CFR-1

The Coral VM

Version 0.1-alpha1

Kateřina Markéta Lisová

September 30, 2014

Contents

Preface

The *Coral Virtual Machine* (the CVM) is a specification for a set of programs and program components. Those programs should enable the Coral programming language (and possibly many more) to execute on virtually any *host platform* that the CVM is implemented on.

Chapter 1

Abstraction vs. the Platform

Like JVM, a CVM should shield users of its programming languages from specifics of each platform as much as possible. E.g. numbers – each platform has specific representations of numbers, some of which may fit into the languages, some may need to be represented in a more virtual form, like the `Decimal` in Coral, which represents virtually any non-complex number digit-by-digit.

However, users may need to access the platform natively, in some limited program scope. For that purpose, a CVM should provide a native interface, which would inherently be specific to the platform and to the technology that a CVM is built with, but most likely it could be just a set of bindings written in the C language.

A CVM has to also provide ways for inter-program communication. A special channel may be available for CVM-to-CVM communications, and platform's standard channels should also be available, e.g. sockets, shared memory, or messaging services. The range of available non-CVM-to-CVM channels is inherently limited by the host platform and implementation status of a CVM.

1.1 Bytecode Portability

A single bytecode format is required for any CVM. A file containing such a bytecode must be executable by any CVM, regardless of the host platform. This single bytecode format is derived from the needs of the Coral programming language, and is tagged with a *dialect* – basically a short identifier of the language that is supposed to serve as its backend. Therefore, the single bytecode format is portable to any host platform that has the backing language installed.

It is however possible for the language to define a CVM extension, that would be able to transform any bytecode format it needs into a CVM-compatible one. Such extension should be portable between each host platform.

If a program that targets a CVM is to contain native code alongside with CVM bytecode, then such program is limited to the host platforms that the native code is compatible with. This native code could be distributed in these major ways:

- A dynamic library to be loaded at runtime as an extension for a CVM. A CVM then provides a runtime environment for the library.
- A separate executable to be executed as a separate process from a CVM, possibly from within a shell. Such an executable is executed in the context of the host platform, not a CVM.

1.2 Host Platforms

The initially supported host platforms should be these:

- Mac OS X (initially 10.9+)
- Windows (initially 8.1+)
- GNU/Linux and BSD (initially only recent versions)

More platforms are indeed welcome to become supported, but these are the main ones.

Chapter 2

The Abstraction

With abstraction, we here have the concepts of human-to-computer and language-to-computer communication in mind. Users will use a programming language of choice to communicate with a host platform, in a way that is friendly to them and independent of the host platform, except where needed.

2.1 The Languages

Initially, the Coral programming language is the primary supported language and may also serve as a intermediary between any further supported languages, using its own data types and instruction sequences.

A language definition for a CVM is basically a set of callbacks to be invoked for various reasons, e.g. method lookup, PSI or AST manipulations and so on. A language should also contain a compiler, which will provide a way to transform a source file set written using it into a form that is understandable by a CVM.

A CVM will provide primarily constructs to support the Coral programming language, but it may also be extended in the future to add more constructs for other languages, if ever needed.

A CVM should support programs written in several languages by keeping track of origins of values – thus each language may use e.g. different object layouts, and yet still make its values available to objects coming from other languages via CVM's shared protocols. Such tracking should be implemented using a small tracking label, possibly a number mapped to loaded language definitions.

For the supported language, as Coral is a hybrid language (it could be seen as both an object-oriented and a non-purely functional), it would be nice if a ML-based or a Lisp-based functional language was supported as well. It might be even possible to create

a trans-compiler from other pre-existing languages into CVM bytecode, e.g. Java, C# or even C. However, such pre-existing languages would need their backing language definition for a CVM to use.

2.2 The Bytecodes

A CVM should natively support only a single bytecode format – the CVM bytecode. Other bytecode formats may be supported by means of CVM extensions, working as a compiler from the custom bytecode into a CVM bytecode.

The CVM bytecode format has to be complex and robust enough to accommodate most languages' needs.

2.2.1 The CVM Bytecode

The CVM bytecode is a binary file containing several parts:

- Metadata – the dialect (i.e. the backing language), bytecode version etc.
- Constants – e.g. number literals, strings, symbols, type paths.
- Instruction sequences – a list of instruction sequences, one per function. The original source file may be a single instruction sequence, as it is with Coral.

