UNIVERSITY OF KENT

DOCTORAL THESIS

Exploring the Implementation Space of Abstract Syntax Tree and Bytecode Interpreters

Author:
Octave LAROSE

Supervisor:
Dr. Stefan MARR

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Programming Languages and Systems Research Group

April 17, 2026

Declaration of Authorship

I, Octave LAROSE, declare that this thesis titled, “Exploring the Implementation Space of Abstract Syntax Tree and Bytecode Interpreters” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

(Octave Larose)

Abstract

Programming language *interpreters* are often the first choice when implementing a new language, and many well-established dynamic language implementations rely on them, such as Java, JavaScript, Ruby, and Python. Interpreters run code by compiling it to some program representation, then executing this representation at the software level.

Out of all possible interpreter designs, *bytecode* (BC) interpreters are often considered the fastest thanks to their compact and linear representation. This often overlooks other designs, such as *abstract syntax tree* (AST) interpreters, and the variety of hybrid designs that are neither clearly AST nor BC. Interpreter design is influenced by several constraints and opportunities, such as the mechanisms provided by the host language (in which the interpreter is written), the requirements of the guest language (the language being executed), but also available engineering effort, or hardware constraints. Overall, when implementing an interpreter, the number of viable design decisions may appear more limited than it truly is.

Therefore, this thesis endeavors to explore the implementation space of interpreters, to help provide a stronger understanding of the available high-level designs, design decisions, as well as their engineering and performance benefits. We focus on the design and performance of AST interpreters and BC interpreters for the SOM language, implemented on top of meta-compilation frameworks, and written in the Rust system-level language. We also describe approaches that lie somewhere in between AST and BC, and we identify key design dimensions for designing and discussing interpreters. Our results find that in terms of run-time performance, AST designs can rival BC designs on top of meta-compilation frameworks, and are not far behind in interpreters using the Rust language. We also find that the meta-compilation context can push BC designs closer to AST designs, while relying on Rust can push AST designs to be closer to BC instead. We synthesize our analysis by providing recommendations for interpreter design, based on the requirements, constraints, and opportunities of the runtime system in question. We hope this work encourages

language implementers to explore a wider range of designs, and assists in opting for design decisions that best meet their specific needs and requirements.

Acknowledgements

I would like to thank my supervisor, Stefan Marr, for his invaluable help throughout my PhD, and for having more patience than I ever will. Thank you for your advice, expertise, and ever-present support. This work would not have been possible without them. I would also like to thank my examiners, Dominic Orchard and Laurence Tratt, for their highly valuable feedback on this manuscript. I also thank Michael Vollmer for his supervision, and its sharp focus on recommending me obscure games from the 90s.

I am deeply grateful for having the family that I have, and I will always be thankful for their faith in me, which was prevalent during these past few academic years, but also before that, and after that (NB: presumably, no proof as of yet). Thank you to my parents, Zoé, and Val. Thank you to our cat Bagha as well, for thoughtfully helping me write this manuscript by trying to sleep on my keyboard, and repeatedly attempting to lick my face. I am not sure yet how that was helpful, but I hope to eventually figure it out.

On the other side of the Channel, I will remember these years spent in the UK fondly because of the great friends I made there: Sophie, Caio, Calvin, Jonah, and Max. Much advice and support was given, many great and terrible movies were watched, and many good times were had. Thank you for everything.

Thank you to everyone from our shared PhD office, and everyone in our department, for being all around great people. It was a pleasure to come to the office knowing the friendly faces I would get to interact with every day. I was also lucky enough to discover climbing while writing this thesis, and I am very grateful to the UKCMC climbing society for all that it has given me. Special thanks to Elliott, for being better than me both in climbing and in chess.

Of course, a special mention for Patrick, whose friendship and support have been a constant in my life since far before this academic adventure of mine. Far too many days of my life to count were made better through your wisdom, stupid jokes, and combinations of both. Thank you.

Last but not least, I thank my partner, Mila, for her ever-present support, curiosity, and enthusiasm. These last few years of working on this PhD were much brighter with you in my life, and I am deeply grateful for that. Thank you for putting up with my dumb jokes, for actually reading what I write, and for pushing me to be the best person that I can be. I am excited for whatever comes next.

Contents

Declaration of Authorship iii

Abstract v

Acknowledgements vii

List of Figures xv

List of Tables xvii

List of Listings xx

1 Introduction 1

1.1 Research Goals 3

1.2 Key Insights 4

1.3 List of Publications 5

2 Background 7

2.1 Interpreters 7

2.1.1 Abstract Syntax Tree 8

Visitor-based Execution 9

Naturally-Recursive Execution 10

Continuation-Passing Execution 12

2.1.2 Bytecode 13

2.1.3 Brief Discussion of Semantics and Terminology 14

2.1.4 AST versus Bytecode 15

2.2 Simple Object Machine (SOM) 15

2.2.1 Overview of the SOM language 16

2.3 Interpreter Optimizations 18

2.3.1 Common Interpreter Optimizations 18
Polymorphic Inline Caching 18
2.3.2 SOM-specific Optimizations 19
Global Variable Caching 19
Inlining Control Structures 20
Lowering of Basic Operations 20
2.3.3 Direct Handling of Trivial methods 21
2.3.4 Optimizations Relevant to Both AST and BC 21
Quickening (BC), Self-Optimization (AST) 21
Superinstructions (BC), Supernodes (AST) 23
2.3.5 BC-specific Optimizations 24
Threading 24
Stack Caching 25
2.4 Meta-Compilation 26
2.4.1 RPython: A Tracing Meta-Compiler 27
2.4.2 Graal: A Partial-Evaluation-based Meta-Compiler 28
2.5 Garbage Collection 30
2.5.1 Reference Counting 31
2.5.2 Tracing Garbage Collection 31
2.6 Overview of the Rust Language 32
2.6.1 Ownership 33
2.6.2 Borrowing 34
2.6.3 Unsafe Rust 34
2.6.4 Garbage Collection in Rust 35
2.7 The Are We Fast Yet Benchmark Suite 36

3 Overviewing The Interpreter Design Space **39**
3.1 Introduction 39
3.2 AST and BC: Commonly Held Opinions 40
3.2.1 Assumptions About Performance 40
3.2.2 Recursive versus Iterative 41
3.2.3 Non-Compact versus Compact 42

3.2.4 Inefficient versus Efficient Control Flow 43
3.2.5 Source-like versus Hardware-like 44
3.3 Hybrid Interpreter Designs: AST or BC? 44
3.3.1 Recursive Bytecode Interpreter Designs 45
3.3.2 Non-recursive AST Interpreter Designs 45
3.3.3 A Bytecode-like AST Interpreter Design 46
3.3.4 AST-like Bytecode Interpreters 47
3.3.5 Conclusion 48
3.4 Design Trade-offs: AST-like and BC-like 48
3.4.1 AST-like and BC-like: Host Language and Target Machine . . . 48
3.4.2 Intersection of AST-like and BC-like 50
3.4.3 Advantages of AST-like 51
3.4.4 Advantages of BC-like 53
3.5 Summary Table of Interpreter Design Dimensions 54
3.6 Related work 56
3.7 Conclusion 57

4 Interpreters on Meta-Compilation Systems 59
4.1 Overview of TSOM/PySOM 61
4.1.1 AST Interpreters 63
4.1.2 BC Interpreters 64
4.1.3 Optimizations Common to All Four Interpreters 66
Polymorphic Inline Caching 66
Global Variable Caching 67
Inlining Control Structures 67
Lowering of Basic Operations 68
Trivial Methods 68
4.1.4 Optimizations Orthogonal to Program Representation 69
4.1.5 Optimizations Applied to a Subset of Interpreters 70
Superinstructions 70
Quickening 70
Supernodes 71

4.1.6 Overview of Differences Between the Interpreters 71
4.2 Methodology to Compare AST and Bytecode Interpreters 73
4.2.1 Interpreter Performance 73
4.2.2 Peak Performance 73
4.2.3 Warmup Performance 74
4.2.4 Memory Usage 75
4.2.5 Impact of Optimizations 77
4.2.6 Measurement Methodology and Reported Statistics 78
4.3 Results of Comparing our Meta-Compilation-based AST/BC Interpreters 79
4.3.1 Interpreter Performance 79
4.3.2 Peak Performance 79
4.3.3 Warmup Behavior During JIT Compilation 81
4.3.4 Memory Usage 83
Overall Memory Impact 83
Memory Usage for Program Representation 84
4.3.5 Impact of Optimizations 85
Impact on Interpreter Performance 85
Impact on Peak Performance 87
4.4 Result Discussion 88
4.4.1 Results and their Generalizability 89
4.4.2 Limitations of this Study 92
4.5 Related Work 93
4.6 Conclusion 95

5 Interpreters in Rust 97

5.1 Introduction 97
5.2 Overview of the SOMrs Implementations 98
5.2.1 High-level Overview 99
5.2.2 AST 100
5.2.3 BC 102
5.2.4 Differences with Interpreters Based on Meta-Compilation . . . 103
5.2.5 Implemented Optimizations 104

5.3 Garbage Collection in SOMrs 106
5.3.1 MMTk: Memory Management ToolKit 107
5.3.2 Overview of our Garbage Collection Approach 108
5.3.3 Ensuring Soundness With Moving GC 109
Example of Unsoundness from Moving GC 109
Primitives Only Relying On The Universe On a Need-to-use Basis 111
AST: Finding Roots, and the Host Language Stack 113
AST: Storing Nodes on the Rust Heap 114
Self-Referential Frames 116
5.4 Methodology to Compare the SOMrs Interpreters 117
5.4.1 Interpreter Performance 118
5.4.2 Memory Usage 118
5.4.3 Experimental Setup 119
5.5 Results of Comparing our SOMrs AST and BC Interpreters 119
5.5.1 Run-time Performance 120
5.5.2 Memory Usage 121
Total allocations 121
5.5.3 Program Representation Size 122
5.6 SOMrs: Interpreter Design Discussion 123
5.6.1 Design Mismatch: Rust and AST-Like Interpreters 124
Mutable Tree Structures in Rust 124
Finding GC Roots on the Rust Stack 125
Serialized AST Interpreter 126
5.6.2 BC Interpreter Design in Rust 127
Bytecodes Not Variable Width 127
Threaded Code 128
5.7 Conclusion 128

6 Interpreter Design Recommendations **131**
6.1 High-level Recommendations 131
6.1.1 Consider Hybrid Approaches 132
6.1.2 Limited Engineering Resources: Utilize Host Language 132

6.2 Recommendations for Interpreters Based On Meta-Compilation 133
6.2.1 Interpreter Performance: Both AST and BC Viable 133
6.2.2 Peak Performance: Likely Orthogonal to AST or BC 134
6.2.3 Non-Compact Representation: Benefits Dynamic Rewritng and Localizing Profiling Information 135
6.3 Recommendations for Rust-based Interpreters 136
6.3.1 Best Performance: Bytecode Interpreters are Likely Preferable . 136
6.3.2 Good Performance: AST Interpreters can Remain an Option . . 137
6.3.3 Recommendation for AST: Non-Recursive Design 138
6.3.4 Mismatch With Host Language Needs Consideration 138
6.4 Conclusion 139

7 Conclusion 143
7.1 Future Work 146

A Background (Chapter 2) 149
A.1 Overview of the Are We Fast Yet benchmarks 149

B Meta-Compilation Systems (Chapter 4) 151
B.1 Abstract Syntax Tree Nodes 151
B.2 Bytecode Set 152

C Rust-based Interpreters (Chapter 5) 155
C.1 Abstract Syntax Tree Nodes 155
C.2 Bytecode Set 155
C.3 Performance comparison between SOMrs and TSOM/PySOM 157

Bibliography 159

List of Figures

2.1 Example represented as AST and bytecode. 9

3.1 Spectrum from AST-like to BC-like, with characteristics that may more closely identify with one end than the other. The left end *benefits from the host language*, and the right end *minimizes distance with the target machine*. 49

4.1 Interpreter run-time performance, on a logarithmic scale, with Java as baseline. While TSOM and PySOM are overall slower than the Java and Node.js interpreters, we can also observe that $\text{PySOM}_{\text{AST}}$ and TSOM_{AST} are faster than the bytecode versions. TSOM_{BC} is the slowest interpreter overall. 80

4.2 Peak performance with just-in-time compilation enabled, a logarithmic scale, with Java as baseline. TSOM_{AST} reaches the performance of Node.js, while the peak performance of the other implementations is a little further behind. 80

4.3 Performance of benchmark iterations over time for the TSOM interpreters as measured on the ahead-of-time-compiled GraalVM Native Image. All data is normalized to the median of TSOM_{AST} and shown with a logarithmic y-axis. While we see performance difference between TSOM_{AST} and TSOM_{BC}, only the Richards benchmark seems to display a noticeable warmup delay on TSOM_{BC}. 81

4.4 Performance of benchmark iterations over time for the PySOM interpreters. All data is normalized to the median of $PySOM_{AST}$ and shown with a logarithmic y-axis. While we see performance difference between $PySOM_{AST}$ and $PySOM_{BC}$, there does not seem to be a major difference in warmup. 82
4.5 Maximum Resident Set Size (RSS) used by the interpreters, on a logarithmic scale. 83
4.6 Performance impact of removing individual optimizations on the interpreter-only speed of the fully optimized interpreters. Run time normalized to the implementation with all optimizations, on a logarithmic scale. Optimizations marked with (*) are specific to AST or bytecode interpreters. Overall, inlining of control structures, inline caching, and lowering of basic operations give the highest benefit. . . . 86
4.7 Performance impact of individual optimizations on the peak performance of the fully optimized interpreters. Run time normalized to the implementation with all optimizations, on a logarithmic scale. Optimizations marked with (*) are specific to AST or bytecode interpreters. Inline caching gives the highest benefit on TSOM, because there it is needed for inlining across SOM methods. PySOM benefits more from inlining of control structures. 87

5.1 Interpreter run-time performance of the Are We Fast Yet benchmarks, on a logarithmic scale, with Java as baseline. SOMrs-AST and SOMrs-BC are roughly at the same level, and are in the same ballpark as CPython 3.10. 120
5.2 Total number of bytes allocated, with SOMrs-BC as a baseline. 121
5.3 Comparison of the memory needed by both SOMrs interpreters to represent the program itself, with SOMrs-BC as the baseline. Higher is worse. 122

C.1 Interpreter run-time performance of the Are We Fast Yet benchmarks, on a logarithmic scale, with Java as baseline. We compare both SOMrs interpreters, TSOM interpreters and PySOM interpreters. 157

List of Tables

2.1 Overview of all Are We Fast Yet benchmarks. The five macro-benchmarks are at the top, and the nine micro-benchmarks underneath. 37

3.1 Table of design dimensions for AST-like and BC-like interpreters, highlighting traits that we identify to be associated with either design. . . . 54

6.1 Summary table for our design recommendations in this chapter. 140

A.1 Code Size Metrics for all Are We Fast Yet benchmarks. 150

A.2 Receiver polymorphism for the Are We Fast Yet benchmarks, i.e. number of potential receivers at any call site. 150

B.1 SOM AST Nodes including Supernodes and Variants. 153

B.2 SOM Bytecode Set with all Superinstructions and Quickening Variants. 154

C.1 SOMrs-AST Nodes including Supernodes and Variants. 156

C.2 SOMrs-BC Bytecode Set. 158

List of Listings

2.1 AST-based interpreter in Rust (simplified slightly), adding the integers 1 and 2. Nodes implement the common `Node` trait. The `Literal` nodes represent values and the `AddNode` adds the results of two subexpressions. 11
2.2 BC-based interpreter in Rust, adding the integers 1 and 2. The bytecodes encode pushing literal values onto the stack, and then add them with an `ADD` instruction. 12
2.3 Example code for the SOM programming language. The `fib:` method returns the *N*th number in the Fibonacci sequence. 16
2.4 The `whileTrue:` method used for *while* loops in SOM defined in the `Block` class. 17
2.5 Sequence of five basic bytecode instructions that is equivalent to the `INC_FIELD_PUSH` superinstruction. 23
2.6 The `IntInc` supernode, which combines an `Addition` node, specializes it to integers, and stores an integer value for the addition. 24
2.7 Example of a BC-based interpreter designed using RPython. 28
2.8 Example of an AST node in a Truffle/Graal-based interpreter. 29
2.9 Example of ownership in Rust. 33
2.10 Example of references and borrowing in Rust. 34
5.1 Part of the AST representation used in SOMrs-AST, stored in a Rust `enum` type. 100
5.2 Part of the SOMrs-BC bytecode set. 102
5.3 Example of unexpected undefined behavior from moving GC, which relies on `unsafe` Rust code. 109
5.4 Example breaking SOMrs-AST code, where we miss the `receiver` root since it is isolated on the Rust stack. 113

5.5 The `evaluate` method in SOMrs-AST for any sequence of nodes, invoked for every block and method. 115
5.6 A example of a frame in SOMrs-BC. Frames in SOMrs-AST are almost identical, though e.g. do not need to store a bytecode index. 116

Chapter 1

Introduction

A programming language *interpreter* is software that executes (i.e. interprets) code written in a guest language, instead of targeting a specific hardware machine by compiling to native code. Interpreters instead typically have a compilation phase to some program representation, and execute this representation themselves directly at the software level.

By relying on an interpreter, execution takes place entirely at the software level as opposed to the hardware level. This typically yields slower execution speeds than by compiling to native code, which targets the underlying hardware directly. For this reason, a common option is to pair interpreters with Just-In-Time (JIT) compilers, which generate native code at run time. Dynamic native code generation relies on assumptions about the execution of the program to generate efficient code, e.g. "this dynamic dispatch always dispatches to the same method, and so we may only consider this case", therefore assumptions which require programs to be executed and profiled before JIT compilation. Moreover, since native code generation is time-consuming, most runtimes want to limit native code generation to only frequently executed code paths, which also need the program to be executed beforehand. Therefore, interpreters can perform the profiling needed by JIT compilers, and so complement them very well.

A typical execution pipeline may thus start off in the interpreter, which will identify valuable compilation targets, so that native code can be generated for them. Interpreters are therefore essential components of runtime systems, but can sometimes be perceived as a stepping stone in the execution pipeline, or what execution falls back to if one compiled based on faulty assumptions.

Interpreters thus tend to receive less attention than JIT compilers, but offer several advantages:

- implementing an interpreter requires far less engineering effort, making them easier to debug and to experiment with;
- their ease of implementation also makes them valuable reference implementations;
- since there is no native code compilation phase, little time is spent between invoking the runtime system and code being executed, which is beneficial e.g. for frequently redeployed applications;
- their code representation can be more compact than native code, which can reduce memory usage (Franz et al. 1997; Pichler et al. 2025);
- they can execute code even when dynamic native code generation is disallowed, which can occur for security reasons.

Therefore, the design of interpreters is worth investigating in more detail, with or without the possibility of JIT compilation. The design of an interpreter is influenced by several constraints and opportunities, such as the mechanisms provided by the host language (the language in which the interpreter is written), the requirements of the guest language (the language being implemented/executed), but also the available engineering effort that can be expended, or possible hardware constraints.

