A Compact and Extensible Portable Scheme VM

Léonard Oest O'Leary
Université de Montréal
Canada
leonard.oest.oleary@umontreal.ca

Marc Feeley
Université de Montréal
Canada
feeley@iro.umontreal.ca

ABSTRACT

Virtual Machines (VM) tend to evolve over their life cycle with features being added regularly and a growing footprint. In a VM designed for resource constrained environments this trend deteriorates the VM's primary quality. We present how extensibility is implemented in the Ribbit Scheme VM that is both compact and portable to multiple languages. Our approach adds annotations to the VM's source code allowing the compiler to generate the source code of a specialized VM extended with user-defined primitives and with needless ones removed. This gives the best of both worlds: an extensible VM packed with all and only the features needed by the source code, while maintaining a small code footprint.

CCS CONCEPTS

• **Computer systems organization → Embedded software**.

KEYWORDS

Virtual Machines, Compiler, Dynamic Languages, Scheme, Compactness

ACM Reference Format:
Léonard Oest O'Leary and Marc Feeley. 2023. A Compact and Extensible Portable Scheme VM. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming (‹Programming› '23), March 13–17, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3594671.3594672

1 INTRODUCTION

There are many aspects that must be considered when designing and implementing a Virtual Machine (VM). Some of the most important are the portability of the VM implementation, the memory footprint for code and data, the code execution speed, and feature fullness. Our work targets resource constrained environments where the code size must be minimized, and also VM embedding in other software. Situations where this is relevant are:

- extending existing software with scripting support
- microcontrollers with small code memory [1]
- space-limited OS boot sectors and firmware
- mobile code where the code's transmission time must be minimized, such as web apps for mobile phones on slow networks, firmware updates on IoT devices [7] or a Mars rover needing code updates from Earth

‹Programming› '23, March 13–17, 2023, Tokyo, Japan

ACM ISBN 979-8-4007-0755-1/23/03.
https://doi.org/10.1145/3594671.3594672

In previous work we have designed and implemented the Ribbit VM (RVM) , a portable and lightweight implementation of a subset of the Scheme programming language [8]. The system supports features such as tail calls, closures, continuations and incremental compilation. In other VM work, portability is defined as the ability to take the VM implementation written in some host language (typically a system language like C/C++) and to compile it on multiple platforms (machines and operating systems). The RVM's portability is at a higher level; it can be ported easily to other host languages and we have done so for Assembly (x86), C, Clojure, Common Lisp, Haskell, Idris, JavaScript, Julia, Lua, ML, Prolog, Python, Scala, Scheme, Zig and even POSIX shell. This is possible because of the RVM's small size: typically only 200-400 lines of code, depending on the language, and some additional lines of code for a garbage collector when the host language does not manage memory automatically, such as C and POSIX shell. It is typically a few days of work to port the RVM to a new language by translating an existing implementation by hand. Thanks to the small size and multiple existing implementations, the barrier to entry is very low for end-programmers, making the RVM attractive to add scripting support to any software, regardless of the language it is written in. This allows for easy integration of the VM, as it utilizes a single memory management system and represents its objects using the same language as the embedding software.

As with any VM development, there has been a desire to extend Ribbit and the RVM with new features: more complete support of the Scheme language (rest parameters, file I/O, floating point, bignums, ...), addition of new primitive procedures to improve execution speed, addition of a Foreign Function Interface (FFI), better debugging support, etc. Unfortunately, any extension increases the footprint of the RVM and this slowly deteriorates the main quality of the RVM, making it less portable and attractive.

Ribbit aims to be a lightweight implementation of Scheme that requires little effort by an end-programmer to adapt to specific tasks. The system can be extended at different levels (the VM itself, the compiler, the runtime library, and the source program) and we want this to be feasible for an end-programmer without having to understand all the inner workings of the system as well as programmers earlier in the supply chain such as library and tooling creators.

In this paper we explain how Ribbit and the RVM have been made extensible without jeopardizing its small footprint. The basic idea is to specialize the RVM to the source program that is being compiled. In other words, parts of the VM are removed if they are not required by the program and new parts are added to the VM for extensions specific to the source program. Not only does this offer the best of both worlds (small size and feature fullness), in some situations the footprint is smaller than the original RVM implementation.

