VMKit Tutorial

Harris Bakiras, Gaël Thomas, Gilles Müller, REGAL team, LIP6 INRIA

Goal

The VMKit tutorial's goal is to implement a minimal virtual machine called « **ToyVM** »based on VMKit. As the tutorial lasts 3 hours (approximatively), it won't be possible to implement a langage virtual machine. ToyVM will only compute the Mandelbrot set. VMKit tutorial's aims to discover VMKit environment, so the code related to computing the fractale will be provided.

ToyVM's implementation takes place in 4 stages :

- Creating ToyVM's core
- Creating main Thread
- Creating GC collectible objects

  (at this stage, computing the fractale possible, allocated objects are collected by the GC)
- Creating the Just in Time compiler (facultative)

Introduction

Before starting coding, make sure that the base code compiles. To do so, a README file is in the archive 'toyVM-base.tar.gz' provided with this statement. The generated binary does nothing for the moment, it is up to you to complete the code by following the questions.

Note : To compile ToyVM, always call « make » from ToyVM's root folder.

During the tutorial, filenames are mentionned, a diagram is provided in the presentation slides (last slide) in order to describe a global view of key files.

1 ToyVM core

The virtual's machine main class is ToyVM class in *ToyVM.h* file. This class will contain two important fileds, one reference to the main thread and one referece to the compiler. ToyVM has a method wich contains the entry point of application.

1.1 Creating ToyVM class

To begin the tutorial, open *ToyVM.h* file and complete the class definition (inheritance) of ToyVM class. You can look into the presentation to find some usefull information.

Reminder : if class A inherits of class B, C++ syntax wille be the following class A : public B.

Some inherited methods are pure virtual ones, you have to declare and implement them (even if the body is empty!) in order to instaciate a ToyVM object. Some methods implementation are provided in ToyVM.cc file. You just have to uncomment them.

1.2 Instanciating the ToyVM

Once *ToyVM* class is implemented, instanciate a ToyVM in main method in file *ToyVM_DIR/tools/toyVM/main.cc* using the *new* operator inherited from PermanentObject class (file *Allocator.h*) and ToyVM's constructor given in base code (file *ToyVM.cc*).

Reminder : To call a given operator new and constructor, use the following syntax

new(...) Constructor (…) ;

Complete the main function's body by calling *runApplication* and *waitForExit* methods of the ToyVM object you just created.

Now you can launch the toyVM binary located in the directory *ToyVM_DIR/Release/bin/toyVM*. What happens ?

2 Main Thread

If you still haven't launch the virtual machine, do it now.

The application gets stuck. To understand why, you just have to look into the *main.cc* file in *ToyVM_DIR/tools/toyVM*. In question 1.2 you added two methods in main function's body. One of them was *waitForExit*. So the ToyVM stucks to this function until some thread calls exit method. ToyVM must launch its main thread in *runApplication* method *(ToyVM.cc)*. Now to solve this problem, you have to implement the ToyThread class which will be the main thread.

2.1 Creating ToyThread class

Use the presentation's slides and the base code to fill the ToyThread class (*ToyThread .h / .cc*) like you have done in question 1.1 for ToyVM (inheritance, inherited virtual methods, commented methods).

2.2 Starting ToyThread, closing ToyVM

Now that ToyThread is implemented, get back to ToyVM class (ToyVM.cc). As it is explained before, the virtual machine starts its main thread in *runApplication* method. This operation is divided in three stages :

- Allocating the main thread using the constructor defined in ToyThread class (*ToyThread.h*).
- Assigning mainThread variable (which is inherited from VirtualMachine class).
- Starting thread by calling *start* method inherited from Thread class (*Thread.h*).

Note : start method needs a static function as parameter, this static function will be *mainStart* function given in *ToyVM.cc* file.

Now fill *mainStart* function to call the thread's *execute* method. Afer that, fill execute method to close the virtual machine (the closing method is inherited from *Thread* class in *Thread.h* file). The ToyVM launches and closes itself correctly.

2.3 Testing GC connexion

You have correctly set the main thread, you can already launch a garbage collection by forcing it. This one will have no effect because there is no collectible object, however it is possible to monitor the progress by implementing logs.

Implement the necessary methods in ToyVM class to print « collection start » and « collection end »

respectively at the beginning and end of collection (see the inherited methods from *VirtualMachine* class related to garbage collector).

You can force a collection by calling the following method *vmkit::Collector::collect() (VmkitGC.h)*

Note : Additional logs can be implemented in tracer methods (see 3.2 for more details) of *ToyThread* and *ToyVM* class in *Tracer.cc* file.

3 Collectible objects

The virtual machine launches a thread but does not compute anything in the meantime. In this section, you will see how to compute Mandelbrot set using collectible objects.

As part of toyVM, a class named ToyRoot is provided in ToyRoot.h file making the connection to the GC via VMKit gc class defined in the file VmkitGC.h. Every collectible object created in the ToyVM has to inherit from ToyRoot.

For recall, during the tutorial, **NEVER FORGET** to tag local variables or function parameters which are collectible objects with the help of TOY_ROOT and TOY_PARAM macros defines in *utli.h* file.