The interpreter design space appears broad, but literature often fixates on a specific design subset: *bytecode* (BC) interpreters, which rely on a compact and linear representation, and are often deemed to be the optimal design choice since they are meant to provide the best possible performance. This is often to the exclusion of other designs, such as *abstract syntax tree* (AST) interpreters, which instead rely on a tree-based representation, and are largely considered to provide worse performance. There also are a variety of hybrid designs that cannot be easily described as either AST or BC, and which are far less frequently considered or documented in the literature. Overall, when implementing an interpreter, the amount of possible designs and design decisions may appear more limited than they truly are. Therefore, this work explores and documents the implementation space for various interpreter designs.

1.1 Research Goals

As we shall see, the implementation space of interpreters is large and complex. Therefore, we work towards better defining this design space by identifying key design dimensions within it, and analyzing different interpreters which feature a variety of these different design choices. To this end, we measure notable concrete metrics, such as run-time performance and memory usage. Moreover, the dominance in use of bytecode interpreters overshadows several design options, such as AST interpreters or hybrid AST/BC designs, notably since bytecode designs are assumed to offer better run-time performance.

As such, a key question we address throughout this work is:

Can AST interpreters be as performant as BC interpreters?

We specifically investigate the design and performance of AST and BC interpreters based on meta-compilation, and system-level languages. Throughout this work, we point out various possible design choices and provide an overview of their associated trade-offs, with the goal of facilitating discussion on interpreter design as a whole.

We focus on the following research questions:

1. **What are the key design dimensions to define the design space of programming language interpreters?** The interpreter design space can appear to be representable as a binary split between AST and BC interpreters. We argue that this is not the best way to represent it, since it fails to account for numerous potential hybrid approaches and so limits design possibilities. Therefore, Chapter 3 gives a broad overview of the interpreter design space, argues against a binary view of interpreter design to instead qualify interpreters as more or less *AST-like* and *BC-like*, and establishes several key design dimensions associated with both terms, to define and reason about the interpreter design space.

2. **How does meta-compilation influence the design and performance of interpreters?** We argue that *meta-compilation* is an essential domain in which to analyze the design of interpreters, since the interpreter is the only part of the system that is manually written, with the JIT compiler being derived from it automatically.

Therefore, Chapter 4 focuses on meta-compilation, and the performance of a strongly AST-like and a strongly BC-like interpreter using these frameworks. We analyze whether bytecode designs necessarily give better performance when dealing with the constraints of these systems.

3. **How does using the Rust language influence the design and performance of interpreters?** Many interpreters are written in system-level languages to achieve the best possible performance. Therefore, after focusing on interpreters written in Java and Python to work with meta-compilation systems, we turn our attention to interpreters written in a language where we achieve faster execution speeds, namely Rust. Chapter 5 thus focuses on the influence of Rust on the design and performance of interpreters written in it, and as with the previous chapter, compares a strongly AST-like and strongly BC-like interpreter.

Based on our findings from each chapter, we give various recommendations for designing interpreters, based on the requirements and constraints of the end user. These recommendations are formulated in Chapter 6.

1.2 Key Insights

As we explore the interpreter design space throughout this thesis, we extract several key insights.

Interpreter Design is Not a Binary Choice Between AST/BC. While these designs are often the only two that are considered, we highlight that a binary view is not an ideal model, since it fails to consider hybrid designs that exist in between both approaches, and exhibit valuable trade-offs. So, we define the terms AST-like and BC-like, as well as a set of key design dimensions associated with both terms, to help communicate design choices more efficiently than with a reductive yet commonly accepted binary model. We describe these identified major design dimensions in section 3.5.

Meta-Compilation: AST Interpreters Can Rival BC Interpreters In Terms Of Run-Time/Peak Perf. While we mentioned that AST interpreters are commonly considered to be outperformed by BC interpreters in terms of run-time performance, we find that this is not always the case. In Chapter 4, we focus on interpreters based on meta-compilation, and we find that by comparing highly optimized AST interpreters and highly optimized BC interpreters, the AST interpreters can rival their BC counterparts in terms of run-time performance, as well as in terms of peak performance achieved with JIT compilation.

System-level language: Good Performance Achievable with Optimized AST Interpreter. We compare optimized AST and BC Rust-based interpreters in Chapter 5. We find that bytecode designs outperform AST interpreters in this context, which aligns with the literature. Though, we also find that the difference in performance between an optimized AST interpreter and an optimized BC interpreter is relatively small, around 16%.

Rust: Design Mismatch With Recursive Interpreters. While we use the Rust language to achieve good run-time performance in our interpreters in Chapter 5, we find that this choice of host language can make some design decisions detrimental to some interpreter designs. Most notably, we find that traditional designs for AST interpreters are not a good fit for Rust , therefore, that the design was at odds with the host language. This highlights a mismatch between host language and interpreter design, related to the idea of semantic mismatch between host and guest language, which we find to be relevant to discussing interpreter design.

We hope that these insights and our work as a whole benefit interpreter implementers, and assist in selecting designs that best meet their specific needs and requirements.

1.3 List of Publications

This thesis was based on the following works:

1. O. Larose et al. (2025). "AST, Bytecode, and the Space In Between: An Exploration of Interpreter Design Tradeoffs". In: *The Journal of Object Technology*. ICOOOLPS'25 24.3 (accepted, to be published). This paper focused on analyzing the interpreter design space as a whole, and deriving recommendations based on our observations and performance experiments. This content provided the foundation for Chapter 3, and guides the structure of the thesis as a whole.

2. O. Larose et al. (Oct. 2023). "AST vs. Bytecode: Interpreters in the Age of Meta-Compilation". In: *Proceedings of the ACM on Programming Languages*. OOPSLA'23 7.OOPSLA2, pp. 318–346. ISSN: 2475-1421. DOI: `10.1145/3622808`. This work focused on interpreters in the domain of meta-compilation, and served as the foundation for Chapter 4.

3. O. Larose et al. (Mar. 2022). *Less Is More: Merging AST Nodes To Optimize Interpreters*. This work defined supernodes, an optimization which was implemented in all of the AST interpreters which we talk about in this thesis.

4. O. Larose (June 2025). "Adding garbage collection to our Rust-based interpreters with MMTk". In: *Workshop on Modern Language Runtimes, Ecosystems, and VMs*. MoreVMs'25. Peer-reviewed blog post on the implementation of garbage collection in our Rust-based interpreters, which we expand on in Chapter 5.

5. Additionally, several parts of Chapter 5 on the design of our Rust-based interpreters relate to blog posts written by myself at `https://octavelarose.github.io/`. Those are "Spending too much time optimizing for loops" and "Inline caching in our AST interpreter".

Chapter 2

Background

This chapter aims to provide the necessary context to understand this work and its contributions. We first provide an overview of programming language interpreters in section 2.1, defining both the abstract syntax tree and bytecode interpreter formats. We then introduce the SOM language in section 2.2, which our interpreters execute. The guest language of an interpreter has a large impact on which interpreter optimizations can be implemented, thus we go over these optimizatons in section 2.3. We also give an overview of meta-compilation in section 2.4, as it is a key part of Chapter 4, before focusing on garbage collection in section 2.5 and the Rust language in section 2.6, which are highly relevant to chapter 5. Finally, since our experiments throughout this work rely on the same benchmark set, we briefly introduce it in section 2.7.

2.1 Interpreters

A programming language *interpreter* is software that executes, i.e. interprets, code of a guest language. This is in contrast to *Ahead-Of-Time* (AOT) compilers, which generate native executables that are executed by a CPU. Interpreters, after typically compiling to an intermediate representation, execute the representation themselves at the software level. As a result, such software can also be referred to as *virtual machines* (VMs), or *runtime systems*. Since terminology related to interpreters can cause confusion, and can raise questions such as what qualifies or not as an interpreter, we briefly address these semantic questions in section 2.1.3.

Since interpreting an intermediate representation is less efficient than executing native code, many high-level language virtual machines implement not only an

interpreter but also a *Just-In-Time* (JIT) compiler to eventually take over the execution of a program. At run time, JIT compilers compile the most frequently executed parts of a program to native code. For dynamic languages or use cases where AOT compilation is impractical, JIT compilers allow a language implementation to reach high performance regardlessly. Since they can offer better run-time performance, JIT compilers often overshadow interpreters, which is why the V8 JavaScript engine originally did not include an interpreter.[1] Though, interpreters are highly valuable when run-time compilation is too expensive, e.g., during program startup, or when native code compilation is limited, impractical, or impossible.

Interpreters are usually categorized based on the program representation they execute. The most common ones, and therefore the most common interpreter types, are *abstract syntax tree* interpreters and *bytecode* interpreters.

2.1.1 Abstract Syntax Tree

Programming languages are often parsed into an *abstract syntax tree* (AST) to represent the program syntax as a tree. Each element is mapped to a node of a specific type. This makes them an intuitive design for representing a program. For example, the node type for a source code element can implement an operation to realize its run-time semantics.

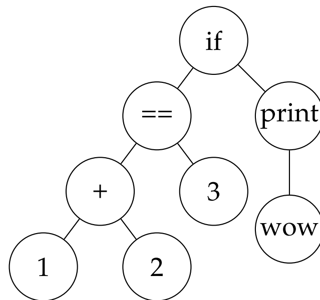
While an AST can be a simple parse tree, i.e., a one-to-one mapping of the program syntax to a tree representation, most ASTs abstract away from the original source to various degrees. This may be done to e.g. represent high-level concepts based on more basic ones, or to optimize performance. A common optimization is lexical addressing (Abelson et al. 1996; Kalibera et al. 2014). Instead of qualifying variables by name, it is often more efficient to represent variables with indices into some array. Other run-time information, e.g., the result of method lookups, can also be cached in the tree as polymorphic inline caches (Hölzle et al. 1991). These optimizations are not unique to AST interpreter designs, and we later expand on them in section 2.3.

So, while an AST remains close to the syntax of the source code, when used as an interpreter representation, it may differ in some ways from the original syntax to allow for more efficient interpretation. One could then refer to these trees as *execution*

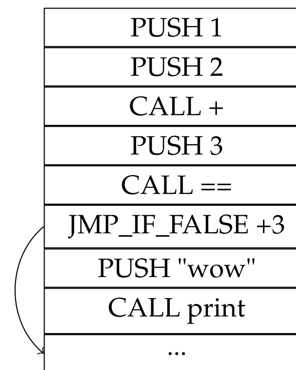
[1] https://v8.dev/blog/ignition-interpreter

```
1 if 1 + 2 == 3 {
2     print("wow")
3 }
```

(A) Source code of the example program



(B) As AST



(C) As Bytecode

FIGURE 2.1: Example represented as AST and bytecode.

trees (Roberts et al. 2019). But since AST is the more common term, we refer to these trees as such for the rest of this thesis.

Figure 2.1b shows a simple AST for the code shown in Figure 2.1a. The root node `if` corresponds to the `if` statement shown in Figure 2.1a, which has two children, one which represents the condition, and the other the body. Usually, execution is defined by a pre-order traversal of the tree: the root `if` node will be visited first, which will in turn check the condition by visiting the `==` node, whch then visits the `+` node, which finally visits its two children in turn: `1` and `2`. The `+` node will perform an addition on the two children, and propagate the value back to the `==` node. The `==` node then compares the value to the value its other child, the `3` node, evaluates to true or false, and execution will then return the result of the comparison to the parent, i.e., the `if` node. Execution will continue from the `if` node with a call to the `print` node if the condition was true, which is always the case in our naive example, or go back to the parent of the `if` node if the condition was false.

Visitor-based Execution

The *visitor* design pattern (Gamma et al. 1994) is often associated with AST interpreters (Nystrom 2021 ch. 5), since it provides a flexible way to implement the semantics of a language in an object-oriented manner. A *visitor* class contains one method

for each node type, which implements the behavior for the corresponding nodes. To execute a program, the visitor walks the tree and for each encountered node, it calls a common method on the node, often called `accept`, passing in the visitor, which in turn calls the specific `visit` method associated with its own type on the visitor, passing itself as an argument so that it can be read from.

This design has engineering benefits in that it allows for easily defining several different sets of operations on the AST. One visitor can interpret the program, another print out the tree structure, and a third might apply optimizations. Each visitor then contains the specific implementation of, for instance a `visitIf` method to either implement the control-flow semantics, print it, or perhaps attempt to constant fold the condition and replace the `if` node with the remaining branch.

Naturally-Recursive Execution

An alternative implementation style in object-oriented languages foregoes the engineering benefits of the visitor pattern and implements `execute` methods directly on the nodes. This design has potential performance implications from removing the visitor, and so avoiding the indirect method calls that it performs. Listing 2.1 illustrates this interpreter variant. Here, the AST consists of an `AddNode` with two children, `Literal(1)` and `Literal(2)`. Execution still traverses the tree in pre-order by calling `execute` on the `AddNode` first. `AddNode.execute` then calls `execute` on each of the two children, and returns the addition of their results. The `execute` method takes in a *frame*, which refers to a call frame, also called a context. For a guest language subroutine call, the call frame stores the local variables, arguments, and general information needed to execute the function.

This tree-based design is elegant in the sense that it directly leverages functionalities of the host language, such as recursion and dynamic dispatch for node behavior.

This naturally-recursive style of interpreters is favored by the GraalVM and Truffle projects (Würthinger et al. 2017, 2012), which started out by implementing a wide range of languages in this style on top of Java. The TruffleDSL (Humer et al. 2014a) further simplifies the implementation of interpreters in this style and makes it easier

```
1  trait Node {
2     fn execute(&self, frame: &mut Frame) -> Value
3  }
4
5  impl LiteralNode {
6    fn new(value: Value) -> Self {
7      Self { value }
8    }
9  }
10
11 impl Node for LiteralNode {
12   fn execute(&self, frame: &mut Frame) -> Value {
13     self.value.clone()
14   }
15 }
16
17 impl AddNode {
18   fn new(child1: Node, child2: Node) -> Self {
19     Self { child1, child2 }
20   }
21 }
22
23 impl Node for AddNode {
24   fn execute(&self, frame: &mut Frame) -> Value {
25     self.child1.execute(frame) + self.child2.execute(frame)
26   }
27 }
28
29 fn main() -> Value {
30   let tree = AddNode(Box::new(LiteralNode::new(1)),
31                      Box::new(LiteralNode::new(2)));
32   tree.execute(Frame())
33 }
```

LISTING 2.1: AST-based interpreter in Rust (simplified slightly), adding the integers 1 and 2. Nodes implement the common `Node` trait. The `Literal` nodes represent values and the `AddNode` adds the results of two subexpressions.

to benefit from meta-compilation in the form of just-in-time compilation. We expand on these projects in section 2.4.2.

Continuation-Passing Execution

Another way to implement an AST interpreter is in a *Continuation Passing Style* (CPS), a technique originating from Scheme (Sussman et al. 1975). When using CPS, functions will take an extra argument in the form of a *continuation*, which corresponds to the program control state. The final expression in such a function will be a call to the continuation, after providing it the return value computed by the current function.

The call to the continuation is the last thing to occur in the function itself, making it a tail call. If the host language guarantees *tail call optimization*, this form of recursion does not consume stack space of the host language and compiles the call to a jump instruction instead.

This design naturally lends itself to walking a recursive data structure like a tree, while allowing for the implementation of language features, e.g. recursion or green threads, without relying on the host language.

```
1 let bytecodes = [PUSH_LIT, 0, PUSH_LIT, 1, ADD];
2 let literals = [1, 2];
3 let mut stack = vec![];
4 let mut i = 0;
5
6 loop {
7   bc = bytecodes[i];
8   i += 1;
9
10  match bc {
11    PUSH_LIT => {
12        let lit_idx = bytecodes[i];
13        stack.push(literals[lit_idx]);
14        i += 1;
15    }
16    ADD => {
17      let (val2, val1) = (stack.pop(), stack.pop());
18      stack.push(val1 + val2)
19    }
20    // ...
21 }
```

LISTING 2.2: BC-based interpreter in Rust, adding the integers 1 and 2. The bytecodes encode pushing literal values onto the stack, and then add them with an ADD instruction.

2.1.2 Bytecode

Bytecode is a virtual instruction set designed to be linear and compact, similar to hardware instruction sets. It is often obtained by compiling the source code into some tree representation (an AST, though one not meant for direct execution), then doing a pre-order traversal to linearize it into a sequence of instructions. In part because of this additional compilation phase to bytecode, a BC interpreter typically requires more engineering effort than an AST interpreter, from having to ensure correct instruction generation and correct traversal during execution.

The concept of bytecode can be traced back to the BCPL language and its *O-code* (Richards 1969), with Smalltalk likely being the first language to use the term *bytecode* instead (Bush et al. 1987). On Smalltalk bytecode, Béra et al. (2016) write "The virtual code [...] is encoded in bytes for compactness. Its byte encoding gives it the name bytecode". As its name suggests, bytecode is designed to be a compact enough byte-based program representation. Bytecode sets require careful design to generate short and efficient sequences of bytecodes to represent the program semantics, as to minimize execution times, while ensuring the bytecode set itself does not grow too large to be encoded as compactly as possible.

Figure 2.1c shows a simple bytecode sequence where each bytecode has an *opcode* representing its type (`PUSH`, `CALL`, etc.) and one or more arguments. Each bytecode instruction is executed in turn, one after the other. This is a *stack-based* bytecode, where values are pushed onto a stack and consumed by other operations, such as the bytecode `CALL` which consumes one or more arguments from the stack and pushes its result onto the stack. Jumps can redirect execution to any other bytecode, either further back or further ahead. Here, after comparing both values using `CALL ==`, the result is on the stack and then consumed by `JMP_IF_FALSE`. If the result was `true`, no jumps will occur and execution will continue at the `PUSH "wow"` point further ahead. Otherwise, execution continues after the `CALL print` bytecode, i.e. the next bytecode in the sequence.

In this simple example, a value was put on the stack by the `CALL ==` instruction, so that it could then be used by the `JMP_IF_FALSE` instruction. In other interpreter designs, it could also have been stored in a register, or in a top-of-stack cache in a

hybrid design (Ertl 1995). Register-based approaches are generally faster than stack-based approaches (Shi et al. 2008; Zhang et al. 2022), though they can incur additional implementation complexity from having to deal with register allocation (Chaitin et al. 1981) when generating bytecode.

Listing 2.2 shows example code for a traditional bytecode interpreter for a stack-based machine. The main bytecode loop iterates over all bytecodes. First, the next bytecode is fetched based on the current bytecode index (`i`), then a `match` expression executes the bytecode handler for the corresponding bytecode. If it is `PUSH_LIT`, the literal index stored at the subsequent index is used to fetch the corresponding literal value, which is then pushed onto the stack. When `ADD` is later encountered, its associated bytecode handler takes the last two values from the stack, adds them together, and pushes the result back onto the stack.

In terms of run-time performance, bytecode interpreters have received a lot of attention over the years, and we go over several BC-specific optimizations in section 2.3.5.

2.1.3 Brief Discussion of Semantics and Terminology

This work focuses on *programming language* interpreters, therefore interpreters that take code written in a programming language as input. We previously briefly contrasted ahead-of-time compilers and interpreters in section 2.1, with interpreters directly executing a program, and AOT compilers first translating (i.e. *compiling*) the input into some other language, often machine code that can be offloaded to the CPU. Yet most interpreters do *not* directly parse and interpret the input program as they read the source file, and instead translate it beforehand to an intermediate format which is often AST or BC, which will be interpreted. In other words, the guest language is *compiled ahead of time*, but to the runtime representation chosen by the interpreter instead of machine code.

