Tutoriel VMKit

Harris Bakiras, Gaël Thomas, équipe REGAL, LIP6 INRIA

Objectif

Implémenter une machine virtuelle minimale, que l'on appellera **toyVM**, basée sur VMKit. Compte tenu de la durée du tutoriel, il ne sera pas possible d'implémenter une machine virtuelle langage. La toyVM exécutera donc uniquement le calcul de la fractale de Mandelbrot écrit en dur dans les sources. L'intérêt du tutoriel étant de découvrir VMKit, tout le code lié aux calculs sera donné.

L'implémentation de la toyVM se décomposera en 4 étapes :

- Création du cœur de la toyVM
- Création du thread principale
- Création des objets collectables
  (à ce stade, le calcul de la fractale est possible, les objets alloués sont collectés via le GC)
- Création du compilateur Just In Time *(facultatif)*

Introduction

Avant de commencer à coder, veillez à faire compiler le canvas de base. Pour cela, un fichier *README* est disponible dans l'archive 'toyVM-base.tar.gz' fournie avec le présent sujet. L'exécutable généré ne fait rien pour l'instant, à vous de le compléter en suivant les questions suivantes.

Note : Pour lancer la compilation de la ToyVM, toujours appeler make depuis le répertoire racine toyVM

Tout au long du tutoriel, des noms de fichiers seront mentionnés, un schéma est fourni en annexe afin de présenter une vue d'ensemble de ces fichiers.

1 Cœur de la toyVM

La classe centrale de la machine virtuelle réside dans la classe *ToyVM* présente dans le fichier *ToyVM.h*. C'est cette classe qui contiendra des références vers le thread principale et le compilateur. *ToyVM* possède le point d'entrée de l'application.

1.1 Création de la classe ToyVM

Pour commencer le tutoriel, ouvrez le fichier *ToyVM.h* et à partir de ce qui a été dit lors de la présentation, complétez la définition (héritage) de la classe *ToyVM*. Rappel : si A hérite de B, la syntaxe C++ sera la suivante *class A : public B*. Certaines méthodes héritées de la classe parente sont virtuelles pures, il faut les déclarer puis les implémenter (même si la méthode est vide!) afin de pouvoir instancier ToyVM. Certaines méthode vous sont données dans le fichier *ToyVM.cc*, il suffit de les dé-commenter.

1.2 Instanciation de la ToyVM

Une fois que la classe *ToyVM* est implémentée, instanciez-la dans la méthode main du fichier *ToyVM_DIR/tools/toyVM/main.cc* à l'aide de l'opérateur *new* hérité de la classe *PermanentObject (Allocator.h)* et du constructeur de ToyVM donné dans le canvas de base (*ToyVM.cc*).

Note : Pour appeler un opérateur new et un constructeur utiliser la syntaxe suivante

new(...) Constructor (…) ;

Complétez le code de la fonction main en appelant les méthodes *runApplication* et *waitForExit* de l'objet ToyVM ainsi créé.

Lancez à présent l'exécutable toyVM situé dans *ToyVM_DIR/Release/bin/toyVM*. Que se passe-t-il ?

2 Thread Principale

Si vous n'avez pas encore lancé la machine virtuelle, faites-le.

L'application reste en attente. Pour comprendre pourquoi, il suffit d'aller regarde dans le fichier *main.cc* du répertoire *ToyVM_DIR/tools/toyVM*. La toyVM doit en fait lancer son thread principale dans la méthode *runApplication (ToyVM.cc)*. Nous allons donc remédier à ce problème en implémentant le ToyThread.

2.1 Création de la classe ToyThread

A l'aide des informations de la présentation et du canvas de base fourni, complétez la classe ToyThread de la même façon qu'en 1.1 pour la ToyVM (héritage, méthode virtuelles héritées, méthodes commentées).

2.2 Lancement du ToyThread, fermeture de la ToyVM

Maintenant que le ToyThread est implémenté, revenons dans la classe ToyVM. Comme expliqué précédemment, la machine virtuelle lance son thread principale dans la méthode *runApplication (ToyVM.cc)*. Cette opération se fera en trois étapes :

- Allocation du thread à l'aide du constructeur défini dans la classe *ToyThread* (*ToyThread.h*)
- Affectation de la variable mainThread (héritée de la classe *VirtualMachine*)
- Lancement du thread via la méthode *start* héritée de la classe *Thread* (Thread.h)

Note : une méthode est nécessaire au démarrage du thread. Cette méthode **statique** sera *mainStart* donnée dans le fichier *ToyVM.cc*.

