

Université de Montréal

**Designing Compilers for Portability with Virtual
Machine Tailoring**

par

Léonard Oest O’Leary

Département d’informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l’obtention du grade de
Maître ès sciences (M.Sc.)
en Compilation

30 avril 2026

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

**Designing Compilers for Portability
with Virtual Machine Tailoring**

présenté par

Léonard Oest O’Leary

a été évalué par un jury composé des personnes suivantes :

Houari Sahraoui

(président-rapporteur)

Marc Feeley

(directeur de recherche)

Ian Arawjo

(membre du jury)

Résumé

Depuis l'avènement des premiers ordinateurs personnels, les machines virtuelles demeurent une technologie clé permettant la portabilité des programmes. Elles se basent sur la standardisation du code intermédiaire assurant une même exécution sur plusieurs machines virtuelles implémentées dans divers environnements. Toutefois, cette standardisation est aussi son talon d'Achille. Elle contraint les implémentations à respecter une spécification minimale, ce qui ne convient pas à tous les environnements.

Ce mémoire présente une nouvelle approche de compilation appelée la conception de machine virtuelle sur mesure (*Virtual Machine Tailoring*). Cette approche enlève la nécessité d'un code intermédiaire standard, ce qui permet la spécialisation de la machine virtuelle au programme et à l'environnement. Cette spécialisation donne un contrôle total au compilateur, permettant de rajouter, depuis le programme interprété, des fonctionnalités comme des primitives manquantes, de nouveaux types primaires et une interface de fonction étrangère.

Pour démontrer les capacités de la compilation par machine virtuelle sur mesure, nous avons développé son implémentation de référence : Ribbit. Ribbit est l'une des implémentations du langage Scheme les plus portables, supportant 25 langages cibles différents allant d'Assembly (x86) à POSIX Shell en passant par Go et JavaScript. De nouveaux langages cibles peuvent être ajoutés en seulement 2 semaines de travail par des non-spécialistes. De plus, Ribbit génère des exécutables qui peuvent facilement être inclus dans d'autres programmes, car ils sont compacts, tiennent en un seul fichier et peuvent facilement être adaptés au programme englobant. Finalement, Ribbit supporte une interface de fonction étrangère vers chaque langage cible sans passer par des protocoles complexes comme des interfaces binaires d'application (ABI).

En résumé, Ribbit démontre que la compilation par conception de machine virtuelle sur mesure est une technique prometteuse pour développer des compilateurs portables.

Abstract

Since the early development of personal computers, virtual machines have been a key technology enabling the portability of programs. They rely on a standardized bytecode, ensuring a consistent execution across diverse virtual machine implementations. However, this standardization is also its downside. It restricts virtual machines from implementing a minimal specification that cannot support all environments.

This thesis presents a new compilation technique called *Virtual Machine Tailoring*, which enables the modular generation of virtual machines. This technique removes the requirements of a standard bytecode, enabling the specialization of a virtual machine to a specific program and environment. This specialization grants full control over the generated virtual machine, allowing the source program to define new virtual machine features such as novel primitives, foreign function interfaces, or new first-class types.

To demonstrate the benefits of Virtual Machine Tailoring, we developed Ribbit as its reference implementation. Ribbit is a highly portable Scheme compiler supporting 25 targets including Assembly (x86), POSIX Shell, Go and JavaScript. The addition of new targets requires limited effort, typically taking two weeks even for non-experts. Furthermore, all programs produced by Ribbit are compact, self-contained, and can be embedded into other programs, making them easily adaptable to new environments. Finally, Ribbit provides a Foreign Function Interface (FFI) to each target without relying on a complex binary interface such as an Application Binary Interface (ABI).

In summary, Ribbit showcases that Virtual Machine Tailoring is a promising technique for the design of portable compilers.

Contents

Résumé	v
Abstract	vii
List of Tables	xv
List of Figures	xvii
List of Acronyms and Abbreviations	xxi
Remerciements	xxiii
Chapter 1. Introduction	1
1.1. Contributions	2
1.2. Chapter Separation	3
Chapter 2. Related Work	5
2.1. Portable Virtual Codes	5
2.2. Compact Lisp and Scheme Virtual Machines	6
2.3. Extensible Virtual Machines	7
Chapter 3. Compilers For Portable Code	11
3.1. Credit	12
3.2. The Benefits of Portability	12
3.2.1. Portability Reuses Development Effort	12
3.2.2. The Amortization Effect of Portability	12
3.3. Exploring Portability with an Example	13
3.4. Defining Portability	14
3.5. Portability as Interfaces and Translators	16

3.5.1.	Interfaces	16
3.5.2.	Environments	17
3.5.3.	Translators	17
3.5.4.	Enabling and Disabling Interfaces	18
3.6.	Compilers for Enabling Portability	19
3.6.1.	Adapting to a Variety of Operational Constraints	19
3.6.2.	Extending Reachable Environments with Virtual Machines	21
3.6.3.	Expressiveness and Portability	23
3.6.3.1.	The Portability Cost of <i>Language Features</i>	23
3.6.3.2.	Minimizing Primitives	24
3.6.3.3.	Foreign Function Interfaces	24
3.7.	Incremental Portability	25
3.8.	Final Thoughts on Portability	27
Chapter 4.	Ribbit	29
4.1.	Credit	30
4.2.	The Ribbit Virtual Machine	30
4.2.1.	Ribs and the Rib Graph	30
4.2.2.	Representation of Scheme Objects	31
4.2.2.1.	Pairs	31
4.2.2.2.	Special Values	31
4.2.2.3.	Vectors and Strings	32
4.2.2.4.	Symbols	32
4.2.3.	Instruction Ribs	33
4.2.3.1.	The Instruction's <i>Argument</i>	33
4.2.3.2.	The set and get Instructions	34
4.2.3.3.	The jump and call Instructions	34
4.2.3.4.	The const Instruction	34
4.2.3.5.	The if Instruction	34
4.2.3.6.	Simple Addition Example	34
4.2.4.	Procedures	35
4.2.4.1.	Primitive Procedures	35

4.2.4.2.	The Apply Primitive.....	35
4.2.4.3.	Controlling the Virtual Machine Control Flow with Primitives	36
4.2.4.4.	Closure Procedures	37
4.2.4.5.	Stack Frames and Free Variables.....	37
4.2.4.6.	The Call Protocol.....	38
4.2.4.7.	Variadic Procedures.....	38
4.2.5.	Memory Management.....	38
4.2.6.	Other Vector Representations.....	40
4.3.	Encoding of the Rib Graph	40
4.3.1.	The Symbol Table	41
4.3.2.	Decoding Instructions.....	42
4.3.2.1.	The <i>SHARE</i> Instruction	43
4.3.3.	Encoding of the RIBN	44
4.3.4.	Encoding of the Decoding Instructions in the RIBN.....	45
4.3.5.	Encoding Specialization.....	46
4.3.6.	Compressing the RIBN	48
4.3.6.1.	LZSS	48
4.4.	The Standard Library.....	49
4.4.1.	A Hierarchy of Library Implementations	49
4.4.2.	Implementation of Portable File I/O Primitives	50
4.4.3.	Writing the R4RS Library Compactly.....	51
4.5.	Final Thoughts on Ribbit.....	51
Chapter 5.	Virtual Machine Tailoring	53
5.1.	Credit	54
5.2.	An Overview of Virtual Machine Tailoring	54
5.2.1.	A Simple Example	54
5.2.2.	The Different Forms of Tailoring.....	55
5.2.3.	Motivation of Virtual Machine Tailoring	56
5.3.	Features and Annotations.....	57
5.4.	Tailoring Virtual Machines with Annotations.....	57
5.4.1.	Syntax and Scope of Annotations	58

5.4.2.	Annotations for Any Language.....	59
5.4.3.	Replace Annotation.....	59
5.4.4.	Primitive Annotations.....	60
5.4.4.1.	Template Generation.....	60
5.4.4.2.	Adding New Primitives from the Source Code.....	61
5.4.4.3.	Adapting a Virtual Machine to Primitive Annotations.....	62
5.4.5.	Feature and Location Annotations.....	63
5.4.5.1.	Feature Annotations.....	63
5.4.5.2.	Location Annotation and define-feature Macro.....	63
5.4.5.3.	The if-feature and use-feature Compiler Directives.....	64
5.5.	Compiling with a Virtual Machine Tailoring Compiler.....	65
5.5.1.	The Liveness Expansion Phase.....	66
5.6.	Case Studies of Virtual Machine Tailoring.....	66
5.6.1.	The Embedding of a Specialized Encoding.....	67
5.6.2.	Improving the Foreign Function Interface.....	68
5.6.2.1.	Manual Conversion of Values.....	69
5.6.2.2.	Defining New Primitives at Runtime.....	70
5.7.	Virtual Machine Tailoring for Portability.....	71
5.7.1.	Keeping a Core and Minimal Virtual Machine.....	72
5.7.2.	Facilitating the Addition of New Targets.....	72
5.7.2.1.	Requirements for New Targets.....	73
5.7.3.	The Implication of Losing Virtual Code Standardization.....	73
5.7.4.	Embeddability as Partial Solution to Inter-Process Communication.....	74
5.7.5.	Targets with Different Feature-Sets.....	75
5.7.6.	Virtual Machine Tailoring for Incremental Portability.....	76
5.7.6.1.	Bounding the Effort to Add New Targets.....	76
5.7.6.2.	Trading Space and Portability for Speed.....	76
5.7.6.3.	Controlling Language Features.....	77
Chapter 6.	Evaluation.....	79
6.1.	Adding New Targets.....	79
6.2.	Improving Targets.....	80

6.2.1. Bootstrapping the Compiler Using the <code>go</code> Target	80
6.2.1.1. Experimental Setup	80
6.2.1.2. Results	81
6.3. Operational Constraints	81
6.3.1. Footprint of the REPL	81
6.3.2. Footprint and Execution Speed	82
Chapter 7. Future Work	85
7.1. Defining Portability for Language Implementations	85
7.2. Generating New Ribbit Virtual Machines with LLMs	86
7.2.1. A Small Experiment at Generating Two RVMs Using Coding Agents	86
7.2.2. Results and Future Directions	86
Chapter 8. Conclusion	89
Appendix A. Survey of Comment Syntaxes in 25 Languages	93
Appendix B. Development log of the Go target extension	95
Appendix C. Experiment Details of Generating RVMs Using LLMs	99
C.1. Setup of the Experiment	99
C.2. Prompts	99
C.3. Evaluation of RVMs	100
References	107

List of Tables

3.1	List of definitions for portability from standards and researchers.....	15
4.1	The default primitive table inside of Ribbit.....	36
4.2	Example of the decoding instructions, their type of argument and their effect on the stack.....	43
4.3	Assignment of ranges for each decoding instruction for the original encoding used in Ribbit.....	47
5.1	Example of functions available inside the replace annotation in Ribbit.....	60
5.2	All variables that can be referred inside the <i>template</i> of a primitives annotation.	61
6.1	Table showcasing the assignment announcement and due date for each edition of the compiling class.....	79
6.2	Footprint of the REPL compiled to the x86 assembly host with different settings.	81
6.3	The first column shows the absolute execution time of each benchmark when using the Gambit Scheme interpreter. The other columns show the relative execution time and footprint of Ribbit's x86 assembly host compared to the first column...	83
6.4	The first column shows the absolute execution time of each benchmark when using the Gambit Scheme interpreter. The other columns show the relative execution time and footprint of Ribbit's NodeJS host compared to the first column.	83
A.1	Survey of syntaxes for comments in each of the languages supported by Ribbit...	93

List of Figures

3.1	Examples of an <code>open_in_browser</code> function for opening a website in a web browser by calling the shell and using the <code>xdg-open</code> utility.	14
3.2	The example from Section 3.3 annotated with all its interface dependencies. As the code is written in Python and uses the standard <code>os</code> package, it depends on the Python language and its standard library. The <code>os.system</code> function calls the operating system's shell to run the <code>xdg-open</code> binary from the xdg specification. The xdg specification from the freedesktop.org organization standardizes the opening of content inside default programs under Linux.	16
3.3	Example of the dependencies of the program in Figure 3.2. Nodes are standard interfaces and arrows represent implementations of these interfaces or <i>translators</i> . Gnome is a common desktop environment that can be seen as a translator between the xdg specification and the Wayland protocol. Weston is the reference implementation of the Wayland compositor.	18
3.4	Figure directly taken from Pittman's article on the tradeoffs between space and time[39] for generating code. Copying is by permission of the Association for Computing Machinery, according to the paper's license.	20
3.5	Example of the usage of a virtual machine to have portable programs.	21
3.6	Figure showcasing how foreign function interfaces (FFIs) bypass the bytecode by having a dependency directly to a new implementation.	25
3.7	Illustration showing the porting distance in the environment space.	26
4.1	Example of a rib containing the values <code>a</code> , <code>b</code> and <code>c</code>	30
4.2	The representation of data in Ribbit as <i>ribs</i>	31
4.3	Example of the representation of the <code>(1 2 . 3)</code> improper list at the top and the <code>'(1 2 3)</code> proper list at the bottom.	32
4.4	Example of the vector <code>#("AB" 42)</code> (top) and the string <code>"AB"</code> (bottom) represented as ribs. Note that the vector contains the string.	32

4.5	Example of the symbol 'a (leftmost rib) containing its string representation "a" inside the <i>name</i> field. In this example, the global variable a contains the number 42 as seen by the <i>value</i> field of the symbol 'a. This could be the result of a direct assignation such as (define a 42).....	33
4.6	All instruction ribs in Ribbit.	33
4.7	Example of the Scheme code (+ 42 43) represented as ribs. The fields of the rib representing the symbol '+ are not shown.	34
4.8	Example of the representation as ribs of procedure > defined as a call to the primitive <. Code for the procedure is displayed on the bottom-right.	35
4.9	Effect of each type of decoding instruction on the stack.	42
4.10	Example of sharing between the two branches of an if statement, using the absolute value function.	44
4.11	This example showcases the differences between embedding a program as a string and as a vector.	44
4.12	An example of the recursive factorial function.	46
5.1	Example of a virtual machine implementing the <i>AddSub</i> language in Python.	55
5.2	Example of an inline feature annotation.	57
5.3	Example of multi-line and nested primitives (plural) and primitive (singular) annotations.	58
5.4	Example of the use of the premature halting parsing token @@ for multi-line comments in OCaml.	59
5.5	Example of two replace annotations. The first one is inserting the program's encoding and the second compile-time information from features.	59
5.6	Example of the example virtual machine from Figure 5.1 annotated with primitive and primitives annotations.	62
5.7	Example of the use of a feature annotation referring to multiple lines and with a condition statement. The example is from the Assembly Ribbit Virtual Machine.	63
5.8	Example of location annotations inside the JavaScript virtual machine.	64
5.9	Example of define-feature and define-primitive compiler construct to define new features and primitives inside the JavaScript virtual machine shown in Figure 5.8.	64

5.10	Example of a compilation of <code>prog.arith</code> to <code>prog.py</code> using a Virtual Machine Tailoring (VMT) compiler. The VM used here is the one from Figure 5.6. As we can see inside <code>prog.py</code> , the new printing primitive is added from the host program and the feature <code>print-final-result</code> is disabled.	65
5.11	Example of the decoding of the optimal encoding.	67
5.12	Example of JavaScript code that converts a string represented in the Scheme to its representation in the host language.	68
5.13	Example of a foreign function interface to the <code>fetch</code> function in JavaScript.	70
5.14	Example of the usage of <code>host-eval</code> to implement a Foreign Function Interface to the <code>fetch</code> JavaScript function.	71

List of Acronyms and Abbreviations

ABI: application binary interface
AOT: ahead-of-time
API: application programming interface
AST: abstract syntax tree
CIL: common intermediate language
CPU: central processing unit
FFI: foreign function interface(s)
FPU: floating-point unit
GSI: Gambit Scheme interpreter
I/O: input/output
ISO: international organization for standardization
IEC: international electrotechnical commission
IEEE: institute of electrical and electronics engineers
JNI: Java native interface
JVM: Java virtual machine
LLM: large language model
POSIX: portable operating system interface
R4RS: revised⁴ report on the Scheme language
R5RS: revised⁵ report on the Scheme language
RAM: random-access memory
REPL: read–eval–print loop
RIBN: Ribbit intermediate byte notation (see Section 4.3)
RSC: Ribbit Scheme Compiler
RVM: Ribbit Virtual Machine (see Section 4.2)
VM: virtual machine(s)
VMT: Virtual Machine Tailoring (see Chapter 5)
Wasm: WebAssembly
WASI: WebAssembly system interface

Remerciements

La maîtrise se compose d'une série de moments, certains marqués par des découvertes réjouissantes et d'autres par des défis complexes. J'ai eu la chance de partager plusieurs de ces moments avec des personnes extraordinaires qui m'ont épaulé, éclairé, encouragé et permis de souffler. Merci du fond du cœur à toutes celles et à tous ceux qui ont fait partie, de près ou de loin, de cette magnifique aventure.

Je remercie tout particulièrement mon superviseur, Marc Feeley, qui a toujours su trouver le temps de répondre à mes nombreuses questions et qui m'a poussé à me dépasser chaque jour. C'est grâce à toi que j'ai découvert, lors de mon baccalauréat, le monde fascinant de la compilation et que j'ai décidé d'y consacrer une maîtrise.

Ma reconnaissance va également aux membres et amis du laboratoire qui ont créé une ambiance motivante tout au long de ma maîtrise et dont certaines discussions ont fait germer des idées qui se retrouvent dans ce mémoire: Olivier, Mathis, Frédéric, Laurent, Marc-André, Maël, Cassandre et Luchino.

Merci également aux professeurs Ian Arawjo et Stefan Monnier. Nos nombreuses discussions m'ont permis de développer un nouveau regard sur l'informatique, que je conserverai toute ma vie.

L'écriture de ce mémoire n'aurait pas été possible sans le soutien de ma famille et de mes amis. Vous m'avez offert un espace pour me reposer et m'avez surpris par vos encouragements constants.

Enfin, j'aimerais remercier chaleureusement ma belle Dominick. Merci d'avoir été mon étoile lors des nuits sombres et ma *cheerleader* lors de mes réussites.

Chapter 1

Introduction

Write once, run everywhere

Announcement of Java 1.0

Sun Microsystems, 1995

“Write once, run everywhere” captured with a simple and catchy sentence a programmer’s dream: never having to rewrite a program because of *where* it is executed. This dream alone justified decades of development by thousands of programmers in the hopes of reaching what is referred to as *portability*. And yet, most programmers tasked to perform a port realize that the portability dream can rapidly transform itself into a nightmare of implementation details and subtle incompatibilities.

In January 1995, Sun Microsystems introduced the Java Virtual Machine and popularized the idea of virtual machines as a definitive solution to *portability*. In theory, executing programs with a virtual machine allows its execution everywhere there is a virtual machine implementation. In practice, to accommodate the ever-growing and complex needs of programs, virtual machines have complexified themselves to keep pace with new features such as the management of memory, foreign function or network interfaces. This added complexity increases the mental load to understand the virtual machine, the performance and memory requirements of the implementation, and the time required to add new target environments. And when virtual machines become harder to understand and implement by programmers, *portability* greatly suffers. This highlights a tension between the portability of *programs* executed by a virtual machine and the portability of *their virtual machine implementations*.

This thesis presents a new compiler technique called *Virtual Machine Tailoring* allowing the *modular* generation of *virtual machines*. This modularity controls the complexity of virtual machines by removing a choke point: the requirement that *all* virtual machines have to implement the *same* bytecode. This permits the generation of featureless virtual machines with a simpler bytecode for more constrained environments while

allowing the generation of fully featured virtual machines when the source program needs it. Overall, this allows the generation of code to more target environments, as the virtual machine can better *adapt itself* without being constrained by the requirements of a common bytecode. This also keeps the core of the virtual machine extremely simple, thus portable, with the possibility to *modularly* add only the required features.

Removing the property that *all* virtual machines implement the *same* bytecode can be seen as losing the portability benefit of virtual machines in the first place. However, Virtual Machine Tailoring generates both the bytecode alongside the modular virtual machine, ensuring that the specialized bytecode can still be executed in the target environment. Furthermore, including a whole virtual machine inside an executable can seem like avoidable bloat. However, a clever design can generate exceptionally small code, as shown by its reference implementation.

To demonstrate the benefits of Virtual Machine Tailoring in practice, we created Ribbit as its reference implementation. Ribbit is a Scheme compiler to 25 different targets, ranging from low-level languages (C, x86 Assembly, Wasm) to high-level languages (JavaScript, Python, Ruby, Posix Shell), with the possibility to add new targets in **two weeks of work** by a third-year computer science bachelor student. This makes Ribbit the most portable Scheme compiler in terms of the number of targets, while also allowing the generation of embeddable code within other programs. Ribbit defines a foreign function interface (FFI) to every target without the usage of complex application bytecode interfaces (ABIs) normally needed to implement such features. Finally, Ribbit artifacts can easily be embedded into other applications as they are compact, self-contained and adaptable. For example, Ribbit generated the world’s smallest R4RS Scheme Interpreter in only 6.5 KB.

With the presentation of *Virtual Machine Tailoring* and its reference implementation *Ribbit*, we aim to empower programmers with effortless portability to environments that have similar constraints, interfaces, or operational characteristics, which we call *incremental portability*. We believe this is a step towards the realization of the portability dream as commercialized by Sun Microsystems in 1995.

1.1. Contributions

The contributions of this thesis can be summarized as follows:

- A discussion of portability, with a focus on the role of *language implementations*.
- The presentation of Ribbit, a compiler whose source code is available under a free and open-source license at <https://github.com/udem-dlteam/ribbit>.
- The explanation and evaluation of Virtual Machine Tailoring.

1.2. Chapter Separation

The rest of the thesis is separated as follows:

- Chapter 2, *Related Work*, situates Ribbit and Virtual Machine Tailoring in the greater body of literature.
- Chapter 3, *Compilers for Portable Code*, defines and presents the notion of portability and its implication for the design of language implementations.
- Chapter 4, *Ribbit*, explains the Ribbit Virtual Machine and its design decisions enabling portability.
- Chapter 5, *Virtual Machine Tailoring*, dives into the new compiling technique of the same name.
- Chapter 6, *Evaluation*, evaluates Ribbit and by proxy Virtual Machine Tailoring.
- Chapter 7, *Future Work*, discusses research directions for portability, Ribbit, and Virtual Machine Tailoring.
- Chapter 8, *Conclusion*, concludes the thesis.

Chapter 2

Related Work

The content presented in this thesis can relate to many different bodies of work and each one is presented in its own subsection.

2.1. Portable Virtual Codes

Although Sun Microsystems commercialized virtual machines (VMs) as a solution to portability in the mid 90s, virtual codes were already used for this purpose as early as the 70s.

In 1973, Niklaus Wirth released Pascal’s p-code[33], one of the first virtual codes designed for portability. At the time, the landscape of hardware architectures was very fragmented, making it difficult to share programs such as compilers. Consequently, the Pascal language kit, called the p-kit, shipped with a Pascal compiler written in p-code and a p-code interpreter written in Pascal. This allowed recipients to either implement a p-code interpreter or modify the compiler to execute Pascal programs[51]. In Wirth’s own words, Pascal’s p-code was designed to “restrict their labour to coding the [p-code] interpreter in their favourite assembler code.”[51]

In 1981, Smalltalk-80[16] was released alongside its VM. Initially, a select set of hardware providers, such as Apple, HP, and Xerox, received a pre-release version of Smalltalk to implement a VM for their hardware. This allowed images containing Smalltalk and its graphical environment to be executed on most hardware on release. Later, an open-source Smalltalk implementation called Squeak[20], reimplemented the Smalltalk interpreter in Smalltalk alongside a Smalltalk to C compiler (written in Smalltalk). This allowed the generation of a Smalltalk VM independently from other programming languages.

Originating from Smalltalk and Pascal, the Java programming language was released in 1995. Developed at Sun Microsystems, Java became the symbol for the “write once, run everywhere” dream and Java Virtual Machines (JVMs) were developed for a variety of

software and hardware. Since its release, all major operating systems and architectures had at least one JVM implementation if not many. JVMs had native implementations inside major browsers until 2016 when Applets got deprecated[47]. JVMs were even ported to hardware implementations with the Java Processor and the Java Card[7], with the latter still being widely used today. This popularity influenced other programming languages to target Java’s bytecode to benefit from its multiple implementations. These languages include Scala, Kotlin, Clojure, Smalltalk, JavaScript and Scheme.

Other notable VMs from this era include:

- Python’s bytecode designed for operating system independence. Although one could believe that Python’s bytecode can easily execute on microcontrollers with MicroPython, a new (and incompatible) bytecode had to be designed.
- Lua’s bytecode is designed for compactness and embeddability within other software. The reference interpreter is written only in standard ANSI C, making the VM implementation, and consequently Lua, very portable.
- .NET Common Intermediate Language (CIL) gives a uniform platform for Microsoft languages (C#, VB.NET, F#) and C/C++.

In 2017, WebAssembly[17] was introduced, citing portability as one of its main design goals, alongside safety, speed, and compactness. Contrary to its name, WebAssembly is not limited to the Web, having implementations for Node.js (V8), the JVM (GraalVM), Rust (WasmTime), and C (wasm2c). These multiple implementations motivated the idea of WebAssembly as a universal bytecode solving portability once and for all[14]. However, the core specification[41] requires over 150 instructions, floating-point numbers, and a validation pass at runtime, while not supporting first-class functions. These constraints make a minimal WebAssembly interpreter complex and therefore unsuitable for certain environments, such as microcontrollers without a floating-point unit (FPU), language runtimes with a native JIT compiler, or drivers that require direct access to machine memory. Nevertheless, WebAssembly has a promising future to support *many* target environments, especially the Web.

2.2. Compact Lisp and Scheme Virtual Machines

Ribbit, presented in Chapter 4, is a virtual machine (VM) for the Scheme programming language. The original objective of Ribbit is the generation of highly compact code while being modular and reasonably performant. This relates to portability as small code can more easily be embedded into other programs and run on size-constrained environments. In the following paragraph, small Lisp and Scheme VMs that influenced the early development of Ribbit are discussed.

Bit[10] and Picobit[45] are the direct predecessors of Ribbit, all designed by Marc Feeley’s research team at the Université de Montréal. They are all ahead-of-time (AOT) compilers supporting first-class functions, call/cc, and most constructs from the Revised⁴ Report on Scheme (R4RS). Compared to Bit and Picobit, Ribbit achieves the smallest generation of programs while providing the most features such as macros and full R4RS support.

TinyScheme[44] and Fentolisp[3] are Scheme and Lisp implementations meant for being embedded inside other programs similarly to Ribbit. TinyScheme implements a large subset of the R5RS specification while Fentolisp supports vectors, exceptions, first-class functions, hash tables, and declares itself highly compatible with Scheme. Both implementations are part of other noteworthy software. The TinyScheme interpreter is part of the GIMP suite and Fentolisp is used as the frontend for the Julia interpreter (accessible by typing `julia --lisp` on the command line).

uLisp[25] is a Lisp implementation designed to run on microcontrollers with limited RAM such as the Arduino Uno. To accommodate all environments, uLisp ships with 5 different versions, offering tradeoffs between language features, RAM utilization and performance. Ribbit has similar goals regarding on-disk memory footprint but is not directly optimized for RAM utilization. Nevertheless, the C Ribbit Virtual Machine (RVM) implementation has multiple parameters that can be tuned to run Ribbit on microcontrollers with limited RAM utilization.

SectorLISP[48] is a unique Lisp implementation that runs directly on bare metal and impressively fits within a 512-byte boot sector. SectorLISP does not include a built-in eval procedure, though users can manually enter one through the read-eval-print loop (REPL). Sector Lisp served as an inspiration for Ribbit, especially for the x86 RVM implementation.

2.3. Extensible Virtual Machines

Planning for *all* future environments where a virtual machine (VM) will need to execute is virtually impossible. For this reason, portable VMs should also provide extensibility to enable programmers with effortless adaptations to new environments. Virtual Machine Tailoring, discussed in Chapter 5, provides portability through extensions. Similar techniques are discussed in the remaining paragraphs.