Caution : When defining a method in a collectible object class, the use of *this* is prohibited because this can not be tagged. A macro names *asSelf* is defined in *util.h* file to overcome this restriction. *AsSelf* defines a local variable *self* which is a tagged copy of *this*. A harmful consequence of this restriction is that private members can not be called threw *self*...

3.1 Creating the objects tree

Create or fill the following collectible object classes in files *Pixel .h/.cc* (inheritence, inherited virtual methods, commented methods) :

- Pixel
- MandelPix, which inherits from Pixel (represents a pixel of the Mandelbrot set)
- Picture (represents the whole computed Mandelbrot set).

Since operator new is forbidden, you have to implement a static method which will be called 'doNew' in order to allocate collectible objects. 'doNew' will have to call explicitly the operator new overloaded in file *ToyRoot.h* and initialize the object.

Reminder : C++ syntax to use for calling operator new in ToyRoot.h is the following.

operator new<'object_class_name'>('object_size') ;

3.2 tracer and print methods

For a better readability, tracer and print methods are respectively implemented in Tracers.cc and Printers.cc files.

About tracer method : It is called during a collection by the collector. It allows to construct the graph of living obects to keep themself. For instance, if a collectible object A has a reference to another collectible object B, then in A's tracer B object must be marked. Marking objects is performed using *markAndTrace* or *markAndTraceRoot* methods implemented in *ToyRoot.h* file. You can add logs in tracer methods to observe object tracing during a collection.

Useless to describe print methods...

So now fill tracer methods of collectible objects and allocate some objects in main thread before triggering a collection to ensure that those ones are correctly traced.

3.3 Compute (@harris : à vérifier)

Compute methods are related to Mandelbrot set computing. A large part of code is provided, fill in the few missing elements.

Once compute methods are implemented, a first Mandelbrot set computing can be achieved in main thread by instanciating a *Picture* object using *doNew* and calling compute method on it.

After that, just call print method to write the resulting picture in a file (mandelbrot.ras).

4 JIT Compiler *(facultative) (@harris : à vérifier)*

The JIT compiler is still missing, so in this part you will see how to add it in the toyVM. The compiler will be as minimal as the toyVM and will only compile one function.

4.1 Creating the compiler

Using the presentation's slides, complete of fill the base code of the compiler in ToyCompiler and ToyJITCompiler classes (file *ToyCompiler .h / .cc*). ToyJITCompiler has mutiple inheritance from ToyCompiler and also from another class seen in the presentation, don't forget inherited virtual methods...

4.2 LLVM compiler initialization

Loading dynamically toyVM's IR code.

With the help of *toyvm_module_path* variable which represents the path to the file containing toyVM's IR, fill *loadSelfModule* code located in file *ToyCompiler.cc*. This function loads in memory toyVM's IR.

Generating computing function's IR

Now that toyVM's IR is loaded in an LLVM module (see presentation slides), you can generate a function which makes a call to *jitCompute (Pixel.cc)*. To achieve this, fill the code of the *generatecode* method in *ToyCompiler.cc*.

Note : IR's optimization will be done in 4.4

4.3 JIT compiler core initialization

JIT compiler's core is the *executionEngine (ToyJITCompiler.h)*, this component translates IR into native code. ExecutionEngine is an LLVM component which has to be integrated into the *ToyJITCompiler* class. A part of initialization code is given in base code. Fill the *ToyJITCompiler*'s constructor in file *ToyJITCompiler.cc*.

Once *executionEngine* is initialized, you can translate the function generated in 4.2 into native code. To perform the translation, fill the *jitCompile* method in file *ToyJITCompiler.cc*.

Add the necessary code into main thread (*ToyThread.cc*) in order to JIT compile Mandelbrot set computing.

4.4 JIT code optimisation

Let's add some optimizations into the generated code (IR) using LLVM optimization passes with the help of a *FunctionPassManager* (FPM, file *PassManager.h*) which is also an LLVM component.

FPM's integration into toyVM can be performed by following those stages :

- Add a field FPM to the compiler (ToyCompiler.h)
- Instanciate and initialize the FPM in a method *initPassManager() (ToyJITCompiler)* which will be called in the JIT compiler's constructor.
- Finally, call the FPM on the generated function in *generatecode* (*ToyCompiler.cc*)

4.5 Stack maps transmission (optionnel)

If generated JIT code contained tagged collectible local variables (ToyRoot reference), those ones would not be collected because GC information (stack maps) is not transmitted to VMKit.

When a JIT function is compiled, a callback is done threw a listener system. To configure the listener, you have to configure the *executionEngine* at its initialization (*ToyJITCompiler.cc*). To do so, just call the *RegisterJITEventListener* method on the *executionEngine* with the *ToyJitListener* as parameter.

The callback made when a function is compiled calls the *NotifyFunctionEmitted (ToyJITCompiler.cc)* method giving as parameter the necessary GC information for tracing collectible variables to the compiler which will transmit it to VMKit.

To implement the GC informaiton transmition, first add a field LLVM GCModuleInfo in the compiler (ToyJITCompiler.h) and then fill the *NotifyFunctionEmitted* function in file (*ToyJITCompiler.cc*) to assign the filed you just created.

Once the compiler contains the GC information after translating the IR into native code, the compiler has to give this GC information to VMKit. This stage is performed by calling the method *addtoVM* which you can find in VMKit file *JIT.h*.