We use *interpreter* as a shorthand for “programming language interpreter”, and we define it as *software* (thus never an abstract machine, i.e. a theoretical model of computation, but it may be an implementation of an abstract machine) that takes in a program written in some guest language. Our interpreters all feature *compilation*

steps, with "compilation" referring to the broad idea of translating one format to another, which we most often use to mean compilation of source code to AST or bytecode, though also sometimes compilation to machine code, e.g. when we refer to JIT compilation.

2.1.4 AST versus Bytecode

ASTs can sometimes be thought of as a format that is only present in the compilation pipeline, as parser output that should be compiled to bytecode, and which should not be directly executed itself. Since bytecode is commonly thought to be a more efficient representation in terms of memory usage that leads to better run-time performance, AST interpreters are often discounted for BC interpreters. As a practical example, Ruby's implementation started out as an AST interpreter, but was eventually replaced by a bytecode interpreter specifically to increase run-time performance (Sasada 2005). The common assumption that good run-time performance mandates a BC design also explains the comparatively lower amount of optimizations for AST interpreters as opposed to BC interpreters, later exemplified by section 2.3 describing more BC-specific optimizations than AST-specific optimizations.

We will question this assumption about the run-time performance of AST interpreters throughout this work, as we do not find bytecode interpreters to yield clearly superior performance in some contexts, or the performance gap between bytecode and AST to always be as wide as one may presume. We will also highlight various engineering benefits from AST interpreter designs, which should not be easily discounted.

2.2 Simple Object Machine (SOM)

The Simple Object Machine language (SOM) is a class-based language and minimal Smalltalk dialect (Haupt et al. 2010)[2]. It is dynamically typed, has class inheritance, closures, and non-local returns. It thus shares many implementation and especially optimization challenges of more commonly used languages such as JavaScript, Python, and Ruby, without reaching their complexity. Given that this work relies on

[2] https://som-st.github.io/

many interpreters to compare them with one another, using such complex guest languages would induce an unrealistic amount of engineering effort and thus make this work infeasible, while SOM helpfully allows us to address similar design challenges with a smaller language specification.

2.2.1 Overview of the SOM language

```
1 Fibonacci = (
2     fib: n = (
3         ^ n <= 1
4             ifTrue: 1
5             ifFalse: [ (self fib: (n - 1)) + (self fib: (n - 2)) ]
6     )
7 )
```

LISTING 2.3: Example code for the SOM programming language. The `fib:` method returns the *N*th number in the Fibonacci sequence.

Listing 2.3 shows an example of SOM code. This code defines the class `Fibonacci` which as with every class in SOM, is a subclass of the `Object` class. Classes contain fields and methods, and this class defines the method `fib:`, which takes in one argument `n`.

Methods have a receiver and any number of arguments, and each colon in the method name specifies that one argument should follow it. For instance, some method "`#do:with:`" would take two arguments, and be invoked on some receiver object. To invoke it on the receiver `a` with the two arguments `b` and `c`, in e.g. Java, one would typically write "`a.doWith(b, c)`". In SOM, a call to this method would instead take the form "`a do: b with: c`". A method call that takes no argument "`unaryCall`", would typically be invoked e.g. in Java with the syntax "`a.unaryCall()`". In SOM, instead, it would have no colon in its name and calling it would take the form "`a unaryCall`".

In our example, with "`(n <= 1) ifTrue: ... ifFalse: [ ... ]`", the receiver is `n <= 1` (very likely a `Boolean` object), the name of the method is "`ifTrue:ifFalse:`", and its arguments are both represented with "`...`". Additionally, "`^`" corresponds to a method return, and code blocks are represented with square brackets `[]`.

As with other dynamic languages, no type for the value of the argument `n` is enforced, which can therefore be any value. If instead of an integer we pass e.g.

a string, the operation `n <= 1` would immediately cause a runtime error and halt execution, since the `String` class does not define `<=` and `-` methods. But the "`fib:`" method would work if provided a floating point value instead of an integer, or some `CustomInteger` class granted that it implements the required methods. Therefore, it is difficult for SOM runtime systems to assume the type of a given value based entirely on static information, and optimizations rely rather on observations made during run time.

To return the correct result, "`fib:`" recursively calls itself, with method calls referred to as *message sends*. Any message send in SOM first checks that the receiver can receive the message, e.g. in this case checking that `self` refers to an object which implements the method "`fib:`", and then dispatches with the provided arguments. Critically, as with Smalltalk, *everything is an object*, and so every operation in SOM is a message send to an object. A message send is either to a SOM method or to a *primitive*, which is standard terminology for a method implemented at the VM-level because it requires access to information internal to the VM, or for performance reasons. Notably, control flow mechanisms are not exempted from this rule. In our example, "`ifTrue:ifFalse:`" is not some special keyword, but is itself a message send to the SOM method of the same name that is defined in the `Boolean` class.

```
1 whileTrue: block = (
2     self value ifFalse: [ ^nil ].
3     block value.
4     self restart
5 )
```

LISTING 2.4: The `whileTrue:` method used for *while* loops in SOM defined in the `Block` class.

As a more complex example, using a *while* loop in SOM is done using the "`whileTrue:`" or "`whileFalse:`" methods defined on the `Block` class, e.g. by doing "`[ check ] whileTrue: [ self doSomething ]`".

Listing 2.4 shows the implementation of the "`whileTrue:`" method. The condition of the loop is the receiver, i.e. `self`, and is evaluated using `self value`, with `value` being a primitive which executes a block and returns its final value. If this value is false, the loop is exited by returning `nil`, and if it is true, the body of the loop is executed with `block value`. Finally, the `restart` primitive is invoked, which returns

execution to the beginning of the current block/method, i.e. right before checking the loop condition, thus going back to the beginning of the loop.

These examples show just how ubiquitous message sends are in SOM, which is why optimizing them is essential to achieve good run-time performance in our VMs.

2.3 Interpreter Optimizations

Interpreter optimizations are design decisions that improve the performance of an interpreter, most notably its run-time performance. This section describes all major optimizations which are relevant to the design of our interpreters throughout this thesis. They fall into two broad categories: structural optimizations to essentially short-cut the generic SOM language semantics in well-known cases, and optimizations that use run-time feedback and thus profiling to improve performance.

The majority of the optimizations described here are present in all of our interpreters. When describing each of our interpreters in later chapters, we will justify which optimizations were implemented and how, as well as which were not implemented and why.

We first go over all common interpreter optimizations in section 2.3.1, then we focus on SOM-specific optimizations in section 2.3.2, before further narrowing the scope by looking at optimizations specific to AST and BC designs in section 2.3.4 and section 2.3.5.

2.3.1 Common Interpreter Optimizations

Polymorphic Inline Caching

Hölzle et al. (1991) introduced Polymorphic Inline Caches (PICs). PICs store the lookup results at a given call site, thus avoiding the lookup cost on subsequent calls.

For instance, if a class implements the method `virtualMethod:`, any of its subclasses can then redefine a similarly named `virtualMethod:`. Any time a method with this name is called, we need to identify the receiver (the class or any of its subclasses), and lookup the method on it, such that it can be dispatched. Using PICs, for every call site, after first looking up the method, we cache this lookup result alongside who

the receiver class was. For any future calls to this specific method from this given call site, the VM can instead check whether there exists a cache entry for the requested receiver class, and if so immediately dispatch it without performing a method lookup. If the receiver is not yet known, a regular method lookup is performed, and a new cache entry for this new receiver is created.

The core assumption is that call sites will nearly always be *monomorphic* (only call one method) or be *polymorphic* (call several possible methods) but rarely require more than a few cache entries. In the pathological case, a call site becomes *megamorphic*, as in it is so highly polymorphic that caching and looking up from the cache becomes less efficient than performing the method lookup in the first place. To avoid this, a maximum number of cache entries is usually set, which varies depending on the VM. After this number is exceeded, the cache is discarded, and a normal lookup is done before each call.

Inline caching is a seminal optimization, and it is still highly relevant today (Kaleba et al. 2022). As SOM is a highly dynamic language with ubiquitous method calls, PICs are essential to achieve good interpreter performance.

2.3.2 SOM-specific Optimizations

We isolate optimizations specific to the SOM language in this section. These optimizations could also be applied to languages with similar properties as SOM, and since it is a Smalltalk dialect, they are also classic Smalltalk optimizations.

Global Variable Caching

Most languages have the notion of global variables, which can be accessed from any part of the code. In SOM, they are used e.g. to represent class objects, though they can also contain arbitrary values. Globals can be read like normal variables and are mutable via a hash-table-like reflective API. Furthermore, access to undefined globals triggers a reflective handler, which in the standard case will attempt to load a class with that name.

Since in most cases globals are only modified once, typically on loading a class or setting an initial value, global variable accesses rely on caches to speed up future accesses.

Inlining Control Structures

SOM uses classes and polymorphic methods to provide control structures, as seen in section 2.2. This includes the methods for conditional branches with `#ifTrue:`, `#ifFalse:`, and `#ifTrue:ifFalse:`, similar `nil`-specific methods with e.g. `#ifNil:`, methods for loops with `#whileTrue:`, `#whileFalse:`, plus the counting loop `#to:do:`, or logical operators with `#and:` and `#or:`.

While this design is a testament to the power of late binding, it makes achieving good performance more challenging, as this means every control flow operation is as expensive as a regular SOM method dispatch.

Therefore, all our interpreters optimize control structures by inlining them at the AST or bytecode level. Thus, our BC interpreters will for instance translate calls to methods for conditional branches into bytecodes when they have literal lambdas, and inline the lambdas into the method with the call, avoiding method call overhead for these lambdas.

Lowering of Basic Operations

As with control structures, basic operations such as arithmetic operations and comparison operators are provided as methods on classes. For example, the `Integer` and `Double` classes have methods for `+`, `-`, <, >= etc. Some of these methods are implemented as primitives directly in the interpreter. Others, including the >= methods, are implemented in the SOM code library as normal methods relying on primitive methods such as the one for <.

A basic SOM implementation would treat all these operations as normal polymorphic method calls, which is elegant but, because of the method call overhead, not particularly fast. Therefore, our interpreters provide primitive versions for methods implemented in SOM's core library to improve performance.

2.3.3 Direct Handling of Trivial methods

Our interpreters have the notion of *trivial methods*, a term which we use to refer to very simple methods such as getters and setters, where we can avoid some of the call overhead to speed up execution. We detect such methods during parsing, and mark them as special-case methods that can avoid allocating a frame object and instead directly perform their corresponding operations.

In our interpreters, a trivial method can be a method that only does:

- a trivial field read, e.g. returns the value of a field (getter method).
- a trivial field set, e.g. writes a value to a field (setter method).
- a trivial literal read, e.g. returns a literal string.
- a trivial global read, e.g. returns a global variable.

These trivial methods are based on methods which we found to be invoked in our benchmark set. Though it could easily be extended to work with other simple methods, we found these four kinds to bring performance benefits on a large amount of benchmarks.

2.3.4 Optimizations Relevant to Both AST and BC

We now focus on optimizations used in BC interpreters, that can be linked to matching optimizations in AST interpreters. For each, we highlight both the BC-specific and AST-specific optimization, and explain how they may differ.

Quickening (BC), Self-Optimization (AST)

In a BC context, *quickening* (Brunthaler 2010a,b, 2021) is an optimization which uses run-time feedback to rewrite bytecodes. Quickening can replace a generic operation with a more specialized one, e.g., replacing a generic `ADD` bytecode with `ADD_INT`, which only handles integers but can fall back to the generic case when needed. This reduces the number of checks at run time and the code complexity of bytecode handlers, which may have secondary effects such as a more effective use of instruction caches.

This is beneficial for dynamic languages, where even basic operators such as addition can have complex semantics. For instance, the ECMAScript specification [3] defines eight steps, which include possible method calls and type coercions. In our SOM interpreters, we use quickening e.g. to turn `SEND` bytecodes into less complex `Q_SEND` bytecodes that rely on polymorphic inline caches.

Similarly, AST interpreters can be *self-optimizing* (Würthinger et al. 2012), meaning nodes modify themselves to benefit from run-time feedback. For example, to optimize a complex and generic `AddNode` for a language such as JavaScript, the node would replace itself in the AST with one designed specifically for integer addition. In case something other than integers is added, it can then rewrite itself to the generic version to support the language correctly.

Another example would be rewriting a generic `DispatchNode` to a more complex `CachedDispatchNode`, which also stores and uses cached values from a previous dispatch. This highlights the flexibility of self-optimizing AST interpreters compared to bytecode quickening. Given the compact nature of bytecode, a bytecode instruction can only be dynamically rewritten to another that occupies the same amount of space or less, without needing to start relying on side tables. On the other hand, since the AST is typically represented as a tree of node objects connected by pointers, rewriting can easily introduce a more complex structure if needed.

Many of the optimizations for such interpreters need to cheaply confirm a precondition at run time, for instance, that the values being added are integers. Qunaibit et al. (2018) perform a high-level analysis that allows to combine such *guards*, i.e., pre-condition checks, into *megaguards*, which may reduce the need to repeatedly check the same precondition in an interpreter.

Self-optimizing AST interpreters were made more popular and easier to implement by the TruffleDSL, which is part of the GraalVM project, and enables the concise implementation of many common patterns (Humer et al. 2014a). We later expand on the design of GraalVM in section 2.4.2, when we focus on meta-compilation systems.

[3] *Standard ECMA-262 Edition 5.1: The Addition operator*, Ecma International, https://262.ecma-international.org/5.1/#sec-11.6.1

Superinstructions (BC), Supernodes (AST)

Superoperators (Proebsting 1995), more commonly known as *superinstructions* (Piumarta et al. 1998), which are combinations of multiple bytecode instructions into a single, more efficient instruction, reducing the number of bytecodes and so dispatch overhead.

For instance, our bytecode interpreters have `INC` and `DEC` bytecodes, which increment or decrement the top-of-stack value. We also always use a `RETURN_SELF` superinstruction, since SOM methods return their `self` value if no explicit return is specified.

Superinstructions can be arbitrarily complex. For instance, some of our interpreters implement the `INC_FIELD_PUSH` bytecode, which increments the value stored in a field and pushes the result onto the stack. Without defining this superinstruction, this would need to be expressed with a sequence of five bytecodes from the basic SOM bytecode set as can be seen in listing 2.5.

```
1 PUSH_FIELD 0 # push the field value onto the stack
2 PUSH_CONSTANT 0 # push by reading from the literal array at idx 0
3 SEND + # send the + message to add the two values
4 DUP # duplicate the top of stack
5 POP_FIELD 0 # pop the top of stack, and write into the field
```

LISTING 2.5: Sequence of five basic bytecode instructions that is equivalent to the `INC_FIELD_PUSH` superinstruction.

Inspired by superinstructions, *supernodes* (Larose et al. 2022) combines multiple AST nodes into one, removing some node dispatch overhead Additionally, this reduces the number of node objects needed to express a program, which lowers memory use since we avoid a parent pointer, and object headers for our interpreters that represent nodes with language objects and not `structs`. While supernodes are not a key focus of this work, they are used in our AST interpreters, which are the subjects of Chapter 4 and Chapter 5.

An example of a supernode mirroring the `INC` bytecode would be `IntInc`, which we added to one of our AST interpreters. It is applied in situations where we have an addition of a literal integer to a value obtained from an arbitrary subexpression. This avoids the extra node for the literal value and specializes the addition to support

```
1 class IntInc:
2   def constructor(value, child):
3     self.child = child # arbitrary subexpression
4     self.value = value # integer constant
5   def execute(frame):
6     return child.execute(frame) + value # avoids one call to execute()
```

LISTING 2.6: The `IntInc` supernode, which combines an `Addition` node, specializes it to integers, and stores an integer value for the addition.

only integers. It also avoids a call to the `execute()` method of the literal node, as illustrated in some example code in listing 2.6.

Both superinstructions and supernodes are selected and implemented based on performance-sensitive patterns that we identified in the interpreters, by inspecting large amounts of generated bytecode and AST. Since adding too many specialized operations could potentially slow down the interpreters by increasing the number of possible jump targets in the bytecode dispatch loop, or the number of call targets in an AST virtual dispatch, we are careful not to add too many superinstructions or supernodes to our interpreters. We therefore only define specialized bytecodes/nodes for what we find to be frequent sequences of operations.

2.3.5 BC-specific Optimizations

This section focuses on optimizations specific to bytecode interpreters, which have no clear equivalent in an AST context.

Threading

Since dispatching to the next instruction can take a large part of the runtime, it can be valuable to optimize instruction dispatch itself. A common bytecode interpreter optimization is to improve the bytecode dispatch using direct or indirect *threaded code* (Bell 1973; Ertl et al. 2003)[4]. Both techniques engineer the loop structure such that bytecode handlers do not jump back to the beginning of the bytecode dispatch loop, but instead to the handler of the next bytecode in the sequence. In terms of run-time performance, the more efficient of the two techniques is meant to be direct threaded code, in which bytecode handlers all end with a jump instruction that takes

[4]Unrelated to *multithreading*, i.e. concurrent threads of execution.

an instruction pointer corresponding to the body of the next bytecode handler in the sequence.

This technique removes a jump instruction for each bytecode dispatch, since handlers no longer jump back to the start of the bytecode loop which then jumps to the next handler, but instead directly jump to one another. Though the largest performance benefit from this technique is meant to stem from generating code that is more amenable to the CPU branch predictor. For every bytecode dispatch, a bytecode loop without this technique can dispatch to N possible bytecode handlers, with N the size of the bytecode set. Therefore, it can be difficult for the branch predictor to accurately predict jumps to bytecode handlers, leading to cache misses and performance hits. Threading limits this issue by having jumps be at the end of bytecode handlers, and using the knowledge that many handlers never jump to all possible N bytecode handlers since bytecode exhibits patterns. For example, the arguments of a method call can often be stored in local variables for the sake of code readability, therefore we may find that a `CALL` bytecode is likely to come after a `PUSH_LOCAL`.

While historically considered a major optimization for bytecode interpreters since these indirect jumps are difficult to predict (Casey et al. 2007; Ertl et al. 2003), branch predictors on modern CPUs have improved enough for instruction dispatch cache misses to now be of less importance for interpreter performance than they were in the past (Rohou et al. 2015). Our interpreters may also be considered *high abstraction-level interpreters* (Brunthaler 2011), where the implementations of bytecode handlers are complex enough that dispatch has a comparatively lower runtime cost, and so we benefit less from optimizations targeting dispatch specifically.

So, our bytecode interpreters do not implement threading, for this reason and others that we later discuss in section 4.4.1 and section 5.6.2.

Stack Caching

When relying on a stack-based bytecode interpreter, *stack caching* (Ertl 1995) can be used to cache the N top-of-stack values.

The intuition behind this optimization is that the vast majority of operations will be performed on top-of-stack values, e.g. calling an `ADD` operation will take the top-of-stack values as operands. Given this knowledge, stack caching stores these values in special registers, essentially providing an interpreter design that lies in-between stack-based and register-based.

The number of cached values can vary. For *dynamic* stack caching, the cost of caching values is that for any N number of values that can be stored, one needs to maintain N additional alternative implementations of every bytecode handler, each corresponding to a different number of values being cached. This is necessary as the interpreter needs to define valid behavior whether we currently have no cached values, or 1, 2, ... N cached values.

