme le code source en un arbre syntaxique abstrait. La représentation générée n'est jamais conservée, et doit être régénéré à chaque nouveau lancement de **Crankshaft**.

La rapidité de cette phase ne justifie pas de s'encombrer de mise en cache quelconque subsistant entre deux lancements du moteur **JavaScript V8**.

ANALYSE DE PORTÉE

Durant cette phase, le moteur **V8** analyse comment les variables sont utilisées dans les différentes portées présentes dans le programme. Les variables globales, capturées (dans les callbacks) ou globales sont traitées différemment.

Cette phase prépare principalement la seconde, puisqu'elle rend ensuite possible le passage de l'arbre syntaxique à une représentation **SSA**, ou **Single Static Assignment**.

HYDROGEN

Hydrogen utilise comme réprésentation interne **HIR**, ou **Hight/Hydrogen Intermediate Rrepresentation**. **HIR** décrit les « basic block » de code, et le flux de contrôle entre ceux-ci. Chacun de ces blocs est représenté par une instruction écrite en **SSA**, ou **Single Static Assignment**. Cette représentation permet de simplifier la tâche de l'optimisation, car chaque variable ne peut être affectée qu'une seule et unique fois, et doit l'être avant son utilisation.

GÉNÉRATION DU GRAPHE

Il s'agit de la phase où se déroule la construction du graph d'**Hydrogen**, utilisant l'arbre syntaxique abstrait, les informations de portée et les différents types de variable. L'inlining prend également place durant cette phase. La représentation en question sera en **SSA, simplifiant la tâche à l'étape suivant d'optimisation.**

OPTIMISATION

Hydrogen essaie ensuite d'optimiser son graphe par divers moyens :

- *Inlining*, il s'agit ici de l'optimisation la plus importante, et la plus souvent mise en œuvre
- *Unboxing*, **JavaScript** étant typé dynamiquement, les types, même naturels (*integer*, *float*), doivent être contenu dans des structures contenant des métadonnées sur la nature de ces valeurs. L'*unboxing* permet de réduire cette charge en prévoyant les vrais types de certaines valeurs et en enlevant la couche de métadonnée devenue dispensable,
- Optimisation de boucles, comme dans le mémoire de nombreuses optimisations peuvent prendre place dans les boucles afin de les rendre plus efficientes (déplacement des invariants, inversion du sens, etc…)

Ces optimisations seraient possibles sans le passage par le **SSA**, mais sont largement simplifiés par celui-ci, permettant un gain de performance non négligeable. Cette phase peut être parallélisée avec l'exécution même du programme, l'un ne dépendant pas de l'autre.

LITHIUM

Lithium représente la phase d'optimisation prenant place après **Hydrogen**, mais en utilisant comme représentation interne **LIR**, pour **Low/Lithium Intermediate Representation**.

Au lieu de directement transformer les instructions optimisées par **Hydrogen** en instructions natives, elles sont transformées par **Lithium** en **LIR**, qui après leur avoir de nouveau appliqué une phase d'optimisation, génèrera enfin du code natif.

Au lieu d'utiliser une représentation **SSA** comme **Hydrogen**, la représentation de Lithium est plus proche du **three-address code** (ou **TAC**). Le **TAC** est une représentation intermédiaire utilisée par de nombreux compilateurs, servant souvent durant les phases d'optimisation. Ces instructions, généralement une opération par ligne (comportant également des sauts conditionnels), se traduisent relativement facilement par la suite en assembleur.

GÉNÉRATION DU GRAPHE

À partir du graphe optimisé de **Hydrogen**, un nouveau graphe (cette fois en **TAC**-*like* et non plus en **SSA**) est généré pour la phase **Lithium**. C'est également durant cette génération que les allocations de registre sont effectuées.

OPTIMISATIONS

L'utilisation de **Lithium** se justifie par sa capacité à être indépendant à la plateforme réelle d'exécution, permettant par exemple des optimisations de registres, l'allocateur de registre étant lui-même indépendant de la plateforme.

D'autres optimisations entrent également en jeu :

- Déduction de type dynamique : utilisant le profileur de code, il est possible de définir des versions optimisées pour des types spécifiques. Il conviendra également de vérifier le type en question avant le traitement, et de rebasculer sur une version non-optimisée en cas de changement.
- *Inlining* : Si une fonction appelée est suffisamment courte et peut être mise en ligne, alors elle le sera. Cette optimisation prend place directement durant la construction du graphe.
- Inférence de représentation : les types naturels n'ont pas besoin d'informations autre que leur valeur et peuvent donc être passés directement
- Etc…

On retrouve également d'autres optimisations ayant déjà pris place aux étapes précédentes, mais réappliquées une nouvelle fois.

GÉNÉRATION DE CODE MACHINE

Lithium écris ensuite les instructions machines dans un buffer, utilisant un assembleur écris en **C++** (qui est quant à lui dépendant de la plateforme). Le passage par un buffer permet quant à lui par exemple l'utilisation de macro et une plus grande flexibilité. Ces instructions machines sont ensuite exécutées.

Les informations concernant la dés-optimisation est aussi générée à ce moment, permettant de revenir à une version non optimisée si les contraintes d'exécution sont violées.

SOURCES

https://wingolog.org/archives/2011/09/05/from-ssa-to-native-code-v8s-lithium-language

https://docs.google.com/document/u/1/d/1hOaE7vbwdLLXWj3C8hTnnkpE0qSa2P--dtDvwXXEeD0/pub

http://wingolog.org/archives/2011/08/02/a-closer-look-at-crankshaft-v8s-optimizing-compiler

http://jayconrod.com/posts/54/a-tour-of-v8-crankshaft-the-optimizing-compiler