Most established VMs provide ways to directly call foreign code, and this mechanism can be used to extend the capabilities of the VM. These include the JVM with the Java Native Interface[28], Lua with its C API[19], Python with ctypes[40], WebAssembly with its Component Model[50], and CIL with P/Invoke[30]. For example, JNI allows a C/C++ function to be linked to the HotSpot JVM to provide the battery level or send desktop notifications, as these operations are not natively supported by Java. However, these foreign

function interfaces (FFIs) are still limited, as they cannot change all properties of a VM. For example, these FFIs cannot implement a new bytecode instruction, change the calling convention used by functions, modify the behaviour of the garbage collector, or implement a FFI. Other extension mechanism that go beyond host-native code interoperability are discussed in the next paragraphs.

Jupiter[9] is an extensible VM born from the pursuit of porting the JVM to a highly parallel environment: a 128-processor cluster of PC workstations. It provides extensibility by implementing the VM as composable building blocks. This allows discrete modifications of the system to prototype and test new ideas. Although the extensibility system gives significant control over the implementation, the VM needs to be recompiled for each modification and users need to thoroughly understand the intricacies of the JVM to make meaningful modifications.

Benzo[5] is a framework for low-level programming at a high-level of abstraction. The system runs on top of the Pharo VM implementation for Smalltalk and permits the runtime modification of VM components using Smalltalk. Using Benzo’s reflective capabilities, the paper showcases the implementation of a Foreign Function Interface (FFI) and JIT Compiler. Interestingly enough, one of the case studies presented in Section 5.6.2 also showcases the implementation of an FFI only using the Virtual Machine Tailoring extension capabilities and the runtime modification of primitives within the VM.

GraalVM[53] is a polyglot platform already supporting JavaScript, WebAssembly, and Python on top of the JVM. GraalVM achieves interoperability between all supported languages via the Truffle framework, requiring an AST interpreter for each new language. While GraalVM aims to unify multiple source languages under one target (the JVM), Virtual Machine Tailoring pursues the inverse objective: supporting one language across multiple targets. Although interoperability is not an explicit goal, Virtual Machine Tailoring inherently supports interoperability between the source language and the target platform.

Vmgen[11], Tiger[6], and, more recently, Deegen[54] are meta-compilers generating efficient implementations of VMs given a description of the source language. This line of work showcases that a generated VM, while being easier to write, can rival the performance of hand-written implementations. However, these tools only generate optimized C/C++ code. It is still unclear if these techniques are generalizable to other programming languages. Virtual Machine Tailoring, presented in Chapter 5, can be viewed as a meta-compiler generating specialized VMs in a variety of programming languages, rather than in performant C/C++.

Interpreter specialization[46, 26] and more practically eJSTK[49] relate to Virtual Machine Tailoring, as they can all be seen specializing a VM to a specific program. eJSTK showcased this in practice by specializing data types of a given program for a JavaScript

interpreter. This includes the selection of a domain-specific layout for data types and the pruning of the VM to remove any routines dealing with unused data types. Virtual Machine Tailoring has more general pruning capabilities, allowing any unused code to be removed from a VM, provided that the programmer manually annotates it.

Chapter 3

Compilers For Portable Code

One of the key benefits of portability is the ability to execute programs in multiple environments. As rewriting programs for each environment is time-consuming, *automatic translators* targeting multiple environments are required for effortless portability. These translators can take multiple forms. For example, some translate operating system interfaces (Docker[29], Wine[1]), while others abstract specific library calls behind a common abstraction layer, such as mobile or machine learning frameworks. Arguably, *compilers* are one of the most powerful and widely used such translators, allowing high-level programming languages to be translated into different, more specific, environments.

Portability has often been studied from the point of view of *programs*. This thesis changes the focus to **the design of programming language implementations for enabling the portability of their programs**. In essence, language implementations are *tools* assisting the programmer while porting a program. These tools are helpful when they give *control* over the generated executable to achieve proper execution inside the new environment.

This chapter starts by motivating portability in Section 3.2 and presenting a short example in Section 3.3. Then, portability is defined in Section 3.4 and further developed by explaining the relation between *translators*, *interfaces*, and *portability* in Section 3.5.

The rest of the chapter discusses how compilers can give back *control* to the programmer, as introduced in Section 3.6. This challenge is further explained in the following sections: generating binaries for a wide range of environment constraints in Section 3.6.1, reducing the effort to add new target environments in Section 3.6.2, and trading off expressivity of the programming language for portability in Section 3.6.3. Finally, the chapter concludes with the notion of *incremental portability* in Section 3.7, offering intuition about the design of compilers for portability.

3.1. Credit

The ideas presented in this chapter have been developed by the author and did not appear in other publications.

3.2. The Benefits of Portability

Before discussing the definition of portability, it is worth answering a more fundamental question: *why do we want portable programs?* This section will focus on the benefits of portability for *individual* projects and then for *communities* as a whole.

3.2.1. Portability Reuses Development Effort

A first observation is that portability serves to *reuse* (and hopefully reduce) *the development effort* invested in a project. This is done by removing the friction induced by *moving* code between environments, avoiding the duplication of code for each environment and *reusing* this development effort. Depending on whether the target environments are known or unknown, portability gives different benefits.

For known targets, portability to different environments reduces the development and maintenance costs by avoiding the duplication of environment-agnostic code. For example, a mobile application required to run on both iOS and Android can reuse the business logic and user interface code as it is not operating system dependent.

For unknown and future targets, portability allows the *reuse* of the already spent development effort by transferring this effort to future environments. In practice, the transfer itself will require some amount of work *determined* by the portability of the software, and development effort is saved when this transfer work is less than rewriting the software[31]. In essence, portability acts as an *insurance* for the already spent development effort, ensuring that this effort is not wasted when the environment changes.

Furthermore, having a portable codebase to new environments can in itself be an advantage, for example, in a competitive race to support a newly released environment. Here, portability will make the difference between being the first or the last to support the new environment, resulting in the gain or loss of market share.

3.2.2. The Amortization Effect of Portability

To run on different environments, programs require abstractions that translate the environment-agnostic code into their environment-specific counterparts. In addition to providing portability, this split between environment-agnostic and specific code separates the work into two: the work to develop a *program* and the work to create the *translator*. At

the scale of a community or large company, the splitting *improves* programmer productivity and *amortizes* the development effort across the community. This subsection further argues those points.

Programming with environment-agnostic abstractions requires less effort as the programmer does not have to think about the specifics of the environment. This saves development effort for the community or company and improves the programmer’s productivity. For example, writing the same program in **Scheme** takes typically less effort than in **C**, which itself takes less effort than in **x86** assembly.

When a community or a company converges on the same few *translators*, the cost of maintaining and improving these few translators is amortized across all programs using them. This leaves a small number of shared translators with the potential to save development effort across the whole community or company. For example, each hour spent improving the portability of Python, Docker, or Java has the potential to save hundreds (if not thousands) of hours of development effort for all programs using them.

Large software corporations know this well and invest resources into the development of specialized portability tools to both save on development costs and amortize their portability effort across their company. As an example, with the announcement of their new M1 CPU chip in June 2020, Apple had to convert all their software from the x86 instruction set to M1. This requires enormous porting efforts, as it targeted most of Apple’s software: their OSes (macOS, iOS, iPadOS) and all the applications running on it. One year later, the first iMac with an M1 chip was released. They were able to perform the port in about 1-2 years because part of their software is written using portable programming languages and because they created a new efficient x86-to-ARM translator: *Rosetta*. This allowed a significant part of their software to be executed on the M1 chip with only the effort of developing or modifying their *translators*. Because of Apple’s tight control over the interfaces used by the company through the development of custom compilers and interpreters, they saved decades of effort compared to redeveloping the software by hand.

3.3. Exploring Portability with an Example

This section discusses the porting of a program to various environments to provide intuition about what portability is. While discussing its portability, readers are encouraged to ask themselves: *What are the key characteristics that make this program portable? Is it the language that it is written in, the structure of the program, or the environment to which we are porting?*

The Python program to port performs some operations assumed to be portable and calls the `open_in_browser` function to open a URL in a web browser as showcased in Figure 3.1.

```
import os

def open_in_browser(url):
    os.system("xdg-open " + url) # Calls the xdg-open utility using the shell
```

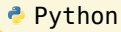


Fig. 3.1. Examples of an `open_in_browser` function for opening a website in a web browser by calling the shell and using the `xdg-open` utility.

What is the portability of this program to different operating systems? This program will work properly on computers running Linux with a graphical environment, but will fail on macOS and Windows due to the missing `xdg-open` utility on these platforms. However, changing this code to make it work on Windows or macOS can be done easily, requiring only a localized change to the `open_in_browser` function or a change to the Windows or macOS machine to add the `xdg-open` binary. One could argue that this program is still quite portable to Windows and macOS because the required changes are minimal.

What about porting this code to a web page? Surprisingly, the Pyodide[8] project allows the execution of Python code within a webpage, but doesn't support calls to `os.system`, needing more adjustments. One could argue that the portability to the web environment is *harder* than porting to Windows or macOS and that this portability is *enabled* by existing tools.

What about porting this code to a headless microcontroller? This would make little sense as headless microcontrollers run without a graphical output, raising the question: What would it even mean to open a URL in a browser in such a context?

From this simple example, we can derive a few statements about portability:

- Portability depends both on the *program* and the targeted *environment*.
- Portability depends on the availability of existing tools, such as compilers.
- Portability is not a binary attribute and some ports are *harder* to perform than others.

3.4. Defining Portability

Definitions of portability have been proposed with various levels of detail by Mooney [32], Brown [4] and several standards [24, 22, 23, 21]. All definitions found while researching portability are included in Table 3.1. Inspired by Mooney [32], the definition of portability used for this thesis is as follows:

The portability of a program to an *environment* is the *effort* to *translate*, *replace*, and/or *remove* its *interface dependencies* such that the program executes correctly inside the environment.

The words in italics are briefly explained here, and further developed in Section 3.5.

Effort refers to the amount of work necessary to perform the porting. In this thesis, we will quantify this work by amounts of time-persons, meaning the estimated time it takes for one programmer to perform the porting.

Interfaces are the contracts that define the communication between different software or hardware systems, for example, a programming language specification or a web server API. *Interface dependencies* are the *interfaces* that the program depends on.

Environment refers to a computer language implementation that is either in software or hardware (an Intel x86 chip, a Scheme interpreter, the CPython interpreter). It also includes its supported interfaces (operating system, file system, available hardware and remote systems) and operational limitations (memory, processing resources, I/O limitations).

Translations can be performed either manually by a programmer or with the use of *automatic translators*. *Automatic translators* are programs that allow the use of one interface as a substitute for other ones. For example, the `gcc` translator can substitute the `C` interface with `x86`, as programs written in `C` can be compiled to `x86` by `gcc`.

Table 3.1. List of definitions for portability from standards and researchers.

Source	Definition of portability
James D. Mooney[32]	A software unit is portable (exhibits portability) across a class of environments to the degree that the cost to transport and adapt it to a new environment in the class is less than the effort of redevelopment.
Peter J. Brown [4]	Software is said to be <i>portable</i> if it can, with reasonable effort, be made to run on computers other than the one for which it was originally written.
ISO/IEC/IEEE 24765 [24]	Ease with which a system or component can be transferred from one hardware or software environment to another.
ISO/IEC 2382:2015 standard[22]	Capability of a program to be executed on various types of data processing systems without converting the program to a different language and with little or no modification.
ISO/IEC TS 25052-1:2022 [23]	Degree to which a cloud service provides the ability to move data and migrate applications from one cloud service to another.
ISO/IEC 10746-2:2009[21]	Property that the reference points of an object allow it to be adapted to a variety of configurations.

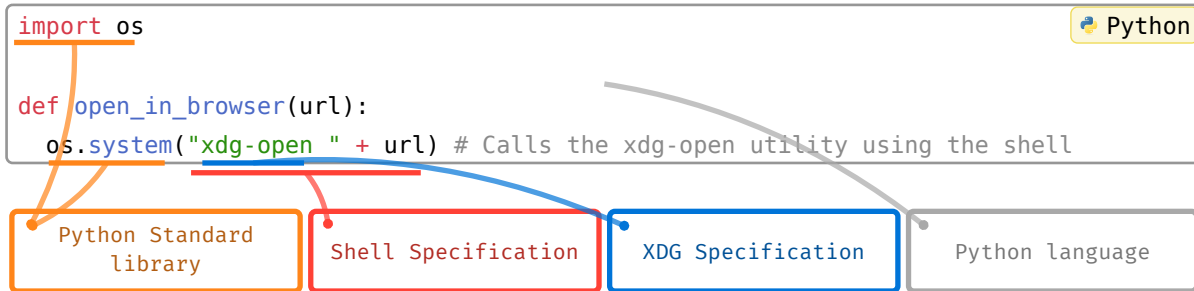


Fig. 3.2. The example from Section 3.3 annotated with all its interface dependencies. As the code is written in Python and uses the standard `os` package, it depends on the Python language and its standard library. The `os.system` function calls the operating system’s shell to run the `xdg-open` binary from the xdg specification. The xdg specification from the freedesktop.org organization standardizes the opening of content inside default programs under Linux.

3.5. Portability as Interfaces and Translators

Programs always depend on existing software or hardware. Even the most low-level program written directly in x86 relies on a CPU and an operating system to communicate with other resources. Communication between the program and its dependencies is done through *interfaces* and a significant part of the porting effort is replacing these interfaces or their implementation. As an example, Figure 3.2 showcases interfaces used by the code in Figure 3.1. This program can be ported to another environment by *translating*, *removing*, or *replacing* the *interface dependencies* with ones running on another environment.

In this section, we will try to understand the relationship between interfaces, environments, and translators in order to understand portability.

3.5.1. Interfaces

At a high level, *interfaces* define how communication between two software or hardware components should occur. For example, any programming language is an interface: the program communicates with the underlying language implementation, which executes it as specified by the programming language. In the same manner, libraries, operating systems and web servers present an interface in the form of standard functions, system calls or URL endpoints. They obey their interface by returning values from functions, changing registers, interrupting the kernel, and responding to web requests.

The actual communication enabled by an interface can have multiple definitions: formal specification, informal documentation, and several implementations. Although multiple definitions exist, the definition that is always realized is the chosen implementation, as it is the one that will be executed. The formalization of an interface in the form of a specification helps when trying to create multiple implementations of the same interface, for example, the C89 standard specification allows the writing of multiple C compilers: `gcc`, `clang` and `msvc`. This ensures that the *portability efforts* stay minimal when a program changes the implementation of an *interface dependency*.

3.5.2. Environments

In this framework, an *environment* can be seen as a set of supported *interfaces*, and programs can only run in a given *environment* if all *interface dependencies* of a program are implemented inside the environment. For example, the program in Figure 3.2 can be executed in the *Linux* environment, as it contains the *C*, *Python* and *xdg* interfaces needed by the program. Note that this assumes that the *Linux* environment has a *C* compiler, *Python* interpreter and uses a graphical desktop environment supporting the freedesktop xdg specification, which might (or might not) be a fair assumption to make.

In addition to the *set of supported interfaces*, an *environment* also has *operational constraints*, meaning the physical limits in terms of memory usage, processing power, or other relevant limits of the environment. For example, running a JavaScript application on a microcontroller might not work because the RAM usage exceeds the amount available inside the microcontroller.

3.5.3. Translators

The action of *porting* can be seen as the process of substituting the *interface dependencies* of a program for other interfaces supported inside the environment. To this end, the program needs to be *modified* in order to change the set of *interface dependencies*, or the *interface dependencies* need to be *translated*. *Modifying* the program in order to change the *interface dependencies* often requires a lot of *effort*, for example, rewriting a program in another programming language interface. As portability is defined as an *effort*, *automatic translation* has to occur whenever possible in order to minimize it.

Automatic translators are defined as all software or hardware that allows the use of one interface as a substitute for another one. These translators come in multiple different forms: libraries (React Native, Flutter), operating system emulators (Docker, Wine), hardware emulators (Rosetta, QEMU, VirtualBox), operating system kernels (Linux, macOS, Windows), compilers (gcc, rustc) and interpreters (CPython, CRuby). In this frame of reasoning, all

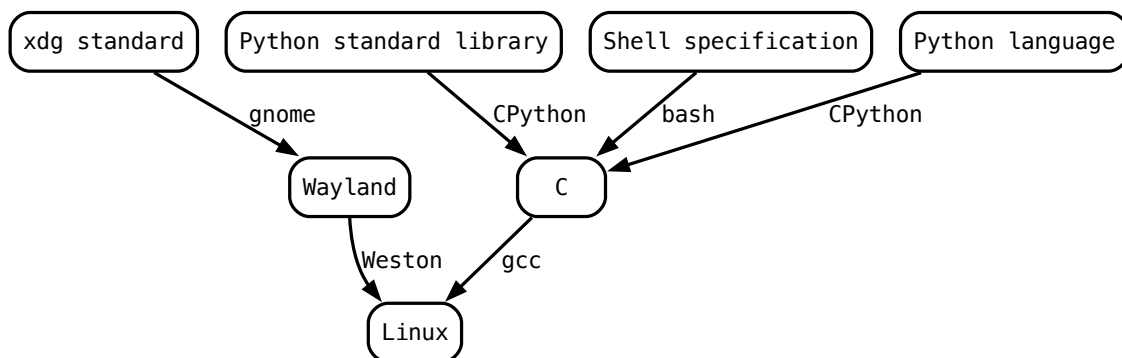


Fig. 3.3. Example of the dependencies of the program in Figure 3.2. Nodes are standard interfaces and arrows represent implementations of these interfaces or *translators*. Gnome is a common desktop environment that can be seen as a translator between the xdg specification and the Wayland protocol. Weston is the reference implementation of the Wayland compositor.

implementations of interfaces are translators, as they convert the interface they implement into the one they are written in.

One property of translators is that they can be used as chains, meaning that their translation can then be passed to another translator and so on until they reach an interface implemented inside the environment. As an example, the Gambit Scheme compiler first compiles a Scheme program into valid C code. Then the C code is translated to the hardware architecture that the user needs depending on the C compiler available on their machine, enabling the execution of Scheme programs on all computers with a C compiler. In reality, most programs rely on a complex chain of *translators* in order to reach any given environment.

3.5.4. Enabling and Disabling Interfaces

Contrary to the connotation of the word *dependence*, the *interface dependencies* themselves are not the limiting factor of portability. Rather, it is their potential to be translated into other environments that can enable or disable the portability of their dependent program.

This observation leads to a few ways in which interfaces can enable portability:

- *Having a heritage of translators* increases the likelihood that newly required environments fall in this set of already supported environments. For example, a program respecting the C language interface can already execute on most hardware architectures known today as translators are plentiful.

- *Trusting the community maintaining the translators* ensures that any future environment will likely be supported by the community. For example, depending on any language officially supported by Microsoft will likely execute on a future Windows version.
- *Minimizing the effort to add missing environments* ensures that programmers can understand and perform the extension to the translators by themselves with limited effort.

Note that any given interface can benefit from some or all of these ways to enable portability. For example, the C language benefits from a very active community that will likely support any new environment while already supporting many targets.

3.6. Compilers for Enabling Portability

Compilers are translators between computer language interfaces. To enable portability, they must enable the programming language interface to be translated effortlessly to a wide range of other interfaces with the *least effort possible by a programmer*.

This goal for compilers leaves a lot of room for interpretation, and showcases the multifaceted property of portability. As an example, components such as documentation, debugging tools, and precise error messages would all reduce the effort a programmer needs to fix bugs encountered while porting and could be considered part of portability.

To clarify our goals, we will focus on three aspects that enable the portability of the programming language interface:

- The difficulty of generating executables that can run on environments having a *wide range of different operational constraints*.
- Virtual machines as a technique to reduce the *effort* to add new target interfaces to an existing compiler.
- Techniques to implement expressive constructs while maintaining portability, such as foreign function interfaces (FFIs).

3.6.1. Adapting to a Variety of Operational Constraints

All computing environments are constrained by operational limits such as computing speed, memory, and power consumption. When generating programs, compilers must generate code within the bounds of these constraints, as otherwise, the program will not run as expected. Consequently, to maximize portability to a wide range of targets, the operational constraint to optimize should be configurable by compilers.

However, there are some practical and theoretical limits to letting users configure which operational constraint gets optimized. The first limitation is that optimizing a program

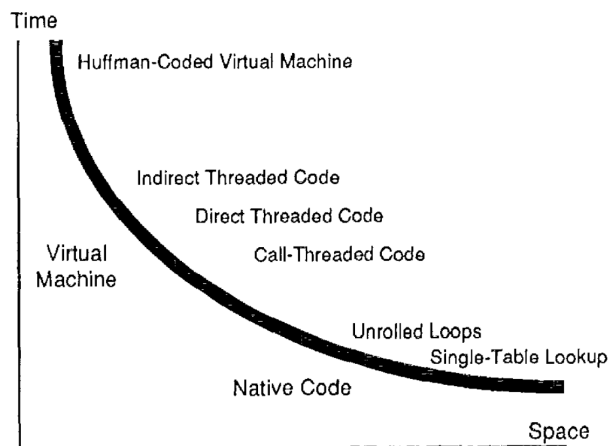


Fig. 3.4. Figure directly taken from Pittman’s article on the tradeoffs between space and time[39] for generating code. Copying is by permission of the Association for Computing Machinery, according to the paper’s license.

requires tradeoffs between the different operational constraints[2, 38], for example, trading speed for memory. To be consistent, compilers make design choices about a main operational constraint to prioritize *over* others. Consequently, it is rare that optimizations for operational constraints other than the main one are implemented, degrading their ability to run on all constrained environments. For example, even the most performant speed-optimizing C compilers have limited options for targeting memory-constrained environments[52], especially compared to all the effort put into making the code run faster.

The second and more fundamental limit is the space-time tradeoff of program generation, discussed by Pittman[39] in the late 80s. Figure 3.4 is taken directly from his paper and summarizes well his viewpoint. To be space-efficient, interpretation has to take place in the generated binary in order to reuse part of the code and make the implementation smaller. To be speed-efficient, specialized code must be generated directly, without using the cost of interpretation. This creates a need for compilers targeting both extremes of memory-constrained and performance-constrained environments to either generate a virtual machine (VM) or native code.

Compilers generating either custom VMs or native code depending on their target environment have yet to appear, and most language implementations work around these limits by having specialized implementations for specific environments. This trend can be observed in the wild, as almost all well-established languages have a secondary specialized implementation for memory-constrained environments: MicroPython (Python), eLua (Lua), JerryScript (JavaScript), .NET Micro framework (C#, F#), TakaTuka (Java), and BIT (Scheme).

This showcases the difficulty of adapting one implementation to the needs of multiple operational constraints, as for all of these specialized implementations, it was easier to rewrite the compiler *from scratch* than to extend or modify the existing compilers. This raises a more fundamental question: *Would it be possible to have one language implementation that can be adapted with minimal effort to generate code to all or most constrained environments?*

3.6.2. Extending Reachable Environments with Virtual Machines

The addition of new target environments is a recurring task for compiler engineers and researchers. For the purposes of this thesis, we will only discuss virtual machines (VMs) as a mechanism for extending the number of targets and omit other techniques such as compiler-compilers (Yacc and Lex) and retargetable compilers (LLVM and Gambit).

VMs are often cited as *the* technique to enable the portability of programs, as commercialized by Sun Microsystems with the launch of Java 1.0. They separate the execution of a program into two steps: the compilation of the program into another machine-readable program, called the bytecode, and the execution of this bytecode. This separation and shared interface are shown in Figure 3.5.

The advantage of having a bytecode as a representation of our program is twofold. First, humans are not expected to read it, allowing it to be optimized for a machine. Most of the time, the bytecode is in a binary format with instructions acting on a stack machine, as this mimics the hardware implementation of computers, simplifying translation to a variety of them. Second, the effort to add a new environment is the same as writing a VM for

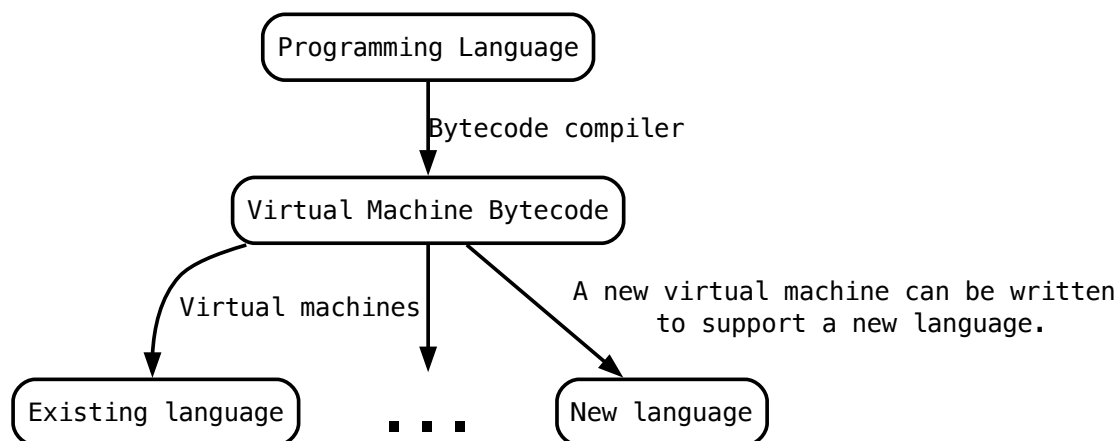


Fig. 3.5. Example of the usage of a virtual machine to have portable programs.

the new environment, allowing the reuse of all the work needed to generate the bytecode representation.

To achieve portability, the effort to create a new VM must be limited. Consequently, the techniques to create VMs are discussed:

- **Writing a new VM from scratch.** When writing a VM from scratch, the design of the bytecode plays an important role in the portability of the system. Bytecodes with a shorter and simpler instruction set allow for the writing of a VM with less effort. However, limiting the instruction set to a minimal core can limit the expressiveness of the language and the optimizations available to speed up the executable.
- **Compiling a VM.** Writing VMs in portable languages is useful for leveraging work already performed by the community and reusing existing translators. However, VMs are programs that require performance and tight control over the running code. To be on the forefront of portability and support targets for which an efficient compiler does not exist, compilation cannot be an option.
- **VM generators.** VM generators take as input a *specification* of the bytecode and generate code that can interpret this bytecode. However, these tools only generate C code, as they need a low-level language to retain sufficient control to properly optimize it. Using this method for generating code to other programming languages remains to be validated.

From the strategies discussed above to create new VMs, only the writing of a VM from scratch extends the number of environments without relying on an already existing solution to portability. For this case, the complexity of a port is directly dependent on the complexity of the bytecode.

When evaluating the current landscape of VMs for common languages, most have extensive instruction sets with complex input/output (I/O) operations. For example, the Java Virtual Machine (JVM) has more than 130 required instructions and the core Wasm specification has a little bit less than 200. This shows that writing a new VM for current VMs is a significant effort, requiring between several months and a couple of years to perform for a single programmer.

This highlights one of the downsides of VMs for portability: their lack of modularity. Because it acts as an *interface*, the standardized bytecode forces VMs to implement *all* defined instructions. This creates a minimal complexity that every VM has to satisfy, making the implementation of new targets more difficult, imposing greater demands on operational constraints of target environments, especially code size, and making it hard to embed VMs in existing programs. Ultimately, forcing a complete implementation of a bytecode hurts portability.

3.6.3. Expressiveness and Portability

To begin the discussion on expressiveness and portability of languages, let's discuss the portability of one of the simplest programming languages to exist: brainfuck. Designed by Muller in 1993, this esoteric language had a simple goal in mind: to create a language that is both Turing complete and has the smallest compiler that implements it. Brainfuck provides a single abstraction, a pointer to an infinite tape where values can be read and written to. Eight operations, represented by single characters, allow the pointer to move left or right on the tape `< / >`, the increase or decrease of the pointed value on the tape `+ / -`, the looping of the program `[ / ]`, and the printing or reading of the pointed value `. / ,`. Programs are a series of these eight characters, making programs very hard to read and understand, hence the name brainfuck.