Static stack caching instead engineers the compiler to keep track of the cache state and generate code accordingly, instead of maintaining N copies of the interpreter. However, it needs to ensure control flow joins rely on the same cache state no matter what branch was taken, since it cannot dynamically dispatch to the correct specialized bytecode handler like dynamic stack caching does.

Both stack caching approaches have the potential to improve the performance of the interpreter, but both also require significant engineering effort. For instance, dynamic stack caching requires essentially maintaining N versions of the main interpreter loop, which makes maintainability more difficult. For this reason, as well as the uncertainty that this optimization would bring worthwhile performance gains, our interpreters do not rely on stack caching.

2.4 Meta-Compilation

While interpreters are comparatively easy to implement, many applications need just-in-time compilers (Aycock 2003) to reach their desired performance. For many languages, one can likely build a simple JIT compiler with moderate effort. However, highly optimizing JIT compilers, as known from Java and JavaScript virtual machines, take hundreds of engineering years of effort.

Meta-compilation reduces the individual implementation effort by making just-in-time compilers reusable and independent of any particular language. RPython

and GraalVM are the main systems in this space. Both approach the problem by compiling the executing program together with the interpreter that is executing it. This combination provides the necessary specialization opportunities to produce efficient native code that rivals custom-built state-of-the-art JIT compilers.

2.4.1 RPython: A Tracing Meta-Compiler

While most JIT compilers choose frequently executed methods as their compilation units, *tracing* JIT compilers (Bala et al. 2000; Gal et al. 2006) use the *trace* of a concrete execution path as their compilation unit typically starting at a loop header or a method entry.

The RPython framework (Bolz et al. 2009, 2013b) uses tracing-based JIT compilation to enable the meta-compilation of interpreters. It is known for PyPy, a widely used Python implementation with JIT compilation. RPython interpreters use annotations to indicate for instance which loops are relevant for JIT compilation, which is then used by RPython to add a JIT compiler and garbage collection. While PyPy uses bytecodes, AST interpreters work also well on RPython.

From an engineering perspective, this approach was found to be highly effective. Marr et al. (2015) were able to build an interpreter with comparably low effort and few code adaptations, the result being a language implementation with good peak performance.

Listing 2.7 is an example of the bytecode dispatch loop of a RPython-based BC interpreter, a simplified version of one of our interpreters in Chapter 4. The `interpret` method is called for a guest-language method, and executes its bytecodes one after the other. This means that each guest language method requires the allocation of a new stack frame, and so that this interpreter is recursive in nature, with guest language methods calling other guest language methods. We refer to this as a *partly recursive design*, which we come back to in section 3.3.1.

To reach optimal performance, interpreter implementers can use annotations to assist the JIT in making informed decisions. In our example, the interpreter uses `jitdriver.jit_merge_point` to define a “merge point” where execution may fall back to the interpreter if the code turns out to have been compiled based on incorrect

```python
def interpret(method, frame):
    current_bc_idx = 0
    stack_ptr = -1
    stack = [None] * max_stack_size

    while True:
        # Define where the JIT may fall back to the interpreter
        jitdriver.jit_merge_point(
            current_bc_idx,
            stack_ptr,
            method,
            frame,
            stack,
        )

        bytecode = method.get_bytecode(current_bc_idx)

        # Promote stack pointer (inform the JIT to assume it is constant)
        promote(stack_ptr)

        if bytecode == Bytecodes.push_global:
            ...
        elif bytecode == ...:
            ...
```

LISTING 2.7: Example of a BC-based interpreter designed using RPython.

guesses. We also use the `promote` function to indicate that a value in a trace is to be considered a constant, which allows subsequent optimizations. Examples of annotations not shown in this code include specifying that loops within a function can be unrolled if deemed beneficial (i.e. have the loop contents be duplicated for each iteration), or promote all arguments of a given function, among others.

2.4.2 Graal: A Partial-Evaluation-based Meta-Compiler

Partial evaluation (Futamura 1999, Originally published in 1971) uses partial input to derive an optimized program. The resulting program contains only computations that depend on the missing input. When the partial input is user code and our interpreter is the program to be partially evaluated, then the result is a compiled version of the user code, which can execute efficiently once the user input is provided. This is considered to be the first Futamura projection, effectively introducing the notion of meta-compilation.

GraalVM uses this approach to partially evaluate an interpreter for a user program, effectively just-in-time compiling it (Würthinger et al. 2017, 2013). Instead of using tracing, the Graal compiler starts from methods and uses aggressive inlining to

determine compilation units. This has the drawback that it cannot rely directly on the values of a concrete execution, as RPython can. Instead, GraalVM relies on interpreters to collect information about types, values, and taken branches while executing, and for them to encode it in the program representation, i.e., the AST, bytecode, and other profiling data structures. Interpreters would typically use self-optimizations, quickening, and similar forms of profiling. The Graal compiler will use the information during partial evaluation to compile an optimized program, eliminating most if not all of the interpreter code in the process.

GraalVM can be used either on top of the HotSpot JVM, or ahead-of-time-compiled with GraalVM Native Image (Wimmer et al. 2019). The native image mode is similar to how RPython would typically be used, and results in a native binary of the interpreter.

Marr et al. (2015) found that interpreters for partial-evaluation-based compilation required more engineering effort, since they need to collect profiling data explicitly. However, GraalVM also had higher peak performance compared to RPython, though with longer warmup phases (which we later observe ourselves in section 4.3.3).

```
1  public abstract class AdditionNode extends Node {
2
3    @Specialization(rewriteOn = ArithmeticException.class)
4    public static final long doLong(final long left, final long argument) {
5      return Math.addExact(left, argument);
6    }
7
8    @Specialization
9    @TruffleBoundary
10   public static final Object doLongWithOverflow(final long left, final long arg) {
11     return BigInteger.valueOf(left).add(BigInteger.valueOf(arg));
12   }
13
14   @Specialization
15   public static final double doDouble(final double left, final double right) {
16     return right + left;
17   }
18
19   @Specialization
20   @TruffleBoundary
21   public static final Object doBigInt(final BigInteger left, final BigInteger
         right) {
22     return left.add(right);
23   }
24
25   ...
```

LISTING 2.8: Example of an AST node in a Truffle/Graal-based interpreter.

Listing 2.8 shows an example, taken from our Truffle-based AST interpreter which we rely on in Chapter 4. The class `AdditionNode` inherits the type `Node`, provided by Truffle, and implements addition for all possible inputs. For instance, the method `doLong` will be called if both inputs are of type `long`, and `doDouble` if both inputs are of type `double`. This is enabled by the `@Specialization` annotation, which specifies that each of these methods corresponds to one node specialization, i.e. a possible state for the node to be in (assumed to always take that input). When compiling, the specialization of a node will be assumed to be constant, which enables optimizations accordingly. Code will be generated by the framework to automatically shift from one specialization to another at run time, as needed. Specializations can specify what may trigger a change to another specialization, seen here with `doLong` and `rewriteOn = ArithmeticException.class`, making it such that an `ArithmeticException` (e.g. an overflow) will change the specialization of this node to one that enables these new semantics (`doLongWithOverflow`). As another example of compiler annotation, `@TruffleBoundary` specifies a boundary for partial evaluation, as in, the aggressive inlining done by Graal, should not be performed for this method.

2.5 Garbage Collection

Garbage Collection (GC) is a form of automatic memory management, usually provided directly by the language implementation. A garbage collector provides mechanisms to *allocate* new objects on a heap, *identify* live objects (i.e. objects that are still in use), and *reclaim* any free memory found. Objects are automatically tracked, and are *collected* when they are no longer live, in contrast to manual memory management e.g. in C, where all memory must be manually allocated and de-allocated by the user.

Given the cognitive overhead from manual memory management and considerably larger risk of security issues (e.g. memory leaks, use-after-free errors), many languages opt to manage memory by relying on garbage collection.

This section serves as an overview of some GC approaches within a vast and complex field, focusing only on the approaches relevant to our interpreters. Our meta-compilation-based systems (seen in Chapter 4) are not concerned with implementing GC themselves, as it is already provided by the underlying meta-compilation systems.

Therefore, this section is most relevant to our Rust-based interpreters, which are the focus of Chapter 5.

This highlights that while our meta-compilation-based interpreters benefit from mechanisms provided by meta-compilation frameworks, e.g. garbage collection, our Rust-based interpreters do not. When relevant in this work, we will refer to interpreters that are not based on meta-compilation as *pure* interpreters.

2.5.1 Reference Counting

One simple approach to implementing garbage collection is to use *reference counting* (Collins 1960). With this technique, GC objects hold a counter corresponding to the number of live references to the object in question. When the counter reaches 0, the language implementation knows that the object is not used by any part of the system and therefore that it should be garbage collected.

In naive reference counting, the counter is always incremented when a reference to the object is created, and gets decremented when a reference goes out of scope and so stops being live. While functional, this approach often struggles with cycles. If an object holds a reference to another object that itself holds a reference to the first one, whether that reference is direct or transitive (i.e. through an object chain), then this forms a cycle that will never get freed. This issue needs to be circumvented carefully using *weak* (non-counted) references, ensuring that cycles cannot be formed, or perhaps implementing a way to detect cycles.

Moreover, naive reference counting tends not to perform well. The counter then needs to be managed every time a new reference is created or removed, which are highly common operations, thus compounding into taking a non-trivial part of the overall runtime. However, more modern and complex reference-counting-based approaches (Zhao et al. 2022) may update counters in more efficient ways.

2.5.2 Tracing Garbage Collection

While reference counting keeps a counter on a per-object basis, *tracing* garbage collection approaches instead periodically identify live objects by *tracing* (or *scanning*) the object graph.

A set of *root* objects is defined to serve as the base from which to start traversal, and edges are pointers to other objects. Each object found gets traced by finding its potential edges to other objects, and those new objects are recursively traced in turn until no new objects can be found. All objects not in the final graph are deemed not live since they are not reachable and so get discarded.

During tracing, the state of the VM must not be modified as to make the object graph no longer up-to-date and accurate. For non-concurrent tracing GCs, this usually means tracing happens after the VM *stops the world*, meaning that it halts its normal execution until tracing is complete. This can happen e.g. when all the memory has been used, and the runtime system needs to reclaim some memory to make room for new objects.

A common and simple approach to tracing GC is *mark-and-sweep* (McCarthy 1960), where a mark phase marks all reachable objects, and a sweep phase discards all other objects. Other approaches may partition the heap, such as *semispace* (Fenichel et al. 1969), where objects are all allocated within one half of the heap (the *from*-space), and any collection will move all live objects to the other half (the *to*-space) before making the from-space become the new to-space, and vice-versa.

There exist many more modern and complex tracing GC implementations such as hybrid approaches with reference counting (Bacon et al. 2004; Zhao et al. 2022), or *mark-region* collectors (Blackburn et al. 2008), though our interpreters rely on less complex GC approaches.

2.6 Overview of the Rust Language

Rust (Klabnik et al. 2023) is a general-purpose programming language with a focus on performance and memory-safety. It aims to supersede low level languages like C or C++, by offering similarly high performance. This section provides a short overview of the language in which we have written interpreters that we focus on in Chapter 5. We aim to provide an introduction to the language for readers who may be unfamiliar with it, establishing a base to explain some of the design decisions taken in our Rust-based interpreters.

The Rust language is interesting for our purposes since it offers high performance while offering more safety guarantees than C and C++. This allows us to write efficient implementations, while benefiting from its safety mechanisms, though they can also be circumvented using the `unsafe` keyword (as we will see in section 2.6.3).

Rust does not feature manual memory management like C or C++, nor is it a garbage collected language. Instead, memory is managed by abiding by *ownership* [5] rules, which we describe in the next sections.

2.6.1 Ownership

Ownership rules define which values can "own" data, as well as when this data gets freed. These rules are enforced statically, that is, programs which break any ownership rules will not compile. The rules are as follows:

- each value must have an owner.
- there can only be one owner at a time.
- a value is dropped when its owner goes out of scope.

```
1 let a = Box::new("hi");
2 let b = a; // ownership of value transferred to b
3 print!("{}", a); // ERROR: a no longer owns its value!
```

LISTING 2.9: Example of ownership in Rust.

Listing 2.9 shows an example. We allocate a string `"hi"` on the Rust heap using the `Box` pointer type. This value is then associated with the variable `a`. By then performing the operation `b = a`, we *transfer ownership* of the string from `a` to `b`, making `b` the new owner of that string. If we then tried to access the value through `a`, we would get a compilation error, as it no longer owns the value at this point in time, and `b` is now the only way of accessing it. And finally, assuming that we removed the final invalid line to allow compilation, as we exit the scope in which `b` was defined, the owner `b` of the string would go out of scope, and so the string would be freed.

[5] https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html

By enforcing the rule that all values have owners, Rust guarantees that it will always be able to reclaim memory used by the program, since memory is owned by variables, and will necessarily be freed when these owners go out of scope.

2.6.2 Borrowing

Since one also needs to reference data, Rust features several pointer types, but the most notable is *references*, which are pointers with safety guarantees. As they refer to existing values without taking ownership of them, creating a reference is referred to as *borrowing*. They are represented with an `&` and can be two distinct types, either immutable (`&T`) or mutable (`&mut T`). The value referred to by an immutable reference cannot be accessed mutably, while any value referred to by a mutable reference can be access mutably and immutably. Crucially, for a given value, immutable and mutable borrows cannot coexist. Moreover, a value may be borrowed immutably an arbitrary number of times, but may only be borrowed mutably once.

```
1 let a = 42;
2 let r1 = &a;
3 let r2 = &a; // OK: several immutable references allowed
4 let r3 = &mut a; // ERROR: immutable references exist!
```

LISTING 2.10: Example of references and borrowing in Rust.

Listing 2.10 shows a code example. Given a variable `a`, we can create several immutable references `r1` and `r2`. Neither of these immutable references shift ownership of `42` away from `a`, an arbitrary amount of them could be created, and they cannot be used to modify the value itself. The value owned by `a` can only be modified if we created a mutable reference, such as `r3`. However, attempting to create this mutable reference leads to a compile-time error, since there are live immutable references when `r3` gets instantiated, namely `r1` and `r2`.

2.6.3 Unsafe Rust

In the previous sections, we described Rust ownership rules, which are restrictions imposed by the Rust language to ensure memory safety. Though, Rust also provides a mechanism to circumvent these rules and perform operations that cannot be guaranteed to be safe by the type system, using the `unsafe` keyword to wrap code blocks.

Unsafe Rust has the same semantics as regular Rust, but allows operations that are normally forbidden. Notably, while safe Rust allows the creation of any raw pointer (e.g. `*const usize`), only unsafe Rust allows dereferencing them, meaning that unsafe Rust allows the access and modification of arbitrary memory, which may introduce undefined behavior.

A frequent use of `unsafe` Rust is to provide more performance, as it grants a large degree of freedom that can be often used to implement more efficient solutions than safe Rust would allow. A simple example is using `unsafe` to circumvent run-time safety checks. An `Option<T>` type may correspond to a `Some(T)` or `None`, and the `unwrap` method will check which one it is and return the corresponding underlying value. But `unsafe` Rust also allows the usage of the `unwrap_unchecked` method, which returns `Some(T)` without performing any safety checks, with the program failing if the value was actually a `None` type. This is easy to misuse, but can achieve better performance by avoiding run time checks if one knows that they are avoidable in practice, even if one cannot necessarily inform the Rust typesystem itself that it is known that this operation will always succeed. This is useful when designing programs that care about attaining good run-time performance, e.g., programming language interpreters. Though, we use the `unsafe` keyword sparingly, as we are mindful of potential misuse potentially breaking execution.

Using `unsafe` is also useful when implementing garbage collection (see section 2.6) in Rust, since garbage collection can easily be at odds with Rust ownership rules as it intrinsically requires widespread control over a large amount of heterogeneous memory. `unsafe` code is relevant to Chapter 5, where we use it both to achieve better performance in some parts of our interpreters, and to more easily implement highly efficient garbage collection.

2.6.4 Garbage Collection in Rust

As we have implemented garbage collection in our Rust-based interpreters, we dedicate this section to describing existing approaches that provide GC in Rust. Though we will later see that our Rust-based interpreters do not use a Rust-specific

GC approach, but instead rely on a language-agnostic GC framework, namely MMTk (see section 5.3.1).

For most Rust programs, ownership rules are sufficient to manage memory, potentially working with the standard library types `Arc<T>` and `Rc<T>`, which provide a simple reference counting (see section 2.5.1) approach to garbage collection, though one that cannot easily free cycles.

So, some programs may benefit from managing memory using tracing GC, and while it is not part of the language core, some libraries provide means to use tracing GC in Rust. Examples include *rust-gc* (Goregaokar 2015), *shredder* (*shredder* 2019), *gc-arena* (West 2024), *shifgrethor* (Aronson 2018), among others.

An alternative approach is that of Alloy (Hughes et al. 2025), which is not a library but instead engineers Rust itself, and uses conservative garbage collection to provide a `Gc<T>` as part of the standard library.

2.7 The Are We Fast Yet Benchmark Suite

For our experiments throughout this thesis, we will rely on the Are We Fast Yet (AWFY) benchmarks (Marr et al. 2016). This suite includes five macro-benchmarks, which behave similarly to parts of applications, and nine micro-benchmarks, which focus on narrow aspects of runtime systems. This gives us sufficient breadth for our experiments. We give an overview of all benchmarks in section 2.7, which provides a short description for each of them. Moreover, we provide additional information about the benchmarks in appendix A.1, such as their size and observed polymorphism.

They are designed for cross-language benchmarking, which allows us to compare our interpreters to state-of-the-art implementations such as Node.js or OpenJDK, to contextualize our performance results when relevant.

They are also designed to exercise the common language abstractions of object-oriented languages. As mentioned in section 2.2, the SOM language shares the typical implementation challenges with other major dynamic languages: specifically, it has objects, classes, collections, lambdas, and exception-like constructs. The Are We Fast Yet benchmarks exercise specifically this common core, and as such, our

Name	Description
CD	Simulation of an airplane collision detector.
DeltaBlue	Classic VM benchmark implementing a constraint solver.
Havlak	Implements a loop recognition algorithm.
Json	JSON string parsing benchmark.
Richards	Classic benchmark simulating an OS kernel.
Bounce	Simulates a ball bouncing within a box.
List	Recursively creates and traverses lists.
Mandelbrot	Calculates the classic fractal.
NBody	Simulates movement of planets in the solar system.
Permute	Generates permutations of an array.
Queens	Solves the eight queens problem.
Sieve	Finds prime numbers using the sieve of Eratosthenes.
Storage	Creates and verifies a tree of arrays to stress the garbage collector.
Towers	Solves the Towers of Hanoi game.

TABLE 2.1: Overview of all Are We Fast Yet benchmarks. The five macro-benchmarks are at the top, and the nine micro-benchmarks underneath.

experiments on SOM evaluate aspects common to a wide range of languages. With this, we believe our results on SOM to generalize to a wide range of other languages of a similar kind, which includes but is not limited to JavaScript, Python, and Ruby.

The key limitation of this benchmark set is that they are relatively small, and with most benchmarks, the results may not generalize to any particular application.