Complétez ensuite la méthode *mainStart* pour appeler la méthode *execute* du thread et fermer la machine virtuelle (la méthode est héritée de la classe Thread). A présent, la ToyVM se lance et se ferme correctement.

2.3 Test du GC

Le thread principale étant en place, vous pouvez dors et déjà lancer un collection en la forçant. Cette dernière n'aura aucun effet puisqu'il n'y a pas d'objet collectable, néanmoins il est possible d'en suivre le déroulement en implémentant des traces de log.

Implémentez les méthodes nécessaires dans la classe toyVM pour afficher les messages « collection start » et « collection end » respectivement en début et fin de collection (voir les méthodes héritées

de la classe *VirtualMachine* liées au garbage collector).

Une collection peut être forcée en appelant la méthode du collecteur *vmkit::Collector::collect()*.

Note : Vous pourrez également implémenter des logs dans les méthodes tracer (expliquée en 3.2) de *ToyThread* et *ToyVM* dans le fichier *Tracer.cc*.

3 Objets Collectables

La machine virtuelle lance un thread mais n'exécute rien pour l'instant. Dans cette partie nous allons lancer le calcul de la fractale de Mandelbrot en utilisant des objets collectables.

Dans le cadre de la toyVM vous est fournie une classe ToyRoot dans le fichier *ToyRoot.h* effectuant la liaison avec le GC de VMKit via la classe gc définie dans le fichier *VmkitGC.h*. Tout objet collectable créé dans la ToyVM devra donc hériter de ToyRoot.

Pour rappel, tout au long du tutoriel, il ne faut **JAMAIS OUBLIER** de taguer les variables locales ou les paramètres qui sont des objets collectables à l'aides des macros TOY_ROOT et TOY_PARAM définies dans le fichier *util.h*.

Attention, dans la définition d'une méthode d'un objet collectable, l'utilisation de *this* est interdite car la variable *this* ne peut être taguée. Une macro *asSelf* est définie dans *util.h* afin de pallier à cette restriction, définissant une variable locale *self* à laquelle est affectée *this*. Une conséquence de cette restriction est qu'on ne peut appeler les méthodes ou champs privés de *this* à travers *self*...

3.1 Création de l'arborescence des objets

Créer ou complétez dans les fichiers *Pixel.h/.cc* (héritage, méthode virtuelles héritées, méthodes commentées) les classes d'objets collectables suivantes :

- Pixel
- MandelPix qui hérite de Pixel (représente un pixel de l'ensemble à calculer)
- Picture (représente l'image calculée)

L'appel à *new* étant interdit, il faut créer une méthode statique que nous appellerons '*doNew*' afin d'allouer des objets collectables. Cette dernière fera explicitement appel à l'opérateur new surchargé défini dans le fichier *ToyRoot.h* et initialisera l'objet en question.

Note : La syntaxe à utiliser pour utiliser l'opérateur *new* présent dans ToyRoot.h est la suivante.

operator new<'nom_de_la_classe'>('taille_de_l_objet') ;

3.2 Les tracers et les print

Pour une meilleure clarté, les méthodes tracer et print seront implémentées respectivement dans les fichiers *Tracers.cc* et *Printers.cc*.

Concernant la méthode tracer : cette méthode est appelée lors d'une collection par le GC. Elle permet de tracer le graphe des objets vivants afin de les préserver. Prenons par exemple deux classes d'objets collectable A et B. Si A contient une référence vers B, alors dans le tracer de A, B doit être marqué. Ce marquage s'effectue a l'aide des méthodes *markAndTrace* ou *markAndTraceRoot* présentes dans le fichier *ToyRoot.h*. Vous pouvez ajouter des traces de log dans les méthodes *tracer* du fichier *Tracers.cc* afin d'observer le traçage des objets lors des collections.

Inutile de décrire la méthode print...

Complétez donc les méthodes (essentiellement tracer) des objets collectables puis créez des objets dans le thread principale en déclenchant une collection afin de vérifier le traçage.