Brainfuck is surprisingly one of the more portable programming languages. Considering the definition of portability in Section 3.4, the effort needed to port any brainfuck codebase is at most the effort to write a compiler or interpreter for this new environment, a very marginal effort compared to most ports (assuming that performance is not an issue). Although it would be a portability paradise to have codebases in brainfuck, it would be hell for programmers maintaining and developing the software. Brainfuck provides only one abstraction (the tape) and one method of communication (the printing and reading of single characters). This heavily limits the writing of any common program.

This little thought experiment showcases the conflicts between expressiveness and portability. The inexpressiveness of the brainfuck language significantly reduces the portability effort. In this case, however, the reduced effort is completely offset by higher development effort caused by the language's inexpressiveness. This doesn't necessarily have to be the case, and this section explores how the *design* of a programming language implementation can mediate this tension by providing expressive yet portable abstractions when possible.

3.6.3.1. The Portability Cost of *Language Features*. It is generally understood that the expressivity of a program comes from its *language features* [13], with most *language features* incurring a cost to portability. These costs come from the additional *code* responsible for the implementation of the feature which negatively affects portability in two ways. First, the additional code complexifies the implementation of the language and sometimes adds dependencies on external interfaces, thus making it more effortful to port and less portable. Second, the additional code consumes more resources from the environment, thus further constraining the environments where it can run.

Some *language features* require changes in the execution model of the program even when the feature itself is not directly used. A typical example is *variadic functions*, meaning

functions that can accumulate excess arguments inside a list. Implementing this feature requires a change to the call protocol, passing explicitly the number of *applied* arguments in order to compare it to the *formal* arguments to construct the list with the remaining arguments. This significantly increases the code’s footprint as this affects *all* function calls, even the ones that are not variadic, having a global negative impact on portability.

Here are a few additional examples of language features with a global negative impact: floating-point numbers requiring additional checks for standard numerical operations, exceptions requiring additional metadata for the stack unwinding, and reflectivity requiring additional metadata about the modified objects.

As some *language features* can have global negative impacts on portability, a language implementation prioritizing portability should tightly control their availability, disabling them to trade portability for expressivity when needed. This is easier to achieve when the whole source program is known, such as in full-program compilation, as these features can be detected or manually disabled for a specific generated executable. For interpreters and VMs that are often already preinstalled on the target environment, the disabling of certain features can cause incompatibilities, as it can change the accepted input programs. However, this can be a necessary compromise as implementing all the language features while maintaining portability to constrained environments can be near impossible. For example, MicroPython chose to implement only a subset of the Python language, leaving out detrimental language features, to better target memory constrained environments.

3.6.3.2. Minimizing Primitives. Every environment has different mechanisms to perform operations such as arithmetic operations, input-output communication, and concurrency. To express these operations inside a programming language implemented on top of the environment, certain language operations need to be translated into these environment-specific operations. This is done through *primitives* callable from the language that execute code inside the language *implementation*.

Primitives are by definition environment-specific and need to be reimplemented when changing environments. To reimplement these primitives with minimal effort, the number of primitives and their complexity must be minimized. However, a small set of simple primitives can sometimes not provide the performance or expressivity needed by the program. Therefore, to accommodate all program needs, different implementations of *primitives* are required to choose the right balance between portability, expressivity, and performance.

3.6.3.3. Foreign Function Interfaces. Foreign function interfaces (FFI) allow the execution of foreign functions, meaning functions written in another programming language than the one they are called from. By definition, FFIs limit portability as they add a dependent

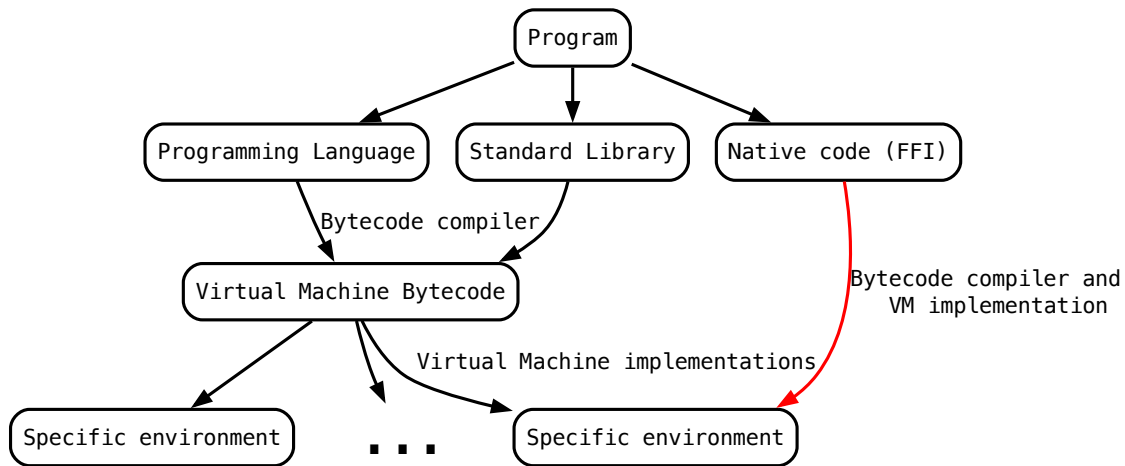


Fig. 3.6. Figure showcasing how foreign function interfaces (FFIs) bypass the bytecode by having a dependency directly to a new implementation.

interface that *is* another programming language as shown in Figure 3.6. In this section, we will only focus on FFIs as an *extension* mechanism for the programming language.

Programming languages provide an interface limited by the *primitives* defined by them. To extend the expressivity of the language and properly adapt to environments, this set of primitives sometimes needs to be extended. FFIs allow the execution of code at a lower abstraction level, such as inside the VM or the target language. For example, Java’s Native Interface allows the linking of C/C++ code to the JVM reference implementation, allowing these primitives to be called from Java code. However, the implementation of FFIs is typically complex, relying on Application Binary Interfaces (ABI) to communicate with the language.

3.7. Incremental Portability

High portability acts like insurance for the development effort put into programs. It ensures that this development effort is not wasted when the environment changes. However, guaranteeing *high portability* is impossible, as predicting the future environments of programs is in the realm of speculation. This is especially true considering that environments are constantly changing, and the environments of tomorrow might not even exist today.

To partially remediate this, this section presents the notion of *incremental portability*. This notion is rooted in the following intuition: *future needed environments will share common characteristics with the one currently supported*. When this intuition is true, *high*

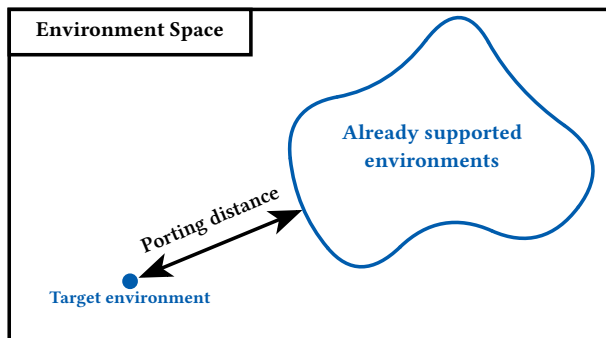


Fig. 3.7. Illustration showing the porting distance in the environment space.

portability can be partially achieved by limiting the effort to port to environments that are *near* currently supported environments.

More precisely, *incremental portability* can be defined as bounding the effort to support an environment by its *porting distance* from the *already supported surface*, as shown in Figure 3.7. In this figure, the *environment space* is a multi-dimensional space where each dimension represents a property of the environment: its available memory, its processing speed, each available interface, the available input/output (I/O) and so on. In the figure, the *environment space* is simplified to two dimensions, but it can be generalized to any number of relevant properties. Because it forms a space, a notion of *distance* is inferred where environments closer to one another share more similar characteristics than the ones further away. Within this space, environments already supported by a program using a language implementation form a multi-dimensional *surface*. The goal to attain *incremental portability* is to ensure the *distance* to any target environment is directly proportional to the *effort* required to support it. In other words, it ensures that portability can be extended progressively with limited additional effort.

Another way to frame *incremental portability* is through the management of tradeoffs. To provide *portability*, Section 3.6 established that the compiler should give *options* to the programmer to trade certain operational constraints against others. To provide *incremental portability*, the compiler should allow these tradeoffs to be made in incremental steps. For example, to trade language expressivity for compactness on disk, the compiler could provide the disabling of the standard library, which is *not* incremental. Instead, the compiler should provide varying levels of already-implemented compact standard libraries, or a control over size-detrimental language features, which allows for a finer-grained control when a program requires only a little bit more space. Allowing for tradeoff options to be incremental enables *incremental portability*, as it limits the effort required by a programmer to port to environments at varying *porting distances*.

Incremental portability is not meant as an absolute goal. Rather, it prompts us to wonder about the various properties of environments that could be constrained past the edge of current support, and whether the compiler’s design is robust to these changes. Moreover, each of the properties could be constrained at varying orders of magnitude, prompting the question of whether the compiler provides *incremental* options for each of these distances. For example, *for a given program, with how much effort can the compiler support environments with slightly less space on disk, significantly less space on disk, running on another programming language, or without a file system?*

3.8. Final Thoughts on Portability

Portability for compilers is often simplified to the number of supported targets, or the compiler’s ability to target a well-supported language such as C. Hopefully, this chapter deconstructed this belief by framing portability as an *effort*, and thinking about the design of compilers to *reduce* this effort. This perspective reveals important ingredients for building compilers for portability, such as a control over the expressivity of the input language, limiting the time to add a new target, and the configuration of which operational constraint to optimize. *Incremental portability* was also presented to give some guidelines to design compilers for portability.

The chapter is meant as an introduction of the several challenges to design compilers for portability. The next chapter presents a compiler for portability called Ribbit, which takes some of the ideas presented in this chapter and implements them in practice.

Chapter 4

Ribbit

Ribbit is a Scheme implementation born from the pursuit of effortless portability. While working towards this goal, a general compiler technique emerged, allowing the generation of modular and portable virtual machines called Virtual Machine Tailoring. This chapter focuses on the original implementation of this compiler technique, Ribbit. It provides background to understand Virtual Machine Tailoring as explained in Chapter 5.

Ribbit is based on a virtual machine (VM) called the Ribbit Virtual Machine (RVM). With the goal of effortlessly adding new targets, the VM algorithm has been designed for easy implementation in all common programming languages. This is done by having a few yet powerful instructions and a uniform model for *all* memory allocations (including the stack, heap, and Scheme code and objects). Furthermore, the virtual machine has been optimized for on-disk code size with the possibility of trading space for performance, allowing execution inside most environments regardless of their operational constraints.

Ribbit's portability achievements suggest an effective design. Ribbit supports 25 target languages with a variety of constraints: Prolog (backtracking execution model), POSIX (no built-in data structures), C (manually managed memory), x86 (no high-level abstractions), Haskell (strong static typing) and Python (high-level and dynamic). Most of the targets have been implemented in less than **two week of work** each, suggesting that new virtual machines can be implemented for almost all targets with limited effort. Furthermore, Ribbit generated the smallest fully-compliant Scheme R4RS interpreter within a 6.5KB binary. Overall, Ribbit is the most portable Scheme implementation in terms of the number of supported targets and the effort to add new ones, while generating very compact code.

The chapter is divided as follows. Section 4.2 explains the design of RVMs allowing their implementation with limited effort and Section 4.3 discusses the design of the bytecode allowing compact code. Note that the Ribbit Scheme Compiler is not explicitly discussed, as it is covered as part of Chapter 5 on Virtual Machine Tailoring.

4.1. Credit

Ribbit was first introduced in a paper by *Yvon and Feeley*[55] and Section 4.2 introduces the same concepts as in the paper. The rest of the sections are original contributions by the author of this thesis, some already covered in already published papers under his name as first author: *A Compact and Extensible Portable Scheme VM* [34] (from Oest O’Leary and Feeley) and *A R4RS Compliant REPL in 7 KB* [35] (from Oest O’Leary, Laroche and Feeley). The author took the liberty of taking small sections of these two last papers literally inside this thesis.

4.2. The Ribbit Virtual Machine

In this section, the design of the Ribbit Virtual Machine algorithm is discussed, with considerations for easy implementation within all languages.

4.2.1. Ribs and the Rib Graph

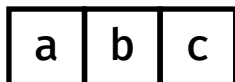


Fig. 4.1. Example of a rib containing the values **a**, **b** and **c**.

All data allocated by the Ribbit Virtual Machine (RVM), including its own stack, the code to execute and the representation of Scheme objects, is solely composed of 3 field vectors called *ribs*. This choice is at the core of Ribbit because it simplifies many components of a virtual machine:

- When porting the RVM to a new programming language, *ribs* can most likely be implemented as a host language object. In host languages that don’t support objects, it simplifies memory management as *all* allocated data has the same fixed size.
- Code and data can be serialized together as they have the same structure.
- The representation of the stack as a linked list of *ribs* ensures there is a single memory model for all objects with no arbitrary limits on the size of the stack.

Ribs contain **3 fields**, and each field can either contain an integer or a reference to another rib. A size of 3 was chosen as it is wide enough to represent pairs and virtual machine instructions without any extra fields: instructions contain an opcode, an operand and the next instruction to execute, while pairs can be represented with their two values and a tag to mark them as a pair. Pairs can then be used to represent other data types, such as lists, vectors and strings.

Ribs are linked together by referencing each other within one of their three fields, forming the *rib graph*. All non-primitive operations performed by the virtual machine are analogous to operations on the *rib graph*. For example, pushing a value onto the stack corresponds to moving the pushed rib to the one pointed by the stack (which is itself a linked list of ribs within the *rib graph*). In essence, the *rib graph* represents the whole running state of the virtual machine.

In the rest of the thesis, we will use the notation `[a,b,c]` to mean a rib with the fields `a`, `b`, and `c`, as shown in Figure 4.1. This also happens to be the implementation of ribs in the Python and JavaScript RVMS, among others. When ribs reference each other, we will visualize this with arrows as shown in Figure 4.3.

4.2.2. Representation of Scheme Objects

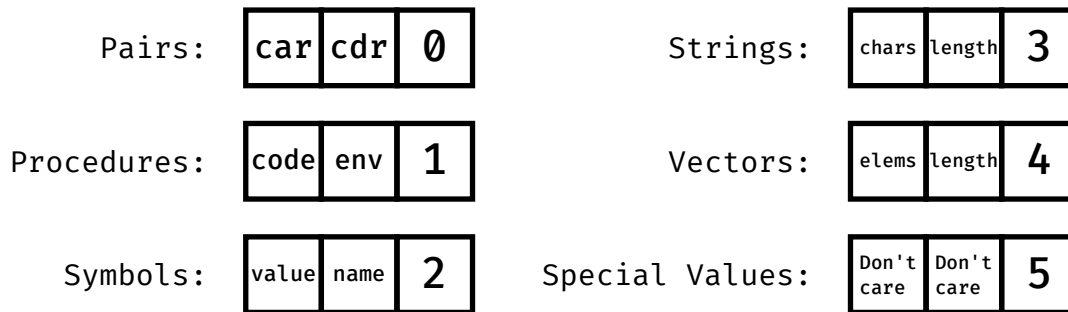


Fig. 4.2. The representation of data in Ribbit as *ribs*.

Scheme objects are represented in the host language using one or more ribs where the last field is an integer indicating the type: 0 for pairs, 1 for procedure, 2 for symbol, 3 for strings, 4 for vectors, and 5 for special values as shown in Figure 4.2. Each type will be briefly discussed in ascending order of complexity, except for procedures which are discussed in Section 4.2.4.

4.2.2.1. Pairs. Pairs are a fundamental data type in Scheme used to create lists by chaining them in their second field. A single rib can be used to represent them where the first two fields are the first and second field of the pair (also referred to as `car` and `cdr` in Scheme). As an example, the Scheme improper list `(1 2 . 3)` is represented using two ribs: `[1, [2, 3, 0], 0]` as shown in Figure 4.3.

4.2.2.2. Special Values. Special values represent the Scheme values `#t` (true), `#f` (false) and `'()` (nil). Usually, these values are created as unique instances of a rib stored directly as variables inside the virtual machine implementation, distinguishing them by their memory

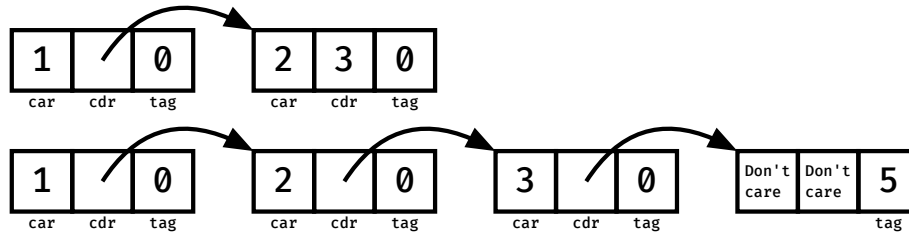


Fig. 4.3. Example of the representation of the (1 2 . 3) improper list at the top and the '(1 2 3) proper list at the bottom.

location. As an example, the proper list '(1 2 3) is represented using ribs as shown in Figure 4.3 ending with the unique rib instance representing the nil value.

4.2.2.3. Vectors and Strings. Strings and vectors are represented using a rib where their first field is a list of their elements and the second field is their length. For strings, the list stores the ASCII codes (integers) forming the string. For vectors, the list can store any value, as vectors are heterogeneous in Scheme. Representing strings and vectors as a linked list of ribs has the advantage of being portable across hosts. However, this also comes at the cost of accessing elements inside vectors and characters inside strings in linear time relative to the accessed position. This can sometimes cause problems and other representations are further discussed in Section 4.2.6.

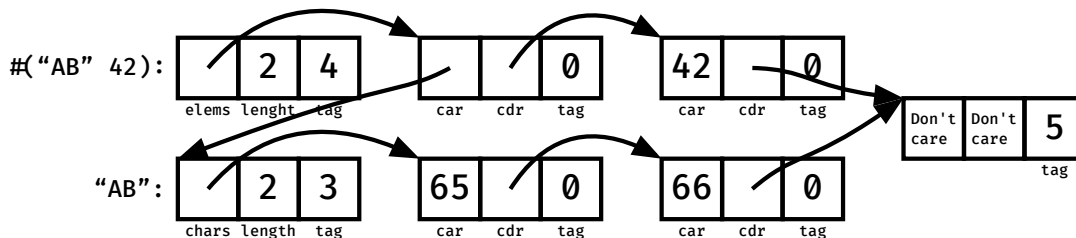


Fig. 4.4. Example of the vector #("AB" 42) (top) and the string "AB" (bottom) represented as ribs. Note that the vector contains the string.

4.2.2.4. Symbols. The symbol table in Ribbit is stored within the representation of symbols, avoiding any extra data structure. This is achieved by storing the value of the global variable represented by the symbol in the first field of the symbol rib. The second field is used to store the string representation of the symbol. Note that sometimes, the compiler avoids including the string representation of a symbol when it can prove that it is never accessed (for example, when the function `symbol->string` is never used).

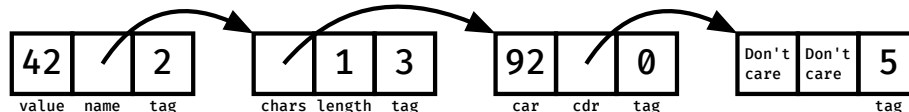


Fig. 4.5. Example of the symbol 'a' (leftmost rib) containing its string representation "a" inside the *name* field. In this example, the global variable `a` contains the number 42 as seen by the *value* field of the symbol 'a'. This could be the result of a direct assignment such as `(define a 42)`.

4.2.3. Instruction Ribs

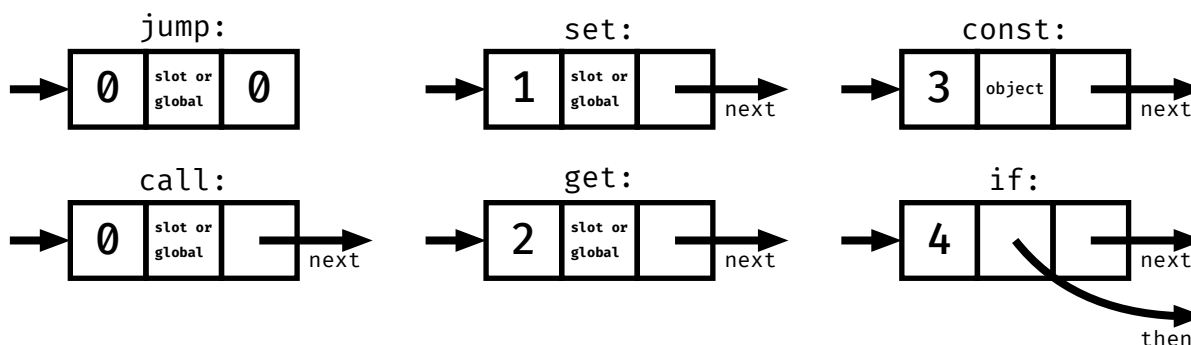


Fig. 4.6. All instruction ribs in Ribbit.

The Ribbit Virtual Machine (RVM) is a stack machine executing instructions stored in the heap as *ribs*, called *instruction ribs*. This is unusual as virtual machine instructions are typically a sequence of bytecode in a dedicated region of memory. This aligns with our portability goals as having a single data structure heavily simplifies the implementation of RVMs.

The RVM executes 6 types of instructions, closely related to Scheme constructs: `jump` (tail call), `call` (non-tail call), `set` (writing a variable), `get` (reading a variable), `const` (literal data), and `if` (conditional execution). The type of an *instruction rib* is contained within the first field as an integer, called the *opcode* as shown in Figure 4.6. The third field of an *instruction rib*, called the *next* field, contains the next instruction to execute (except for `jump`).

4.2.3.1. The Instruction's *Argument*. The `jump`, `call`, `set`, and `get` instruction ribs contain an *operand* in their second field that identifies either a stack slot or a global variable depending on the kind of the *operand*. If the *operand* is an integer, the *operand* denotes an index on the stack. If the *operand* is a reference, this reference points to the symbol of the global variable denoted by the *operand*. In both cases, the *operand* identifies a rib

containing the value that is acted on by the instruction: a pair from the stack list or the symbol containing the global variable's value. The value contained in the rib identified by the *operand* is called the *argument* of the instruction. Note that in both cases, the value is contained in the first field of the identified rib.

4.2.3.2. The `set` and `get` Instructions. The `set` and `get` instructions are stack operations. The `set` instruction mutates the *argument* with the top of stack (TOS) and `get` retrieves the *argument* and pushes it onto the stack.

4.2.3.3. The `jump` and `call` Instructions. The `jump` and `call` instructions have the same opcode, the integer 0, and are distinguished by their *next* field containing either a reference to the next instruction to execute (`call`) or the integer 0 (`jump`). The `jump` instruction is guaranteed to perform a tail-call optimization, a necessity in the Scheme standards allowing the writing of loops as recursive functions. When executed, the `jump` and `call` instructions change the *program pointer* of the RVM according to the *call protocol* described in Section 4.2.4.6.

4.2.3.4. The `const` Instruction. The `const` instruction pushes the content of its second field onto the stack. This instruction must not be confused with the `get` instruction, which pushes the content of the *argument* (Section 4.2.3.1) instead of the literal value contained inside the second field.

The `const` instruction is commonly used to define new procedures at the start of the program by pushing their literal content onto the stack and then using the `set` instruction to mutate them inside their global variable's value.

4.2.3.5. The `if` Instruction. The `if` instruction is the only operation to reference two other instruction ribs in its *then* and *next* fields, corresponding to the two branches of an *if*. When the TOS is a reference to the special value rib `#f`, the program pointer is mutated to the *next* field. Otherwise, the program pointer is mutated to the *then* field.

4.2.3.6. Simple Addition Example. As also shown in Figure 4.7, the operation `(+ 42 43)` is represented using both the `const` and `jump` instructions: `[3, 42, [3, 43, [0, '+, 0]]]` where `'+` is a reference to the symbol `+` (a rib), 3 is the opcode of the `const` instruction (push to stack) and 0 is the opcode for the `jump/call` instructions.

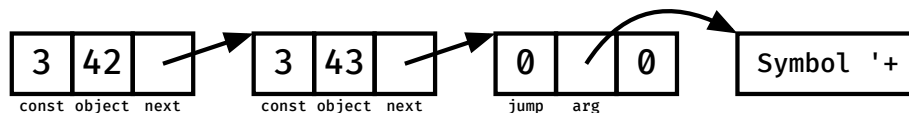


Fig. 4.7. Example of the Scheme code `(+ 42 43)` represented as ribs. The fields of the rib representing the symbol `'+` are not shown.

4.2.4. Procedures

Procedures are the only objects that can be called by the `jump` or `call` instructions of the virtual machine and are distinguished from other objects by having a *tag* of 1 as shown in Figure 4.8. Procedures come in two forms: *primitives* and *closures*.

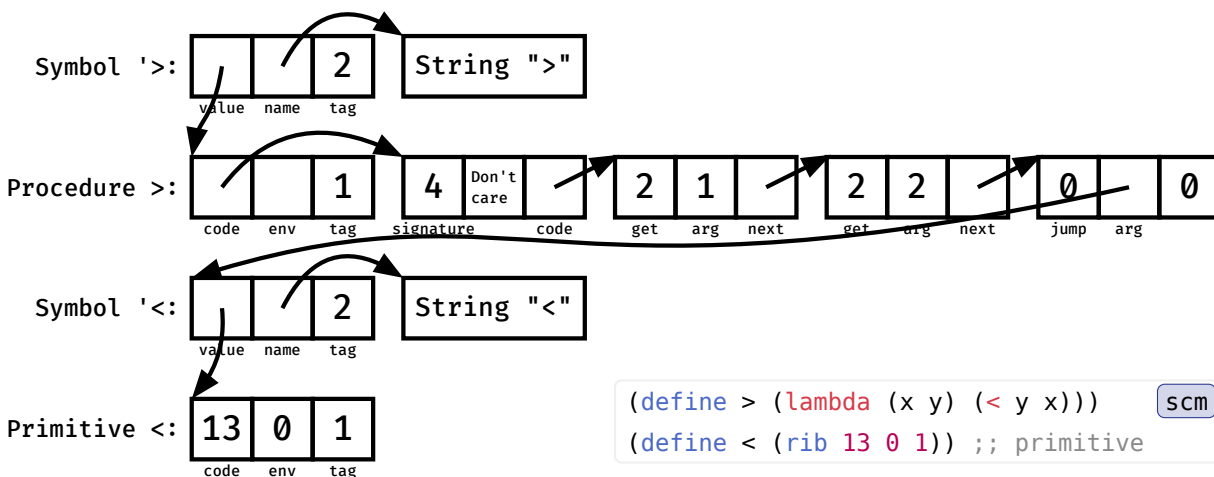


Fig. 4.8. Example of the representation as ribs of procedure `>` defined as a call to the primitive `<`. Code for the procedure is displayed on the bottom-right.

4.2.4.1. Primitive Procedures. *Primitive procedures* enable the execution of primitive operations, such as addition or input/output operations. They always contain an integer in their *code* field identifying a host primitive in the host primitive table as shown in Figure 4.8. When a *primitive procedure* is executed by a `jump/call` instruction, the identified host primitive is executed and is expected to pop from the stack the required arguments and push back the result. For example, the `+` primitive pops two numbers from the stack and pushes back their addition.

All default primitives in Ribbit are shown in the Table 4.1. Note that this set of primitives is not enough to implement the full R4RS specification as it misses, for example, file I/O primitives. To alleviate this, compile-time extensions to the virtual machine are performed with Virtual Machine Tailoring. This is further explained in Chapter 5.

4.2.4.2. The Apply Primitive. The `apply` procedure in Scheme takes a procedure and a list as arguments, calling the provided procedure with the provided list as its argument. For example, `(apply + '(1 2 3))` results in `(+ 1 2 3)`. Implementing `apply` requires a tight control over the interpreter loop, as the call to `apply` has to be replaced by a call to another function.