Chapter 3

Overviewing The Interpreter Design Space

3.1 Introduction

As seen in section 2.1, interpreters are a lightweight and efficient way to implement a programming language, with two typical approaches identified by the program representation used: *abstract syntax tree* (AST) or *bytecode* (BC).

The distinction between AST and bytecode is however not as clear cut as one might expect, and interpreter designs can blur the line between the two. Many characteristics commonly associated with AST interpreters can be found in BC interpreters, while conversely a BC interpreter can be implemented like an AST interpreter. Representations such as serialized ASTs lie in between AST and BC representations, being linear and tree-like simultaneously. As such, strict definitions for AST and BC interpreters are less useful for designs that lie in the middle.

This chapter examines commonly held opinions about the characteristics of AST and BC designs, and highlight various hybrid interpreter designs that challenge these assumptions. Based on these observations, we provide an overview of the interpreter implementation design space, comparing different design decisions and their trade-offs. We posit that AST and BC are not separate designs but rather the ends of a spectrum, and we contribute a novel spectrum-based characterization of interpreter designs from *AST-like* to *BC-like*, recognizing the benefits of either utilizing the host language, or moving closer to the target machine. From this high-level overview of the interpreter design space, we establish a set of traits which we believe

to be associated with AST-like and BC-like designs. This chapter thus serves as a foundation for Chapter 4 and chapter 5, which focus on AST-like and BC-like interpreters implemented using different languages and within different contexts, such that we can justify design decisions as being associated with traits which are typical of AST or BC interpreters, specifically. We summarize the findings in this chapter in section 3.5, where we outline key traits associated with AST-like, and BC-like interpreters.

3.2 AST and BC: Commonly Held Opinions

To lay the groundwork for a discussion of the differences between AST and BC, this section highlights the design decisions and attributes which are most commonly associated with either design. We find these observations useful for reasoning about interpreter design as a whole, and so several of them later help us formulate design dimensions for what we define as AST-like and BC-like interpreters, in section 3.5. Though, importantly, we find that none of these observed traits is a clear indication that an interpreter is strictly AST or BC, since we highlight counterexamples for each of them.

3.2.1 Assumptions About Performance

AST interpreters are often criticized for their performance. Section 2.1.1 discussed the visitor-based AST interpreter design, which is considered to result in low run-time performance, since it requires two method dispatches for each operation: one to the node's `accept` method, which takes the visitor as argument, and then one to invoke the `visit` method, which implements the node type's behavior.

Reducing dispatch overhead has long been identified as a valuable optimization target for interpreters (Basso et al. 2023; Bell 1973; Casey et al. 2007; Larose et al. 2022), as it can take up a large portion of the run time.

Consequently, AST designs that aim to minimize run-time overhead tend to choose a structure where the next node is invoked with a single method dispatch. Such a design may be a naturally recursive design as discussed in section 2.1.1. However, each node requires a virtual dispatch to an `execute` method. Moreover, in

AST-based designs, nodes can vary in size and are therefore often stored as separate heap objects accessed via machine-word-sized pointers, making them less compact. Such nodes are typically also not guaranteed to be adjacent to each other, possibly causing suboptimal cache behavior. Therefore, such AST designs are often criticized for doing *pointer chasing*, which limits their performance.

Thus, the less compact structure of an AST contributes to the idea that AST-based designs offer worse performance than BC-based ones. Practical examples of this include Ruby and JavaScript implementations. Ruby's implementation started out as an AST interpreter, but was replaced by a bytecode interpreter to increase run-time performance (Sasada 2005). Similarly, Webkit's JavaScript engine replaced an AST-based interpreter with a BC one, citing performance as a key motivation for this change.[1] Since run-time performance is often a major concern for language implementers, relying on bytecode instead of AST is often viewed as natural. We will revisit the notion that ASTs are less compact in section 3.2.3.

3.2.2 Recursive versus Iterative

The AST interpreters discussed in section 2.1.1 use recursion, as an AST is a recursive data structure. Seaton (2016) writes "An AST interpreter executes a program by recursively walking its AST". To compare AST and BC interpreters, Zhang et al. (2014) write "A bytecode interpreter is iterative [...] An AST interpreter on the other hand is recursive", making it a common distinction between both designs. So, execution of an AST is typically thought of as walking the tree structure in a recursive fashion, accessing one node, then its children, with each node execution being a function call, thus on the host language call stack. CPS-style interpreters as described in section 2.1.1 may rely on tail-call elimination to turn function calls into direct jumps, rather than requiring the allocation of new frames on the host stack.

In contrast, bytecode interpreters tend to avoid relying on the host language call stack. Since the bytecodes are already a linearization of a possible AST, they are handled one by one, instead of recursing on an AST.

Although it is a natural way to implement an AST interpreter, we do not view recursion as a mandatory characteristic of an AST-based design. We give examples of

[1] https://webkit.org/blog/189/announcing-squirrelfish/

non-recursive AST designs in section 3.3.2, and of recursive BC-based interpreters in section 3.3.1 for instance on top of Truffle (see section 2.4.2).

3.2.3 Non-Compact versus Compact

A bytecode instruction is typically represented by one byte or a few bytes. Both Smalltalk-80 and Java Virtual Machine used designs that are roughly one byte for the opcode, and zero or more bytes to encode additional arguments (Goldberg et al. 1983; Lindholm et al. 1999). CPython uses two bytes per bytecode, one for the opcode and one for an optional argument.[2]

A sequence of bytecodes is executed one by one, though, jump bytecodes can change the control flow by specifying an offset or bytecode index to be executed next. Bytecodes are usually designed with a constant number of arguments for a given opcode, which means they have a specific size, after which the next bytecode is encoded. A bytecode sequence is thus representable as a contiguous and compact byte array.

In contrast, an AST is often a less compact and non-contiguous data structure in memory. As a tree structure, its nodes are linked by edges, which can be naturally modeled using pointers. Thus, a simple AST would be based on heap objects for instance for interpreters using the visitor pattern (Gamma et al. 1994) or Truffle-based implementations (Humer et al. 2014a; Würthinger et al. 2012). Since any given node with children nodes needs to store pointers to them, any node with children will need more memory than a typical bytecode. This makes AST interpreters likely to require more memory overall. When comparing AST and BC interpreters on top of meta-compilation systems, we later confirm this in this thesis (see section 4.3.4) that ASTs use more memory than bytecode to represent a program.

Since linearized instructions are usually stored in a compact contiguous encoding, they better utilize CPU cache lines, while the non-contiguous encoding of an AST is more likely to incur cache misses when dispatching the next instruction. This is likely one of the reasons why AST interpreters are known for being slower than BC

[2] https://github.com/python/cpython/blob/275056a7fdcbe36aaac494b4183ae59943a338eb/InternalDocs/interpreter.md#instruction-decoding

interpreters. However, an AST can also be represented as a compact structure, which we explore in section 3.3.3.

3.2.4 Inefficient versus Efficient Control Flow

AST interpreters that do not utilize continuation-passing style, are often criticized for having inefficient control flow. Basso et al. (2023) observed that "widely used implementations of AST interpreters rely on costly run-time exceptions to model the control flow of the interpreted language." Basso et al. work with AST interpreters based on the TruffleDSL.

In this and similar interpreter implementations, control-flow mechanisms in the guest language such as non-local returns, exceptions, or leaving a loop early with constructs like a `break` keyword, may require the use of an exception-like mechanism in the host language. These interpreters are recursive, but do not utilize tail-call optimizations and rely on the host language stack. For host languages such as Java, run-time exceptions are the only available mechanism to unwind the host stack (Nystrom 2021 ch. 10).

In contrast, bytecode interpreters are commonly designed to be not bound by either host language stack or other host language mechanisms. Thus, they are often non-recursive and realize control flow as either `JUMP` bytecodes or explicit manipulation of the guest-language stack representation, which can let complex control flow be more easily represented, and potentially more efficiently.

However, the cost of non-local control flow is not an issue for all types of AST interpreters. For instance, in continuation-passing style, the problem is trivially avoided by making the continuation explicit and passing it on. Thus, it is rather a consequence of how guest-language function application is realized. If the interpreter uses recursion, exceptions or similar costly features are needed. If the AST interpreter is non-recursive, as for instance illustrated in section 3.3.2, then its stack and control flow can be manipulated more directly.

This idea of avoiding being constrained by the host language has been around for a long time, as Sussman et al. (1975) wrote for the Scheme interpreter: "we must not use recursion in the implementation language to implement recursion in the

language being interpreted. [...] The reason for this is that if the implementation language does not support certain kinds of control structures, then we will not be able to effectively interpret them."

3.2.5 Source-like versus Hardware-like

An AST is typically designed to closely resemble the syntax of the guest language code. While a parser may omit irrelevant elements and decompose high-level syntax into more fundamental AST nodes, it stays close to the guest language.

Bytecode is, however, an instruction set for a virtual machine, often resembling a stack or register machine (Shi et al. 2008) and thus, is inspired by hardware instruction sets (D'Hondt 2008). Consequently, bytecode is typically closer to hardware instruction sets than it is to the syntax of the guest language.

However, this is a rather vague pair of contrasting traits, since "resemblance" is not a specific term itself, as it could refer to different characteristics and to varying degrees of closeness. On the one hand, a bytecode set could be designed to be "source-like" by having each bytecode have a direct equivalent as a source code keyword or construct, e.g. a `loop` bytecode as used by Wasm (Rossberg 2022). One can then fairly directly reconstruct the source code from the bytecode. On the other hand, an AST interpreter could more closely resemble hardware instruction sets by relying on some `GotoNode` to traverse the tree using direct jumps, or have nodes manipulate a global set of registers.

We find this distinction between source and hardware interesting nonetheless, as it highlights that the proximity between AST designs and source code, and the proximity between BC designs and hardware. We base our definitions of the terms *AST-like* and *BC-like* on similar ideas, as we later see in section 3.4.

3.3 Hybrid Interpreter Designs: AST or BC?

In the previous section, we highlighted characteristics from the literature that are typically associated with AST interpreters or BC interpreters. However, there are counterexamples which prevent clearly categorizing each characteristic as either

belonging to AST or BC interpreter designs. This section discusses designs that lie in between AST and BC.

3.3.1 Recursive Bytecode Interpreter Designs

While bytecode interpreters generally traverse bytecodes iteratively, their overall design may still rely on recursion for the implementation of function application. While this partly-recursive design is uncommon in classic bytecode interpreters, in our experience it helps meta-compilation systems such as RPython (see section 2.4.1) and the GraalVM (see section 2.4.2) to produce better just-in-time-compiled code (Marr et al. 2015), by more explicitly separating frame objects, i.e., activation records. For example, the SOM bytecode interpreters (Larose et al. 2023; Marr et al. 2015) on RPython and the GraalVM have a classic bytecode loop within a single function, but instead of representing all state in a frame object for the guest language, the bytecode index is kept in a local variable, and the function with the bytecode loop is invoked recursively for guest language functions. We gave an example of this design in section 2.4.1. This design helps both meta-compilation systems, and is even enforced on top of GraalVM.

Thus, while these interpreters are bytecode interpreters, they also rely at their core on recursion, which moves them closer to what one would consider AST interpreters. For us, this means that the code representation (AST or BC) and the interpreter design strategy (recursion or iteration) can be independent of one another in specific aspects. In the following section, we explore this design dimension by discussing how an AST interpreter can be designed without relying on recursion.

3.3.2 Non-recursive AST Interpreter Designs

Since any recursive algorithm can be turned into an iterative one by using an explicit stack, interpreting a tree does not have to be done using recursion. Thus, one can implement an AST node dispatch loop similar to a bytecode dispatch loop, which stores and executes a current node instead of a bytecode index pointing to the current bytecode.

Dispatching the next node in the sequence could still be handled by reading from any given node and finding out the address and type of its potential children, but it would not rely on the host language and dynamic dispatch to execute the next node and instead be dispatched by the main AST loop. Such a design may handle values on its own stack or set of registers. It would rely less on the host language and therefore help with efficiently representing more complex control flow, as seen in section 3.2.4. Crucially, the AST representation remains the same, with nodes still being distinct heap objects.

Interpreters in this style can be found, among other examples, in implementations of the Scheme programming language. Scheme supports first-class continuations, so any interpreter for Scheme in a host language without this feature cannot use the host program stack. For example, Dybvig (1987) describes a Scheme interpreter that compiles code into a kind of tree-based intermediary representation, which can be efficiently interpreted by an iterative interpreter based on the SECD abstract machine (Landin 1964). Similarly, continuation-passing style, as discussed in section 2.1.1, is often used for implementing Scheme interpreters, as it elides the stack entirely and instead represents on-hold computations as continuations in the heap (Felleisen et al. 1987; Sussman et al. 1975).

As mentioned in section 3.3.1, these examples further support the point that the interpreter representation and the main strategy used in the design of the interpreter (recursion or iteration) are independent of each other.

3.3.3 A Bytecode-like AST Interpreter Design

As noted in section 3.2.3, an important difference between AST and BC interpreters appeared to be that bytecode is a more compact representation, with ASTs often being made up of individual objects instead. Edges are commonly represented by pointers, and the AST nodes themselves can be scattered in memory, preventing an optimal use of CPU cache lines.

However, depending on the host language, it can be fairly natural to place the tree node consecutively in memory and represent edges implicitly or with compact offsets instead of machine-word-sized pointers. Thus, every node knows its own

position, and instead of a pointer, children are found by their offset from the parent node. Because of Rust's restrictions around pointers, such a representation can be more desirable.[3] This assumes that the nodes are not moved or changed in size so that the offsets are constants. For memory use, this type of representation is beneficial since offsets can typically be much smaller than generic pointers. Such an interpreter could be considered a *serialized AST interpreter*. One example is the Gibbon compiler (Vollmer et al. 2017), that transforms a tree structure into a packed version of itself by doing a pre-order serialization pass.

However, by packing the tree structure, the difference between an AST and bytecode becomes much smaller.

Since AST traversal can be implemented in a main dispatch loop (see section 3.3.2), a recursive traversal of the tree (section 3.2.2) is not necessary. With such an interpreter design, we rely on a flat and compact data structure (section 3.2.3) and can also avoid recursion for function application, which would allow us to handle non-local control flow (section 3.2.4) efficiently. At this point, we have all traits typically associated with bytecode instead of AST interpreters.

3.3.4 AST-like Bytecode Interpreters

Expanding on the previous section, it is possible to implement a bytecode interpreter in an AST-like manner. Some interpreters implemented with the TruffleDSL (Humer et al. 2014a), originally designed for AST interpreters, were instead implemented as bytecode interpreters.

GraalSqueak (Niephaus et al. 2018) is designed to execute Squeak/Smalltalk bytecode, but models each bytecode as a Truffle node object. It features a main bytecode dispatch loop where each bytecode object is executed by calling a virtual `execute` method, a method shared by all AST nodes in Truffle. Since it is explicitly defined to execute a bytecode set, the authors specifically refer to it as a bytecode interpreter, although each bytecode is represented by an AST-like node object, which comes with the memory overhead of objects instead of being a compact bytecode.

[3] An example for a use in Rust: Flattening ASTs, Adrian Sampson, 2023-05-01: https://www.cs.cornell.edu/~asampson/blog/flattening.html

Sulong (Rigger et al. 2016) is another bytecode-based design on top of Truffle which interprets LLVM bitcode. It relies on a main dispatch loop similar to bytecode interpreters and GraalSqueak, but only to dispatch LLVM basic blocks. The instructions within basic blocks are represented using AST-like node objects. They explicitly refer to their implementation as a “hybrid bytecode/AST interpreter approach”, showing that it is not easily categorized as either AST or BC.

3.3.5 Conclusion

Given that many of these examples blur the line between AST and BC designs, a natural question may be how one can categorize any of these designs as either AST or BC. It may be possible to come to an agreement on a clear definition of both terms, for example, deem any interpreter to be a BC interpreter if and only if its representation is compact and has reified jumps as direct offsets.

In our eyes, however, this is not the most valuable angle of approach. Although it might be possible to precisely define the terms AST and BC, these hybrid designs highlight that interpreter design is not easily represented as a binary choice between two distinct representations.

3.4 Design Trade-offs: AST-like and BC-like

In the previous section, we explored different hybrid interpreter designs that were neither clearly AST nor bytecode interpreters. Based on this, we argue that interpreters can be somewhere in-between two possible extremes and have traits that put them closer to an AST or a BC design. In other words, we argue that interpreters can be more or less *AST-like* and *BC-like*.

3.4.1 AST-like and BC-like: Host Language and Target Machine

We find it useful to conceptualize AST and bytecode interpreters on a spectrum, which we show in fig. 3.1. On one end, we have a AST interpreter, recursive and relying on virtual dispatch, e.g., with a visitor-based or naturally-recursive design (see section 2.1.1). On the other end, we have a well-performing interpreter that relies

on a compact program representation and a main dispatch loop, i.e., a traditional bytecode interpreter (see section 2.1.2).

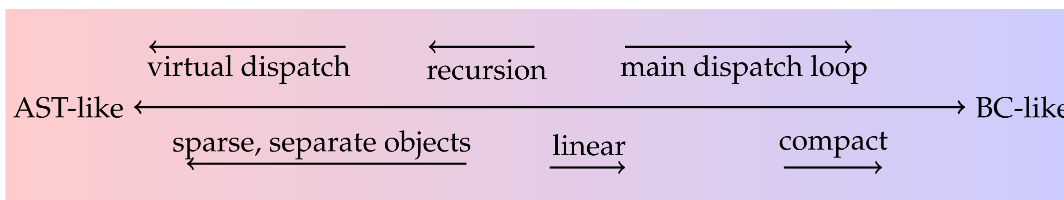


FIGURE 3.1: Spectrum from AST-like to BC-like, with characteristics that may more closely identify with one end than the other. The left end *benefits from the host language*, and the right end *minimizes distance with the target machine*.

A key distinction between both is that one maximizes the use of the host language and the other intentionally avoids depending on it, to instead attempt to maximize performance by minimizing the distance with the underlying machine[4]. Therefore, we define AST-like as *benefiting from host language abstractions* and BC-like as *minimizing distance with the target machine*.

Based on section 3.2, we conclude that AST interpreters are commonly assumed to rely heavily on host language features and abstractions (and so tend to rely on host languages structured similarly to the guest language, which can otherwise lead to semantic mismatch - see section 3.4.4). Many typical AST designs use recursion to traverse the representation, which relies on the host language call stack. They usually navigate from one node to another by dereferencing a pointer and invoking a method on the node, for instance to execute a subexpression. To return to the parent, they simply return from the method call. However, to return from nested expressions e.g. when throwing a guest language exception, they may use host-language exceptions to unwind the host language call stack.

Bytecode interpreters, on the other hand, tend to implement these operations explicitly with data structures they have full control over. For example, they may use offset-based jumps for local control flow or modify a list of activation records when handling a guest language exception.

Bytecode sets are generally designed to be a linearized and compact program representation. This has the benefit of fitting multiple bytecodes in a single CPU cache line, which can be important for performance. Thus, when we say BC-like

[4]We assume that the machine in question has a Von Neumann architecture, and is not e.g. a LISP machine, which is outside the scope of this work.

interpreters *minimize the distance with the target machine*, this does not require that bytecodes match hardware instructions. Instead, we mean that the overall interpreter design aims to utilize the hardware as effectively as possible, typically by using lower-level programming constructs, and often forgoing host-language abstractions with the aim for greater control over the language implementation.