3.3 Compute (@harris : à vérifier)

Les méthodes compute concernent le calcul de la fractale de Mandelbrot. Une large partie du code est donnée, il ne reste qu'à compléter les zones marquées.

Une fois les méthodes compute implémentées, un premier calcul de la fractale peu être lancé dans le thread principale en instanciant un objet Picture (*doNew*) et en appelant la méthode compute.

Une fois le calcul terminé, un appel à print écrira l'image dans un fichier au format rasterfile (.ras).

4 Compilateur JIT *(facultatif) (@harris : à vérifier)*

Nous voulons à présent ajouter un compilateur JIT à notre toyVM. Ce dernier sera tout aussi minimal que la toyVM et ne consistera qu'à faire un appel de fonction qui lancera le calcul de la fractale.

4.1 Création du Compilateur

En vous servant de la présentation effectuée, complétez le canvas de base du compilateur (classes *ToyCompiler* et *ToyJITCompiler*).

Sachant que ToyJITCompiler possède un héritage multiple de ToyCompiler mais également d'une autre classe vue dans la présentation, pensez aux méthodes virtuelles héritées...

4.2 Initialisation du compilateur LLVM

Chargement dynamique du code IR de toyVM

A l'aide de la variable *toyvm_module_path* représentant le chemin vers le fichier contenant l'IR du module LLVM toyVM, complétez le code de la méthode *loadSelfModule* qui charge l'IR de la toyVM en mémoire.

Génération de l'IR de la fonction de calcul

Maintenant que l'IR de la toyVM est chargé dans le module LLVM, générez une fonction qui fait un appel à *jitCompute*. Pour cela, complétez le code de la méthode *generatecode*.

Note : L'optimisation de l'IR se fera en 4.4

4.3 Initialisation du cœur du compilateur JIT

Le cœur du compilateur JIT n'est autre que l'*executionEngine*, le composant qui transforme l'IR en code natif. C'est un composant LLVM qui doit être intégré dans le *ToyJITCompiler*. Une partie du code d'initialisation est donnée dans le canvas de base. Il ne reste plus qu'a le compléter !

Une fois l'*executionEngine* initialisé, on peut l'appeler sur la fonction générée pour obtenir un pointeur vers du code natif. (compléter la méthode *jitCompile*)

Ajoutez le code nécessaire dans le thread principale *(fichier ToyThread.cc)* afin de générer en JIT le calcul de la fractale de Mandelbrot.

4.4 Optimisation du code JIT

Nous allons ajouter des optimisations au code généré en utilisant des passes d'optimisations LLVM à l'aide d'un *FunctionPassManager (FPM)*.

L'intégration du *FPM* à la toyVM se fait en suivant les étapes suivantes :

- ajout d'un champs *FPM* au compilateur
- instanciation et initialisation du FPM dans une méthode *void initPassManager()* qui sera ensuite appelée dans le constructeur du compilateur. Vous pourrez appliquer à l'initialisation les passes d'optimisation décrites dans la toyVM.
- Enfin, lors de la génération de l'IR LLVM, appeler le le FPM sur la fonction générée.

4.5 Transmission des stack maps (optionnel)

Si le code JIT généré contenait des variables locales collectables *(ToyRoot)* taguées objet GC, ces dernières ne seraient pas collectées car les informations GC ne sont pas communiquées à VMKit.

Lorsqu'une fonction JIT est compilée, un callback est effectué via un système de listener. Pour paramétrer ce listener, il faut configurer l'*executionEngine* à son initialisation en lui attribuant le ToyJitListener via *RegisterJITEventListener*.

Ce callback, *NotifyFunctionEmitted*, renvoie les informations GC nécessaires au compilateur qui à son tour devra les transmettre à VMKit pour tracer les variables collectables.
Pour implémenter la transmission des informations GC, il faut dans un premier temps, créer un champs LLVM *GCModuleInfo* dans le compilateur, puis compléter le code du ToyJitListener dans le fichier *ToyJITCompiler.cc*

Une fois que le compilateur possède les informations GC, après compilation de l'IR en code natif, c'est à la charge du compilateur de passer les informations GC à VMKit par le biais de la méthode *addToVM*.