Table 4.1. The default primitive table inside of Ribbit.

Name	Opcode	Semantics
rib	0	$z \leftarrow \text{pop}(); y \leftarrow \text{pop}(); x \leftarrow \text{pop}(); \text{push}(\text{rib}(x, y, z))$
id	1	$x \leftarrow \text{pop}(); \text{push}(x)$
arg1	2	$y \leftarrow \text{pop}(); x \leftarrow \text{pop}(); \text{push}(x)$
arg2	3	$y \leftarrow \text{pop}(); x \leftarrow \text{pop}(); \text{push}(y)$
close	4	$x \leftarrow \text{pop}(); \text{push}(\text{rib}(x[0], \text{stack}, 1))$
rib?	5	$x \leftarrow \text{pop}(); \text{push}(x \text{ is a rib})$
field0	6	$x \leftarrow \text{pop}(); \text{push}(x[0])$
field1	7	$x \leftarrow \text{pop}(); \text{push}(x[1])$
field2	8	$x \leftarrow \text{pop}(); \text{push}(x[2])$
field0-set!	9	$y \leftarrow \text{pop}(); x \leftarrow \text{pop}(); x[0] \leftarrow y; \text{push}(y)$
field1-set!	10	$y \leftarrow \text{pop}(); x \leftarrow \text{pop}(); x[1] \leftarrow y; \text{push}(y)$
field2-set!	11	$y \leftarrow \text{pop}(); x \leftarrow \text{pop}(); x[2] \leftarrow y; \text{push}(y)$
eqv?	12	$y \leftarrow \text{pop}(); x \leftarrow \text{pop}(); \text{push}(x \text{ is identical to } y)$
<	13	$y \leftarrow \text{pop}(); x \leftarrow \text{pop}(); \text{push}(x < y)$
+	14	$y \leftarrow \text{pop}(); x \leftarrow \text{pop}(); \text{push}(x + y)$
-	15	$y \leftarrow \text{pop}(); x \leftarrow \text{pop}(); \text{push}(x - y)$
*	16	$y \leftarrow \text{pop}(); x \leftarrow \text{pop}(); \text{push}(x * y)$
quotient	17	$y \leftarrow \text{pop}(); x \leftarrow \text{pop}(); \text{push}(x // y)$
getchar	18	$x \leftarrow \text{getchar}(); \text{push}(x)$
putchar	19	$x \leftarrow \text{pop}(); \text{putchar}(x); \text{push}(x)$

To allow the implementation of `apply` using primitives in Ribbit, a convention is used within the virtual machine implementations. Primitives, which are often implemented as host functions, can *return* a procedure to be called immediately by the virtual machine. This allows the implementation of the `apply` primitive by first setting up the stack with the values contained within the list and then *returning* the function to be called.

4.2.4.3. Controlling the Virtual Machine Control Flow with Primitives. As most hosts implement host primitives as host functions, the *return value* of the host functions can be used as a general mechanism to control the flow of execution within the virtual machine. Depending on the constraints of the environment, such a mechanism can be required. For example, when executing JavaScript inside a browser, control must be given back to the browser once in a while to avoid hanging the page. In this case, this mechanism can be

implemented by exiting the virtual machine when a primitive *returns* the *false* JavaScript value, giving back control to the browser.

4.2.4.4. Closure Procedures. Closures are distinguished from primitives by their *code* field being a reference to a rib instead of a number as shown in the Figure 4.8. The *code* field references a rib containing in its third field a reference to the instruction ribs of the procedure and in its first field the signature number. The signature number specifies the number of formal arguments and whether more actual arguments are accepted (variadics). The least-significant bit is used to encode the variadicity and the rest encode the number of formal arguments. For example in Figure 4.8, the `>` procedure has a signature number of 4 (100 in binary) meaning 2 formal arguments (10 in binary) and no variadicity (binary 0).

Closures, as their name suggests, capture free variables present in their lexical scope. In Ribbit, this capture can be done in two ways: using the `close` primitive or by using *safe-closures*.

Using `close` after defining a procedure populates its *env* field with the current stack pointer, allowing access to free variables through it. However, referencing the whole stack within a procedure leaks more data than necessary: variables not bound by the procedure and in previous call frames will still be reachable from this procedure, preventing the garbage collector from collecting them. Closures created this way are not *safe-for-space*[43, 37], but can be preferred because they generate less code to be closed, only needing one primitive call.

Safe-closures ensure that only the free variables present inside the lexical scope of a procedure are captured. When *safe-closures* are activated, the compiler performs a *closure conversion*, generating additional instructions to create a list of ribs containing only the free variables. This list is then assigned to the *env* field of a procedure. This ensures that closures are *safe-for-space*[43, 37], but requires more instructions to be generated, inducing a bigger footprint. This technique is similar to *flat closures*, however, free variables are not necessarily stored in a contiguous region of memory because the *env* list is created with 3-field ribs.

4.2.4.5. Stack Frames and Free Variables. Variables belonging to different functions in the call hierarchy are distinguished with stack frames. In Ribbit, as the stack is a linked list of ribs, frames are separated with a special rib at the boundaries of function calls called the *continuation rib*. The *continuation rib* is linked instead of a pair within the stack and can be identified by having a reference instead of the integer 0 in its last field. The *continuation rib* contains all the information to give back control to the calling function: a reference to the previous stack in its first field and the previous program pointer in its third field.

The second field of the *continuation rib* contains the procedure currently under execution, giving access to free variables with only the use of the `get` instruction rib. As the

`get` instruction accesses the stack at a specified index by following references in its second field, free variables can be accessed by providing an index greater than the position of the *continuation rib* effectively accessing the *procedure* followed by the *env* field of the procedure containing the free variables.

4.2.4.6. The Call Protocol. When a `call` instruction is executed with a *closure procedure*, a new *continuation rib* is allocated and populated with the called *procedure* and the *next* instruction of the *program pointer*. The *continuation rib* is then linked with the current stack, starting a new stack frame yet to be assigned to the current stack. Then, the *number of arguments* is retrieved from the *closure procedure* and the new stack frame is filled by popping each argument from the current stack. The continuation rib is then mutated with the value of the current stack. Finally the *program pointer* is changed to the procedure's code and the current stack is changed to the new stack.

When a `jump` instruction is executed with a *closure procedure*, the procedure call is tail-call optimized slightly changing the call protocol. Instead of populating the continuation rib with the current stack and *next* instruction of the *program pointer* (which doesn't exist for `jump`) the *continuation rib* is populated with the *stack* and *program pointer* of the *next continuation rib in the stack*. This leaves the in-between stack frame unreachable from the current stack and program pointer.

4.2.4.7. Variadic Procedures. Variadic procedures are implemented by pushing the *number of arguments* onto the stack before each function call. By comparing this pushed value with the number of arguments accepted by the procedure (derived from the *arity number* of the procedure), the virtual machine can determine if the function call is valid and whether excess arguments must be accumulated into a list as a *rest* parameter.

Implementing variadic parameters by pushing the number of arguments onto the stack increases the footprint of the generated program, as an additional `const` instruction must be generated as part of each function call. For programs that never use variadics, this leads to unnecessary bloat as the possibility of having variadics forces every call to contain an additional instruction.

4.2.5. Memory Management

Ribbit Virtual Machines (RVMs) aim to be implemented in all programming languages regardless of their memory allocation strategies and this is simplified with the usage of *ribs* as a single data structure for *all* memory allocations: the stack, global variables, symbol table, Scheme objects, and code. Languages that can implement *ribs* using host values can reuse their memory management strategy, while simplifying the implementation of a garbage collector as all memory allocations are homogeneous.

For hosts managing their memory automatically, RVM implementations can reuse the language’s memory collecting strategies on the condition that heterogeneous nested lists are supported as a host value inside the language, meaning a data structure with at least 3 fields containing both integer values and references to another instance of the data structure (sometimes cyclic). This comes from the necessity to track the references of Ribs *by the host language* in order to determine unreachable memory by its underlying memory collecting strategy. Most languages managing their memory implement heterogeneous nested lists and here are a few examples: Python and JavaScript through native arrays, PostScript through values/references on the native stack, and Haskell with algebraic data types.

For hosts allocating memory manually, such as C or x86, the simple memory model allows for the writing of a garbage collector succinctly. As all objects have 3 fields, no header is needed to keep track of the size of the object. The garbage collector only needs to track whether the value contained within a *rib* needs to be followed or not (is an integer or another *rib*). In C and x86, this distinction is encoded using pointer tagging, reusing one bit within the object’s reference pointer. As the code is represented as heap allocated ribs, there is no need to distinguish return addresses from object pointers and dead code is deallocated for free. Furthermore, the garbage collector only needs to be aware of two roots, aside from the constants `#t`, `#f`, and `'()` : (1) the *rib* at the top of stack (TOS) linking all variables of the environment including previous frames and (2) the *program pointer* (pp) as instructions are also ribs that can be collected. Because of these simplifications, stop-and-copy and mark-and-sweep garbage collectors have been implemented in under 200 lines of code in both the x86 and C hosts. Ribbit was also used to prototype a novel garbage collector algorithm, the arborescent garbage collector[27].

Overall, the usage of *ribs* simplifies memory management for almost all languages. Indeed, languages that manage their own memory are typically expressive enough to allow the implementation of *ribs* as host values, and languages that do not manage their memory can easily implement a garbage collector. This tendency has been observed for 23 of the 25 hosts that Ribbit supports. The two outliers were POSIX shell and BASIC, which managed their own memory but did not have a native data structure to implement *ribs*.

To implement an RVM in these languages, a heap is simulated inside a large array with references as indexes to this array. As indexes are simple integers and not seen as references by the language, a garbage collector is needed to clean up unreachable memory. POSIX-shell was especially tricky to implement as the standard does not define any arrays, requiring the simulation of arrays with variable names and the arithmetic evaluation[18]. For example, the value of `array[n]` can be retrieved with `$((array_$i))` where `i` is the index of the array.

4.2.6. Other Vector Representations

By default, vectors are implemented as a linked list of ribs. This choice can enable portability as it reuses the already implemented *rib* data structure, simplifying memory management and host implementations. At the same time, this choice can disable portability as having a linear time access into vectors can lead to unacceptable performance slowdowns, needing additional efforts to optimize the program. This subsection explores other vector representations that trade ease of implementation or program size for performance.

One approach is to implement vectors as ternary trees. This representation accesses elements in $O(\log(n))$ where n is the size of the vector and avoids any modification of the target by reusing the *rib* data structure. However, this representation leads to bigger generated programs to include the implementation of ternary trees and can be slower than the linked-list implementation for small indexes inside large vectors.

Another approach is to implement vectors as primitives inside the target language. This option allows the access speed of elements inside vectors to be the same as inside the target language. However, this representation leads to bigger generated programs and complicates the implementation of the target language, especially when the target virtual machine implements a garbage collector, breaking the invariant that everything is a 3 sized rib.

Note that all three representations can be implemented inside libraries by defining the `make-vector`, `vector-set!` and `vector-get` procedures as primitive or closure procedures.

4.3. Encoding of the Rib Graph

When compiling a program, Ribbit embeds an encoding of the *rib graph* as a string or table literal within the generated program. The representation of the graph as an embeddable value inside the program is called the Ribbit Intermediate Byte Notation (RIBN), pronounced *ribbon*. This section explains how the encoding and decoding of the rib graph is performed.

At a high level, the RIBN is decoded by a small stack-based state machine, reconstructing the graph using *decoding instructions*. These instructions allocate and link new ribs, recreating the *rib graph* in memory. Once the rib graph is decoded, it can be executed by the Ribbit Virtual Machine (RVM) interpreter as described in the previous section (Section 4.2).

To be as portable as possible, programs generated by Ribbit need to have a small footprint, requiring a compact encoding of the *rib graph*. This calls for the optimization of both the size of the RIBN and the decoder's implementation, as they are both included in the final program generated by Ribbit. This guided our choice towards a simple decoding algorithm with compact decoding instructions.

4.3.1. The Symbol Table

To construct the instruction ribs, references to symbols are sometimes required. For example, calling the global variable `hello` is written as: `[0, 'hello, 0]`, where `'hello` is a reference to the symbol `hello`. For this reason, the RIBN is composed of two main parts: the symbol table followed by the decoding instructions. This allows the decoder to pre-construct the symbol table and reference it while executing the decoding instructions.

In memory, the symbol table is represented as a linked list of symbols, where the position of a symbol in the list is its index. When encoding the symbol table inside the RIBN, the list, as well as the comma-separated string representation of symbols, are encoded in reverse order:

```
esolc%,2gra%,*%,di%,1gra%,<%,-%,f,,,bir%%
```

Representing symbol strings and lists in reverse allows their reconstruction more easily: starting with the nil value, a rib can be created that contains both the current element and a reference to the previously created instance. For example, the symbol `%%close` can be created by starting with nil: `[0, 0, 5]` and linking (in reverse): `[e, [0, 0, 5], 0]`, `[s, [e, [0, 0, 5], 0], 0]`, etc. The same is done for the symbol table with its symbol elements. For example, here the **last** symbol of the symbol table is `%%close` (found at the beginning of the encoded symbol table in reverse). The **first** symbol would be `%%rib`.

Some symbols don't need a string representation and are therefore not included to save space in the RIBN, as presented by the missing values between the commas after the character `f` in the encoded format. Because there might be multiple unnamed symbols, the first character of the RIBN includes the number of unnamed symbols to include encoded as a character. Here, the character `#` corresponds to the creation of 1 unnamed symbols and the character `;` is used to distinguish the symbol table representation from the decoding instructions:

```
#esolc%,2gra%,*%,di%,1gra%,<%,-%,f,,,bir%%;<decoding instructions>
```

The AOT compiler will optimize the order of the symbols depending on their usage in the program, ensuring that the most used symbols have lower indexes, reducing the number of instructions to decode them.

4.3.2. Decoding Instructions

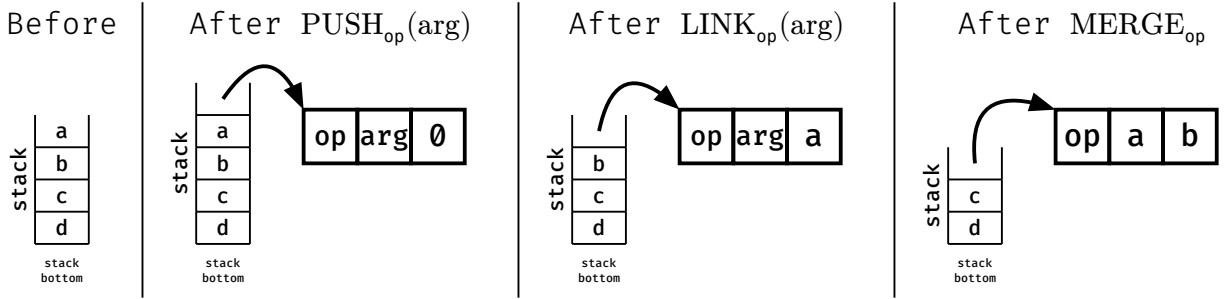


Fig. 4.9. Effect of each type of decoding instruction on the stack.

Decoding instructions are executed on a state machine of rib references with the goal of recreating the *rib graph* to be executed by the Ribbit Virtual Machine (RVM) interpreter. This section discusses the definition of these instructions and the next section discusses how they are encoded in the RIBN.

There are 4 categories of decoding instructions depending on how the stack of the decoder is affected: *PUSH*, *LINK*, *MERGE*, and *SHARE* as shown in Figure 4.9. *PUSH* adds a new rib onto the stack, *LINK* references the previous TOS inside a newly allocated TOS, and *MERGE* takes the first and second elements of the stack and forms a new rib with them. The *SHARE* instruction is discussed in Section 4.3.2.1.

In addition to their type, decoding instructions have two possible arguments: *op* and *arg*. These correspond to the *opcode* (a number between 0 and 5) and *operand* of the generated instruction rib, and are denoted as $INSTRUCTION_{op}(arg)$ in the rest of the text. To avoid arbitrarily limiting the number of symbols and global variables of the source program, the value of *arg* has no upper limit. Furthermore, as instruction ribs can take a reference to a symbol or a literal number as *operand*, *arg* can specify an index in the symbol table or a literal number depending on its type: *int* or *sym*, written as *int* *n* or *sym* *i*. For example, the instruction $PUSH_0(int\ 1)$ will push the rib $[0, 1, 0]$ onto the decoding stack, while $PUSH_0(sym\ 0)$ will push the rib $[0, '%%rib, 0]$, as *%%rib* is the symbol at the beginning of the symbol table.

The decoding instructions used by Ribbit are shown in Table 4.2. These are specialized versions of the general types of decoding instructions for the case of instruction rib, often containing a literal integer in their *op* field. For example, $LINK_2$ creates the **get** instruction, as 2 is the opcode for **get** in the RVM instructions. The only instruction that differs from the categories defined in Figure 4.9 is $MERGE_3$. Instead of including *y* directly in the second field multiple ribs are created: $[[arg, 0, y], 0, 1]$. This corresponds to the creation of

Table 4.2. Example of the decoding instructions, their type of argument and their effect on the stack.

Decoding instruction type	Argument type (<i>arg</i>)	RVM instruction generated	Effect on decoding stack state ($\langle \text{current} \rangle \rightarrow \langle \text{next} \rangle$)
$PUSH_0$	int or sym	jump	$stack... \rightarrow [0, arg, 0] stack...$
$LINK_0$	int or sym	call	$x stack... \rightarrow [0, arg, x] stack...$
$LINK_1$	int or sym	set	$x stack... \rightarrow [1, arg, x] stack...$
$LINK_2$	int or sym	get	$x stack... \rightarrow [2, arg, x] stack...$
$LINK_3$	int or sym	const	$x stack... \rightarrow [3, arg, x] stack...$
$MERGE_3$	int	const	$y x stack... \rightarrow [3, [[arg, 0, y], 0, 1], x] stack...$
$MERGE_4$	none	if	$y x stack... \rightarrow [4, y, x] stack...$
$SHARE$	int	none	$x stack... \rightarrow list\text{-}tail(x, arg) x stack...$

a global function (closure with no free variables) as the *env* field is 0. $MERGE_3$ also has an argument (*arg*) positioned as the arity number for the created procedure (see Section 4.2.4).

Because of the linked nature of the rib graph where all leaves (terminal nodes) are jump instructions, the rib graph is recreated by first pushing a jump instruction (with $PUSH_0$) and then linking following instructions in the reverse order of their execution by the RVM interpreter (with $LINK$ and $MERGE$). This explains the presence of $PUSH_0$ as the only $PUSH$ instruction used in Ribbit.

4.3.2.1. The $SHARE$ Instruction. The $SHARE$ instruction enables the decoding of a directed acyclic graph rather than a tree by allowing the *sharing* of previously created ribs. This is especially useful for the encoding of non-returning if statements, as encoding this control flow as a tree requires the duplication of the code after the *if* (its continuation). When non-returning *ifs* are used one after the other, this duplication can have an exponential effect on the size of the generated instructions, because each *if* multiplies by 2 the number of encoded instructions following each *if*. This was a limitation of the first version of Ribbit that is now resolved with the $SHARE$ instruction.

The $SHARE$ instruction works by extracting an already existing rib accessible from the top of stack and starting a new code sequence with it. The argument specifies which rib to access by providing the number of RVM instructions to skip starting from the TOS. For example, when the code graph of Figure 4.10 is converted to a RIBN a $SHARE$ decoding instruction with an argument of 1 is used after the false branch of the *if* has been constructed.

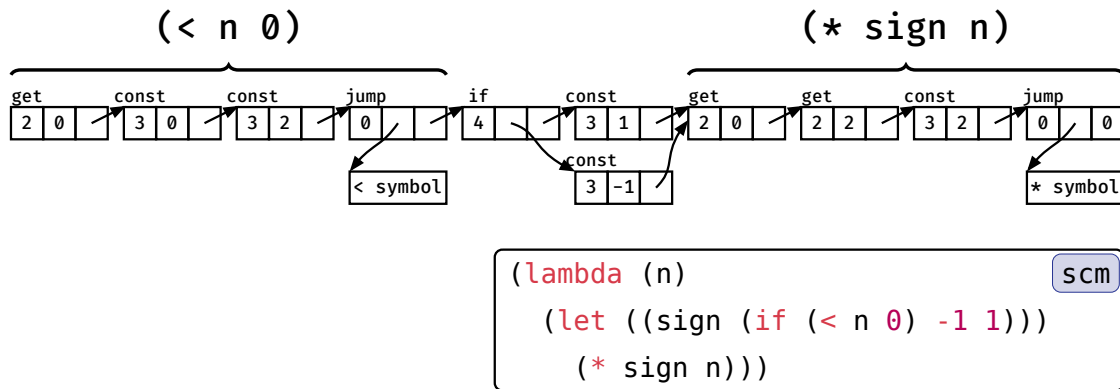


Fig. 4.10. Example of sharing between the two branches of an if statement, using the absolute value function.

Then the true branch is added and a *MERGE₄* decoding instruction is used to create the if.

To determine what part of the code graph has sharing, a hash-consing algorithm is used by the Ribbit Scheme Compiler to construct the code graph. Hash-consing can determine if two nodes, including all of their children, are equal. Equal code graph tails will automatically be shared in the constructed code graph. Other benefits also emerge from hash-consing such as the ability to optimize certain forms of duplication in the source code. For example, the two following expressions will compile to the same code graph: (if (< x y) (f x) (f y)) and (f (if (< x y) x y)).

4.3.3. Encoding of the RIBN

The RIBN is encoded literally in a program's source code and depending on how the target is executed, the size of the resulting program will be affected differently. For compiled targets, such as C or x86, the compiler usually converts literal byte-sized arrays into a series of bytes in memory, typically using one byte in the final executable for each number in the array. For interpreted targets, such as Python and JavaScript, the source program is used to calculate the size, meaning that a literal byte-sized array needs more than 1 byte to be encoded for elements exceeding 9 (as 10 requires 2 bytes to be written). For this reason,

```

(define input #(92 93 94 95 96)) // better for compiled targets
(define input "abcde")           // better for interpreted targets

```

Fig. 4.11. This example showcases the differences between embedding a program as a string and as a vector.

compiled targets encode the RIBN as an array literal and interpreted targets use a string literal as shown in Figure 4.11.