Chapter 3

The Components of a Virtual Machine

3.1 Language Interface

3.2 Command Line Interface

3.3 Threads

A CVM runtime is all around *threads*. A thread models a computation stream, therefore computations may be parallel. Moreover, each thread may compose of multiple *fibers*, which are modelled using separate call stacks, some of which do not need to be complete copies. This also enables continuations, both unlimited and delimited.

A CVM should prefer native implementation of threads wherever possible, providing an abstraction to the languages, which in turn may provide abstraction to its users. Fibers, on the other hand, are not really easy to implement using native resources.

Fibers may be passed between threads, but for this to actually work, synchronization has to be implemented to make the threads able to pick up the new fiber. This may happen on a language level as well as user level, or even both (as in Coral, where the language enables passing of fibers between threads, but users have to make up the logic in which the receiving thread will pick up the fiber object).

3.4 Stacks

A *stack* is the low-level structure of a CVM. It contains a stack of stackable frames, which are structures containing reference to an instruction sequence, maintaining an instruction pointer (program counter), containing references to argument values, the

result value and thrown values, and optionally some more metadata, e.g. line numbers and corresponding files.

A stack may or may not represent a whole fiber execution, but there is always at least one stack that does. The other, incomplete ones, are delimited and used with constructs such as other fibers and continuations.

A stackable frame internally increments reference count to the instruction sequence that it references, and decrements it again upon being popped off its stack.

A stackable frame may be a control frame, containing the aforementioned elements, or a native frame, filling in for a native method invocation via `FFI` (Foreign Function Invocation). A native frame does not need to contain any program counters due to the nature of its execution. The native function is given access to its stackable frame, so it can e.g. return or throw a value.

3.5 Address Spaces & Values

A CVM has to give its languages memory zones for allocating new objects. In the context of a CVM, an object and a value are referring to the same concept – it represents a value that has some address, properties and size.

Languages are given APIs to handle the memory zones that are managed by a CVM. A language may be happy to work with a single memory zone itself, without giving its users access to creation of new memory zones, but it has to be able to accept objects from multiple memory zones – this has to be handled transparently.

The concept of memory zones (each having its own address space) is to abstract a continuous memory block. Every program running on top of CVM is given one main memory zone with preset behavior. New memory zones may have different settings, e.g. one can create a memory zone designed specifically for lightweight objects that are to be released from memory quickly, or even all at once with the whole memory zone. Making changes to a memory zone is protected by a read/write lock, so that single memory operations are atomic. A set of multiple memory operations still has to be manually synchronized.

Every object has its address, locating it in one of memory zones. Therefore the address is two-part, first identifying its memory zone, and second identifying it within its memory zone. Each address is a view – an object does not know anything about its address.

For memory management, every object is tagged with a reference counter, and a list of references of variables for managing weak references, which are essential in reference counted environment.

Objects are referred to via variable values, which are generally placeholders for object addresses. Addresses may be cached per-thread, unless the variable is specified to be volatile.

An object can have any number of properties, and also a set of flags that are common throughout every language (e.g. the “frozen” state for immutable objects). A property can be another object’s address, or it can be a special value provided by a CVM. Some of these include: fiber references, memory zone identifiers, or most importantly, numbers. This abstraction goes deep enough to say that a number can replace an object’s address in both properties and variables. Every language is required to be able to handle this level of abstraction.

3.6 Program Structure Information

A program structure information (`PSI`) is a forest of tree structures that hold both compile-time and runtime information about types in a program. One of the most important data stored in it are method references.

Languages may opt in to implement class objects by having their objects point to various subtrees of a PSI.

3.7 Inter-VM communication

A CVM has to implement a mechanism for inter-VM communication between programs. Furthermore, such VMs do not have to run on the same computer. The toolchain provided for this communication does not need to be embedded in a CVM and can be shipped separately.

Also a separate CFR is to be made for this component of a CVM.

3.8 Native Components

A CVM has to also provide guided access to some components that are underneath mapped to native host platform components, e.g. everything input/output or thread synchronization primitives. This is so that a language can be useful (input/output) and also parallelized (synchronization).