CFR-1

The Coral VM

Version 0.1-alpha1

Kateřina Markéta Lisová

October 19, 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 a custom bytecode into a CVM bytecode, primarily to support sort of a transpilation, e.g. from Java's `.class` format directly into the CVM bytecode format, if such extension is ever built.

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

Chapter 3

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 sequence graph – graph structures of instruction sequences, one per function. Such a graph is represented by a serialized sequence of instruction nodes along with links to other nodes as specified by each node. The original source file may be a single instruction sequence graph, as it is with Coral.

This chapter describes the CVM Bytecode file format, `.coralb`. Each `.coralb` file contains definition of a main instruction sequence graph along with additional graphs, which is unlike Java's **`.class`** file format, which only contains definition of a single class.

A `.coralb` file consists of a stream of 8-bit bytes. All 16-bit, 32-bit, 64-bit and 128-bit quantities are constructed by reading in two, four, eight, and sixteen consecutive 8-bit bytes, respectively. Multibyte data items are always stored in big-endian order, where the high bytes come first.

The types `u1`, `u2`, `u4`, `u8` and `u16` represent an unsigned one-, two-, four-, eight-, or sixteen-byte quantity, respectively.

In order to save disk space, numbers can be also expressed with multi-byte constructs. The types `m`n, where $1 \leq n \leq 20$, then represent 8-bit byte sequences in big-endian order, where only the last 7 bits are used to represent the corresponding number and the very first bit indicates whether the byte is the last byte in a sequence (the bit is `0`), or if it does not end a sequence (the bit is `1`). The types `m1`, `m2`, `m3` etc. then respectively represent unsigned numbers of 7-bit, 14-bit, 21-bit. The pseudo-type `mx` then represents any of these types.

The `.coralb` file format is presented here using pseudo-structures, written in a C-like notation. The contents of these structures are referred to as *items*. An item may also be a *table*, which consists of zero or more variable-sized items. Due to that fact, a table index may not be used to directly reference a byte offset in the file.

3.1 The `File` Structure

A `.coralb` file consists of a single `File` structure:

```
File {
  u4 magic;
  u4 major_version;
  u4 minor_version;
  dialect_info dialect;
  mx constant_pool_count;
  cp_info constant_pool[constant_pool_count];
  mx graphs_count;
  graph graphs[graphs_count];
}
```

The items in the `File` structure are as follows:

`magic`
: The `magic` item supplies the magic number identifying the `.coralb` file format; for the CVM bytecode, it has the value of `0xC0124189`.

`major_version, minor_version`
: The `major_version` along with the following `minor_version` items determine the version of the bytecode file format. A particular CVM may only support file formats in a certain range.

`dialect`
: The `dialect` item specifies the identifier and version of the backing language of the `File` structure.

`constant_pool_count`
: The value of the `constant_pool_count` item is equal to the number of entries in the `constant_pool` table.

`constant_pool`
: The `constant_pool` item is a table of structures representing various string constants, symbols, numbers and other constants that are referred to within the `File` structure and its substructures. Each entry in it is tagged with its format. The `constant_pool` is indexed from `1` to `constant_pool_count`.

`graphs_count`
: The value of the `graphs_count` item is equal to the number of entries in the `graphs` table.

`graphs`
: The `graphs` item is a table of structures representing instruction sequence graphs. The `graphs` table is indexed from `1` to `graphs_count`.

3.2 The `constant_pool` Table

Instructions refer to items in the `constant_pool` table. All entries have the following general format:

```
cp_item {
  u1 tag;
  u1 data[];
}
```

Each item in the `constant_pool` table must begin with a 1-byte tag indicating the kind of `cp_item` entry. The contents of the `data` array vary with the value of `tag`. The valid tags and their values are listed in Table 3.1. The format of the additional information in `data` varies with the `tag` value.

Table 3.1: Constant pool item tags

Constant tag	Value
`Const_UTF8`	1
`Const_Type_Path`	2
`Const_Int16`	3
`Const_Int32`	4
`Const_Int64`	5
`Const_Int128`	6
`Const_UInt16`	7
`Const_UInt32`	8
`Const_UInt64`	9
`Const_UInt128`	10
`Const_Float32`	11
`Const_Float64`	12
`Const_Float128`	13
`Const_Decimal`	14
`Const_Symbol`	15
`Const_String`	16

3.2.1 The `Const_UTF8_Item` Structure

A CVM must use the unmodified UTF-8 format. It is an universal item for text representation, used in both strings and symbols.

The `Const_UTF8_Item` structure is:

```
Const_UTF8_Item {
  u1 tag;
  mx length;
  u1 bytes[length];
}
```

The items of the `Const_UTF8_Item` are the following:

`tag` The `tag` item of the `Const_UTF8_Item` structure has the value `Const_UTF8` (1).

`length`
: The value of the `length` item gives the number of bytes in the `bytes` array (not the length of the resulting string or symbol, due to the nature of multi-byte UTF-8). It is not null-terminated, as that would be redundant. If the array ends with a null byte, it is a real part of the data.

`bytes`
: The `bytes` array contains the bytes of the string. The bytes may have any value.

3.3 Type Paths & Scopes

A type path is a simple list structure that consists of:

- A variable symbol.
- A selection symbol.
- An application symbol sublist.
- A type application symbol sublist.
- An expected type path.