To be compact, the characters allowed in a literal string cannot contain the whole range of ASCII characters. This is because some characters are encoded in the string using two or more characters as they need to be escaped, such as `\0` for NULL or `\n` for newline respectively. Furthermore, as we do not enforce a particular file encoding for the source code, character codes exceeding 127 are not used as they have different representations in the file's memory, sometimes needing the use of more than one byte (such as in UTF-8). For these two reasons, characters encoded in strings are limited to 92 printable ASCII characters with space, " and \ removed:

```
!#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

In both cases, each element of the RIBN can be viewed as a number between 0 and 255 for a literal byte-sized vector and between 0 and 91 for strings. If we generalize, we can define this upper bound as the byte's base (bb), with $bb = 256$ for tables and $bb = 92$ for strings. This allows us to only think of the RIBN as a series of numbers with a particular byte base and not as any particular representation in the final program.

4.3.4. Encoding of the Decoding Instructions in the RIBN

We will now discuss how *decoding instructions* are encoded in the RIBN. As discussed in Section 4.3.3, the RIBN can be viewed as a series of codes capped by its byte base, reducing the problem to mapping *decoding instructions* to codes between 0 and byte base (bb), which usually has a value of $bb = 92$ or $bb = 256$.

The encoding of each decoding instruction contains its type (*PUSH*, *LINK*, *MERGE* and *SHARE*), its operand (number 0 to 5), its argument's type (*sym* or *int*), and an unbound positive number, the argument's value. The argument's value is encoded either with a single RIBN code, called the *short* form, or across multiple ones, called the *long* form. For each set of decoding instruction type, operand, argument's type, and form, a range of RIBN codes is assigned as exemplified by Table 4.3. The specific value within the range determines the value of the argument. For the *short* form, the argument's value will be the difference between the code and the lower boundary of the range. For the *long* form, the argument will utilize the VLQ encoding, with a starting value in the accumulator equal to the difference between the code and the lower boundary of the range. For instance, if the range of codes 50..55 is assigned for $PUSH_0$ with a *long* encoding, then the two RIBN codes 53 4 will represent the instruction $PUSH_0(int\ 3 \times bb/2 + 4)$. When the RIBN is encoded

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

λScheme

Fig. 4.12. An example of the recursive factorial function.

as a string with $bb = 92$, this generates $PUSH_0(int\ 142)$, thus pushing the `rib[0, 142, 0]` onto the decoding stack.

This encoding is very compact for code, as one byte can represent the type, operand, argument's type, and argument's value of a decoding instruction. For example, the function shown in Figure 4.12 uses 12 decoding instructions: $PUSH_0(sym\ *)$, $LINK_0(sym\ fact)$, $LINK_0(sym\ -)$, $LINK_3(int\ 1)$, $LINK_2(int\ 1)$, $LINK_2(int\ 0)$, $PUSH_0(sym\ id)$, $LINK_3(int\ 1)$, $MERGE_3$, $LINK_0(sym\ =)$, $LINK_3(int\ 0)$, and $LINK_2(int\ 0)$. When encoding this function using the ranges defined in Table 4.3, most decoding instructions fit in a single byte: small integers (0 and 1) fall within range, and most symbols will be assigned small indexes in the symbol table as they are frequently used in this particular program (*id*, *-*, *=*, *fact* and ***). This means that this function will most likely be encoded in 12 bytes or 96 bits. By comparison, this is less than the size used by a single vector register in modern Intel or AMD CPUs (AVX/XMM), typically using 128, 256 or 512 bits.

The ranges defined in Table 4.3 were determined by minimizing the RIBN size through a trial and error process on the source code of the REPL with a fixed byte base of 92 ($bb = 92$). However, the chosen ranges can be suboptimal for a given program, and its specialization is discussed in the next section.

4.3.5. Encoding Specialization

To ensure that *decoding instructions* using frequent arguments are encoded in the least bytes possible, the compiler specializes the ranges assigned to each decoding instruction to the source program. This problem can be seen as filling the first column of Table 4.3 with ranges that minimize the size of the RIBN .

Ribbit uses a greedy algorithm to optimize the ranges. The algorithm starts by assigning a range size of 1 for the *long* form and a range size of 0 for *short* forms for all sets of decoding instructions, argument type, and opcode. Then, the algorithm chooses the best range to increase by an integer n . To calculate the number of bytes saved by an increase of n , the occurrence of decoding instructions and their arguments are kept in memory, and the bytes saved are computed by counting their occurrences. The best range can be picked as the one

Table 4.3. Assignment of ranges for each decoding instruction for the original encoding used in Ribbit.

Range	Size	Decoding instruction	Argument	Form	RVM instruction
0-19	20	$PUSH_0$	sym	short	jump
20-20	1	$PUSH_0$	int	long	jump
21-22	2	$PUSH_0$	sym	long	jump
23-52	30	$LINK_0$	sym	short	call
53-53	1	$LINK_0$	int	long	call
54-55	2	$LINK_0$	sym	long	call
56-56	1	$LINK_1$	int	long	set
57-59	2	$LINK_1$	sym	long	set
59-68	10	$LINK_2$	int	short	get
69-69	1	$LINK_2$	int	long	get
70-71	2	$LINK_2$	sym	long	get
72-82	11	$LINK_3$	int	short	const
83-83	1	$LINK_3$	int	long	const
84-85	2	$LINK_3$	sym	long	const
86-89	4	$MERGE_3$	int	short	const (procedures)
90-90	1	$MERGE_3$	int	long	const (procedures)
91-91	1	$MERGE_4$	-	-	if

that has the best ratio between the number of bytes saved and n . Finally, this process is repeated until the sum of all ranges equals the byte base.

As an example, suppose that a source program often refers to the number 4. Then, the decoding instruction $LINK_3(int\ 4)$ will appear frequently in the program's RIBN. Consequently, the number of bytes saved by increasing the range size of $LINK_3$ with *int* type argument and *short* form from 0 to 5 will be large and thus picked (note the 5 because 0 is included in the range).

These specialized ranges are then embedded inside the algorithm that decodes the program. To embed these ranges, Ribbit uses Virtual Machine Tailoring and the process is explained in Section 5.6.1.

4.3.6. Compressing the RIBN

Typically, when evaluating the efficiency of compression algorithms, the size of the decompression algorithm is not considered. However, in the case of Ribbit, the decompression algorithm is included alongside the RIBN inside a generated RVM, meaning that both the decompression algorithm and its efficiency at compressing data must be taken into account. For this reason, fancier algorithms like Zip or Bzip2 are not considered as their implementation will most likely take more space than the space saved by their compression of the RIBN. This problem is similar to the compression problems demoscene artists have[42], as their entire binary, including decompression algorithms, must be under a specified number of bytes usually 1KB, 4KB or 64KB.

4.3.6.1. LZSS. LZSS is a general-purpose compression algorithm heavily inspired by LZ77 that replaces recurring slices of text with back-pointers to previous occurrences. An LZSS decompressor is particularly compact and has been implemented in less than 50 lines of x86 assembly code. It offers effective compression of the RIBN for the amount of space taken.

The LZSS algorithm works on units of information that we will call bytes since in practice they correspond to 8 bits even though in theory it could be different. The decompressor takes a stream of bytes and outputs a sequence of RIBN codes.

One of the key issues in applying the LZSS algorithm to compress the RIBN is the encoding of back-pointers, which are composed of an offset and a size. The offset is the backward distance in number of RIBN codes to the end of the section that is repeated and the size is the length of the section.

To achieve good compression rates it is important to encode back-pointers in as few bytes as possible. We chose to always use two bytes. When the decompressor encounters a byte whose value is lower than the RIBN base (rb), it represents a RIBN code that is output as is. Otherwise the byte is in the compression-range and it is the first of the two bytes that encode a back-pointer BP. The two bytes are combined with the formula $BP = (byte1 - rb) \times bb + byte2$. The size and offset are then extracted using the size base (sb):

$$size = BP \bmod sb + 3 \tag{4.3.1}$$

$$offset = BP \div sb \tag{4.3.2}$$

where $\div$ and $\bmod$ are the integer division and modulo operators. There is no gain in using back-pointers to encode repeated sequences of size 1 or 2, so the minimum size is 3 and the maximum size is $sb + 2$. The maximum offset is $((bb - rb + 1) \times bb - 1) \div sb$.

The value chosen for size base determines the balance between the range of sizes and the range of offsets that can be encoded by a 2 byte back-pointer. Given that the optimal size base depends on the source program, it is the Ribbit Scheme Compiler (RSC) that determines the value. RSC will iterate over a small reasonable range of values for size base (7 to 13) and pick the one that produces the best compression. When using LZSS, a RIBN base of 186 is used, as this gives the best compression for the REPL.

4.4. The Standard Library

Ribbit implements different libraries with particular emphasis on generated code size and portability. This section explores the various libraries available, the techniques to implement I/O primitives in a portable fashion, and techniques to write compact code.

4.4.1. A Hierarchy of Library Implementations

Ribbit implements different standard libraries offering a choice between functionality and code size. These libraries are structured as a hierarchy, with featureless libraries at the bottom and fully featured libraries at the top. This allows a programmer to pick the right tradeoff between the features available inside the library and the operational constraints imposed by its implementation. This aligns with the objective of *incremental portability* (Section 3.7).

The `min` and `max` libraries are the two most barebones libraries available. They implement usual Scheme functions with the possibility of having a read-eval-print loop (REPL). As their name suggests, the `min` library generates smaller code than the `max` library, at the cost of having less functionality. For example, the `min` library supports fewer expressions inside the `eval` function than the `max` library. The first paper on Ribbit [55] demonstrated that the `max` library can generate a minimal Scheme REPL contained inside a 4 KB binary.

The R4RS library is a more recent implementation of a standard library and implements all necessary functions defined by the R⁴RS standard. This library adds file input-output primitives, the `apply` primitive and variadic functions for arithmetic operations, trading code size for more functionality.

The Revised⁴ Report on Scheme (R4RS) library can be configured to enable type-checks. Adding type checking further increases the size of the generated program as runtime checks and error reporting routines are added inside the generated program. This also comes with the benefit of more safety as arguments passed to the standard functions check their type before executing.

4.4.2. Implementation of Portable File I/O Primitives

The way of interfacing with I/O varies greatly between languages. To solve this problem, Ribbit reduces the responsibilities of each Ribbit Virtual Machine (RVM) to a minimum by implementing most of the logic in Scheme code directly. This allows Ribbit to present a unified API that can adapt to all RVMs while complying with the behaviors expected by R4RS.

Ribbit separates input-port and output-port into two distinct data types, as required by R4RS. An input-port is a rib with this layout: `[fd, peeked-char/open?, 8]`. The `fd` field is reserved by the RVM for implementation dependent file descriptors. Each host language has its own way of communicating with the file system, and this object bridges the gap between the host language and the RVM. For example, the x86 assembly implementation uses an integer representing the Linux file descriptor, while the Python implementation keeps a reference to a Python File object.

The `peeked-char/open?` field has two purposes. Its first role is for caching the last read character after a peek. It is used in the implementation of `peek-char` and `read-char`. It is needed because the RVM's file interface does not necessarily conform to the R4RS standard. This field is the empty list by default, meaning there is no character peeked. The second role of the field is to indicate if the port is open or closed. To do so, the field is set to `#f` when the port is closed and checking the state of a port becomes a simple test for whether it is open. It is necessary to provide this information as R4RS mandates that a port may be closed any number of times without causing an error, a guarantee not shared by all languages. For example, NodeJS will throw an exception while attempting to close an already closed port.

An output-port is a rib with this layout: `[fd, open?, 9]`. The `fd` field and the `open?` field have the same meaning as those fields in an input-port. To support I/O the following primitive procedures must be defined by an RVM: `%%stdin-fd`, `%%stdout-fd`, `%%get-fd-input-file`, `%%get-fd-output-file`, `%%read-char-fd`, `%%write-char-fd`, `%%close-input-fd` and `%%close-output-fd`.

`%%stdin-fd` and `%%stdout-fd` return the `fd` host-dependent value used in the standard input and output port. `%%get-fd-input-file` and `%%get-fd-output-file` take a filename and return the `fd` host-dependent value used in the input and output port for that file. `%%read-char-fd` takes a `fd` and reads a character code (as an integer) from the file, while `%%write-char-fd` takes a character code (as an integer) and a `fd` and writes the character to the file. Finally, `%%close-input-fd` and `%%close-output-fd` take a `fd` and close the corresponding port. All those procedures either take or return the implementation-dependent object `fd` that is used to retrieve or write data to the file system using the host language. Scheme procedures are defined on top of these primitives, and they take care of caching the

peeked character, checking if the port is open, closing the port when necessary, returning the end-of-file object, and converting the character code (integer) read into a Scheme character.

4.4.3. Writing the R4RS Library Compactly

The Ribbit implementation of the R4RS library is optimized for size at the cost of execution speed. This approach can be seen in the implementation of `+`, `*`, and `-`, which are all defined using the fold procedure, as a call to fold takes less space than an explicit loop to compute the sum or product. The fold procedure, even if absent from the R4RS standard and therefore not required, is used often enough to make up for the space its implementation needs. The pattern of using higher order procedures to generalize a behavior is used frequently in the R4RS implementation, as it often avoids code repetition.

Another trick is to define procedures in terms of other procedures. For example, `(char>? c1 c2)` is defined as `(char<? c2 c1)` instead of the faster, but larger `(> (char->integer c1) (char->integer c2))`.

Another common technique is to define a few internal procedures of a general nature and to call them in many different places. This makes the AOT compiler rank those procedures at low indexes in the symbol-table, reducing the cost of accessing them to, in the best case, as little as one byte.

4.5. Final Thoughts on Ribbit

Ribbit is a complete system that executes code on a unique virtual machine to implement the Scheme programming language. It provides multiple benefits regarding portability: a small on-disk size and a virtual machine algorithm based on a unique memory representation, the *rib*, that can be easily implemented in different programming languages. Virtual Machine Tailoring, presented in the next section, enables Ribbit to become even more portable through the specialization of virtual machines.

Chapter 5

Virtual Machine Tailoring

Virtual Machine Tailoring (VMT) is a novel compiler technique that enables portability by removing the requirement of a fixed standard bytecode between the compiler and the virtual machine (VM). Instead, the compiler generates a specialized bytecode for the source program and a *tailored* VM that can interpret it. This allows the specialization of every component of the VM to the source program, such as the primitives exposed to the interpreted language, the call protocol, the first-class value types and the memory management model.

Template VMs are used to generate *tailored* VMs. These templates contain language-agnostic annotations specifying the meaning of its source code to the compiler. This allows the compiler to prune unused code by the interpreted program, such as primitives that are never called, and to insert additional extensions that might be required by the specific application.

To provide concrete examples, two extensions using only VMT annotations are presented as case studies: the specialization of the bytecode decompression algorithm to the source program and the implementation of a Foreign Function Interface providing runtime primitive definitions.

This chapter opens with an overview of VMT in Section 5.2. Then, Section 5.3 introduces the concept of *features* and *annotations*, while Section 5.4 details the specific annotations used in Ribbit. Section 5.5 explains the compilation process of a VMT compiler, and Section 5.6 demonstrates the usefulness of VMT through case studies. The chapter concludes with Section 5.7 relating VMT to portability, for example, deriving the implication of losing bytecode standardization.

5.1. Credit

The presentation of Virtual Machine Tailoring as a compiler technique is an original contribution of the author. The workshop paper *A Compact and Extensible Portable Scheme VM* [34] (from Oest O’Leary and Feeley) presents preliminary results of this technique without framing it as Virtual Machine Tailoring.

5.2. An Overview of Virtual Machine Tailoring

Tailoring consists in the modification of the generated VM to the specific needs of the source program or environment. For example, *tailoring* can: extend the dispatch table with primitives defined in the source program, change the call protocol to support multiple-value return only when multiple-values are used, or specify a new VM native type such as floating point numbers.

This section gives an overview of Virtual Machine Tailoring by first giving an example in Section 5.2.1, explaining the possibilities enabled by this new technique in Section 5.2.2 and motivating it in Section 5.2.2. Note that this section skips over the technical details, which can be found in later sections.

5.2.1. A Simple Example

To understand what Virtual Machine Tailoring (VMT) is, we first define a simple language and VM that will be used throughout this chapter. The language, called *AddSub*, consists of arithmetic expressions with only + and - as operations written as s-expressions. For example, `(- (+ 12 5) 10)` yields 7. The *AddSub* language can be implemented with a stack-based VM with two operations: pushing a number and calling the + or - primitives that add or subtract the top two elements on the stack. For example, operations to compute `(- (+ 12 5) 10)` are `PUSH 10, PUSH 5, PUSH 12, CALL +, CALL -`.

A VM in Python implementing the *AddSub* program `(- (+ 12 5) 10)` is shown in Figure 5.1. This VM represents the program inside the `program` variable as a list that either contains numbers or strings. The VM’s interpreter loop is a `for` loop that traverses all elements of the list and either pushes the value when it is a number or computes the arithmetic operation + or - when it is a string.

To compile another program, a VMT compiler takes the VM and substitutes the value of the `program` variable with the new program. For example, compiling the *AddSub* program `(- (+ 20 5) 7)` yields the same result as Figure 5.1, except that the first line is: `program = [7, 5, 20, "+", "-"]`. This method to generate new programs may appear as a complex form of bytecode compilation, with the whole VM generated alongside the embedded bytecode. This

```

1 program=[10, 5, 12, "+", "-"]
2 stack=[]
3 for token in program: # interpreter loop
4     if isinstance(token, str): # + and - operations
5         a = stack.pop()
6         b = stack.pop()
7         if token == "+":
8             stack.append(a + b)
9         elif token == "-":
10            stack.append(a - b)
11     else: # number
12         stack.append(token)
13 print("Result:", stack[0]) # print the result

```

 Python

Fig. 5.1. Example of a virtual machine implementing the *AddSub* language in Python.

key difference, however, allows for changes to other parts of the VM, enabling more powerful forms of tailoring that cannot easily be replicated with classical bytecode compilation. These forms of tailoring are discussed in the next section.

5.2.2. The Different Forms of Tailoring

A first form of tailoring is the removal of unused pieces of code from the virtual machine. As an example, when compiling a program without any subtractions, the code to perform subtractions inside the VM is no longer needed. In Figure 5.1, this corresponds to removing lines 9 and 10. This optimization can be performed by analyzing the primitives used by the source program and removing code related to unused primitives, provided that the compiler knows where they are. In addition to primitives, the compiler can detect unused language features or data types, for example, removing code for managing variadic functions or floating point arithmetic when they are not used in the source program. This enables the generation of a minimal VM for any given source program.

Some parts of the VM can also be removed according to the environment in which they execute. Even under the same programming language, the interface to communicate with the underlying operating system or external system can be different. To have a complete implementation, code supporting all interfaces must be included, which can bloat the VM implementation. *Tailoring* allows the removal of these unused pieces of code only when a more specific environment is known. For example, when generating a VM in JavaScript using Ribbit, the programmer can specify whether the VM should run in the browser, Node.js or both, and a minimal VM will be generated for the specific use case.

A second form of tailoring is the extension of the VM with additional primitives. As the VM is generated *at compile time*, new primitives can be instructed by the programmer with *compile-time* macros. For example, the *AddSub* language from Section 5.2.1 can be extended with a compile time instruction containing the code in Python of a new primitive. To implement this primitive natively, the compiler can add a new `elif` clause to the primitive dispatch mechanism in Figure 5.1 (after line 11). This allows the calling of native code from the source language, similar to Java’s Native Interface (JNI) or Microsoft’s P/Invoke, but without any complex API or ABI. The VM is simply generated with the additional primitive and without overhead.

A third form of tailoring is the modification of the VM execution model. For example, Ribbit can:

- Change the call protocol to support variadics only when they are used in the Scheme source program or library.
- Specialize the bytecode’s decoding algorithm according to the usage of opcodes and operands in the bytecode, as discussed in Section 5.6.1.
- Implement a new *foreign* native type inside virtual machines only when a Foreign Function Interface is used inside the source code as discussed in Section 5.6.2.
- Implement a Foreign Function Interface *only using VMT extensions* as discussed in Section 5.6.2.

5.2.3. Motivation of Virtual Machine Tailoring

Compilers are typically based on one of two techniques that both have their limits regarding portability: bytecode or native compilation. Bytecode compilation eases the implementation of new targets by writing only a VM. However, forcing the representation of programs into a standard bytecode limits portability. Indeed, constrained environments might not be able to implement the standard bytecode, and some program features might not be representable in the standard bytecode. In contrast, native compilation retains full control of the generated code by skipping the bytecode representation entirely. However, this comes at the cost of reimplementing the backend for each target, which is typically a time-consuming task which hurts portability.

Virtual Machine Tailoring (VMT) is a third technique that solves this seemingly irreconcilable tension between ease of implementation and control over the generated code. Through the *generation of the VM*, new targets can still be added by only writing a template VM while keeping the full control over the generated artifact. Overall, VMT can be seen as a hybrid technique between bytecode and native compilation that simplifies the writing of

compilers for portability. Additional motivations are given in Section 5.7 once the technical details are explained.

5.3. Features and Annotations

To *tailor* any VM, a general mechanism must be introduced that can precisely shape the generated VM. This mechanism consists of *annotations* inside template VMs and *features*.

Features are compile-time variables that control the generation of VMs through annotations. Inside template VMs, *annotations* associate the values of features with parts of the VM. This allows features to control which parts of the template VM are removed, kept, or extended in the generated VM. Features also control the inclusion of primitives: when a feature with the same name as a primitive is enabled, the primitive is included in the generated VM. Here are all the ways features can be enabled or disabled:

- **By the programmer using the command line options.** In Ribbit, this is done with the command line options `-f+ feature-to-enable` or `-f- feature-to-disable`.
- **With dependencies among the features.** Features can define dependencies with other features with the `(use ...)` clause. A fix-point algorithm is used to determine the set of features to enable or disable.
- **By the compiler.** If the compiler detects certain optimizations, it can enable or disable features. For example, in Ribbit, the `rest-param` feature controls whether the call protocol collects excess arguments passed to a function in a list. When no variadic functions are detected, this feature can be disabled.
- **In the source program.** Compile-time directives within the source program enable or disable features. For example, defining a primitive within the code activates the feature with the same name as the primitive when it is used.

Note that the value of features is not limited to a binary value. They can contain any arbitrary value, and their values can be included within the generated VM as long as it has a textual representation understandable by the host language.

5.4. Tailoring Virtual Machines with Annotations

Annotations are written inside template VMs, embedded inside host language comments. Then, the generation of VMs can be *tailored* according to the values of features inside

```
#include <stdio.h> // @@(feature stdio)@@
```



Fig. 5.2. Example of an inline feature annotation.

```

1  switch (prim_index) {
2  // @@(primitives (gen "case " index ":" body)
...
23  case 19: // @@(primitive (##putchar c) (use stdio)
24      putchar(NUM(tos())); break; // print top of stack
25  // )@@
26  case 20: // @@(primitive (##exit n)
27  {
28      vm_exit(NUM(tos())); break;
29  } // )@@
30  // )@@
31  }

```

Fig. 5.3. Example of multi-line and nested **primitives** (plural) and **primitive** (singular) annotations.

the template. In this section, the syntax and semantics of annotations used in Ribbit are discussed.

5.4.1. Syntax and Scope of Annotations

Annotations have a syntax similar to s-expressions, but start and end with `@@(` and `)@@` instead of a single parenthesis character. They are composed of a name and zero or more arguments separated by spaces. Figure 5.2 showcases an annotation with the name **feature** and the argument **stdio**.

Annotations refer to some part of the host code, called the *scope* of the annotation. The *scope* depends on where the annotation starts and ends. If the annotation starts and ends on the same line, the scope of the annotation is that line only. For example, the scope of the annotation in Figure 5.2 is the line where `#include <stdio.h>` resides in the Ribbit Virtual Machine (RVM) written in C.

If the annotation spans multiple lines, its scope begins on the line *after* the annotation starts and ends on the same line as the annotation ends. In Figure 5.3, the scope of the `%%putchar` primitive annotation is from line 24 to line 25 (inclusive) and the scope of the `%%exit` primitive annotation is from line 27 to line 29 (inclusive).

Annotations can also be nested. For instance, the **primitives** (plural) annotation includes multiple **primitive** (singular) annotations in Figure 5.3. This allows some lines to be part of the scope of multiple different annotations. In Figure 5.3, the line 24 is part of the scope of the **primitives** (plural) annotation and the `%%putchar` primitive annotation. This makes sense as this line implements the primitive which is part of the primitive table.

```
(* @@(replace "..." (encode 92) @@ *)
let input = "..."
(* )@@ *)
```

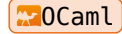


Fig. 5.4. Example of the use of the premature halting parsing token @@ for multi-line comments in OCaml.

5.4.2. Annotations for Any Language

As the goal of Virtual Machine Tailoring (VMT) is portability, annotations need to be insertable in as many programming languages as possible. We surveyed 25 programming languages in Appendix A to ensure the syntax and semantics of comments allowed the use of our annotation syntax.

In the surveyed languages, most allowed either inline or multi-line comments. Inline comments work well with our annotations and are the most prominent form of comments. However, multi-line comments can cause trouble when parsing annotations as they need an ending token at the end of the annotation line, and this token can be wrongly perceived as arguments to our annotation. For this reason, a special token @@ can be used in annotations to prematurely end the parsing of the annotation. This allows us to support languages with only multi-line comments such as OCaml, Pascal, or C89, as shown in Figure 5.4.

5.4.3. Replace Annotation

Replace annotations serve a dual purpose. They are used to include the encoded program inside the VM and to substitute any other compile-time values from features. The syntax of the replace annotation is: @@(replace <pattern> <replace-code>)@@. Where:

- <replace-code> is an s-expression that will be evaluated at compile time.
- <pattern> is a string that replaces the result of the evaluation of <replace-code>.

Variables allowed inside <replace-code> are features and encoding functions. Encoding functions are used to generate the encoded program. In Ribbit, allowed functions are found in

```
// @@(replace "{0}" (encode-as-bytes 256 "{" " ", " }"))
unsigned char compressed_input[] = {0}; // RVM code that prints HELLO!
// )@@

unsigned char input[00]; // @@(replace "00" compression/ribn-size)@@
```



Fig. 5.5. Example of two replace annotations. The first one is inserting the program's encoding and the second compile-time information from features.

Table 5.1. Example of functions available inside the replace annotation in Ribbit.

Encoding function name	Purpose
(encode-as-bytes prefix sep suffix)	Encodes the program as a list of numbers starting with prefix , separated with sep and ending in suffix .
(encode-as-string bits)	Encodes the program inside of a string using the encoding default on either 92 or 256 bits.
(encode-writable-ascii chars)	Encodes a program as a string containing only writable ascii characters and leaving out all characters in the string chars .

Table 5.1. These encoding functions allow for host-specific configurations through arguments, for example, specifying the syntax for generating a literal table. For instance, in Ribbit, a literal C table can be encoded using (encode-as-bytes 92 "{" " ", " "}"), as inline tables are written as `char[] input = {45,92, ... }`.

The replace annotation can also be used to include the value of any features. For example, the C Ribbit Virtual Machine (RVM) performs compression of the encoded program and would normally need a resizable heap-allocated array to store the result of the compression, as its size is unknown. Figure 5.5 shows how the replace annotation can be used to insert the size of the decompressed program, a value known at compile-time, avoiding the use of a dynamic array.

5.4.4. Primitive Annotations

The primitive annotations come in two forms: **primitives** (plural) indicates where the primitive dispatching code resides and **primitive** (singular) indicates where each primitive is. This is used by the compiler to reorder, remove and add new primitives, for example, with compiler directives inside the source code.

5.4.4.1. Template Generation. When the compiler generates the primitive table, specific information about the host language must be known, such as how to structure each primitive. Figure 5.3 showcases the particular form needed for the C target. In this case, primitives are implemented using a switch statement, needing an explicit index to branch to the right primitive. This extra information is given to the compiler through the **gen** clause inside the **primitives** (plural) annotation, taking as arguments a *template* that will be used by the compiler to generate each singular primitive.

Templates can either contain literal strings or variables from Table 5.2 and will be concatenated to form the final generated primitive. As an example, Figure 5.3 defines the

Table 5.2. All variables that can be referred inside the *template* of a `primitives` annotation.

Variable	Content	Example from <code>%%putchar</code>
body	Code within the scope of the annotation	<code>putchar(NUM(tos())); break;</code>
head	Code on the starting line of the annotation	<code>case 19:</code>
index	The index of the primitive	Determined at compile time.

template as `"case " index ":" body`. When generating the `%%putchar` primitive with the index of 7, the concatenated result yields `case 7:putchar(NUM(tos())); break;`.

To generate the entire primitive table, each singular primitive is generated by the compiler using the template. Then the result is concatenated together and replaces the `primitives` (plural) annotation.

5.4.4.2. Adding New Primitives from the Source Code. In Ribbit, new primitives are added in Scheme with the following directive: `(define-primitive <signature> <primitive-content>)`, where:

- `<signature>` is the name and arguments required to call the *primitive procedure* assigned to the primitive.
- `<primitive-content>` is the string representation of the host code to be executed when the primitive is called.

Within the compiler, when a new primitive is defined, the compiler first assigns an index to the primitive. Then, the compiler creates a *primitive procedure* (Section 4.2.4.1) with the corresponding index and assigns it to the symbol corresponding to the name of the primitive. Finally, the compiler generates the virtual machine (VM) containing the new primitive inside the dispatch table using the template defined by the `gen` clause of the `primitives` (plural) annotation.

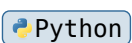
To give a concrete example, a primitive that squares a number in python can be defined as such: `(define-primitive (square x) "lambda: push(pop())**2")`. If the index 3 is assigned to the primitive, the compiler generates the following code in place of the primitive's definition: `(define square (rib 3 0 1))`. This corresponds to a *primitive procedure* calling the primitive at index 3. Finally, when generating the VM, the compiler adds at the index 3 of the primitive table the following code: `lambda: push(pop())**2`.

This mechanism allows for a foreign function interface (FFI) to the target language without any complex API or overhead. Leveraging primitive annotations to create a two-way FFI is later discussed in Section 5.6.2.

5.4.4.3. Adapting a Virtual Machine to Primitive Annotations. The primitive annotations require the code of primitives to be textually next to each other, line-by-line and can require modifications of the VM structure. To showcase the needed adjustments, primitive annotations have been added to the first example presented in this chapter (Figure 5.1) resulting in Figure 5.6.

The first modification is to refer to primitives by their number instead of their name, as is usually the case within a bytecode. This is not strictly needed, but it would require modifications of the variables allowed in the template to include the name of the primitive (see Table 5.2). In Figure 5.6, the bytecode was changed to a string where `pN` signifies a call to the primitive with index `N`.

The second modification is the dispatch mechanism. Primitives need to be grouped textually next to each other, and this can be achieved in Python by using a list containing lambda expressions for each primitive as shown in Figure 5.6. A switch statement or an if/then/else statement could have been used as well.



```

program="10 5 12 p0 p1" # @@(replace "10 5 12 p0 p1" (encode-as-
string))@@
stack=[]
push = lambda x: stack.append(x)
pop = lambda: stack.pop()
primitives = [
# @@(primitives (gen body)
    lambda: push(pop() + pop()), # @@(primitive (+ a b))@@
    lambda: push(pop() - pop()), # @@(primitive (- a b))@@
# )@@
]

for inst in program.split(): # Interpreter loop, executes all instructions
    if inst[0] == 'p': # Primitive call
        prim_no = int(inst[1:])
        primitives[prim_no]()
    else: # Push operation
        push(int(inst))

print("Result:", pop()) # @@(feature print-final-result)@@

```

Fig. 5.6. Example of the example virtual machine from Figure 5.1 annotated with primitive and primitives annotations.

5.4.5. Feature and Location Annotations

Feature annotations are used to control the inclusion of various parts of the VM depending on the value of *features*. They can be defined within the VM's source code with feature annotations or within the program's source code with macros.

5.4.5.1. Feature Annotations. Inside the VM, feature annotations have the following syntax: `@@(feature <cond>)@@`. Where `<cond>` is an s-expression evaluated at compile time containing *features* composed with the following boolean operations: `and`, `or` and `not`.

When the condition expression `<cond>` evaluates to `#t` at compile time, the code within the scope of the annotation is included inside the VM. Note that a fix-point algorithm is required to compute the value of features, as the inclusion of code can itself activate other features. The liveness expansion phase is further discussed in Section 5.5.1.

Figure 5.2 showcases an example of an inline feature annotation. A more complex multi-line feature can be found in Figure 5.7 from the Ribbit x86 assembly host. This condition includes the code from `prim_close_fd` only if either the primitive `%%close-input-fd` or `%%close-output-fd` is activated. It turns out that the implementation in x86 of the closing of a file descriptor is the same for input and output files.

5.4.5.2. Location Annotation and define-feature Macro. Location annotations indicate places where additional code can be inserted by the compiler as shown in Figure 5.8. This is useful to extend the VM from the program's source code without having to change the VM's source code (except for adding the locations). The syntax of the location annotation is `@@(location <name>)@@` where `<name>` is a symbol identifying the name of the location.