We start with an overview of Ribbit's original architecture and then explain how it was modified for extensibility.

2 RIBBIT

Ribbit's central component is the Ribbit Scheme compiler (`rsc`), an Ahead Of Time (AOT) moderately optimizing compiler that generates code executed by the targeted RVM implementation (in the host language selected at compile-time). The source program and runtime library are combined and compiled as one unit. This allows the compiler to analyze the whole code and determine the procedure definitions that are dead and that can be eliminated from the generated RVM code. The RVM code is then converted to a compact string representation. The compaction algorithm takes into account the static frequency of the operations in the program to encode frequent operations using fewer characters, typically one character per RVM operation. This string is then embedded into the target RVM implementation as a string literal. When the RVM starts executing, it will use this string to build in the main heap a symbol table (see below) and a more convenient representation of the code as a linked graph of instructions.

Although Ribbit implements a subset of the Scheme language, it is still quite powerful. Indeed the `rsc` AOT compiler is itself written in this subset of Scheme so it can bootstrap itself. Consequently both `rsc` and programs compiled by `rsc` are portable in the sense of our high-level portability.

The runtime library notably includes the procedures `read`, `write`, and `eval` which are the basis for implementing a Read-Eval-Print-Loop (REPL) at the console. This means that a compact Scheme interpreter can be created by compiling a simple Scheme program with `rsc`. For example, the interpreter's total footprint is less than 4K bytes with the JavaScript RVM. Moreover a Scheme program compiled with `rsc` can easily include a REPL functionality for run time debugging and experimentation. The `eval` procedure is based on the `compile` procedure that implements a non-optimizing incremental compiler that converts a Scheme expression to its RVM code wrapped in a parameterless procedure. Calling this procedure has the effect of evaluating the expression. The generated RVM code uses the same linked graph representation as AOT compiled code.

For implementation simplicity Scheme's symbol type is represented at run time as a heap allocated structure with a field containing the symbol's name (a Scheme string) and a field containing the value of the global variable with that name (for example the `+` symbol has the Scheme addition procedure in its global variable field). A symbol table must be maintained at run time to implement the `string->symbol` procedure in order to create only one symbol and global variable with a given name. In many programs, space can be saved by not maintaining a symbol table (which is simply a list of symbols) and the names of the symbols. Not only does this save heap space at run time but it reduces the size of the generated RVM code's compacted string representation which contains the names of the source program's symbols, a saving of 25%-30% for typical programs. This is determined during the dead code analysis. If `string->symbol` is dead there is no need for a symbol table. If in addition `symbol->string` is dead then there is no need to store the symbol's name. Note that `read` depends on `string->symbol` and `write` depends on `symbol->string` and the REPL depends on both. This means, for example, that a program using `write` but not `read` will avoid the symbol table but still store the names in symbols. To improve the space saving when both `string->symbol` and `symbol->string` are live, such as when a REPL is used, the programmer can add a `(export sym1...)` declaration to the program. The compiler will initialize the run time symbol table with the symbols in the list and will assume those global variables are live. The list usually contains the global variable names that might be referred to from the REPL, for example all the Scheme predefined procedure names in the case of the interpreter.

3 RVM

Ribbit's VM is designed to be compact. It is a stack machine with a classic code interpretation loop that dispatches on the next instruction to execute. An unusual aspect of the RVM is the simplicity of memory management: all data structures are built solely out of fixed size cells with 3 fields, called *ribs*. The run time stack, RVM code and Scheme objects are all represented using linked ribs. A Scheme object is either an integer or a rib whose third field is an integer indicating the type of object (0=pair, 1=procedure, 2=symbol, 3=string, etc).

The RVM instructions closely match Scheme's basic constructs: `jump`, `call`, `get`, `set`, `const`, and `if`. Procedure calls are performed with `jump` (tail call) and `call` (non-tail call). They can call one of 20 primitive procedures that are built into the RVM, or closures created using a `lambda`-expression. The `get` and `set` instructions are for reading and writing local and global variables. These 4 instructions contain a parameter that is either an integer indicating the location of a local variable on the stack, or a symbol indicating a global variable. Any other values required by the instruction are passed on the stack. The `const` instruction pushes to the stack the data that is in the instruction. The `if` instruction pops a value from the stack and follows one of two paths depending on whether the value is false or not.