For example, by utilizing more low-level mechanisms, bytecode interpreters can minimize the bytecode dispatch overhead with optimizations such as threaded code (see section 2.3.5) which uses low-level constructs, e.g. `goto`, instead of the more high-level `switch/case`. Such choices move bytecode interpreter designs closer to the target machine for the benefit of performance. However, these choices also come at the cost of more engineering effort and moving away from the host language's core features, which typically means that debugging tools become less direct, and interpreter implementers need to implement their own alternatives, for instance to print out or visualize the guest-language stack.

3.4.2 Intersection of AST-like and BC-like

Many design decisions are enabled by one another, and seem to move an interpreter design further towards the AST-like or the BC-like end of the spectrum.

Starting with a very AST-like design, as one moves away from the host language closer to the target machine, one may choose to rely on an iterative design rather than a recursive one (section 3.2.2), such that complex control flow can be represented more efficiently than using host language mechanisms (section 3.2.4). One can then apply a pre-order transformation to rely on a compact tree structure and offset-based references (section 3.2.3) and thus, benefit from improved cache locality and possibly improve performance. The representation can then more easily leverage the underlying machine in other ways (section 3.2.5), such as by relying on a set of registers instead of a stack, have more efficient dispatching e.g. using threaded code, and overall exercise greater control over the language.

With this model of a spectrum for the design space, a compact tree structure would represent roughly a midpoint, being ambiguously both AST-like and BC-like, which

matches our observations in section 3.3.3 that this design narrows the difference between AST and BC.

Though, not all possible design decisions will necessarily be comparable or enable one another, as all points of a spectrum do not necessarily form a total order. So, we find it also helpful to think of design decisions as forces that push or pull an interpreter closer towards one end of the spectrum. For instance, a CPS-style interpreter as discussed in section 2.1.1 may rely on recursion. This design decision implies a degree of reliance on the host language, and pushes it towards the AST-like end of the spectrum. However, it also reifies the program control state as a continuation, which reduces its dependence on host language mechanisms, which pulls it closer to the BC-like end of the spectrum. One can assume these forces to have different magnitudes depending on the design decision in question, though even roughly quantifying them is a difficult problem.

3.4.3 Advantages of AST-like

Benefits of Utilizing the Host Language Having defined AST-like to mean "benefiting from host language abstractions", these interpreters tend to leverage the existing host language facilities, such as the host language call stack. This makes them easier to implement (Basso et al. 2023; Humer et al. 2014b; Kalibera et al. 2014; Würthinger et al. 2012), and it is a key advantage. It makes getting a first implementation to a functional state faster, it allows for quicker experimentation, and saves on engineering effort that can then be spent elsewhere. As BC-like designs work with a representation closer to that of machine code rather than source code, this can make it harder to reason about and require custom tooling for debugging. While a structure like that of bytecode relies on jumps to represent control flow, an AST close to the original code syntax would keep the control-flow structures present in the guest language, e.g., `if` and `while` statements. An AST then represents the control-flow operations with specific nodes, e.g., `IfNode`, `WhileNode`.

This closeness to explicit structured control-flow mechanisms can make debugging an interpreter easier. It can help to reason about the execution path of a program

by seeing the AST structure, which is harder with a BC structure. For an AST interpreter that relies on the host language stack, they benefit from the debugging support of the host language, and variables will be named and stored explicitly on the host language stack. For a bytecode interpreter with a custom stack or registers, one may have to maintain custom debugging information to identify stored variables. Though, this debugging information may be more easily centralized since it is not scattered across host language stack frames, which may help inspect the current context and therefore make debugging easier for bytecode interpreters. However, this does not remove the implementation cost of this custom tooling, and we nonetheless find our AST interpreters, used in Chapter 4 and Chapter 5, to be easier to debug overall.

AST nodes are also likely to be implemented using host language objects, which enables us to inspect them in a debugger. Many languages also provide facilities to pretty-print host languages objects for debugging, such as the `Debug` trait in Rust. In comparison, bytecode is more likely to be implemented in a custom way, so that we need a custom decompiler to show it as readable text.

In general, relying on the host language reduces the need for custom tooling, e.g. bytecode disassemblers, making debugging and implementation easier, and therefore reducing the overall engineering effort. In addition to these engineering benefits, parts of the implementation may also see performance benefits. For instance, in a recursive AST interpreter, it is straightforward to have more coarse nodes to implement loops with a `ForLoop` node or a `ForEachLoop` node. While a bytecode interpreter would typically encode these semantics with multiple bytecodes, each of which incurs dispatch overhead, an AST interpreter makes it natural to use coarser nodes, which also benefit from the host language stack for intermediate values, instead of having to store them on a guest stack or in a register.

Benefits of a Non-compact Representation When AST nodes are represented as host-language objects, an advantage of a pointer-based representation is that it enables specialization of nodes by replacing them based on run-time feedback, for instance in the sense of self-optimizing AST interpreters (Würthinger et al. 2012), which also facilitate partial evaluation to enable just-in-time meta-compilation (Würthinger et al. 2017, 2013). With pointer-based representations, specialization and optimization have

many freedoms, where an in-place specialization may be limited by the available space of the original node.

For bytecodes, quickening (Brunthaler 2010a,b) is such an optimization, but because bytecode are generally a compact representation, quickening a bytecode to a more complex structure requires additional mechanisms, especially when it needs more memory. For instance, one can use side tables to represent cached information. One example is the Bytecode DSL (*Bytecode DSL* 2025; Humer et al. 2022) for Truffle interpreters, which uses side tables to enable, for instance, type specialization and caching. Another is Wizard, which uses side tables to enable efficient in-place interpretation of Wasm bytecode (Titzer 2022).

Alternatively, the bytecode sequence could be reallocated to make space for additional information to be inserted. However, such inserting may then require jump offsets to be adjusted, or one to use a trampoline or self-adjusting mechanisms, where offsets are adjusted lazily on the next execution, introducing more complexity and maintenance challenges.

For any regular jump offset or trampoline, tooling and correctness checks are desirable to ensure that it is pointing to the correct bytecode. Meanwhile, a pointer to a host language method in the AST will have been intrinsically validated by the host language and possible type checks.

3.4.4 Advantages of BC-like

Since we defined BC-like interpreters to rely less on existing host language mechanisms to be closer to the target machine, avoiding these mechanisms implies greater engineering effort. Though, moving away from host-language mechanisms can also be beneficial, such as when the guest language is limited by the host language, i.e. that there is a *semantic mismatch* (Bolz et al. 2013b). For instance, a `goto` mechanism or `call/cc` can be cumbersome or impossible to efficiently implement with maximal host language use. This instead becomes much easier when using a bytecode interpreter and the guest-language stack is reified.

Another advantage is that since such interpreters typically work with compact

structures, e.g., a byte-based program representation, this can help reduce the memory footprint of a program. Specialized hardware such as for embedded devices may only have a limited amount of available memory, making a compact program representation mandatory.

Finally, a key argument for minimizing distance with the target machine is, of course, better run-time performance. It is a primary concern for many language implementers, and so a major argument for BC-like designs. Gaining finer-grained control over the execution enables many optimizations, and helps ensure excellent performance. As such, a highly optimized BC interpreter is widely considered to be faster than a highly optimized AST interpreter (Nystrom 2021).[5]

3.5 Summary Table of Interpreter Design Dimensions

In this chapter, by highlighting the existence of several hybrid approaches in section 3.3, we motivated the terms *AST-like* and *BC-like* which we defined in section 3.4. Based on these definitions, and on our observations of commonly held opinions about AST and BC interpreters in section 3.2, we now define a set of traits that we find to be commonly associated with AST-like and BC-like designs. We find these contrasting traits to be the most notable when analyzing the interpreter implementation space by focusing on constrating AST and BC interpreters.

	AST-like	BC-like
Overarching design decisions		
Closer to:	Host language	Target machine
Presumed Performance	Overall lower	Overall higher
Design dimensions		
Program Repr. Layout	Sparse/Separate, Non-Linear	Compact, Linear
Program Repr. Uses:	Host language objects	Is Reified/Byte-based
Main dispatch mechanism	Virtual dispatch	Main dispatch loop
Typically Uses Recursion	Yes	No, avoids host stack

TABLE 3.1: Table of design dimensions for AST-like and BC-like interpreters, highlighting traits that we identify to be associated with either design.

[5]While there are many anecdotes to this effect, we are not aware of any experiments showing this scientifically for a system-level host language.

These design dimensions are outlined in table 3.1. We first show overarching high-level traits of AST-like and BC-like interpreters, with AST-like interpreters being closer to the host language with generally lower performance, and BC-like interpreters leaning closer to the target machine. A key reason for BC-like interpreters prioritizing the target machine is run-time performance, and so generally speaking, we can assume they have better run-time performance than AST-like interpreters. Though, in this thesis, we find this to not necessarily always be the case, as we will most notably see in Chapter 4.

We then define some design dimensions, which can be broadly divided into two categories, the representation of the interpreted format, and how execution takes place.

Regarding the format, we define AST-like interpreters to rely on a sparse and non-linear representation (see section 3.2.3), while BC-like interpreters rely on a purposefully compact and linear representation instead. This often comes with AST-like interpreters relying on host language objects, which are usually allocated by the host language sparsely on the host language heap, while BC-like interpreters often instead directly rely on a set of bytes which they keep contiguous in memory, and so rely less on the host language.

For execution, we find AST-like interpreters to rely not just on host language objects, but also on host language dispatch mechanisms, i.e. virtual dispatch. Meanwhile, BC-like interpreters usually reify this dispatch mechanism themselves by implementing a main dispatch loop, with a conditional branch redirecting to the correct handler based on the current instruction, and with no virtual dispatch involved. This main dispatch loop is especially important since it usually means an interpreter does not rely on a recursive structure (see section 3.2.2) as it avoids the use of the host language stack entirely. Recursion can occur at the level of expressions, though as we saw in section 3.3.1, one can also design BC-like interpreters that are recursive only at the level of guest-language functions. We find recursion to be a trait associated with AST-like interpreters, as this means we rely on the host language stack for part of the core interpreter execution, which can warrant the use of host language exceptions (see section 3.2.4).

As we consider these traits to be key interpreter design dimensions, we build

upon them to discuss the design of our runtime systems in the following chapters, as they allows us to distinguish design decisions in our interpreters as AST-like or BC-like. Defining these traits thus enables the next chapters, and so these dimensions are prevalent thoughout this work.

3.6 Related work

D'Hondt (2008) explored the idea of bytecode being an outdated program representation, having been originally designed to mimic machine code and not having been revisited since. He argued that staying as close as possible to the semantics of the program, by relying on a graph-like structure, may not be as suboptimal as often assumed, and that bytecode was not the pinnacle of interpreter design. Our work also supports the idea of exploring designs beside bytecodes, but focuses on identifying and discussing different design dimensions where AST and bytecode are merely ends of a spectrum.

Körner et al. (2021) compare AST and BC interpreters implemented in Prolog. They note that recursive AST interpreters work well with Prolog's recursive execution model. They achieve good performance with an AST interpreter and find that it outperforms a bytecode interpreter on top of Prolog. This is a good example of how the host language needs to be considered when choosing an interpreter design, and one needs to make sure that the two fit together.

Several works compare AST and bytecode interpreters in terms of run-time performance. In Chapter 4, we will investigate the performance in the context of meta-compilation systems where one implements an interpreter and the runtime system provides JIT compilation. Within the context of meta-compilation, we find that AST interpreters can rival the performance of bytecode interpreters. Niephaus et al. (2018) on GraalSqueak, and Kalibera et al. (2014) on FastR also study the performance angle in different contexts. Though, all three of these works focus on performance and do not explore the design space itself, which is what our work focuses on.

Last but not least, Midtgaard et al. (2013) investigated how to engineer definitional interpreters to reach performance closer to that of practical bytecode interpreters. They focus on what they call control context, i.e., where and how execution continues

after evaluating an expression. They experiment with families of interpreters that either use the host language, reify control context as a data structure, or a continuation. They also experiment with designs close to bytecodes, but stay brief since their host language OCaml limited them, thus being an example of semantic mismatch. Most notably, they evaluated hybrid designs that use a mix of control context representation for different parts and achieve good results. Interestingly, for them, using the host language for control context seems to have given them their fastest interpreter METACON-LETTMP.

3.7 Conclusion

In this chapter, we have explored characteristics typically associated with either *abstract syntax tree* or *bytecode* interpreters. We then showed that these characteristics could feature in either design and that hybrid approaches are possible and useful. We call interpreters with hybrid characteristics *AST-like* and *BC-like*. We propose to reason about AST and BC interpreters in a novel way: as extremes on a spectrum, with AST-like meaning *close to the host language* and BC-like designs to mean that they *minimize the distance with the target machine*.

We believe that there are more than a select few options when designing an interpreter: it is more than AST or bytecode, or merely stack-based BC or register-based BC. Instead, reasoning about the design space in terms of proximity to the host language or to the target machine allows us to consider engineering as well as performance trade-offs. In the end, there is a fundamental push-and-pull between better run-time performance and the cost of engineering, which is a classic software problem extending far beyond interpreters.

This high-level overview of the interpreter design space, and the terminology we established, will be relevant for the next chapters to go more in-depth with specific aspects of interpreter design. Though, since the terms *AST-like* and *BC-like* were coined by our work and are not standard terminology, we will henceforth refer to classic AST interpreter designs as simply "AST", and classic BC interpreter designs as simply "BC". We will rely on the terms "AST-like" and "BC-like" as needed, to highlight deviations from more typical interpreter designs.

Chapter 4

Interpreters on Meta-Compilation Systems

Interpreters execute code directly and immediately, but have low performance. They are thus commonly paired with just-in-time compilers, which can compile efficient native code at run time, but first need the program to be executed and profiled to gather run-time feedback and identify valuable compilation targets.

While interpreters are lightweight and comparatively easy to implement, JIT compilers are much more complex systems that require far more engineering effort. However, thanks to *meta-compilation* systems such as RPython (see section 2.4.1) and GraalVM (see section 2.4.2) it became possible to use existing just-in-time compilers, garbage collectors, multi-threading support, tooling (Van de Vanter et al. 2018), and language interoperability (Grimmer et al. 2018), all without investing the hundreds of person years of engineering effort that went into systems such as the HotSpot JVM, GraalVM, and V8 JavaScript. This can be very beneficial for many language projects, since most do not have such a great amount of resources. To benefit from meta-compilation, one only needs to implement an interpreter. Like with most VMs, this interpreter is used to execute code immediately and identify targets for the JIT compiler, but the interpreter is also the base from which the JIT code gets generated.

Therefore, meta-compilation is a key domain in which to analyze the design space of interpreters, given the central role played by the interpreter in this context, notably in terms of run-time performance, peak performance, and memory usage. And a major decision in the design of any interpreter is the choice of representation, from *AST-like* to *BC-like*.

Notably, several interpreters implemented on top of GraalVM are AST interpreters, with naturally-recursive (see section 2.1.1) designs. While Graal's partial evaluation approach, combined with just-in-time compilation, allows language implementations to reach peak performance that is comparable with V8 (as we later see in section 4.3.2), it can not yet compete with custom-built systems when it comes to memory use and warmup performance. Given these issues and commonly held negative views on AST interpreters (see section 3.2), one may assume AST to be the cause, which may contribute to GraalVM now facilitating the implementation of bytecode interpreters as well (*Bytecode DSL* 2025; Humer et al. 2022).

However, on meta-compilation systems, we find that this generally accepted truth does not hold given the constraints and opportunities of such systems. To carefully study the trade-offs between interpreters at both extremes of our design spectrum, we implemented four interpreters: an AST and bytecode interpreter each on top of both RPython and GraalVM. PySOM is built on top of RPython and comes with a self-optimizing AST interpreter as well as a bytecode interpreter that supports quickening. TSOM (TruffleSOM) is built on top of GraalVM and uses the TruffleDSL to implement a self-optimizing AST interpreter and has a custom bytecode interpreter that utilizes many of the self-optimization techniques, too. By investigating different interpreter designs on both RPython and GraalVM, we believe our conclusions may generalize to other meta-compilation systems.

All four interpreters implement the SOM language (see section 2.2), and are designed with the design dimensions commonly associated with AST and BC designs in mind (see section 3.2) to justify labeling our systems as AST/BC interpreters, though our BC interpreters have some AST-like traits, as we will see in section 4.1.2. The interpreters are as similar as possible to each other to facilitate direct comparison, while exploiting some minor performance opportunities where available (see section 4.1.6). They also implement many common interpreter-level optimizations, including polymorphic inline caching, inlining of control structures, self-optimization and bytecode quickening, supernodes and superinstructions, as well as optimizing object storage models and storage strategies for arrays.

Our interpreters reach peak performance similar to V8 using the Are We Fast Yet benchmarks (see section 2.7) as we show in section 4.3.2. Thus, the meta-compilation

systems of RPython and GraalVM are effective, and the SOM implementations are fast enough to enable a study of trade-offs between bytecodes and ASTs in the context of such systems.

To assess the trade-offs between AST and bytecode interpreters, we compare interpreter performance as well as just-in-time-compiled peak performance. We find that AST interpreters are on par in run-time performance, and in some cases even moderately faster. Thus, bytecode interpreters do not have the generally expected advantage on top of meta-compilation systems. When looking at the overall memory use, we even find that bytecode interpreters have drawbacks because boxing avoidance is not as applicable, and their interpreter structure requires additional allocations. However, we can also confirm that bytecodes are more compact in memory, which is important for large codebases. But, this benefit narrows when the code is executed and collects profiling information.

For completeness, we also assess the warmup behavior and the performance impact of optimizations that improve AST or bytecode performance.

The contributions of this chapter are:

1. the implementation of TSOM_{AST}, TSOM_{BC}, $\text{PySOM}_{\text{AST}}$ and PySOM_{BC}, to compare interpreters on top of GraalVM as well as RPython, and an overview of their design based on the design dimensions that we defined in Chapter 3.
2. the design of an experimental methodology to identify run-time performance and memory usage trade-offs between AST and bytecode interpreters.
3. an analysis that shows that bytecode interpreters cannot be assumed to be faster on top of current meta-compilation systems.

4.1 Overview of TSOM/PySOM

The experimental subjects for this chapter are the four SOM interpreters that we will refer to for brevity as TSOM_{AST}, TSOM_{BC}[1], $\text{PySOM}_{\text{AST}}$, and PySOM_{BC}[2], distinguishing the abstract syntax tree (AST) and bytecode (BC) variants. Since TSOM is

[1] https://github.com/SOM-st/TruffleSOM
[2] https://github.com/SOM-st/PySOM/

based on GraalVM using partial evaluation, and PySOM builds on RPython using meta-tracing, we cover both meta-compilation approaches.

All four interpreters were designed with these priorities in mind:

- being as strongly AST-like and BC-like, as possible, to meaningfully compare interpreters on both ends of our design spectrum as much as meta-compilation systems allow, and explore their associated design decisions.
- ensuring the notable difference between $TSOM_{AST}$/$TSOM_{BC}$ and $PySOM_{AST}$/$PySOM_{BC}$ to be the program representation and its implications.
- similar designs between $TSOM_{AST}$/$PySOM_{AST}$ and $TSOM_{BC}$/$PySOM_{BC}$ to enable meaningful comparisons across meta-compilation frameworks.
- run-time performance.