A type path—relatively to its used position and scope—determines a type, either by pointing it to a node of PSI (§4.6), or, with a variable symbol, to an actual type of a value in a variable. The list of possibilities of type paths may get extended with support for more backing languages.

A type path is built as a sequence of references to symbols in the symbols list and a sequence of designators for each of its node (determining whether a node is a variable symbol, a selection symbol etc.).

A type path can also refer to a type that is made available with a type scope, which includes mechanism that can mark a scope being a parent scope of another scope (this is important for reuse of type scopes – and type scopes can possibly be reused across different bytecode files as well). When talking Coral, this can be done by using **`use`** clauses to import names into a particular scope – such types are then available with type paths that start with a selection symbol.

A type scope can also have a wildcard import, by using an extension of a backing language. When talking Coral, a scope can be tagged with a list of types from which it is built up, along with their versions at the time that import was performed. Thus, Coral is able to detect a dynamic change in a scope and next time that scope is accessed, the language is given an opportunity to refresh the scope and check for duplicates (which would be an error).

A language has to define the order of search for a type, since a type scope is not the only place where it may look for. The way it defines that is irrelevant to CVM, and it does so by implementing the language bindings (§4.1).

- The type of the function itself, if the language supports higher-order function types and their extension.
- The type of receiver, if any.
- Super-type(s) of receiver, if any.
- The given type scope assembled while evaluating an instruction sequence graph.
- Another related types to the type of receiver (if any) or the type of the function, such as a namespace or a module.
- Use a dynamic callback.

To resolve a type that is present in the PSI (note that PSI contains both named and anonymous types), a language binding has to be invoked with the assembled type scope and the corresponding type path with variable types resolved automatically by a CVM.

The backing language then with its implementation defines the order of search (which is arbitrary to each backing language) and either returns a type from PSI, or signals that it could not find a type, which is an error state that is further resolved. Such resolution may be implemented directly in the instruction that failed as a fallback instruction, or if that fails too or is not set up, another language binding (e.g. in Ruby, that could lead to an invocation of `const_missing`).

A very interesting case is when a backing language implements implicit conversions. E.g., say that a type path contains a function application and for some (or maybe even all) of the argument types, the corresponding member is not applicable to the given argument types. The language can make use of an implicit scope – e.g. search the type scope for a type that can convert a particular argument type to a type that the corresponding member is applicable to, or search some another type for such a conversion.

To allow the aforementioned behavior, all types referenced from a type scope are tagged with an "implicit flag", which (when set) makes the type eligible for implicit search & querying.

A CVM helps the backing language by providing an API to look for an implicit conversion on demand (therefore, a particular backing language does not need to make sense of the implicit flags at all). A CVM also tells the language where does that conversion come from, so that the language may choose whether to use it or not. A CVM can also be instructed to only look into the type scope, or some arbitrary other type, to collect all implicit values filtered by the backing language's query.

With an implicit value for conversion found, the backing language may instruct a CVM to apply the conversions before applying the function that requires the conversions.

An implicit conversion based on an implicit value is not the only option of converting an argument type though – a backing language has also the following options of converting an argument type (or even the selection itself):

- An implicit value conversion on argument.
- An implicit value conversion on selection. [1]
- A conversion specified by the language itself (e.g. numeric widening).
- Application conversion. [2]
- Eta-expansion. [3]
- Type application. [4]

A language may indeed choose multiple such conversions (based on its rules) for each component of a type path, and even multiple conversions for one component (e.g. an

[1] Used in Coral for view application.

[2] Used in Coral for method conversion – evaluation and empty application.

[3] Basically a conversion from a complex type path into a simple path that ends with an application symbol, created by evaluating the preceding nodes up to the last function reference.

[4] Used in Coral for value conversion – type instantiation, and also type inference that does not need to be deferred on selections, as the whole selection is made available with the type path, and Coral chooses to treat empty applications as selections.

implicit value conversion followed by an application conversion). It is therefore possible to even replace the whole type path with conversions and cache the resulting conversion (the key in a cache is the type path in the given used position).

When talking Coral, the language may also implement implicit parameters, along with implicit conversions. Such a feature may be rendered by using an application conversion on a modified type path, adding a flag that such an extra application is implicit (a CVM has to know of such possibility), and then what CVM does is: if the original function application is directly followed by another application (without selecting another member on the previous result), it ignores that application conversion added to the type path, and if not, then another function application happens, based on the newly added "virtual" application conversion (and yet again, the language is tasked with resolving each argument type, which is now tagged as implicit, so that it knows that it should[5] only look for implicit values).

[5]Really just "should", a CVM is not supposed to check whether it did or did not.

Chapter 4

The Components of a Virtual Machine

4.1 Language Interface & Bindings

4.1.1 Compiler Interface

4.1.2 Runtime Interface

4.2 Command Line Interface

A command line interface to a CVM should enable two main important functions: to start a CVM with a specified program, and to cover compilation of source of programs and libraries.

4.3 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).

4.4 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 graph 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.

4.5 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.

4.6 Program Structure Information

A program structure information (`PSI`) is a forest of graph 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 subgraphs of a PSI.

4.7 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.

4.8 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).