Ribbit's functionality is mainly defined by the set of primitive procedures. The 20 primitives support basic operations: `rib`, `rib?`, `field[0/1/2]`, `field[0/1/2]-set!`, `eqv?`, `<`, `+`, `-`, `*`, `quotient`, `getchar`, `putchar`, `id`, `arg1`, `arg2`, and `close`. These primitives were carefully chosen to allow the definition of more complex Scheme procedures in the runtime library. For example, the primitive procedure `rib` constructs a rib from the value of its 3 fields, and the primitive procedures `field0`, `field1`, and `field2` extract each of those fields from a rib. A Scheme pair is represented with a rib whose first two fields contain the first (`car`) and second (`cdr`) field of the pair. Consequently the Scheme procedures `cons`, `car`, `cdr`, and `pair?` are defined in the runtime library as:

```
(define (cons car cdr) (rib car cdr 0)) ;; 0 = pair type
(define (car pair) (field0 pair))
(define (cdr pair) (field1 pair))
(define (pair? x) (and (rib? x) (eqv? 0 (field2 x))))
```

The primitive `id` (identity) eliminates the need for a *return* instruction since a tail call to `id` has the same effect. The primitive `arg1` has the same effect as a *pop* instruction to discard unneeded results (for Scheme's `begin` sequential execution construct). The primitive `close`, which captures the stack to create a closure, is also used for implementing the `call/cc` procedure which also needs to capture the stack.

4 EXTENSIBILITY

To make Ribbit extensible we exploit the fact that the compiler takes a source program and generates the source code of a RVM that embeds the RVM code generated for the program. It is natural to extend this model to make deeper changes to the source code of the RVM when new functionality is needed by the source program. Providing low-level interfaces is critical to an extensible VM, as it widens the range of uses and allows application-specific optimizations [6].

Adding primitives to the RVM is a natural place to start. These primitives could be added directly to the RVM's source code, like would be required in typical VMs, but we also want to allow end-programmers less knowledgeable in the system's inner workings to do this. We added support to `rsc` for a `define-primitive` construct that can appear in the source program or the runtime library. It defines a new RVM primitive procedure with a name and a string containing the implementation in the RVM's source code. For example, a `square` primitive could be added to the Python RVM with:

```
(define-primitive (square x) ;; only name matters
  "lambda: push(pop()**2)")  ;; Python code added to RVM
```

If `rsc` determines that `square` is live, an index will be assigned to that primitive, say 20, and the `define-primitive` will be replaced with the creation of a primitive procedure:

```
(define square (rib 20 0 1)) ;; 1 = procedure type
```

Which assigns to `square` a procedure represented as a rib. The compiler will also modify the primitive procedure dispatch logic of the RVM to handle index 20 by executing the code specified in the `define-primitive` construct. This serves as a simple FFI/intrinsic mechanism, with the advantage that there is very little overhead for calling extensions.

Generating a specialized VM also allows removing from the RVM source code all primitives that are determined dead by `rsc`. For example, the `getchar` primitive could be removed when the program does not read input, the `close` primitive could be removed when no closures are created, etc.

The host code in `define-primitive` constructs is usually specific to one host language. We added to the `cond-expand` conditional expansion construct a test so that the host can be taken into account at compile-time. A primitive can have one implementation for each relevant host language like this:

```
(cond-expand ((host py) ;; Python host
              (define-primitive (square x)
                "lambda: push(pop()**2)"))
             ((host c)  ;; C host
              (define-primitive (square x)
                "{ int x = pop(); push(x*x); }")))
```

This is convenient to modularly extend domain-specific runtime libraries with interfaces to services that exist in multiple host languages (access to filesystems and databases, fetching web documents, cryptographic hashing, etc).

Since the beginning of the project, the RVM's source code for each host has been a self-standing file, including a predefined *hello world* RVM code string. This allows most of the development of a new RVM to be done with the usual developer tools and code editors without requiring the execution of `rsc`. This is a valuable quality that we want to preserve. In addition the RVM's source code must be modifiable by `rsc` without having to embed knowledge of the host language in `rsc` which would make it harder to port to new host languages. For that reason our approach is based on annotating the RVM source code. These annotations are parsed by `rsc` to determine how the RVM's source code needs to be modified. To avoid including a parser for every host language in `rsc`, these annotations are placed in host language comments and have the easy-to-find marker `@@(...)@@` and a Lispy syntax.