While we previously mentioned several hybrid AST/BC approaches on top of GraalVM and RPython (see section 3.3.4), our focus in this chapter is on comparing AST and BC interpreters, and so our systems were designed to be as strongly AST-like and BC-like as possible, basing this definition on the design dimensions we previously listed in section 3.5.

Within a meta-compilation system, we ensure that the main difference between the AST interpreter and BC interpreter is the program representation, and the design impact of relying on this representation. We aim to share as much as possible between the implementations that is independent of AST-like or BC-like design decisions, which includes for instance the implementation of built-in functions such as loading of files and producing output. They also share the same object model and storage strategies, as we will discuss in section 4.1.4. Moreover, any optimization is also implemented in the other interpreter, or has an optimization that serves as a counterpart implemented instead. And wherever possible, design decisions are matched in interpreters across meta-compilation frameworks. This also implies that our systems have all received a similar amount of engineering effort.

Finally, run-time performance is a key concern, which is true of both interpreter performance and peak performance. This led to the implementation of many optimizations, and impacted the design of our interpreters in ways that we describe shortly, in section 4.1.3.

In preparation of the experiments sketched in section 4.2, we give an overview of the four interpreters, and describe the key high-level design decisions taken in our systems to make them AST-like and BC-like. We also detail the optimizations implemented, and discuss relevant differences between the implementations.

4.1.1 AST Interpreters

As previously mentioned, TSOM is built on top of the Truffle framework and uses the TruffleDSL to implement a self-optimizing AST interpreter. PySOM on the other hand, is implemented in RPython, which gives us a bit more freedom but also does not provide the conveniences of the TruffleDSL. In practice, this means we often keep the RPython interpreter simpler, but without sacrificing performance, as we will see in section 4.3.1.

Both AST interpreters implement a variety of nodes, including nodes for reading and writing method arguments and local variables, nodes for reading global variables, literals, instantiating blocks, or sending messages, i.e., looking up and activating methods on a receiver object. A full list of nodes is given in appendix B.1.

Our AST interpreters were designed to be strongly AST-like. We now discuss this by going over several traits and features implemented in both TSOM_{AST} and $\text{PySOM}_{\text{AST}}$, which we previously found to be associated with AST interpreters in section 3.2.

Non-Compact Representation Each AST node is represented by a host language object. These objects thus come with metadata such as object headers, and refer to one another with machine-word-sized pointers, which has an additional memory cost but allows us to benefit from host-language tooling for implementation and debugging, as well as enabling self-optimization (Würthinger et al. 2012). Additionally, they are not guaranteed to be stored next to one another in memory, therefore the representation

itself is fragmented and may lead to pointer chasing. Though, despite this potential issue, our AST interpreters achieved good performance, as we see in section 4.3.1.

Recursive Design Both AST-based interpreters have a naturally-recursive design (see section 2.1.1). Each node implements a virtual `execute` method which defines its behavior, and takes in the frame of the current context. For example, the `execute` method for an `ArgumentRead` node may simply read and return the Nth argument from the frame. Since this method is virtual, any node can store a reference to any other node as one of its children, and invoke it by doing a call to its `execute` method. Such a design naturally uses the host language stack, which is populated by recursive calls from nodes to other nodes. A similar AST interpreter design was previously shown in listing 2.1.

This design was mandated by the TruffleDSL, which is designed to work with nodes implementing such virtual methods. We matched this design decision in PySOM to meaningfully compare TSOM_{AST} and $\text{PySOM}_{\text{AST}}$, and because we find it to be a natural way to implement an AST interpreter, easily handling all the many different possible nodes, and allowing for easier integration with host language tooling, and self-optimization.

Exceptions For Non-Local Control Flow We use host language exceptions to model non-local control flow, as described in section 3.2.4. As TSOM and PySOM are written in Java and Python respectively, handling SOM non-local returns is done most idiomatically using host language exceptions, since they unwind the host language stack. This allows the language implementations to unroll the stack until finding the correct context to return to.

4.1.2 BC Interpreters

Both TSOM_{BC} and PySOM_{BC} are stack-based. Just as our AST interpreters were designed to be strongly AST-like, our bytecode interpreters were designed to be as BC-like as possible. Though because of the constraints of meta-compilation, our BC interpreters leverage the host language in more ways than others, and so are AST-like in a few aspects.

Compact Bytecode Set Our interpreters rely on a compact, byte-based representation. Each bytecode and each argument of every bytecode is represented with a single byte, stored in a byte array containing all bytecodes for a given scope.

The original SOM interpreter had merely 16 instructions[3], while our bytecode set was designed to be more complex, to achieve good performance. Both TSOM_{BC} and PySOM_{BC} have roughly 70 different bytecodes, with the full set listed in appendix B.2. We optimized it by introducing variants that directly encode certain parameters. To give two examples, pushing `self` on the stack, i.e., the first method argument, is very common and thus, we introduced `PUSH_ARG_0`. Pushing the `nil` value is also common, and done with a specialized superinstruction `PUSH_NIL`. These specialized bytecodes avoid the extra byte to encode the argument, and at the same time, they can be slightly more efficient, because for instance `PUSH_NIL` does not need to load the value from the literal array but pushes a constant directly to the stack.

Recursive/Iterative Design Our interpreters rely on recursion at the guest-language-method level, and were previously used as an example in section 3.3.1. That is, they feature a main bytecode dispatch loop, but within a function that gets invoked recursively for each guest language function. The bytecode index is kept in a local variable as opposed to stored within a frame object. Therefore, while they rely on an iterative main dispatch loop, which iterates over all bytecodes in the sequence, this dispatch loop does get invoked recursively for each guest-language method.

There are two main reasons for this design choice. First, for Truffle-based interpreters, an implementation is forced to be recursive in this way at the guest-language method level, since each of these methods must be invoked after providing an `Object[]` argument array and a `Frame` object. Second, we found that it helped PySOM_{BC} generate more efficient code, and so achieve better peak performance. This may be because this design may make it easier for meta-compilation systems to identify precisely which frames correspond to which methods, and so which methods are being executed, which is important information to generate well optimized code.

Though we find this design to be valuable to our interpreters, we find it to be an example of the host language, or underlying system, pulling the design of the

[3] https://github.com/SOM-st/som-java/blob/original/src/som/interpreter/Bytecodes.java

interpreter closer towards one end of the AST-like/BC-like spectrum. In this case, we argue that this decision makes the interpreter more AST-like, since it introduces recursion to the core interpreter loop. Indeed, this use of the host language stack also means that non-local returns are handled using host language exceptions.

Exceptions For Non-Local Control Flow As with our AST interpreters, because our designs rely on recursion, we need exceptions as a host language mechanism to unroll the stack. Non-local returns in TSOM_{BC} and PySOM_{BC} thus also use a `ReturnException`, which is caught by exception handlers which we add to previous scopes. While this decision stems from our recursive design, which we found to be a valuable design in our case, this may negatively impact the performance of the interpreter (Basso et al. 2023), though such a design grants better peak performance in PySOM and is unavoidable in TSOM.

Therefore, our meta-compilation-based BC interpreters have some AST-like traits because of the context of the meta-compilation frameworks that we rely on. We focus on more strongly BC-like interpreters in Chapter 5, where we rely on interpreters written directly in a system-level language.

4.1.3 Optimizations Common to All Four Interpreters

We first compare optimizations present in all AST and bytecode interpreters, so that we may later analyze their individual effects in section 4.3.5. Our general approach is to have the interpreters as similar as practical, which also includes optimizations. The optimizations described in this section were previously defined in section 2.3, though we also introduce a few optimizations specific to our meta-compilation-based interpreters in section 4.1.4.

Polymorphic Inline Caching

All of our interpreters rely on polymorphic inline caches (see section 2.3.1). For GraalVM, this optimization also enables inlining across SOM methods. While this is not strictly needed for RPython with its meta-tracing, the PIC avoids unnecessary repeated lookups during tracing, too.

In both TSOM and PySOM, the maximum number of entries in the polymorphic cache is six, a maximum which is very rarely reached in practice with our benchmark set.

Global Variable Caching

As seen in section 2.3.2, SOM has the notion of global variables. While they can be mutated, this hardly happens in practice, and so we cache their values and treat them as constants. This is highly valuable for the JIT compiler since treating values as constants leads to generating more optimized code, and potentially enables constant folding and constant propagation (i.e. precompute constant values, and substitute the known constants for their values). In case a global is mutated, compiled code using them will be invalidated to ensure correct semantics.

In the AST interpreter, access to globals is implemented with the `Global` nodes. For `true`, `false`, and `nil`, we have nodes with the corresponding value. For other globals, they start out in an uninitialized state, and rewrite themselves to a `CachedGlobal` after looking up the global.

The bytecode is very similar and has `PUSH` operations. Though, we only added `PUSH_NIL` for literals, since `true` and `false` are very rarely used directly. For normal globals, `PUSH_GLOBAL` will rewrite itself to `Q_PUSH_GLOBAL`, which uses the result of the lookup.

Inlining Control Structures

The bytecode interpreters support the inlining of control structures (see section 2.3.2) for the methods for conditional branches, the `#and:` and `#or:` methods on booleans, and while loops. SOM features a `#to:do:` counting loop which was not included since it did not yield the desired performance. To support the inlining, we added the `JUMP` bytecodes listed in the appendix B.2.

The AST interpreters inline the same methods as the bytecode interpreters, but also `#to:do:`. During inlining, method call nodes are replaced by specific nodes for the control flow structures. e.g., an `InlinedIfElse` node (see appendix B.1). The lambda arguments are inlined into the calling method on the AST level, too.

Lowering of Basic Operations

Like control structures, basic operations such as arithmetic operations and comparison operators are also implemented as normal SOM methods, which is not very fast (see section 2.3.2). Therefore, PySOM and TSOM provide primitive versions for methods implemented in SOM's core library to improve performance.

Since $TSOM_{AST}$ is using the TruffleDSL, it also uses what we call *eager operations replacement*. Instead of relying on the polymorphic inline cache to resolve operations, we directly replace standard method call nodes for operations such as + and >= with AST nodes that specialize on argument types and implement the necessary semantics. This avoids the method call overhead and enables boxing elimination on the statement level $TSOM_{AST}$, because statements that work on integers can use code paths that are consistently typed for integers, without having to fall back on a representation that encapsulates the integer value in an object.

Since RPython does not have the TruffleDSL, and the eager replacement is not applicable to bytecode interpreters, $PySOM_{AST}$, $PySOM_{BC}$, and $TSOM_{BC}$ do not use eager operations replacement. However, the extra flexibility of RPython allows us to handle activation of basic operations differently without extra overhead, such as by specializing primitive dispatches based on one, two or three arguments instead of always providing an array. Furthermore, all interpreters benefit from the primitive versions of SOM library methods.

Trivial Methods

All our interpreters directly handle trivial methods, as described in section 2.3.3. This means that simple methods like getters or setters are detected at compilation-time and are marked as *trivial*, meaning that their behavior is simple enough that we can avoid allocating a stack frame for them. Given that these methods are relatively common, speeding up their dispatch is beneficial to reducing the overall run time.

TSOM and PySOM directly handle getters, setters, methods that simply return a literal value, and methods that simply return a global value.

4.1.4 Optimizations Orthogonal to Program Representation

PySOM and TSOM implement a number of other optimizations to reach the performance level of Node.js. Though, since they are orthogonal to the program representation, we will only briefly sketch the object model and use of storage strategies as the most important ones, and do not assess their performance impact.

Unboxed Number Storage in Object Fields The object model takes inspiration from maps (Chambers et al. 1989) and the object model used by GraalVM (Wöß et al. 2014). Its main benefit is unboxed storage of SOM integer and double values, which is especially beneficial in compiled code. Though, the object model is simplified to avoid the need of a transition tree and reduce the problem of high map-polymorphism (i.e. polymorphism from a large number of possible maps/object layouts). Instead, each class tracks only the latest layout, i.e., mapping between logical fields and physical storage. Objects are transitioned to the latest layout lazily, avoiding a global impact when the layout changes.

Unbox Storage in Arrays PySOM and TSOM implement storage strategies (Bolz et al. 2013a; Pape et al. 2015), an optimization which allows for efficient storage of homogeneous collection types. Since collections can store data of different types, implementations usually rely on boxing values. But if the collection is homogeneous, boxing can be avoided entirely by instead representing the collection in memory with all of its elements stored consecutively, and without storing type information for individual elements.

To achieve this, collections are able to shift between different *strategies* for specific types, and fall back to a generic strategy if the data becomes heterogeneous. Our interpreters currently support strategies for SOM integers, doubles, and booleans. In addition, they also support a strategy for empty arrays, i.e., arrays that store only `nil` values. In this case, the array only stores the length, which avoids allocating memory when not yet needed.

As with the object model, since strategies help avoid storing values in their boxed form, this is especially beneficial in compiled code.

4.1.5 Optimizations Applied to a Subset of Interpreters

Because of the differences in implementation platform, and between AST and bytecode, the following optimizations are only available in some interpreters. We will detail key differences in section 4.1.6. Note that superinstructions, quickening, and supernodes are nominally different optimizations. However, we apply them in a way that the resulting interpreters are as similar as possible to ensure fairness in the comparison.

Superinstructions

We use several superinstructions (see section 2.3.4) that we identified as common patterns based on bytecode profiles. This includes `INC` and `DEC` bytecodes, which increment or decrement the top of stack value, `INC_FIELD` and `INC_FIELD_PUSH`, which increment the value stored in a field, and possibly push the result onto the stack, as well as `RETURN_SELF` or `RETURN_FIELD_n`, which combines taking the `self` value or reading a field from `self` with returning either at the end of a method. Getters are perhaps the most common methods that end with reading a field.

Quickening

Our interpreters use quickening (see section 2.3.4) to incorporate run-time feedback into bytecode, for instance for the method-call bytecodes. The initial bytecodes initialize the polymorphic inline caches on first execution, and then rewrite themselves to quickened versions `Q_SEND[_n]`, which merely use the caches. Similarly, as discussed in section 4.1.3, we use quickening for access to global variables and introduced the `Q_PUSH_GLOBAL` bytecode. For these examples, quickening avoids run-time checks and improves the performance of method calls and access to globals. In the AST interpreters, we use self-optimization (see section 2.3.4), which is very similar to quickening in that it also relies on rewriting at run time, however, because of the tree structure of ASTs, it is more flexible.

Supernodes

We applied supernodes (see section 2.3.4) to $TSOM_{AST}$ based on frequent node patterns observed in the interpreter. They reduce dispatch overhead for common operations by avoiding virtual calls to the `execute` methods of the now-merged child nodes, but supernodes are also beneficial since they reduce memory use. The number of node objects is reduced by combining several nodes into one, which in turn removes several Java object headers, and pointers to link nodes with one another, including parent pointers.

An example of a supernode we implemented is the `IntInc` supernode, matching the `INC` bytecode (previously used as an example of a supernode in Chapter 2 with listing 2.6). It is applied in situations where we have an addition of a literal integer to a value obtained from an arbitrary subexpression.

Another set of supernodes fuses method call nodes with an argument read node for the receiver, e.g., with the `BinaryArgSend` node. We also added common comparisons with literal values, which simplifies the comparison logic since the literal value is statically known. Our supernodes for a few selected arithmetic expressions include squaring a value and squaring a value read and stored back into a field, which enables us to significantly reduce redundant type checking for this pattern.

4.1.6 Overview of Differences Between the Interpreters

Given the constraints and guidance provided by RPython and GraalVM with its TruffleDSL, there are some differences between the interpreter implementations that may lead to different trade-offs.

GraalVM provides a more strict framework requiring the use of a predetermined calling convention between guest-language-level methods and a provided `Frame` abstraction that does type profiling of local variables, including boxing avoidance for primitive values. For the TSOM interpreters, this means calling between SOM methods has a relatively high overhead because an `Object[]` array needs to be allocated to pass method arguments, and the `Frame` structure needs to be constructed, which involves multiple objects and arrays. This means our Truffle-based interpreters must necessarily be based on recursion at the level of guest language methods, for

both our AST and BC interpreters, and so our interpreters always rely on the host language in that aspect.

Since RPython does not provide and require such a strict framework, we had more leeway. We represent method activations with a simple array, and constructing frames is not as complex as it is for Graal. Similarly, there is no calling convention, which allowed us to specialize the calls to methods with one, two, and three arguments by providing explicit methods for it, and only requiring an allocation for an array to pass arguments when four or more arguments are needed, which happens much less frequently. So while TSOM_{BC} had more rigid design constraints, we could have instead opted not to rely on recursion in PySOM_{BC}. Though, meta-tracing gave better performance from a recursive interpreter structure, as previously mentioned in section 4.1.2.

On the other hand, the TruffleDSL makes it straightforward to provide specializations for primitive values such as `long` and `double`, which enables us to avoid boxing of such values on the statement level in the TSOM_{AST} interpreter (Humer et al. 2014a; Würthinger et al. 2012). This enables nested expression nodes to pass temporary values between them as unboxed values.

In PySOM, we do not do any such specializations and all values in the interpreter are always boxed, i.e., stored in an object, except when storing a value in a SOM object or array (see section 4.1.4).

The TSOM_{BC} interpreter does not benefit as much from avoiding boxing since the bytecode structure does not directly benefit from the TruffleDSL-provided specializations, and temporary results of expressions are stored in an explicit stack, which is realized as an `Object[]` array, and thus, requires the boxing of `long` and `double` values. However, the TSOM_{BC} interpreter still uses the same node objects for basic operations that the TSOM_{AST} interpreter uses. This allows us to benefit from the profiling built into these nodes, to facilitate partial evaluation.

4.2 Methodology to Compare AST and Bytecode Interpreters

Our goal is to characterize the performance trade-offs of AST and bytecode interpreters in the context of meta-compilation systems. We focus on run-time performance and memory usage, since these are the two key aspects commonly named when preferring bytecode over AST interpreters.

Since meta-tracing and partial-evaluation-based meta-compilation have different trade-offs (Marr et al. 2015), we use both platforms to get a complete picture. This experimental methodology is designed to compare AST and bytecode interpreters, and is therefore not limited strictly to the domain of meta-compilation. So, we also later apply it in Chapter 5, where we compare AST and BC interpreters written in the Rust language.

4.2.1 Interpreter Performance

The perhaps most important aspect for us is the run-time performance of the interpreters before just-in-time compilation. Here the interpreters have the highest impact on the observable performance.

For a wide range of workloads, the interpreter performance is crucial for the user experience. This includes highly interactive and often short-running applications such as scripts on websites, many short-running command-line scripts, large-scale applications with frequent redeployments many times a day (Ottoni et al. 2021), and perhaps most important for the acceptance of new language implementations: the run time of unit test suites.

To assess interpreter performance, we compile the interpreters to stand-alone binaries without JIT compilation support. Both RPython and GraalVM are able to produce binaries with the interpreters ahead-of-time compiled for best interpreter performance.

4.2.2 Peak Performance

In the context of meta-compilation systems, we are also interested in the best possible performance that our language implementations can achieve. Thus, we want to assess

the performance with JIT compilation enabled, and the benchmarks running long enough to be fully compiled.