```
prim_dispatch_table:
;; @@(primitives (gen body)
    dd  prim_close_fd ;; @@(primitive (%%close-input-fd fd))@@
    dd  prim_close_fd ;; @@(primitive (%%close-output-fd fd))@@
;; )@@
;; @@(feature (or %%close-input-fd %%close-output-fd)
prim_close_fd:
    shr  LAST_ARG, 1
    mov  ebx, SYS_CLOSE
    xchg eax, ebx
    CALL_KERNEL
    ret  ;; )@@
```

Fig. 5.7. Example of the use of a **feature** annotation referring to multiple lines and with a condition statement. The example is from the Assembly Ribbit Virtual Machine.

To insert code from the program's source code, features can be defined with the `define-feature` compiler directive having the following syntax: `(define-feature <cond> <decls>)`, where:

- `<cond>` is an s-expression evaluated at compile time containing *features*.
- `<decls>` is a binding list similar to `let` expressions that relates *locations* to *code*.

Similarly to feature annotations, the code defined inside `<decls>` will only be included when the condition expression `<cond>` resolves to `#t` at compile time. Figure 5.9 shows a `define-feature` for the template VM shown in Figure 5.8.

5.4.5.3. The `if-feature` and `use-feature` Compiler Directives. The `if-feature` compiler directive acts like a Scheme `if` statement, except that the condition is evaluated at compile time and contains *features* composed with the `and`, `or` and `not` boolean operations. This directive is used when Scheme code depends on the activation of certain features. For example, the `eval` function, implemented in Scheme, needs to implement function calls that can slightly change depending on the activation of certain features such as `rest-param`.

The `use-feature` compiler directive adds a dependency from the source code to features passed as arguments. This directive is subject to liveness analysis, meaning that its appearance within a function will only activate features when the function is used. This directive is useful when Scheme code depends on certain features to be active and to give control to the programmer on which feature is active within the source code.

```
let os = require('os')
// @@(location decl)@@
function run() { /*Decodes and executes the program*/ }
// @@(location start)@@
run()
// @@(location end)@@
```

JsJavaScript

Fig. 5.8. Example of location annotations inside the JavaScript virtual machine.

```
(define-feature rib_eater (use scm2host)
  ((decl "collected_ribs = []; eat_rib=(r) => collected_ribs.push(r);")
    (start "console.log('Rib eater is activated');")
    (end "console.log('Here are my eaten ribs : ', collected_ribs)")))
(define-primitive (eat rib) (use rib_eater)
  "()" => {eat_rib(pop()); return push(FALSE)},")
```

λScheme

Fig. 5.9. Example of `define-feature` and `define-primitive` compiler construct to define new features and primitives inside the JavaScript virtual machine shown in Figure 5.8.

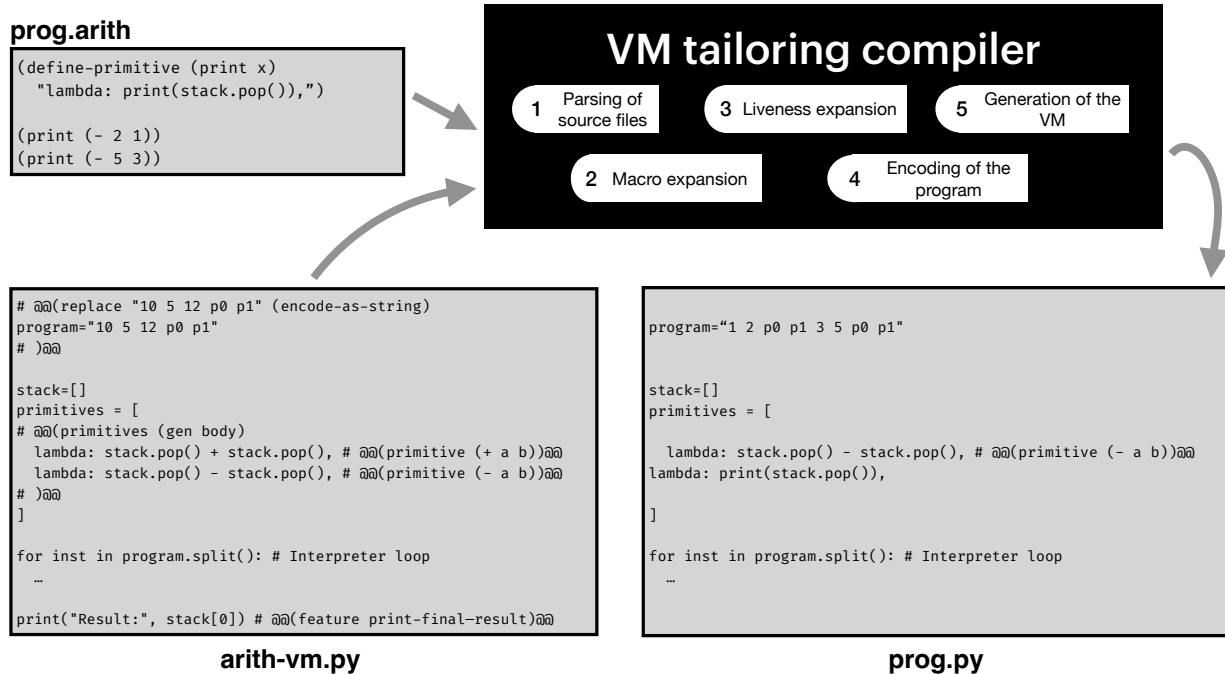


Fig. 5.10. Example of a compilation of `prog.arith` to `prog.py` using a Virtual Machine Tailoring (VMT) compiler. The VM used here is the one from Figure 5.6. As we can see inside `prog.py`, the new printing primitive is added from the host program and the feature `print-final-result` is disabled.

5.5. Compiling with a Virtual Machine Tailoring Compiler

The compilation process using a Virtual Machine Tailoring (VMT) compiler is shown in Figure 5.10. The compiler takes as input the program as well as a template VM and then:

1. Parses the source program and the annotated VM.
2. Expands any user-defined macros inside the source program.
3. Performs the liveness expansion phase of the source program as described in Section 5.5.1.
4. Encodes the program into an embeddable value within the VM. In Ribbit, this is done by encoding the rib graph into a string or a literal table, as explained in Section 4.3.
5. Evaluates annotations inside the template VM to generate a *tailored* VM with the embedded bytecode.

Note that optimizations and other compiler passes can still occur to optimize the encoding of the program, making it smaller, faster, or both.

The example in Figure 5.10 showcases an example where a new primitive for printing numbers is added to the VM from the source program (`prog.arith`) and where the `+` primitive is never used. The compiler only generates the `-` and `print` primitives with indexes 0 and 1, yielding the encoding: `"1 2 p0 p1 3 5 p0 p1"`. The feature `print-final-result` is also disabled, as the printing functionality has been moved to a primitive. The resulting program contains only the needed primitives and code as shown in Figure 5.10.

5.5.1. The Liveness Expansion Phase

The dependency of a feature on another feature can be indicated by the `use` clause in the `feature` and `primitive` annotations as well as in the `define-feature`, `define-primitive`, or `use-feature` compiler directives. As the activation of a feature can cause code to be considered live or features to be activated, a fix point algorithm that performs code expansion and a liveness analysis of both code and features must be implemented in a single phase.

To understand the recursive relationship between feature activation and code expansion, first consider that features can cause the inclusion of additional code through the usage of the `if-feature` compiler directive. This newly considered code can then activate additional features by:

- using the `use-feature` compiler macro.
- using a previously unused primitive directly or through the usage of Scheme functions referencing it.
- defining new primitives or features using other features through the `use` clause.
- using a previously unused language feature tracked by the compiler, such as rest parameters.

The recursive dependency of features and code is resolved by first considering all features as deactivated, except for the ones forced to be active by the programmer using the command line. Then, a liveness analysis is performed activating additional features and resolving `if-features` with the current set of activated features. While performing each pass, features can only go from a deactivated to an activated state. This ensures that the liveness expansion phase finishes as the set of activated features can only grow monotonically and there is a finite number of definable features. When no additional features are added after two consecutive passes, the liveness expansion phase is finished and the code is expanded one last time with the fixed set of activated features.

5.6. Case Studies of Virtual Machine Tailoring

Annotations and compiler directives, presented in Section 5.4, serve as the building blocks to *tailor* VMs to their source program. This section studies how the composition of these

building blocks enables specializations and optimizations that are hard to replicate with traditional techniques.

5.6.1. The Embedding of a Specialized Encoding

One of the objectives of Ribbit is to generate compact code. To achieve this goal, the compiler minimizes the size of the program's encoding according to the occurrences of instructions in the source program. More precisely, this is done by optimizing the size of ranges assigned to decoding instructions, as explained in Section 4.3.5. As this information is required by the decoding algorithm, these range sizes are embedded within its implementation using Virtual Machine Tailoring (VMT).

In practice, the `replace` annotation is used to embed a vector containing each range size. In Figure 5.11, the `list->host` function is used to convert at compile time the `encoding/optimal/start` feature containing a Scheme list of range sizes into a literal vector in the host language. As the syntax of literal vectors changes between programming languages, the `list->host` takes 3 additional arguments specifying the `prefix`, `separator`, and `suffix` of the vector to embed.

This paragraph explains how the argument value (`arg_value`) and the decoding instruction index (`range_index`) are computed from the embedded range vector (`ranges`) in Figure 5.11. At a high level, the goal is to find the value of `arg_value` and `range_index` such that the following two equations hold: $code = arg_value + \sum_{i=0}^{range_index-1} ranges[i]$ and $arg_value < ranges[range_index]$. Here, *code* is the value read from the program's

```
// @@(feature encoding/optimal
// @@(replace "[...]" (list->host encoding/optimal/start "[" "," " "))
int[] ranges = [...];
// )@@
while (1) {
    int code = get_code(); // Retrieve the RIBN code.
    int arg = code;        // Argument of the decoding instruction.
    int range_index = 0;   // Index in the range list.
    while (1) {
        if (arg < ranges[range_index]) break; // Stop when in range.
        arg_value += ranges[range_index]; range_index++;
    }
    // ... determine the decoding instruction from code, arg_value, and range_index
} // )@@
```

Fig. 5.11. Example of the decoding of the optimal encoding.

embedding and $\sum_{i=0}^{range_index-1} ranges[i]$ is the cumulative sum of ranges up to `range_index` - 1. To find this value, a loop goes over each range size and subtract the current range size from `code` until it is within the current range size. Finally, the decoding algorithm can deduce the combination of decoding instruction, opcode, argument type and form from the `range_index` as this information is constant across the compiler and Ribbit Virtual Machine (RVM) implementation.

As a particular RVM can support multiple encodings, to choose the right decoder implementation the feature `encoding/<encoding-name>` is used to include or exclude the code used to decode a particular encoding of the RIBN, where `<encoding-name>` is the name of the encoding. As an example, the code defined in Figure 5.11 will only be included when the feature `encoding/optimal` is activated.

This case study showcases how the replace and feature annotations with feature values can be used to embed specialized information within the bytecode decoding algorithm. This allows the specialization of a decoding algorithm to the source program, enabling precise optimization of the overall footprint of the generated program.

5.6.2. Improving the Foreign Function Interface

A Foreign Function Interface (FFI) is a language feature that allows calling code written in another language than the source language, for example, calling a `C` routine from `Python` code. Virtual Machine Tailoring (VMT) allows for a natural FFI to the host language with the `define-primitive` construct. However, `define-primitive` can be cumbersome to use as it requires manual manipulation of the stack to access arguments passed to the primitive and does not support the conversion of Scheme values to the host language values automatically.

```
// @@(feature scm2str
function scm2str(rib_str) {
  let result = "", rib_chars = rib_str[1] // Chars are in the 1st field of the rib
  while (rib_chars !== NIL) {
    result += String.fromCharCode(rib_chars[0])
    rib_chars = rib_chars[1]
  }
  return result
} // )@@
```

Fig. 5.12. Example of JavaScript code that converts a string represented in the Scheme to its representation in the host language.

This case study shows two ways to improve the FFI using VMT constructs. First, feature annotations and a naming convention demonstrate how conversion functions can be implemented to improve **define-primitive**. Second, a **host-eval** primitive for the JavaScript host in Ribbit is showcased, allowing two-way conversion of Scheme values to JavaScript values including functions, thus removing the need to manipulate the stack manually and enabling the definition of new primitives at runtime.

5.6.2.1. Manual Conversion of Values. To convert values from the host language to the target language, the VM needs to be improved with code that allows this conversion. To make sure this code is only included when conversions are performed, **feature** annotations are used around the conversion functions as shown in Figure 5.12. Then, primitives can declare dependence on these features using the **use** clause, as shown in Figure 5.13. Note that this conversion function can also be defined inside the source code using the **define-feature** compiler directive.

To avoid remembering each conversion function name, a convention is used for both the surrounding feature and the host function name performing the conversion:

- **scm2<target-type>** converts a Scheme object to the targeted type **<target-type>**. For example, **scm2str** converts a Scheme string to a host string in Figure 5.12.
- **<target-type>2scm** converts a host object to a Scheme value. For example, **list2scm** can be used to convert the command line arguments to a Scheme list.

With **define-primitive** and the **scm2str** feature, all R4RS primitives needed for file access can be implemented, reusing the underlying file access capabilities of the host language. The **scm2str** conversion function is used for transforming the Scheme filename to a host value that can be passed to host-specific reading functions inside the **%%get-fd-input-file** and **%%get-fd-output-file** primitives.

To give a more concrete example of the **define-primitive** syntax to define new primitives, Figure 5.13 showcases the creation of a **fetch** primitive essentially calling the JavaScript **fetch** function. This primitive performs the following action in order when no exceptions occur:

1. Convert its argument (the URL) into a JavaScript string using the **scm2str** conversion function.
2. Call the **fetch** function with the converted JavaScript string.
3. Call the **text** function on the response to get the website content.
4. Convert the website content into a Scheme string.
5. Push the Scheme string as the return value of the **fetch** primitive.
6. Return **true** to signal to the Ribbit Virtual Machine (RVM) to continue the execution

```

(define-primitive (fetch url)
  (use scm2str str2scm)
  "async () => {
    try{
      let url = scm2str(pop());
      let result = await fetch(url);
      let website_content = await result.text();
      return push( str2scm(website_content) );
    } catch (e){
      console.err(\"Error calling Fetch: \", e);
      return push(FALSE);
    }
  },")
(define ip-address (fetch "https://api.ipify.org"))
(if ip-address
  (begin (display "Your ip is: ") (display ip-address) (newline))
  (display "Cannot find your IP"))

```

Fig. 5.13. Example of a foreign function interface to the `fetch` function in JavaScript.

When an exception occurs, the false value is returned allowing Scheme to handle this case. The Scheme code then calls this primitive with the `https://api.ipify.org` website that returns the IP address of the requester that is then displayed in Scheme.

5.6.2.2. Defining New Primitives at Runtime. The previous example defined the `fetch` primitive at compile time using `define-primitive`. In Figure 5.14, the same primitive is defined at runtime using the `host-eval` primitive that can *automatically* convert a JavaScript function into a Scheme procedure. More precisely, the `fetch` Scheme procedure automatically converts its arguments to JavaScript values, calls the underlying `fetch` JavaScript function, and returns the result back to Scheme. The rest of this section discusses the implementation of `host-eval` using the constructs defined by VMT.

The addition of new primitives at runtime can be done in several ways. One way is to represent the primitive dispatch table as a dynamic array of host functions, where new functions can be appended. Note that this requires the support of first class functions in the host language. Another way is to extend the VM with a native type that can contain host values: foreign objects. The latter is used for the implementation of `host-eval` for the JavaScript host.

Foreign objects are represented as a rib containing a tag (third field) of 7 and a host object in their second field. For the JavaScript host, this allows the storing of JavaScript objects, such as website Requests or functions, directly within the rib graph. However, this

```

(define fetch (host-eval "fetch"))
(define .text (host-eval "(response) => response.text()"))