The `@@(` token indicates the start of an annotation, followed by a name and optional parameters on the same line. If the annotation ends on the same line with a `)@@` token, then this annotation refers to code on this line (see the example below). If it ends later on, the annotation refers to everything after the starting line, until and including the line containing the matching closing bracket `)@@`. Note that annotations can embed other annotations, for example, the `primitives` (plural) annotation indicates the section of code containing *all* the primitives. Inside it, `primitive` (singular) annotations indicate the location of a specific primitive. For languages like Pascal that require a comment-ending token, we can use `@@` to halt annotation parsing early.

For code generation the annotations inform `rsc` about the host syntax, in particular for RVM code string literals and the primitive dispatch logic code (`switch` statement, array of procedures, etc). For maximum flexibility annotations can embed Scheme code generating host code as a string.

Multiple sections of RVM source code may be needed to implement specific features. For example, to implement the `getchar` primitive the C RVM needs a `#include <stdio.h>` at the top, an auxiliary C function that calls the C `getchar`, and code in the primitive dispatch logic. Moreover the `putchar` primitive also depends on `#include <stdio.h>`. For this reason features are named and annotations can express dependencies between the sections of RVM source code. Here is the C RVM's implementation of the `putchar` primitive:

```
#include <stdio.h> // @@(feature stdio)@@
...
switch (prim_index) {
 // @@(primitives (gen "case " index ":" body)
 ...
 case 19: // @@(primitive (putchar c) (use stdio)
   putchar(tos()); break; // print top of stack
   // )@@
 ...
 // )@@
}
```

The `gen` argument of the `primitives` form instructs `rsc` on how to generate a single primitive. It includes a template employing special *variables* like `index` and `body`. The `index` represents the dispatch number of the primitive, while `body` is the code enclosed by the `primitive` annotation. In this context, we notify `rsc` that for the `C` host, primitives need to be wrapped in a `case` statement. The `use` form declares dependencies between the current primitive and features, in this case, the `stdio` feature.

Like the `define-primitive` form, the `define-feature` form allows programs to define new features. For instance, consider a primitive that's designed to write a string to the console. It would need a conversion function that translates RVM strings into host

language strings, which can also be beneficial for other primitives. Below is an example of how this could be implemented in the JavaScript RVM:

```
(define-feature rvm-to-host-string
  (decl "function rvm2host_string(s) { ... }"))

(define-primitive (write-string s)
  (uses rvm-to-host-string)
  "console.log(rvm2host_string(pop())); push(FALSE);")
```

In the `define-feature` construct, the `decl` symbol preceding the JavaScript function definition is used to denote the *location* in the RVM source code where the feature should be inserted. The need for a location marker arises because definitions can be location-specific within host languages. Named locations are indicated in the source code with : `@@(location` *<location-name>*`)@@`.

A `@@(replace` *<string>* *<replacement>*`)@@` annotation is also available to replace a string in the RVM source code with another string. This is mainly used to replace the default *hello world* RVM code string with the RVM encoded program. This is how the annotation is used inside the Python host:

```
# @@(replace ");'u?>vD?>v ... lkv6y" (encode 92)
input=");'u?>vD?>v ... lkv6y"
# )@@
```

Our goal is to encode the program in the fewest bytes possible, and doing so is dependent on the host language. Some languages (like x86) allow strings of arbitrary bytes. Conversely, in languages like JavaScript, encoding bytes within a vector would not be the most efficient approach, as each byte would be encoded with up to four characters. The `(encode 92)` argument instructs the compiler to encode the string using a safe set of 92 ASCII characters, but the RVM implementation can ultimately decide which is best. Future extensions could offer a variety of options for finer control of the encoding.

Lastly, the option to enable or disable features during the compilation process is provided through the command line. This empowers the programmer to exercise control over the elements included in the host RVM, and ultimately, in the final program. For instance, the arity-check feature directs the compiler to validate the number of arguments prior to each call. This requires `rsc` to push the number of arguments before each call, which might result in a slightly less efficient code with a larger footprint. Despite this, it is advantageous for debugging purposes. Further, feature options might encompass adding or removing the garbage collector, or making host-specific modifications such as distinguishing between a Node.js and a web target on the JavaScript host. Our feature system possesses the adaptability to accommodate all these scenarios.

5 RELATED WORK

Jupiter is a Java Virtual Machine (JVM) that aims flexibility through software design choices [4]. New extensions can be easily written by extending existing classes and interfaces. However, the VM needs to be modified to do so. It doesn't offer dynamic primitive creation, and it is unclear if this is even possible given the JVM context.

Benzo is a framework for low-level programming [2] at a high-level of abstraction. It allows dynamic modifications of components, like our annotation system. It does this through a FFI similar to `define-primitive`. However, Benzo is not a VM, it runs on a single host language, and it doesn't have a liveness analysis or target resource constrained systems.

Maté is a VM with a similar code size as Ribbit [7]. It targets sensor networks and offers on-the-go network capabilities with abstractions for simplifying the writing of asynchronous applications. In terms of extensibility, eight instructions have been reserved for users to define. Ribbit's annotation system has no such limitation and through primitives it could define powerful network abstractions as well.

The VM generation tools vmgen[5] and Tiger[3] can generate and specialize a C implementation of a VM with the main goal to improve the execution speed. They analyze and profile the VM code to extract superinstructions, specialized instructions, and speed-related optimizations. The speed improvement of the VM code interpreter comes at the cost of a larger VM. In Ribbit, we are concerned with the VM's size and portability across host languages. Our approach modifies VM implementations without any knowledge of the host language syntax through the use of annotations in comments, making it portable and modular.

6 CONCLUSION

In this paper, we showcased an annotation language to allow specializing a portable VM to the needs of the program. Extensions are expressed in the source program and the code of standard features is labelled in the VM's source code. This allows the compiler's dead code analysis to be applied to the whole program including runtime library and VM's source code. The system's extensibility does not compromise the size of the final executable because it contains only the needed parts. In future work we think it will be interesting to explore how to allow more global properties of the VM to be deactivated/activated, such as support for tail calls, first class continuations, threads, etc. It will also be interesting to see if this allows a complete implementation of RnRS Scheme to fit in a small code memory for typical programs.

REFERENCES

[1] Sean Bartell. 2021. *Optimizing whole programs for code size.* Thesis. University of Illinois Urban. https://hdl.handle.net/2142/113862

[2] Camillo Bruni, Stéphane Ducasse, Igor Stasenko, and Guido Chari. 2014. Benzo: Reflective Glue for Low-level Programming. In *Proceedings of the International Workshop on Smalltalk Technologies.* Association for Computing Machinery, New York, NY, USA. https://hal.inria.fr/hal-01060551

[3] Kevin Casey, David Gregg, and M. Ertl. 2005. Tiger - An Interpreter Generation Tool., Vol. 3443. 246–249. https://doi.org/10.1007/978-3-540-31985-6_18

[4] Patrick Doyle, Carlos Cavanna, and Tarek S. Abdelrahman. 2004. The design and implementation of a modular and extensible Java Virtual Machine. *Software: Practice and Experience* 34, 3 (2004), 287–313. https://doi.org/10.1002/spe.565

[5] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. 2002. Vmgen—a generator of efficient virtual machine interpreters. *Software: Practice and Experience* 32, 3 (2002), 265–294. https://doi.org/10.1002/spe.434

[6] Tim Harris. 1999. An Extensible Virtual Machine Architecture. In *Proceedings of the OOPSLA'99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design.* Association for Computing Machinery. https://www.microsoft.com/en-us/research/publication/an-extensible-virtual-machine-architecture/

[7] Philip Levis and David Culler. 2002. Maté: a tiny virtual machine for sensor networks. *ACMSIGPLAN Notices* 37, 10 (Oct 2002), 85–95. https://doi.org/10.1145/605432.605407

[8] Samuel Yvon and Marc Feeley. 2021. A small scheme VM, compiler, and REPL in 4k. In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL 2021).* Association for Computing Machinery, New York, NY, USA, 14–24. https://doi.org/10.1145/3486606.3486783

Received 2023-01-22; accepted 2023-02-06