(let ((response (fetch "https://api.ipify.org")))
  (if response
    (let ((ip (.text response)))
      (display "Your ip is: ")
      (display ip))
    (display "Cannot find your IP")))

```

λScheme

Fig. 5.14. Example of the usage of `host-eval` to implement a Foreign Function Interface to the `fetch` JavaScript function.

breaks the invariant that the rib graph is only composed of numbers or ribs, which can cause trouble for serialization, memory managed implementations, or implementations that cannot express heterogenous arrays. For the purpose of this case study, we will ignore these problems and only concentrate on the JavaScript implementation.

Once foreign objects are implemented, a `host-call` primitive can be implemented that receives a foreign function and a list of Scheme arguments. This primitive converts the list of arguments to JavaScript, retrieves the JavaScript function from the foreign object, calls the JavaScript function with the arguments, converts back the result to Scheme, and returns it. This requires the implementation of most conversion functions between the two languages, with a fallback on foreign objects when the value cannot be expressed in Scheme.

Finally, the `host-eval` primitive can be implemented as a wrapper around the `eval` JavaScript function and the `host-call` primitive. When `host-eval` needs to return a function, a Scheme closure executing the `host-call` primitive with the foreign JavaScript function is returned instead. This is effectively the magic behind the `host-eval` implementation in JavaScript.

This case study showcases the flexibility of VMT. With only feature annotations and the addition of new primitives, a powerful Foreign Function Interface allowing the automatic conversion of host functions can be implemented. One of the strengths of VMT is that all the additional modifications to the VM are only included when the programmer uses the `host-eval` primitive, ensuring that a minimal VM can still be generated for more constrained environments.

5.7. Virtual Machine Tailoring for Portability

Now that Virtual Machine Tailoring (VMT) is properly introduced, it's time to discuss the portability implications of this new compiler technique. As explained in Section 3.6,

compilers for portability should limit the effort to translate programs to other environments. This involves multiple aspects of portability, such as the addition of new targets, the expressiveness of language features, and the operational constraints of environments.

VMT enables portability by attacking all of these aspects: facilitating the addition of new targets by writing template virtual machines, having a tight control over the language features, and providing tradeoff options for operational constraints. However, VMT has particularities that can be disadvantages depending on the context, such as lacking guarantees about the feature-completeness of targets and losing virtual code standardization. All of these aspects are discussed in the following subsections. Finally, this section concludes with VMT for incremental portability.

5.7.1. Keeping a Core and Minimal Virtual Machine

Every bytecode compiler and VM implementation has to make a design decision about the feature completeness of the standard virtual code, such as the number of primitives, instructions, and natively supported language types. This choice has a direct impact on portability as more featureful bytecodes lead to more complex implementations that are harder to port. At the same time, feature-less bytecodes limit the expressivity of the program they execute, hindering productivity. This creates a dilemma that seems irreconcilable, making either the VM harder to port or programs harder (or impossible) to write. In practice, traditional VMs tend to choose expressiveness over portability to support all language features, such as network interfaces or foreign functions.

Virtual Machine Tailoring (VMT) provides a solution to this problem by ensuring that a core VM can stay minimal. This is possible because of two properties of VMT: the removal of a standard virtual code and extensibility through annotations and features. First, having a specific program encoding for each program ensures that a bytecode standard does not force a particular feature-set upon the VM implementation, allowing it to be as simple as possible. Second, the extension capabilities of VMT ensure that a core and minimal VM does not limit the expressivity of programs, as it can be extended with additional features when needed.

5.7.2. Facilitating the Addition of New Targets

A core and minimal VM facilitates the addition of new targets because this VM *is* minimal and consequently effortless to port. This goes beyond supporting environments running under a specific programming language or hardware architecture. Having a minimal core ensures that unusual environments can be supported as well, such as programs running inside another program (embedding) or programs running with limited resources.

In practice, the Ribbit Virtual Machine has been ported 18 times to new programming languages by non-experts as detailed in Section 6.1 of the Evaluation section. Most ports were performed in 2 weeks of work, showcasing the speed of implementation even for non-experts.

5.7.2.1. Requirements for New Targets. Another aspect simplifying the addition of new targets is the limited requirements for new target languages. The only assumptions made for new targets are:

1. The VM can embed the program as a value within the language, for example, as literal strings or arrays.
2. The language allows the writing of a dispatch mechanism with code that is textually next to each other, line by line. This is used for referencing the primitives according to their numbers. For example, switch, if/else statements, or tables containing functions are all mechanisms that work.
3. The language’s syntax must support either inline or multi-line comments.
4. The language must support the writing of the logic of the VM within a single file.

These four requirements are present in the vast majority of languages, as shown by the 25 languages that Ribbit supports. However, these are just the requirements *for the use of VM tailoring*, not the implementation for the underlying VM.

While developing Ribbit, some programming languages were discovered as having arbitrary limits that made compiling to them harder. For example, Java limits the size of literal strings to 65534 bytes. This limits the size of the embeddable bytecode, thus the size of the Scheme program that can be compiled to Java. Hopefully, this limit can be circumvented quite easily, either by concatenating multiple literal strings or by reading the bytecode from another file.

5.7.3. The Implication of Losing Virtual Code Standardization

Traditionally, the *virtual* code executed by a *virtual* machine is standardized. Although this can be seen as enabling portability, this choice limits VM implementations as it requires the complete implementation of the standard, which limits the environments where they can be implemented. This is further discussed in Section 3.6.2 of Chapter 3.

Virtual Machine Tailoring removes the need for standardization, as the final generated program *is* the VM, allowing its specialization alongside the virtual code. Losing *virtual code* level portability may initially appear to introduce certain disadvantages:

1. VMs are often seen as relying on the standardized *virtual code* to enable the portability of programs. However, this relation is misleading: portability does not come from the *standardization of the virtual code*, but from the *VM implementation* that

translates the *virtual code* into specific target code. In other words, it does not matter that the virtual code is standardized for portability purposes, as long as there is a VM implementation that can execute this virtual code in the wanted environment, which is the case in VMT.

2. Including the VM inside each generated program can be seen as avoidable bloat. Although it is true that some parts of the VM could be reused between programs, thus reducing the overall footprint of multiple programs running on the same VM, specialized VMs remain the most space-efficient method to encode single programs. Indeed, encoding a single program efficiently requires reusing common instruction patterns under common abstractions, which can be done with a specialized bytecode as discussed in Section 3.6.1. In practice, Ribbit achieves very small footprints, on the orders of kilobytes. For example, Ribbit is the first compiler to generate a full R4RS Scheme interpreter within 6.5KB of memory (including the VM and virtual code), which is the world’s smallest known such implementation.

Losing *virtual code* standardization also has real disadvantages. As there is no standard, there is no possibility for the virtual code to become a standardized platform used by other compilers (such as Scala compiling to the Java’s bytecode). Furthermore, serializing and deserializing code between VMs is not directly possible, although we believe that synchronization of the virtual code of multiple VMs is possible, allowing them to communicate with the same virtual code.

When portability at the bytecode level is needed, Ribbit can always choose to standardize any particular bytecode, as it has full control over the generated VM. This essentially falls back to classical bytecode compilation, with its upsides and downsides.

5.7.4. Embeddability as Partial Solution to Inter-Process Communication

Communication between two processes requires orchestration with the executing host, adding complexity inside a language implementation when this functionality is provided by the language. This complexity comes from the creation of a communication channel, such as a shared memory region or operating system sockets, and the translation of values into ones that can be deserialized and understood by the other process. This is especially tricky when processes don’t share the same programming language or for values such as *functions*, for which serialization doesn’t always exist. This added complexity hurts portability, as the language has to implement environment-specific primitives to communicate with the executing host, often one of the operating systems macOS, Linux, or Windows. This also makes hosts more complex, harder to implement and thus less portable.

Virtual Machine Tailoring provides a simple solution to interprocess communication: *embedding* one program into another as a single process. Using this method, values already have a common representation within the enclosing program simplifying the translation of values and can directly communicate with the enclosing program’s language features.

This solution is already employed by certain programming language implementations, particularly interpreters such as CPython and CLua, which can be embedded inside C programs. However, these often rely on a single reference implementation and do not support embedding in other programming languages. In contrast, Ribbit already supports 25 languages into which Scheme programs can be embedded, allowing for inter-process communication between Scheme and any of these 25 languages. Furthermore, when communication with an unsupported language is required, the new language can be implemented with limited effort.

However, *embedding* is not a complete solution for inter-process communications. Programmers may lack control over the other running process, require a security boundary between processes, or need to communicate with several other processes.

5.7.5. Targets with Different Feature-Sets

In Virtual Machine Tailoring, targets do not need to support the same feature-set. This can be frustrating from the point of view of a programmer, as programs executing on top of one target might not execute on top of another one because of the lack of supported extensions. However, extending targets to support those features is a one-time effort and extensions are designed to be as effortless as possible.

In Ribbit, standard extensions have been proposed to ensure that a virtual machine having the same standard extensions can execute the same programs. For example, these extensions include variadic functions, file input-output primitives, the `apply` primitive, and the `command-line` primitive. At the time of writing, 7 out of the 25 targets have full R4RS support, meaning that they implement enough extensions to execute any program that respects the R4RS specification.

To evaluate the time needed to write new extensions, the author of this thesis extended the `go` target to support R4RS and the full bootstrap while keeping a written log of all the changes. In total, the 7 extensions needed for the full bootstrap were performed under 8 hours of full-time work. The full results can be found in Section 6.2.1.

5.7.6. Virtual Machine Tailoring for Incremental Portability

Incremental portability, defined in Section 3.7, ensures *high portability* by bounding the effort to support a new environment to the *porting distance*. In addition to providing portability, Virtual Machine Tailoring (VMT) enables *incremental portability* throughout its design which is discussed in each subsection.

5.7.6.1. Bounding the Effort to Add New Targets. When porting a program to a new programming language, VMT allows for the splitting of the efforts into distinct steps: the implementation of a core VM in the new language, and the development of each extension used by the program. This aligns with *incremental portability* as the core VM and each extension are implemented step-by-step with limited effort.

Drawing from our experience with Ribbit, splitting the porting process into clear milestones, such as the core VM, limits the overall effort required to add new targets. These milestones can be tested independently and become predictable to implement with practice. Moreover, when milestones are sufficiently constrained, they can be executed and understood by non-experts in a short amount of time, as shown by the implementation of core VMs in two weeks by third-year undergraduate students (see Section 6.1).

From a more theoretical point of view, the implementation of modular extensions also ensures that the porting effort stays focused on *the program* instead of *the language*. In other systems, new targets have to be complete before they are considered mature, which typically requires significant work. This is especially true for traditional VMs, as a standard bytecode forces a feature set that might not be required by the program, such as variadic functions or floating-point numbers. With VMT, extensions can be granular and composed together to support a specific program, ensuring minimal porting effort for minimal programs.

5.7.6.2. Trading Space and Portability for Speed. When a program requires faster execution speeds, VMT can move slow routines written in the source language into faster routines written directly in the target language. This is done by writing *primitives* specialized for the program, trading space inside the generated VM and a dependency on a host-dependent primitive for speed. Writing these primitives requires effort from the programmer, but each primitive is added *incrementally*, steadily increasing execution speed to meet the requirements of the program.

In the case of Ribbit, this technique can be very effective for targeting size-constrained environments. By default, Ribbit generates programs in their most compact form, likely allowing for their execution in a size-constrained environment. Then, some of the remaining space can be traded for more speed when execution time is too prohibitive. This contrasts with other compilers that optimize for speed first, making it harder to trade speed for space

afterwards. A classic example of trading space and portability for speed is the implementation of native vectors, as discussed in Section 4.2.6.

5.7.6.3. Controlling Language Features. As discussed in Section 3.6.3.1, some language features can have globally negative effects on portability, requiring control over their availability. VMT, through its use of *feature* annotations, provides this control to ensure these features *can* be disabled when unused by the source program.

Moreover, *tightly* controlling language features aligns with *incremental portability*. All programming languages impose minimal constraints through the implementation of language features. Sometimes, these minimal constraints prevent the porting of a program to certain target environments. When this happens, porting can be achieved by removing the dependence of the program on detrimental language features and disabling those features, provided that the implementation allows it. When *granular* control of language features is allowed, programmers can pick the right trade-off between language expressivity and portability for any particular program. Overall, this enables *incremental* portability, as a programmer can spend a constrained amount of effort to refactor the original program (removing the detrimental language features) to target new constrained environments. For example, a program can be ported to size-constrained environments by removing all variadic functions.

To enable portability by disabling language features, implementations need to remove all traces of these language features in the generated programs. This can be achieved with ahead-of-time (AOT) compilation, for example, the Gambit Scheme compiler has the option to disable floating-point numbers. Traditional VMs cannot perform this optimization as they need to support all features required by their bytecode. In other words, removing language features from their implementation would change which bytecode is accepted.

VMT allows for both *tight* control of language features and AOT compilation to truly remove them. With the *feature* system, programmers can precisely control what is included in the final executable. New language constructs can be annotated to be controlled, and existing ones can be split up into granular options when needed. Combined with the ability to specialize VMs, this creates a system where VMT enables *incremental portability* through the control of language features.

Chapter 6

Evaluation

This chapter evaluates the portability of Ribbit and Virtual Machine Tailoring. As no metric exists that directly assesses the portability of a compiler, other aspects are measured: the time required to add and improve target environments and the number of supported targets.

The footprint and execution speed are also evaluated in the Section 6.3. Note that these results already appeared in *A R4RS Compliant REPL in 7 KB* [35] (from Oest O’Leary, Laroche and Feeley).

6.1. Adding New Targets

18 of the 25 targets currently supported by Ribbit have been implemented by third-year bachelor students in the context of the Compiling class at the Université de Montréal taught by Marc Feeley. This gives a good baseline for estimating the time required by non-experts to write a new target for Ribbit, as most students were not familiar with either Scheme or Ribbit when entering the class.

These RVMs were implemented over 3 editions of the class: Winter 2022, Winter 2023 and Winter 2025. Between 13 and 15 days were given to students to perform the implementation of the RVM as shown in Table 6.1. Note that a typical curriculum recommend

Table 6.1. Table showcasing the assignment announcement and due date for each edition of the compiling class.

Class year	Announcement of the assignment	Assignment due	Time required
Winter 22	19 of January	1st of February	14 days
Winter 23	18 of January	1st of February	15 days
Winter 25	22 of January	4th of February	13 days

students to have 5 courses per semester, meaning that they had to attend other classes and assignments during the completion of this assignment. They also had access to both the teacher and the teaching assistant to ask questions about the assignment.

Most students submitted a working implementation that passed all tests for the original implementation of Ribbit.

6.2. Improving Targets

Virtual Machine Tailoring implies the existence of virtual machines at different stages of development. To support features needed by certain programs, virtual machines need to be improved with additional primitives or features. This subsection evaluates the time required to improve these virtual machines to support a wider set of programs.

6.2.1. Bootstrapping the Compiler Using the `go` Target

This experiment has the goal of measuring the time required to improve the `go` RVM in order to support the bootstrap of the compiler. This particular goal has been chosen as supporting the bootstrap can be objectively assessed. Indeed, the Ribbit compiler, which is a Scheme program, can be executed by two different Scheme implementations and the programs generated by both implementations can be compared byte-to-byte. In our case, the two implementations were Gambit and Ribbit (`go` RVM). The program was the Ribbit compiler.

6.2.1.1. Experimental Setup. Prior to this experiment, there was already an RVM implemented for the `go` target. However, the RVM did not support the bootstrap of the compiler due to the absence of these seven enhancements:

- The implementation of variadic functions
- The `primitives` annotation within the RVM
- The implementation of the `apply` primitive
- The implementation of file I/O primitives
- The implementation of command line arguments
- The implementation of the `exit` primitive.
- The implementation of the `optimal` encoding

The author of this thesis implemented these features while keeping a written log of the activities performed at each moment and a git log of all changes made. The aggregated log of commits and activities can be found in Appendix B.

The `go` target was chosen as the author did not have any knowledge of the language before the experiment started.

Overall, the experiment lasted two days and started on 13 April 2026 at 11h00 and finished on 14 April 2026 at 12h58. Two pauses were allowed: a lunch pause between 13h51 and 15h27 on the 13th of April and a night rest between the 13th of April at 17h20 and the 14th of April at 11h00.

6.2.1.2. Results. The time necessary to improve the `go` RVM to support the bootstrap was **7 hours 12 minutes** without any pauses and **1 day 2 hours 58 minutes** with pauses. More precisely, here is the time taken for each enhancement: variadic functions (41m), primitive annotations (15m), apply primitive (22m), file I/O primitives (2h51m), the command line primitive (25m), the addition of the optimal encoding (2h35m) and the exit primitive (4m).

6.3. Operational Constraints

6.3.1. Footprint of the REPL

We are interested in measuring the footprint of our R4RS implementation as a standalone executable as well as the execution speed. The footprint of the read-eval-print loop (REPL), including the generated RVM for the x86 assembly host, is shown in Table 6.2. Each column represents a different compilation setting. The first column is the baseline, which is the size of the generated code without any optimizations. The available compilation settings are:

Baseline.: The `original` encoding is used.

Prim-no-arity.: The procedure call argument count is normally pushed to the stack, costing one byte per encoded call. If rest parameters are not used with primitives, this can be skipped for primitives. The space saving is appreciable as all calls to primitives are encoded with one fewer byte and the argument count check in the RVM can be removed if rest parameters are not used in the source program.

Optimal (92).: The `optimal` encoding with 92 codes per byte is used.

Optimal (256).: The `optimal` encoding with 256 codes per byte is used.

LZSS.: LZSS compression is applied to the generated code before writing it to the RVM.

Note that LZSS compression works only with optimal (256) encoding.

	Baseline	Prim-no-arity	Optimal (92)	Optimal (256)	Prim-no-arity Optimal (92)	Prim-no-arity Optimal (256)
No LZSS	14 KB	13 KB	9.8 KB	9.2 KB	9.0 KB	8.4 KB
LZSS	-	-	-	6.6 KB	-	6.5 KB

Table 6.2. Footprint of the REPL compiled to the x86 assembly host with different settings.

The most compact executable for the REPL is obtained, unsurprisingly, with the combination of Prim-no-arity, Optimal (256), and LZSS: a footprint of 6.5 KB. Not using Prim-no-arity has a minor 2% impact on footprint when using LZSS.

6.3.2. Footprint and Execution Speed

To test the footprint and execution speed for specific source programs, the x86 assembly and the JavaScript host have been benchmarked, as well as the Gambit Scheme interpreter (GSI) [12]. Tests are taken from the Gambit benchmarking suite[12]. The test machine is a 4.5 GHz Intel i7-9750H with 16 GB of RAM running Linux. The NodeJS version is v10.24.1. The Gambit version is v4.9.5. In the Table 6.3 and Table 6.4, different settings were used to compile/execute the benchmark:

gsi.: GSI used as a reference for execution speed comparison.

pna.: Refers to the Prim-no-arity compilation option. This is the same as in Table 6.2.

Note that we use the label **pa** to indicate not using that option (i.e. the primitives do check arity).

tc.: The R4RS library comes in two forms: with and without type checking. The **tc** option means that the type checking version is used. Note that type checking is not required for conformance with R4RS.

x86 REPL.: Execution using the REPL compiled for the x86 assembly host. It was compiled with Prim-no-arity, Optimal (256), LZSS, and the non type checking R4RS library (footprint of 6.5 KB).

Note that a few entries are missing from the tables. Some of the benchmarks use named-**let** and internal **define** which are not supported by the REPL because they are not required by R4RS. This is indicated with N/A in Table 6.3. In Table 6.4, a few tests with the type checked version timed-out. The limit was 5 minutes for the compilation and the execution of the benchmark.

	gsi (secs)	pna	pa	tc pna	tc pa	x86 REPL
ctak	0.2 s	0.9 × 2.1 KB	1.0 × 2.2 KB	1.5 × 8.8 KB	1.8 × 9.3 KB	2.8 ×
fib	26.2 s	0.3 × 2.0 KB	0.4 × 2.0 KB	1.9 × 8.6 KB	2.2 × 9.1 KB	4.9 ×
ack	2.2 s	0.4 × 2.0 KB	0.4 × 2.0 KB	1.6 × 8.6 KB	1.9 × 9.1 KB	5.7 ×
tak	2.3 s	0.6 × 2.0 KB	0.6 × 2.0 KB	1.5 × 8.6 KB	1.7 × 9.1 KB	3.5 ×
takl	2.5 s	0.9 × 2.2 KB	1.0 × 2.2 KB	0.8 × 8.7 KB	1.0 × 9.2 KB	1.0 ×
primes	1.4 s	0.8 × 2.3 KB	1.0 × 2.3 KB	1.5 × 8.8 KB	1.8 × 9.3 KB	2.6 ×
deriv	0.7 s	6.8 × 2.7 KB	8.2 × 2.7 KB	26.3 × 9.2 KB	32.6 × 9.8 KB	7.3 ×
mazefun	1.4 s	0.7 × 4.0 KB	0.8 × 4.1 KB	1.7 × 9.9 KB	2.0 × 11 KB	N/A
nqueens	2.0 s	0.8 × 3.4 KB	0.9 × 3.5 KB	1.7 × 8.8 KB	2.0 × 9.2 KB	N/A
sum	19.4 s	0.3 × 2.0 KB	0.4 × 2.0 KB	2.1 × 8.6 KB	2.5 × 9.1 KB	N/A

Table 6.3. The first column shows the absolute execution time of each benchmark when using the Gambit Scheme interpreter. The other columns show the relative execution time and footprint of Ribbit’s x86 assembly host compared to the first column.

	gsi (secs)	pna	pa	tc pna	tc pa
ctak	0.2 s	10.6 × 2.7 KB	13.2 × 2.7 KB	18.3 × 11 KB	22.0 × 11 KB
fib	26.2 s	4.0 × 2.4 KB	5.1 × 2.5 KB	FAIL 11 KB	FAIL 11 KB
ack	2.2 s	4.8 × 2.4 KB	5.3 × 2.5 KB	21.5 × 11 KB	22.9 × 11 KB
tak	2.3 s	7.0 × 2.4 KB	8.3 × 2.5 KB	20.4 × 11 KB	23.1 × 11 KB
takl	2.5 s	11.5 × 2.7 KB	13.5 × 2.7 KB	12.5 × 11 KB	13.7 × 11 KB
primes	1.4 s	10.6 × 2.8 KB	12.3 × 2.9 KB	19.7 × 11 KB	25.0 × 11 KB
deriv	0.7 s	94.2 × 3.3 KB	108.0 × 3.4 KB	FAIL 11 KB	FAIL 12 KB
mazefun	1.4 s	8.5 × 5.0 KB	10.5 × 5.2 KB	23.8 × 12 KB	28.0 × 13 KB
nqueens	2.0 s	10.2 × 4.2 KB	11.8 × 4.4 KB	24.7 × 11 KB	27.6 × 11 KB
sum	19.4 s	4.2 × 2.4 KB	5.3 × 2.5 KB	FAIL 11 KB	FAIL 11 KB

Table 6.4. The first column shows the absolute execution time of each benchmark when using the Gambit Scheme interpreter. The other columns show the relative execution time and footprint of Ribbit’s NodeJS host compared to the first column.

The results for the x86 assembly host demonstrate good space and execution speed characteristics when the programs are compiled with the AOT compiler. These relatively small benchmark programs (15-200 lines of code) compile to executables in the 2-4 KB range when the non type checking R4RS library is used. They demonstrate the effectiveness of the AOT compiler to remove unused parts of the R4RS library and RVM source code. In terms of execution speed, when the type checking R4RS library is used, the footprint grows considerably to the 9-11 KB range. This is due to the frequent use of `set!` to redefine the predefined procedures with type checking variants which interferes with the effectiveness of the dead code elimination.

Execution speed compares well with the GSI. All programs except `deriv` are faster when compiled with the Ribbit AOT compiler and without the type checking library. The AOT compiler does not optimize `deriv` well due to the presence of higher-order procedures and rest parameters. The x86 REPL fares reasonably well in terms of execution speed with a factor of no more than $7.3\times$ slower than the GSI.

The results for the JavaScript host show that the footprint is consistently about 0.5 KB larger than with the x86 assembly host. On the other hand, the execution time is about one order of magnitude larger, which can be explained by the use of a higher-level host language without low-level control.

Chapter 7

Future Work

7.1. Defining Portability for Language Implementations

In the context of language implementations, portability is a catch-all word. It takes different meanings depending on the operational constraint relevant for the particular language implementation. For example, *portability* for language implementations can mean:

- A “high” number of supported targets.
- The ability to compile code for embedded devices.
- The compliance of the language implementation with the C standard.
- The generation of executables with a “low” number of dependencies.
- The ability for generated executables to run inside a browser.
- A small language implementation size.
- The ability to embed the language implementation into other software.

While all of these definitions concern a certain aspect of portability, they do not get to the root of it. This lack of precise definition has negative consequences for the design of language implementations as it obscures the process of making them more portable. It leaves this process to ad-hoc techniques and intuition, which might inadvertently hurt other aspects of portability.

Chapter 3 is an attempt at defining portability precisely and applying this meaning to language implementations. However, Section 3.6 specifically refocuses the discussion around three aspects of portability which may or may not represent all the challenges encountered while porting. Further theoretical research into the implications of portability on language implementation could identify other important tradeoffs to consider while designing highly portable systems.

7.2. Generating New Ribbit Virtual Machines with LLMs

The core Ribbit Virtual Machine (RVM) is surprisingly simple. It is an algorithm that can be implemented in fewer than 1000 lines of code in most programming languages, suggesting that RVMs could entirely be generated using Large Language Models (LLMs). At the same time, virtual machines need precise engineering where any small mistake or misunderstanding of the target’s language semantics can yield unusable virtual machines, suggesting that LLMs might have trouble generating one fully automatically.

7.2.1. A Small Experiment at Generating Two RVMs Using Coding Agents

A small experiment was conducted to generate two core RVMs automatically. The author selected two programming languages: Ruby and APL, as they represent two extremes of implementation difficulty. On one hand, an RVM in Ruby should be easy for an agent to implement. Code is plentiful online (3M+ repositories with the Ruby tag on GitHub), and the semantics of the Ruby language are very similar to Python and JavaScript, both of which already have mature RVM implementations. On the other hand, APL should be harder for an agent to implement. Code is less available online (with fewer than 600 repositories on GitHub with the APL tag), and APL uses a programming language paradigm, the array programming style, that no current Ribbit target has. Before conducting any experiment, the author’s hypothesis was that the agents could generate an RVM for Ruby easily, but would fail to generate a valid RVM for APL.

The full setup of the experiment can be found in Appendix C. The results are summarized here. GLM-5.2[15] with OpenCode[36] managed to generate a Ruby RVM passing all tests of the core RVM specification. To generate the Ruby target, the agent processed 3.6M tokens and cost \$0.81 USD of the \$11.70 USD allocated for this experiment. The same setup did not manage to generate an RVM passing all tests for the APL target after consuming the rest of the allocated budget for the experiment. The generated RVM prints `HELLO!` on execution and passes 16/35 tests. To generate the APL RVM, the agent processed 46M tokens and cost \$8.37 USD, which is more than 10 times the resources allocated for the Ruby target.

7.2.2. Results and Future Directions

These results show that the current capabilities of agents are inconclusive concerning the generation of *arbitrary* RVMs. For well-known languages having semantics similar to the existing targets, agents can generate RVMs with limited time and resources as shown

by the generation of the Ruby target costing less than \$1. However, for more complex environments such as APL, more research is required to know whether agents could generate RVMS automatically. It is believable that with more precise prompts, better or different models, or more resources, the agent could generate a fully-working APL RVM, especially because the generated RVM is partially working.

Furthermore, the experiment did not evaluate the extension of targets. The generated RVMS were only core RVMS, meaning RVMS that support basic Scheme constructs without variadics, file I/O, or R4RS support. It is believable that agents could generate simple extensions with limited resources, but it is still unclear whether they could generate complex extensions *automatically*, such as a threading system, the embedding of a program into a very specific environment, or new memory management strategies.

If this method is found to be reliable, this would allow a new kind of portability where any porting effort could be performed at a predictable cost using an agent. This would ensure the effortless portability of code written for Ribbit, as any change in *where* the code should execute could be resolved by chatting with an LLM. Contrary to other systems, Ribbit is particularly well-suited for this usecase, as its system is compact and simple, thus easily understandable by coding agents.

Chapter 8

Conclusion

This thesis presents a new technique called Virtual Machine Tailoring with a promising application in the design of compilers for portability. Virtual Machine Tailoring provides the control of native compilation and the convenience of bytecode compilation by generating *tailored* virtual machines alongside a specialized bytecode. This hybrid approach to compilation is particularly useful when programs require high portability: the *tailored* VM can be adapted to specific environments at compile time, new targets can easily be added similarly to bytecode compilation, and the source language can be extended or restrained depending on the portability needs.

The first part of this thesis (Chapter 3) introduces the concept of portability and its implications for language implementations at a theoretical level. The chapter defines portability as the *effort* to translate all interfaces a program depends on into ones supported by a target environment. Within this definition, a *portable* language implementation should limit the effort to translate the programming language interface into relevant interfaces. To do so, Section 3.6 discusses three strategies: adapting programs to various operational constraints (such as less memory, compute, or I/O), extending the compiler with more targets, or trading expressiveness of a programming language for portability. Section 3.7 also presents the concept of *incremental portability*, which gives insight into the design of compilers for portability.

The second part of this thesis (Chapter 4) presents Ribbit, the reference implementation of Virtual Machine Tailoring. Ribbit is based on a unique VM called the Ribbit Virtual Machine, where *everything* from the stack, code, and data is represented as 3-field vectors called *ribs*. Having a single memory model for all objects combined with the small set of instructions and primitives makes the Ribbit Virtual Machine very simple to implement and understand, thus portable to a wide range of targets. In practice, Ribbit is very portable.

It supports 25 target languages, with the ability to add new targets in 2 weeks of work by third-year bachelor students.

In Ribbit, programs are serialized into the Ribbit Intermediate Byte Notation (RIBN) to generate compact programs, necessary for portability to memory-constrained targets. As programs are graphs of ribs, the RIBN is composed of *decoding instructions* acting on a stack-based VM to reconstruct the graph. Section 4.3 also discusses the various optimizations to reduce the size of the textual representation of the RIBN, notably, using compression algorithms and specializing the encoding of *decoding instructions* based on their frequency inside the RIBN. Overall, these optimizations yield very compact programs, in the order of KiloBytes. For example, Ribbit compiles a full R4RS-compliant Scheme interpreter in a 6.5 Kb binary with no dependencies, which is the smallest ever recorded.

The third part of this thesis (Chapter 5) details the technique to perform Virtual Machine Tailoring. In this approach, the compiler takes an annotated VM as input to generate a specialized VM given a set of activated features. This allows the input program, accompanying libraries, or compiler to alter the generated VM using features, allowing VM extensions to be expressed inside the compiler, VM, program, or accompanying libraries. Furthermore, the compiler or VM can activate features to conditionally include code from the input program or accompanying library, allowing the adaptation of the input code to the compiler or environment. This technique yields a very expressive system that can be used to *tailor* any given program for specific environments or programs.

To showcase the expressivity of Virtual Machine Tailoring, Section 5.6 presents two case studies. The first case study shows the specialization of Ribbit’s decoding algorithm to the input program. This optimization shows how *the compiler* can guide the generation of specialized VMs. The second case study demonstrates the implementation of a runtime two-way foreign function interface (FFI) from a Scheme library. This illustrates the writing of virtual machine extensions from Scheme code, in this case, adding a new *foreign* native datatype to the generated VM alongside the FFI implementation.

Finally, Section 5.7 discusses the implications of Virtual Machine Tailoring as a compiler technique on portability. For example, Virtual Machine Tailoring allows for a portable minimal core VM that can be extended as needed using expressive extensions. This controls the complexity of VM implementations, limiting the negative impact on portability of requiring fully-featured VM implementations (which is typically the case for large general VMs). However, this implies that different targets might support a different set of extensions, which can confuse programmers who want to port programs for a new platform. Ultimately, Virtual Machine Tailoring comes with different tradeoffs, but its advantages align closely with portability as defined in the first chapter.

The chapter concludes with the evaluation of Ribbit and, by extension, Virtual Machine Tailoring. Chapter 6 measures the time to implement and extend various targets, taking two weeks for third year undergraduate students (non-experts) and 2 days for the author (an expert) respectively. Section 6.3 also evaluates the execution speed and footprint of Ribbit. Ribbit’s execution speed is in the same order of magnitude as the Gambit Scheme interpreter (GSI) for most benchmarks except one while being significantly smaller.

Overall, Virtual Machine Tailoring is a promising technique for the design of compilers for portability. Most of its benefits align with the goal of *incremental portability* (Section 5.7.6), raising the question whether a practical system could ensure portability through small incremental efforts. Ribbit offers a glimpse into this vision of portability that might one day change the “write once, run everywhere” motto to “extend once, run everywhere.”

Appendix A

Survey of Comment Syntaxes in 25 Languages

Table A.1. Survey of syntaxes for comments in each of the languages supported by Ribbit.

Language	Inline Comment	Block/Multiline Comment
Ada	--	<i>None</i>
Assembly (x86)	; (NASM)	<i>None</i>
BASIC	REM or ' (must be at the start of line)	<i>None</i>
C	// (since C99)	/* ... */, no nesting.
Clojure	;	<i>None</i>
Common Lisp	;	# ... #, nestable.
Dart	//	/* ... */, nestable.
Go	//	/* ... */, no nesting.
Haskell	--	{- ... -}, nestable.
Idris	--	{- ... -}, nestable.
Java	//	/* ... */, no nesting.
JavaScript	//	/* ... */, no nesting.
Julia	#	#= ... =#, nestable.
Kotlin	//	/* ... */, nestable.
Lua	--	--[[...]] nesting with extra syntax.
OCaml	<i>None</i>	(* ... *), nestable.
Perl	#	<i>None</i>
POSIX Shell	#	<i>None</i>

The table continues on the next page.

Table A.1 from previous page.

Language	Inline Comment	Block/Multiline Comment
PostScript	%	<i>None</i>
Prolog	%	<i>/* ... */</i> , no nesting.
Python	#	""" ... """ or ''' ... '''. '''
Ruby	#	=begin/=end, no nesting (must start at the start of line)
Rust	//	<i>/* ... */</i> , nestable
Scala	//	<i>/* ... */</i> , nestable.
Scheme	;	# ... #
WebAssembly (WAT)	;;	(; ... ;), nestable.
Zig	//	<i>None</i>

Appendix B

Development log of the Go target extension

Listing B.1. Development log of the Go target extension

```
April 13
11h00 - Cloning of the Ribbit repository
11h03 - Ensuring that the 'go' RVM works with the min/max REPL (it
      does!)

Step 1: Implementation of the variadics (11h03 to 11h44)
11h04 - Modified the makefile to run tests related to variadics
11h05 - Noticed that variadics tests loop indefinitely
11h15 - Searched about ternary operator in go
11h18 - Searched about while loops in go
11h21 - First implementation of the arity-check feature
11h25 - Pause
11h27 - Resume
11h32 - Fixed type errors and wrongly named variables
11h32 - Implemented the arity-check feature, rest-param is still
      required
11h38 - Fixed type errors
11h43 - Fixed off-by-one error with rest param calculation
11h44 - All tests pass with the variadics

Step 2: Add primitive annotations to the RVM (11h44 to 11h59)
11h54 - First addition of the primitive annotations within the go
      RVM
11h58 - Fix error related to feature dependencies
11h59 - All tests pass with the primitive annotations
```

12h00 - Pause
12h06 - Resume

Step 3: Implementation of the Apply primitive (12h06 to 12h28)

12h13 - Added logic inside RVM to directly call a returned rib by a primitive
12h25 - First implementation of the Apply primitive
12h28 - Fixed bug where the returned rib was not assigned properly
12h28 - All tests pass with the apply primitive

Step 4: Implementation file I/O primitives (12h28 to 13h51 and 15h27 to 16h55)

12h30 - Researching about the file API in go
12h39 - First implementation of the scm2str feature necessary for converting a Scheme filename into its host counterpart
12h49 - First implementation of the 6 I/O primitives: ‘%%stdin-fd’, ‘%%stdout-fd’, ‘%%get-fd-input-file’, ‘%%get-fd-output-file’, ‘%%read-char-fd’, ‘%%write-char-fd’
12h52 - Fixed syntax errors inside scm2str
13h00 - Research about file descriptors and finalizers in go
13h15 - New iteration of file I/O primitive using file descriptors instead of file handlers
13h26 - Debugging file I/O primitives with existing tests
13h46 - Test 00-ribbit/71_write_file.scm working
13h51 - Lunch pause
15h27 - Resume from Lunch
15h27 - Trying to find why the error ‘write: bad file descriptor’ appears for programs not opening any files
16h09 - Found that the stdout file descriptor is being closed for no apparent reason
16h37 - Partially fixed finalizers closing the file descriptors upon the collection of file handlers
16h37 - Almost all tests related to file I/O passes
16h41 - Fixed bugs with feature dependencies
16h55 - Rewrote file I/O primitives using ‘os.syscall’ directly, avoiding finalizers
16h55 - The file I/O primitive and R4RS repl now works!

Step 5: Implementation of command line argument primitives (16h55 to 17h20)

17h04 - First implementation of list2scm, necessary for converting host command line arguments to a Scheme list
17h14 - First implementation of the 'command-line' primitive
17h17 - Fixed bug where there was one too many argument inside of the list
17h17 - All tests related to the primitive pass
17h20 - Stopping for the rest of the day

Step 6: Implementation of the optimal encoding (11h00 to 12h35)

April 14

11h00 - Resume the experiment
11h24 - First implementation of the optimal encoding
11h33 - Fixed compiling bugs with the first implementation
11h43 - Fixed one error with the decoding algorithm
11h53 - Pause
11h58 - Resume
12h27 - Fixed multiple errors with the decoding algorithm implementation
12h33 - Fixed a bug with the instTail function
12h35 - All tests pass with the optimal encoding

12h38 - Pause
12h42 - Resume

Step 7: Add the %%exit primitive (12h42 to 12h46)

12h46 - Added the %%exit primitive

Step 8: Debug the Bootstrap (12h46 to 12h58)

12h49 - Fixed bug with 'command-line' primitive
12h58 - Bootstrap is now working with no diffs!

Appendix C

Experiment Details of Generating RVMs Using LLMs

C.1. Setup of the Experiment

The coding agent used was OpenCode[36] 1.15.12 with GLM-5.2[15], one of the highest-rated open-source models for coding as of August 5, 2026. The LLM provider used was OpenRouter with a total budget of \$11.70 USD for the whole experiment (\$10.00 USD in initial credits and \$1.70 USD in overcharges due to delays from OpenRouter in stopping token processing). This budget covered all LLM interactions, including the initial prompt engineering and the generation of both RVMs.

C.2. Prompts

In total, the author crafted three prompts with the help of GLM-5.2: a general prompt for translating an RVM for any target (see Figure C.1) and one specific prompt for each language: APL (see Figure C.3) and Ruby (see Figure C.2). All prompts were crafted before agents performed any work, making sure both trials were independent. All three prompts were written on disk, and the instruction for the agent was as follows, where `<language>` was replaced with the specific language, either Ruby or APL: “Implement a new `<language>` target for Ribbit. Read `ribbit.md` to have general information about the implementation of a new target for Ribbit and `<language>.md` for specific information about the `<language>` target.” Once the prompt was started, no other human interaction was involved, except accepting all the commands that the agent asked to execute.

According to OpenCode, the generation of the Ruby target cost \$0.81 USD, using 288,939 input tokens, 8,792 output tokens, 44,080 reasoning tokens, and 3,299,968 read-cache tokens, for a total of 3.6M tokens. The generation of the APL target cost \$8.37 USD, using 2,045,332

input tokens, 101,798 output tokens, 176,609 reasoning tokens, and 43,853,568 read-cache tokens, for a total of 46M tokens.

C.3. Evaluation of RVMS

The Ruby and APL RVMS were evaluated using the `core` test suite of Ribbit that can be found under the `src/tests/00-ribbit` folder in the Ribbit repository. This suite tests the basic functionalities of an RVM, such as the printing of characters, addition of two numbers, the calling of a function, the reading of S-expressions, and the evaluation of the Fibonacci function using a REPL. The directory for each target contained a stub RVM implementation with a working command to execute all tests.

The generated Ruby RVM passed all tests related to the RVM `core` tests.

The generated APL RVM did not pass all tests related to the RVM `core` tests. When executed, the RVM printed the `HELLO!` string and 16/35 tests from the core Ribbit test suite passed.

Listing C.1. General prompt used to generate both RVMS.

```
# Plan: Implement a new Ribbit RVM host target

## Context & Approach

The target lives at 'src/host/<ext>/'. It currently contains a stub 'rvm.<ext>' that just
prints the HELLO! bytecode, a 'makefile', a placeholder 'minify', and a 'tests/00-empty.
scm'. The goal is to translate a reference RVM into the host language so that running
the interpreter on 'rvm.<ext>' prints 'HELLO!', then passes 'HOST=<ext> make check' from
'src/'.

The best reference is the annotated hello.py (on the 'leo-ard/document-contributing'
branch: 'src/host/py/hello.py'), which maps 1:1 to the 6 implementation sections
described in 'CONTRIBUTING.md'. The production 'src/host/py/rvm.py' gives the canonical
feature-annotated form for later (AOT compiler substitution), but for step 1 the hello-
style core-only RVM is sufficient. The task is essentially a faithful "human compiler"
translation from Python to the host language. No '@@(feature)@' annotations for
this step — core-only.

## Step 1 — Implement 'rvm.<ext>' (core RVM)

Translate 'hello.py' section-by-section, keeping the same structure so it remains a faithful
translation:

- *(0) Input string — Preserve the '@@(replace ...)' marker exactly so 'rsc.scm' can
  substitute the real bytecode later:
  ```
 # @@(replace " ");'u?>vD?>vRD?>vRA?>vRA?>vR:?>vR=!(:lkm!':lkv6y" (encode 92)
 <input assignment holding the HELLO! bytecode>
  ```
```

```

# )@@
'''

The marker must remain intact and unaltered for AOT compilation. Running the file
unmodified must print 'HELLO!'.

- **(1) Rib representation & VM defs** — Ribs are 3-field structures; in dynamically-typed
  hosts, use the host's array/list type. Implement 'get_byte', 'get_code', 'get_int', '
  push'/'pop' (stack as a rib-linked-list on field 1), and 'TRUE'/'FALSE'/'NIL' as
  distinct '[0,0,5]' objects. 'is_rib?' checks whether a value is a rib (array). Output
  via stdout write+flush for 'putchar'; stdin read of 1 byte for 'getchar' returning '-1'
  on EOF.

- **(2)/(3) Primitives** — Build the 21-entry 'primitives' array in the exact order from
  'hello.py' ('%%rib!..%%exit'). Use 'prim1'/'prim2'/'prim3' wrappers (lambdas/closures).
  Critical semantics:
  - '%%id' — pop and push back (no-op).
  - '%%arg1' — pop (discard top).
  - '%%close' — builds '[rib[0], stack, 1]'.
  - '%%rib?' / '%%eqv?' — return the 'TRUE'/'FALSE' ribs, not the host's native booleans.
  - '%%eqv?' — uses identity (pointer equality) when either side is a rib.
  - '%%quotient' — truncated division toward zero, not floored.
  - '%%getchar' — returns '-1' (integer) on EOF.
  - '%%putchar' — may need to flush output.
  - '%%exit' — exits the program.

- **(4) Symbol table** — Translate the decode loop: read count of empty symbols, then chars
  building reversed names as rib-lists, splitting on ',' and ending on ';'. 'symbol_ref(n)
  ' via 'list_tail'.

- **(5) Code graph decode** — Direct translation of the tight 'while 1' loop with the
  '[20,30,0,10,11,4]' op table and the 'x>90'/'op==0'/'op>4' branches.

- **(6) Execution loop** — Translate the main 'while 1' dispatch ('jump/call', 'set', 'get',
  'const', 'if', 'halt'), plus 'get_opnd', 'get_cont', 'set_global' for the 4 globals (
  primitive-0, FALSE, TRUE, NIL) and the primordial continuation '[0,0,[5,0,0]]'.

Verification: running the interpreter on 'src/host/<ext>/rvm.<ext>' should print 'HELLO
!'.

## Step 2 — Verify against the test suite

From 'src/': run 'HOST=<ext> make check'. This invokes 'run_test.sh' for each 'tests/*.scm'
plus 'host/<ext>/tests/*.scm' using the 'HOST_INTERPRETER' (or 'HOST_COMPILER') from the
makefile. Iterate on failures.

## Risks / things to get right (general)

1. '%%quotient' — must truncate toward zero; many hosts' integer division floors toward
   negative infinity for negatives.
2. Identity for '%%eqv?' — 'TRUE'/'FALSE'/'NIL' are distinct rib objects; comparisons
   must use object identity, not value equality.

```

3. **getchar EOF** — return `-1` (integer), not `null/None`.
4. **Stack discipline** — every primitive must leave exactly one value pushed.
5. **Shared state** — the decode/execute blocks share `'stack'/'syntbl'/'pos'/'pc'`; ensure the host's scoping (globals, module vars, or a struct) allows mutation across functions.
6. **'@@(replace ...)' marker** — must remain intact and unaltered for AOT compilation.
7. **Comment syntax** — the `'@@(replace ...)' / '(')@@'` and any `'@@(primitive ...)@@'` tags must be in the host's comment syntax so `'rsc.scm'` parses them.
8. **Primitive prefix** — Primitives are prefixed with `%%`, such as `'%%id'` or `'%%+'`. Older versions used `##`, so documentation or older implementations might contain `##` instead of `%%`. Make sure to use `%%`.

Reference files to consult during implementation

- `'src/host/py/hello.py'` (on `'leo-ard/document-contributing'` branch) — annotated 6-section reference.
- `'src/host/py/rvm.py'` — production, feature-annotated form (for later AOT support, not required for step 1).
- `'src/host/js/rvm.js'` — second reference (dynamically typed, closures).
- `'CONTRIBUTING.md'` (on `'leo-ard/document-contributing'`) — section "Adding a new host" / "Hello World RVM".
- Paper: "A Small Scheme VM, Compiler, and REPL in 4K" (Figure 4 = primitive list).

Listing C.2. Specialized prompt used for generating the Ruby RVM.

```
# Ruby-specific notes (extension 'rb')

This file holds the Ruby-target-specific portion of the Ribbit RVM implementation plan. The
general plan lives in 'ribbit.md'.

- Files & location: 'src/host/rb/rvm.rb', 'src/host/rb/makefile' (already correct: 'HOST
=rb', 'HOST_INTERPRETER ?= ${RB_HOST_INTERPRETER:-ruby}', 'TEST_TAGS ?= rb core'), 'src
/host/rb/tests/'.
- Interpreter: 'ruby' (system Ruby 2.6.10 available at '/usr/bin/ruby'). Invoked as '
ruby rvm.rb'.
- Ribs: use Ruby 'Array' of length 3 ('[f0, f1, f2]'); field access by index, mutation
by 'arr[i]='.
- Comment syntax for markers: Ruby '#' line comments. The '@@(replace ...) / '(')@@' and '
@@(primitive ...)@@' markers must be on '#' comment lines so 'rsc.scm' recognizes them.
Use the form:
  ``ruby
  # @@(replace " );'u?>vD?>vRD?>vRA?>vRA?>vR:?'>vR=!(:lkm!':lkv6y" (encode 92)
  $input = " );'u?>vD?>vRD?>vRA?>vRA?>vR:?'>vR=!(:lkm!':lkv6y"
  # )@@
  ``
- Shared mutable state: use global variables ('$input', '$pos', '$stack', '$syntbl', '
$pc') to mirror Python's 'global' declarations, or wrap state in instance variables of a
singleton object. Globals are the simplest faithful translation.
```

```

- **Primitives as closures:** Ruby procs/lambda capture by reference; ‘prim1’/‘prim2’/‘
  prim3’ return lambdas that pop N times and push the result. Note Ruby block arity is
  strict — use a proc accepting ‘*args’ or a lambda with explicit arity matching the pop
  count.
- **%%quotient:** Ruby integer ‘/’ floors toward ‘-infinity’, so implement as ‘(x.to_f / y
  ).truncate’ (or ‘x.divmod(y)’ adjusted for sign) to truncate toward zero like Python’s ‘
  int(x/y)’.
- **Identity (‘%%eqv?’):** use ‘a.equal?(b)’ (object identity via ‘object_id’), not ‘==’.
  Ensure ‘TRUE’/‘FALSE’/‘NIL’ are each a single shared ‘[0,0,5]’ array instance.
- **‘is_rib?’:** ‘x.is_a?(Array)’.
- **%%putchar:** ‘$stdout.write(c.chr); $stdout.flush’.
- **%%getchar:** ‘c = $stdin.read(1); c.nil? || c.empty? ? -1 : c.ord’.
- **%%exit:** ‘exit(pop())’ or simply ‘Kernel.exit’.
- **‘get_byte’:** ‘$input[$pos].ord’ after incrementing ‘$pos’.
- **EOF semantics:** ensure no off-by-one on ‘$stdin.read(1)’ returning ‘’ at EOF.
- **Whitespace sensitivity:** Ruby uses ‘end’ for block termination (not indentation), so
  the dense decode/execute loops translate cleanly; just append ‘end’ for each ‘while’/‘if
  ’/‘def’.

```

Listing C.3. Specialized prompt used for generating the APL RVM.

```

# APL-specific notes (extension ‘apl’) — GNU APL

This file holds the APL-target-specific portion of the Ribbit RVM implementation plan. The
general plan lives in ‘ribbit.md’.

- **Files & location:** ‘src/host/apl/rvm.apl’, ‘src/host/apl/makefile’ (already present and
  correct — see invocation below), ‘src/host/apl/tests/’.
- **Interpreter & exact invocation (from the makefile):**
  ‘‘‘
  HOST = apl
  HOST_INTERPRETER ?= ${APL_HOST_INTERPRETER:-apl —script —OFF —noSV -f }
  TEST_TAGS ?= apl core
  TEST_FEATURES ?=
  ‘‘‘
  i.e. GNU APL (‘/opt/homebrew/bin/apl’, the ‘gnu-apl’ Homebrew package) run as ‘apl —
  script —OFF —noSV -f <file>’. The flags mean: ‘—script’ (batch/script mode, no
  session prompt), ‘—noSV’ (no supervisor), ‘-f’ (load the file as a script and exit),
  ‘—OFF’ (exit cleanly). The trailing space is intentional so the test harness appends
  the filename cleanly. Confirm ‘—script’ exits non-zero on error so ‘run_test.sh’
  detects failures.
- **GNU APL, not Dyalog:** all constructs must be GNU APL compatible. Avoid Dyalog-only
  features (e.g. ‘□NS’ namespaces-as-objects, ‘:Namespace’ scripts, ‘□SE’). GNU APL
  supports nested arrays, tradfns with ‘:While’/‘:If’/‘:Select’, and ‘□IO’. Ribbit’s ‘rsc.
  scm’ accepts ‘@’ as a comment character (confirmed).
- **Index origin — critical:** set ‘□IO-0’ at the top of ‘rvm.apl’ so all ‘[0,0,5]’, field
  indices (‘rib[0]’, ‘rib[1]’, ‘rib[2]’), and op-table indexing translate directly from
  Python with no off-by-one. This is the single most important APL-specific decision.
- **Comment syntax for markers:** GNU APL ‘@’ line comments (confirmed accepted by ‘rsc.scm
  ’):

```

```

‘‘‘apl
Ⓜ @@@(replace " );'u?>vD?>vRD?>vRA?>vRA?>vR:?'>vR=!(:lkm!':lkv6y" (encode 92)
input←");'u?>vD?>vRD?>vRA?>vRA?>vR:?'>vR=!(:lkm!':lkv6y"
Ⓜ )@@
‘‘‘

```

— **Shared mutable state** — ‘)NS‘ namespace (decided):** create a namespace ‘st ←)NS ‘st’ and put **all** shared state in its fields: ‘st.input’, ‘st.pos’, ‘st.stack’, ‘st.symbtbl’, ‘st.pc’, and the rib memory ‘st.mem’ / ‘st.is_rib_flags’ (for the handle representation). Inside any tradfn, ‘st.pos ← st.pos + 1’ mutates the namespace field (not a local), because the *name* ‘st’ is only ever read, never reassigned. This is the APL analogue of a singleton module of globals and sidesteps GNU APL’s local-shadowing-on-assignment trap entirely. Translate ‘hello.py’'s ‘global x; x ←...’ patterns directly to ‘st.x ←...’.

> Why ‘)NS‘ over plain globals: GNU APL tradfns make a name local on assignment by default, with no ‘global’ keyword. A top-level ‘pos ← 0’ is a global, but ‘pos ← pos + 1’ inside a function creates a *local* ‘pos’ (shadowing), so the global is never updated. Workarounds require every helper to return new values and the caller to reassign — which fights ‘hello.py’'s side-effecting ‘get_byte’/‘push’/‘pop’/‘get_int’ style. A ‘)NS‘ namespace ‘st’ fixes this: the name ‘st’ is read-only everywhere (never reassigned), and field assignment into an existing namespace is a *mutation*, not a new binding. So ‘st.pos ← st.pos + 1’ mutates ‘st’'s field from any function. This is the APL analogue of a singleton object / module of globals.

— **Rib representation** — flat global matrix + integer handles (decided):**

- ‘st.mem’ is a growable matrix (or nested vector of 3-vectors) indexed by handle; a rib is an integer *handle*.
- Field access = ‘st.mem[handle; field]’; mutation = indexed assignment ‘st.mem[handle; field] ← v’.
- The stack is a rib-handle linked list (a handle whose field 1 is another handle, terminating in ‘NIL’).
- ‘TRUE’/‘FALSE’/‘NIL’ are three distinct sentinel handles (special reserved integers, e.g. 0/1/2) with their rib fields set to ‘[0,0,5]’.
- **Tagging**:** tag handles so non-rib integers don’t collide — e.g. a parallel boolean mask ‘st.is_rib_flags’ (a boolean vector where ‘is_rib_flags[handle] = 1’ iff ‘handle’ is a live rib), or reserve a high bit / a sentinel range. An integer ‘5’ must be distinguishable from handle ‘5’.

— **is_rib?**:** check ‘st.is_rib_flags[x]’ (or the chosen tagging scheme).

— **Control flow**:** use **tradfns** with ‘:While’/‘:End’, ‘:If’/‘:Else’/‘:End’, ‘:Select’ for the dense decode/execute loops (dfns are recursion-only and would blow the stack on the tight inner loops). Use **dfns** ‘{...}’ for the small primitive wrappers (‘prim1’/‘prim2’/‘prim3’).

— **prim1’/‘prim2’/‘prim3**:** dfns returning a dfn that pops N and pushes the result. **Operand order is reversed relative to Python** (‘lambda y,x: x-y’ pops ‘y’ first then ‘x’). APL evaluates right-to-left, so explicitly reverse to keep ‘%%-’, ‘%%<’, ‘%%quotient’, ‘%%field0-set!’ operands correct.

— **get_byte**:** ‘c ← st.input[st.pos]’ (a character) then ‘ⱮUCS c’ for codepoint; increment ‘st.pos’. Ensure ‘st.input’ is a character vector.

— **%%putchar**:** ‘Ɱ’ normalizes whitespace — use ‘Ɱ’ (raw output) to write bytes without formatting. GNU APL ‘Ɱ’ writes raw; verify output is not post-processed in ‘—script’ mode.

```

- **%%getchar: reading one byte from stdin is awkward in GNU APL. Recommended approach:
  read all of stdin into a vector up front ('st.stdin ← ⍴' or '⍴GET'/file-based) and index
  it, advancing 'st.stdin_pos'; return '-1' (integer) on EOF. Acceptable for the test
  harness.
- **%%quotient: '[' floors, '[' ceilings. Truncate toward zero = apply '[' if 'a' and 'b'
  have the same sign, '[' if they differ, on the float quotient 'a÷b'. Verify against
  negatives explicitly.
- **Identity ('%%eqv?): with handles, compare handle integers directly (two ribs are 'eqv
  ?' iff same handle). 'TRUE'/'FALSE'/'NIL' are three *distinct* sentinel handles.
- **%%exit: under '--script'/'--OFF', exiting cleanly means returning from the top-level
  tradfn / reaching end of script; use '→' (branch out) or simply end execution. Do **not
  ** rely on Dyalog's '⍴OFF'; GNU APL uses '⍴OFF' which is handled by the makefile's '--
  OFF' flag.
- **Recursion depth: the decode and execute loops are long-running; tradfn ':While' is
  fine. Avoid dfn recursion for the inner interpreter loop.
- **Pre-flight checks before full translation:
  1. Confirm 'apl --script --OFF --noSV -f rvm.apl' runs a trivial '⍴HELLO!' script and
  exits with code 0.
  2. Confirm '⍴IO←0' and '⍴--comment '@@(replace)' markers work end-to-end (comment-char
  confirmed; still worth a 1-minute smoke test of the full '@@(replace)' round-trip).

## Decisions settled
1. APL comment char '⍴' accepted by 'rsc.scm' — confirmed.
2. APL shared state — '⍴NS' namespace 'st' (decided).
3. APL rib representation — flat matrix 'st.mem' + integer handles with 'st.is_rib_flags'
  tagging (decided).
4. No '@@(feature)⍴' annotations — core-only for step 1 (decided).

```


References

- [1] Bob Amstadt and Michael K Johnson. 1994. Wine. *Linux Journal* 1994, 4es (1994), 3.
- [2] Sean Bartell. 2021. *Optimizing Whole Programs for Code Size*. Ph. D. Dissertation. University of Illinois at Urbana-Champaign, USA. Advisor(s) Adve, Vikram. AAI30395044.
- [3] Jeff Bezanson. 2012. *Femtolisp*. <https://github.com/JeffBezanson/femtolisp>
- [4] Peter J. Brown. 2003. *Software portability*. John Wiley and Sons Ltd., GBR, 1633–1634.
- [5] Camillo Bruni, Stéphane Ducasse, Igor Stasenko, and Guido Chari. 2014. Benzo: Reflective Glue for Low-level Programming. In *International Workshop on Smalltalk Technologies*. Cambridge, United Kingdom. <https://inria.hal.science/hal-01060551>
- [6] Kevin Casey, David Gregg, and M. Anton Ertl. 2005. Tiger – An Interpreter Generation Tool. In *Compiler Construction*, Rastislav Bodik (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 246–249.
- [7] Zhiqun Chen. 2000. *Java Card Technology for Smart Cards: Architecture and Programmer’s Guide*. Addison-Wesley Longman Publishing Co., Inc., USA.
- [8] The Pyodide development team. 2021. *pyodide/pyodide*. Pyodide. <https://doi.org/10.5281/zenodo.5156931>
- [9] Patrick Doyle and Tarek S. Abdelrahman. 2002. A Modular and Extensible JVM Infrastructure. In *2nd Java Virtual Machine Research and Technology Symposium (Java VM 02)*. USENIX Association, San Francisco, CA. <https://www.usenix.org/conference/java-vm-02/modular-and-extensible-jvm-infrastructure>
- [10] Danny Dubé and Marc Feeley. 2005. BIT: A Very Compact Scheme System for Microcontrollers. *Higher-Order and Symbolic Computation* 18, 3 (2005), 271–298. <https://doi.org/10.1007/s10990-005-4877-4>
- [11] 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>
- [12] Marc Feeley. 2023. *Gambit Scheme*. Université de Montréal. <https://gambitscheme.org>

- [13] Matthias Felleisen. 1991. On the expressive power of programming languages. *Science of Computer Programming* 17, 1 (1991), 35–75. [https://doi.org/10.1016/0167-6423\(91\)90036-W](https://doi.org/10.1016/0167-6423(91)90036-W)
- [14] Nick Fitzgerald. 2024. *Making webassembly and wasmtime more portable*. Bytecode Alliance. <https://bytecodealliance.org/articles/wasmtime-portability>
- [15] GLM-5 Team. 2026. GLM-5: from Vibe Coding to Agentic Engineering. <https://doi.org/10.48550/arXiv.2602.15763> arXiv:2602.15763 [cs.LG]
- [16] Adele Goldberg and David Robson. 1983. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., USA.
- [17] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (Barcelona, Spain) (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. <https://doi.org/10.1145/3062341.3062363>
- [18] Laurent Huberdeau, Cassandre Hamel, Stefan Monnier, and Marc Feeley. 2024. The Design of a Self-Compiling C Transpiler Targeting POSIX Shell. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (Pasadena, CA, USA) (SLE '24)*. Association for Computing Machinery, New York, NY, USA, 70–83. <https://doi.org/10.1145/3687997.3695639>
- [19] Roberto Ierusalimsky, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. 1996. Lua—an extensible extension language. *Softw. Pract. Exper.* 26, 6 (June 1996), 635–652.
- [20] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the future: the story of Squeak, a practical Smalltalk written in itself. *SIGPLAN Not.* 32, 10 (Oct. 1997), 318–326. <https://doi.org/10.1145/263700.263754>
- [21] ISO/IEC 2009. *Information technology — Open distributed processing — Reference model: Foundations (ISO/IEC 10746-2:2009)*. Technical Report. ISO/IEC.
- [22] ISO/IEC 2015. *Information technology — Vocabulary (ISO/IEC 2382:2015)*. Technical Report. ISO/IEC.
- [23] ISO/IEC 2022. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE): Cloud services — Part 1: Quality model (ISO/IEC TS 25052-1:2022)*. Technical Report. ISO/IEC.
- [24] ISO/IEC/IEEE 2017. *International Standard – Systems and software engineering—Vocabulary (ISO/IEC/IEEE 24765:2017(E))*. Technical Report. ISO/IEC/IEEE. 1–541 pages. <https://doi.org/10.1109/IEEESTD.2017.8016712>

- [25] David Johnson-Davies. [n.d.]. uLisp: Lisp for microcontrollers. <http://www.ulisp.com/>. Accessed: 2026-06-30.
- [26] Neil D. Jones. 2004. Transformation by interpreter specialisation. *Science of Computer Programming* 52, 1 (2004), 307–339. <https://doi.org/10.1016/j.scico.2004.03.010> Special Issue on Program Transformation.
- [27] Frédéric Lahaie-Bertrand, Léonard Oest O’Leary, Olivier Melançon, Marc Feeley, and Stefan Monnier. 2025. Arborescent Garbage Collection: A Dynamic Graph Approach to Immediate Cycle Collection. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on Memory Management* (Seoul, Republic of Korea) (*ISMM ’25*). Association for Computing Machinery, New York, NY, USA, 14–26. <https://doi.org/10.1145/3735950.3735953>
- [28] Sheng Liang. 1999. *Java Native Interface: Programmer’s Guide and Reference* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.
- [29] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.
- [30] Microsoft Corporation. 2023. *Platform Invoke (P/Invoke)*. Microsoft. <https://learn.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke> Accessed: 2026-06-30.
- [31] James D Mooney. 1993. Issues in the specification and measurement of software portability. In *15th International Conference on Software Engineering, Baltimore*.
- [32] James D. Mooney. 2004. Developing Portable Software. In *Information Technology*, Ricardo Reis (Ed.). Springer US, Boston, MA, 55–84.
- [33] K. V. Nori, Urs Ammann, Kathleen Jensen, H. H. Nageli, and Christian Jacobi. 1981. Pascal-P Implementation Notes. In *Pascal - The Language and its Implementation*, David W. Barron (Ed.). John Wiley, 125–170.
- [34] 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* (Tokyo, Japan) (*Programming ’23*). Association for Computing Machinery, New York, NY, USA, 3–6. <https://doi.org/10.1145/3594671.3594672>
- [35] Léonard Oest O’Leary, Mathis Laroche, and Marc Feeley. 2023. A R4RS Compliant REPL in 7 KB. (2023). <https://doi.org/10.48550/ARXIV.2310.13589> arXiv:2310.13589 [cs.PL] This paper was presented at the Scheme and Functional Programming Workshop (SFPW’23), part of ICFP’23.
- [36] OpenCode Contributors. 2026. *OpenCode: Open-Source Model-Agnostic AI Coding Agent*. Anomaly. <https://opencode.ai>

- [37] Zoe Paraskevopoulou and Andrew W. Appel. 2019. Closure conversion is safe for space. *Proc. ACM Program. Lang.* 3, ICFP, Article 83 (July 2019), 29 pages. <https://doi.org/10.1145/3341687>
- [38] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy efficiency across programming languages: how do energy, time, and memory relate?. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (Vancouver, BC, Canada) (*SLE 2017*). Association for Computing Machinery, New York, NY, USA, 256–267. <https://doi.org/10.1145/3136014.3136031>
- [39] T. Pittman. 1987. Two-level hybrid interpreter/native code execution for combined space-time program efficiency. In *Papers of the Symposium on Interpreters and Interpretive Techniques* (St. Paul, Minnesota, USA) (*SIGPLAN '87*). Association for Computing Machinery, New York, NY, USA, 150–152. <https://doi.org/10.1145/29650.29666>
- [40] Python Software Foundation. 2026. *ctypes — A foreign function library for Python*. <https://docs.python.org/3/library/ctypes.html>
- [41] Andreas Rossberg. 2019. *WebAssembly Core Specification*. Technical Report. W3C. <https://www.w3.org/TR/wasm-core-1/>
- [42] Vincent Scheib, Theo Engell-Nielsen, Saku Lehtinen, Eric Haines, and Phil Taylor. 2002. The demo scene. In *ACM SIGGRAPH 2002 Conference Abstracts and Applications* (San Antonio, Texas) (*SIGGRAPH '02*). Association for Computing Machinery, New York, NY, USA, 96–97. <https://doi.org/10.1145/1242073.1242125>
- [43] Zhong Shao and Andrew W. Appel. 2000. Efficient and safe-for-space closure conversion. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 129–161. <https://doi.org/10.1145/345099.345125>
- [44] Dimitrios Souflis, Kevin Cozens, and Jonathan S. Shapiro. 2006. *TinyScheme*. <https://tinyscheme.sourceforge.net/home.html>
- [45] Vincent St-Amour and Marc Feeley. 2010. PICOBIT: A Compact Scheme System for Microcontrollers. In *Implementation and Application of Functional Languages* (Berlin, Heidelberg, 2010), Marco T. Morazán and Sven-Bodo Scholz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17.
- [46] Scott Thibault, Charles Consel, Julia L. Lawall, Renaud Marlet, and Gilles Muller. 2000. Static and Dynamic Program Compilation by Interpreter Specialization. *Higher-Order and Symbolic Computation* 13, 3 (Sept. 2000), 161–178. <https://doi.org/10.1023/A:1010078412711>
- [47] Daniil Titov. 2016. Jep 289: Deprecate the applet API. <https://openjdk.org/jeps/289>

- [48] Justine Tunney. 2020. SectorLisp. Github Repository. <https://github.com/jart/sectorlisp>
- [49] Tomoharu Ugawa, Hideya Iwasaki, and Takafumi Kataoka. 2019. eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems. *Journal of Computer Languages* 51 (2019), 261–279. <https://doi.org/10.1016/j.cola.2019.01.003>
- [50] W3C WebAssembly Community Group. [n. d.]. *Component Model Design and Specification*. WebAssembly. <https://github.com/WebAssembly/component-model>
- [51] N. Wirth. 1996. *Recollections about the development of Pascal*. Association for Computing Machinery, New York, NY, USA, 97–120. <https://doi.org/10.1145/234286.1057812>
- [52] Michael Wolfe. 2005. How compilers and tools differ for embedded systems. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (San Francisco, California, USA) (*CASES '05*). Association for Computing Machinery, New York, NY, USA, 1. <https://doi.org/10.1145/1086297.1086298>
- [53] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) (*Onward! 2013*). Association for Computing Machinery, New York, NY, USA, 187–204. <https://doi.org/10.1145/2509578.2509581>
- [54] Haoran Xu and Fredrik Kjolstad. 2026. Deegen: A JIT-Capable VM Generator for Dynamic Languages. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 138 (April 2026), 29 pages. <https://doi.org/10.1145/3798246>
- [55] 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* (Chicago, IL, USA) (*VMIL 2021*). Association for Computing Machinery, New York, NY, USA, 14–24. <https://doi.org/10.1145/3486606.3486783>