This *peak performance* is especially relevant for long-running applications. However, even short-running applications may benefit when the executed code is compiled early on, and thus, can reduce the overall response time.

To assess the best possible, i.e. peak, performance, we use both RPython and GraalVM with JIT compilation enabled. In the case of RPython, this is a native binary of the interpreter with the JIT compiler included as well. For GraalVM, we run TSOM on top of the HotSpot JVM using the Graal compiler for JIT compilation of the SOM programs. This configuration is recommended to gain the best performance overall, since for instance the garbage collector and other runtime elements of HotSpot are more sophisticated than in the ahead-of-time compiled binary we used to assess interpreter performance. However, using HotSpot may have a negative impact on warmup performance, because TSOM starts executing like any other Java program, which means first it is executed by HotSpot's Java interpreter and the TSOM interpreter code itself may be JIT compiled by HotSpot before the Graal compiler can JIT compile the running SOM program.

Both RPython and GraalVM rely on activation counters that once they reach a threshold trigger compilation. For RPython, we verified that the standard thresholds lead to a full compilation of the used benchmarks. The GraalVM's runtime system is more sophisticated, which however also makes it less predictable. To ensure full compilation, we adapted its settings by reducing the compilation threshold, disabling multi-tier compilation, and dynamic compilation thresholds.

While there is no guarantee that an arbitrary program can reach peak performance (Barrett et al. 2017), we adapted compilation settings and we execute the benchmarks long enough to ensure that all parts are compiled.

4.2.3 Warmup Performance

The phase between starting execution and reaching peak performance is the warmup phase.

In RPython, tracing and compilation happen on the same thread as the program execution. Thus, compilation stalls application progress and may be noticeable in the warmup behavior. RPython starts to trace and compile a loop after 1039 iterations. For functions, the threshold is higher at 1619 activations. This will give hot loops, which are likely more relevant for performance, a chance to compile before individual functions, typically resulting in better overall performance.

GraalVM supports background compilation, has a multi-tiered compiler, and uses a system of dynamic compilation thresholds, which is meant to carefully balance the compilation work with the resources needed by the application. In contrast to RPython, the background compilation ensures that the application can continue executing, while another thread compiles the function that reached the compilation threshold. To measure warmup on this system, we run both on top of Hotspot and the ahead-of-time compiled version. The goal here is to see whether the difference between AST and bytecode interpreters has an influence on the warmup behavior in either system. We also use the ahead-of-time compiled version, since here the interpreter performance is best, and has a higher impact during warmup.

4.2.4 Memory Usage

Since our AST interpreters are AST-like, their memory representation leverages the host language to represent programs as trees of objects. Our BC-like interpreters instead encode programs into an array of bytes, which avoids the cost of object headers and machine-word-sized pointers. Consequently, bytecodes should be more compact, and we need to assess the memory overhead of an AST-based approach compared to a BC-based approach. However, our interpreters also collect profiling data in the form of lookup caches. TSOM's bytecode interpreter also uses the TruffleDSL-generated nodes for basic operations, which keep additional profiling information (as seen in section 4.1.6). Thus, in our bytecode interpreters designed for meta-compilation, the memory use of the program representation is not simply the number of bytes emitted by the bytecode compiler.

In addition to characterizing the difference between program representations, memory usage is also relevant in practice. When hosting applications in the cloud,

memory is a direct cost. Moreover, since the program representation is subject to garbage collection, it may also have a secondary run-time cost by increasing the time garbage collection takes.

With meta-compilation and garbage collection in the mix, measuring all impact of the differences of the program representation is not straightforward. For instance, tracing and partial evaluation will be affected, and thus may use different amounts of memory.

To have a single consistent metric for both systems, we will use the maximum Resident Set Size (RSS). RSS corresponds to the maximum heap size the language implementation requested. It thus is an overapproximation of all memory used by the interpreters. While measurements could potentially be skewed by temporary spikes in memory usage, we did not observe such abnormal behavior in our benchmark set, and so did not see fit to report RSS over time instead of the simpler maximum RSS observed. In programs that use large working memory, the impact of the program representation is naturally de-emphasized. However, for programs that load many lines of code, we will get a clearer picture of the cost of the program representation. Thus, while not precise, the maximum RSS reflects what users of a language implementation may care about.

Overall Impact. To get an impression of the high-level impact on memory usage, we measure the maximum RSS for the benchmark execution on the interpreter, as well as for peak performance. This will give us an intuition of the practical differences. However, especially for TSOM, the numbers are not only reflecting the difference in program representation, but also the difference between optimizations applied to the interpreters. As discussed in section 4.1.3, TSOM_{AST} is able to perform boxing elimination to a higher degree than TSOM_{BC}, which is reflected in the memory usage.

Impact of the Program Representation. To more clearly measure the memory used by program representation, we need to dominate the overall memory use with it. Unit tests are a common extreme case for many language implementations, since much of the code is executed exactly once.

To measure the impact of the program representation, we thus compare the memory use between different sizes of unit test suites. Specifically, we measure the memory use of the interpreter for running 1 to 100 copies of the SOM unit tests. For 100 copies, this corresponds to 2,528 files with 197,756 LOC in total, excluding comments and blank lines. With this setup, memory use is dominated by the program representation.

A second version of this experiment merely loads the code. Thus, memory is allocated to represent the program, but neither run-time objects nor profiling data are allocated, providing a lower bound for code that is not executed.

With this approach, we can minimize the impact of other allocations, approximating the memory used to represent the program as AST or bytecode.

4.2.5 Impact of Optimizations

We previously described the optimizations that we have implemented in section 4.1. Several of these optimizations leveraged profiling information e.g. by caching, which is beneficial in the interpreter, but can also be useful to generate more efficient JIT code.

By assessing the performance impact of a number of optimizations, we want to gain a better understanding of how performance relates to the program representation on meta-compilation systems and whether any possible additional memory use may translate into better run-time performance.

As a secondary goal for this work, since it is a relatively unexplored aspect of interpreters built with meta-compilation systems, we also want to gain a better understanding of which optimizations are most important for interpreter performance, peak performance, and warmup performance. This may help fellow language implementers to focus their efforts depending on whether they aim for a design closer to that of an AST or bytecode interpreter.

However, measuring the impact of individual optimizations is nontrivial. Optimizations can interact in unpredictable ways. Some optimizations may even be relevant at the beginning, but become later less important because they are redundant in some way. Assuming the goal is overall best performance, we take our fully

optimized interpreters and remove specific optimizations to identify their impact on the final performance of the systems. Thus, for each optimization under investigation, we have an interpreter that removes it completely to assess its impact.

4.2.6 Measurement Methodology and Reported Statistics

All experiments were run on a system with Ubuntu 20.04.6 LTS (kernel 5.4.0-146), with two 6-core Intel Xeon E5-2620 CPU at 2.40GHz, and with 16GB RAM. TSOM is based on the GraalVM development branch. PySOM uses RPython from the version 7.3.11 release of PyPy. For the baseline comparisons, since our benchmark suite is cross-language (see section 2.7), we compare TSOM and PySOM with well-established virtual machines for languages other than SOM, namely Node.js 17.9 and JDK 20.

We use ReBench (Marr 2023), in version 1.2, to configure the machine for benchmarking, and to collect the results. ReBench instructs Linux to use the performance governor, disables Turbo Boost, runs the benchmarks at the highest priority, and avoids scheduling other processes on the cores used for benchmarking (CPU shielding). This is intended to reduce the measurement noise.

To measure interpreter performance, just-in-time compilation is disabled, and we start the interpreter 100 times for a benchmark, and the benchmark harness measures how long one iteration of the benchmark takes.

We generally report the median run time factor over a baseline as well as the minimum and maximum to characterize the full range.

For measuring peak performance, the just-in-time compilers are enabled, and we start the interpreter once, and the benchmarking harness measures the time of 2000 iterations. Thus, all iterations execute in the same process, and benefit from the JIT compilation. We looked at the over-time performance of all benchmarks, and used the last 1000 measurements for the statistics. They are generally free from further compilation.

Since most experiments compare a set of benchmarks against a baseline, we report the median ratio over a baseline. The results are depicted as boxplots on a logarithmic scale, which leads to a more intuitive understanding of differences, since both a factor of 0.5 and 2 have the same physical distance from 1 on the resulting plot.

4.3 Results of Comparing our Meta-Compilation-based AST/BC Interpreters

We follow the high-level methodology outlined in section 4.2 to identify the trade-offs between the different program representations.

Our experiments can be reproduced through the artifact submitted alongside the original publication associated with this chapter.[4]

4.3.1 Interpreter Performance

As discussed in section 4.2.1, we start by assessing the run-time performance of our interpreters with JIT compilation disabled. Figure 4.1 shows the results. We use Java as the baseline, as it performs best overall. However, most interpreters have at least one benchmark that is faster than on Java. For instance on Node.js, the Json parser benchmark takes only 33% of the time it takes on Java.

Mean run times (shown with ⋄) are slightly ahead of the median runtimes, though they lead to the same order as sorting by median. When we focus on whether AST or bytecode interpreters on top of meta-compilation systems are faster, we see that the PySOM interpreters have very similar interpreter performance. The median slowdown of $\text{PySOM}_{\text{AST}}$ compared to Java is 3.16× (min. 0.91×, max. 8.94×), while the one for PySOM_{BC} is 3.51× (min. 0.94×, max. 8.12×), and thus only slightly slower. However, the difference on top of GraalVM is larger for our interpreters. TSOM_{AST} is 3.07× (min. 0.90×, max. 7.07×) slower than Java, while TSOM_{BC} is 4.96× (min. 1.18×, max. 8.59×)

Based on these results, we conclude that AST interpreters can be on par with bytecode interpreters in run-time performance, and may even have an edge in some benchmarks and on some meta-compilation systems.

4.3.2 Peak Performance

Since one of the main reasons to use a meta-compilation system is to benefit from the JIT compilation, we look at the peak performance results in fig. 4.2. Here we omit the warmup phase, which we will come back to in section 4.3.3.

[4] https://github.com/OctaveLarose/ast-vs-bc-experiments

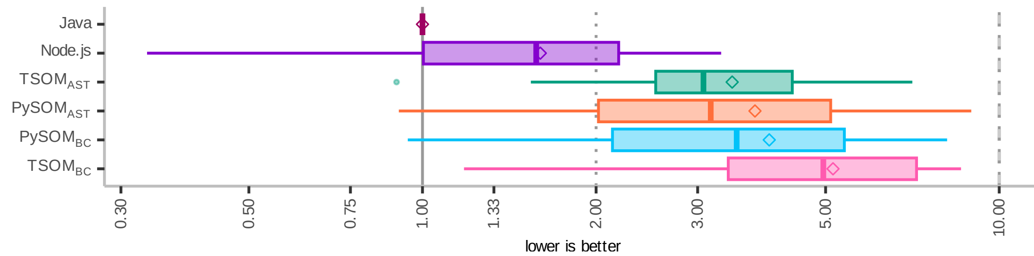


FIGURE 4.1: Interpreter run-time performance, on a logarithmic scale, with Java as baseline. While TSOM and PySOM are overall slower than the Java and Node.js interpreters, we can also observe that $\text{PySOM}_{\text{AST}}$ and TSOM_{AST} are faster than the bytecode versions. TSOM_{BC} is the slowest interpreter overall.

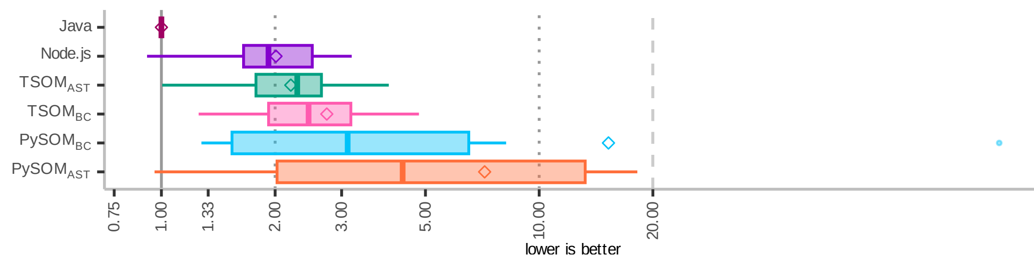


FIGURE 4.2: Peak performance with just-in-time compilation enabled, a logarithmic scale, with Java as baseline. TSOM_{AST} reaches the performance of Node.js, while the peak performance of the other implementations is a little further behind.

We use again Java as the baseline, since it has the best overall performance. The median run time of Node.js is about 1.92× (min. 0.92×, max. 3.19×) higher than Java. .

From our implementations, TSOM_{AST} is overall the fastest and takes about 2.29× (min. 1.00×, max. 4.00×) more time than Java, which is roughly at the same level as Node.js since the range over the benchmarks is well overlapping. TSOM_{BC} is not far behind with 2.45× (min. 1.25×, max. 4.81×).

For PySOM, PySOM_{BC} is the fastest implementation, taking about 3.12× (min. 1.28×, max. 165.13×) more time than Java. Though, the mean run time of PySOM_{BC} is much behind that of any other interpreter, due to the CD benchmark being much slower. This still overlaps well with Node.js, but has a few more benchmarks that do not quite reach the same level of performance. $\text{PySOM}_{\text{AST}}$ reaches 4.35× (min. 0.96×, max. 18.19×). A closer investigation shows that highly recursive benchmarks do not reach the same performance as on other systems. Here, RPython's trace-based compilation struggles to make the most optimal decision of when to stop tracing, which is a known issue. [5] While the bytecode version comes out faster, this result is

[5] https://web.archive.org/web/20241126035337/https://foss.heptapod.net/pypy/pypy/-/commit/928f810b36

dominated by a more suitable recursion inlining threshold.

Overall, there does not seem to be a clear benefit for either AST or bytecode interpreters in terms of peak performance. For TSOM, the results are arguably too close together, and for PySOM, the particularly slow recursive benchmarks make a strong conclusion seem undesirable. However, this in itself is a useful result. Even when focusing on peak performance, there does not seem to be a clear winner, which may allow us to make decisions in favor of one or the other program representation based on other factors (e.g. advantages described in Chapter 6).

4.3.3 Warmup Behavior During JIT Compilation

As discussed in section 4.2.3, we collected and analyzed the warmup data for TSOM on Hotspot and the ahead-of-time-compiled interpreter, and PySOM. For brevity we only include plots for the ahead-of-time-compiled systems. However, the results are similar for TSOM on top of Hotspot.

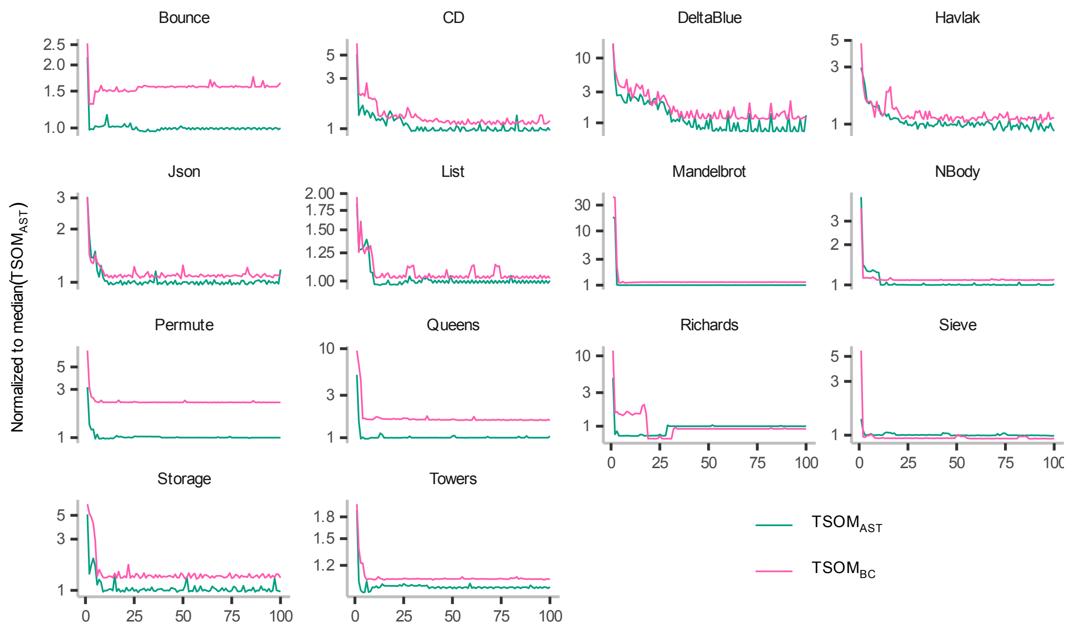


FIGURE 4.3: Performance of benchmark iterations over time for the TSOM interpreters as measured on the ahead-of-time-compiled GraalVM Native Image. All data is normalized to the median of $TSOM_{AST}$ and shown with a logarithmic y-axis. While we see performance difference between $TSOM_{AST}$ and $TSOM_{BC}$, only the Richards benchmark seems to display a noticeable warmup delay on $TSOM_{BC}$.

We illustrate the data in fig. 4.3 and fig. 4.4, showing the iteration times for each benchmark normalized to the median value of the baseline. For consistency, we chose

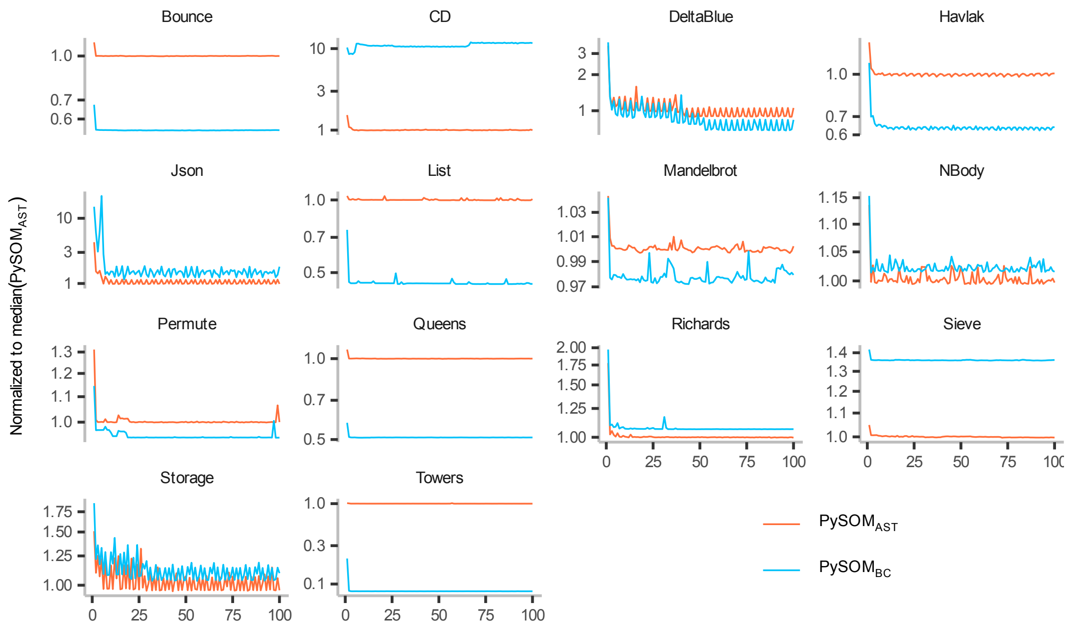
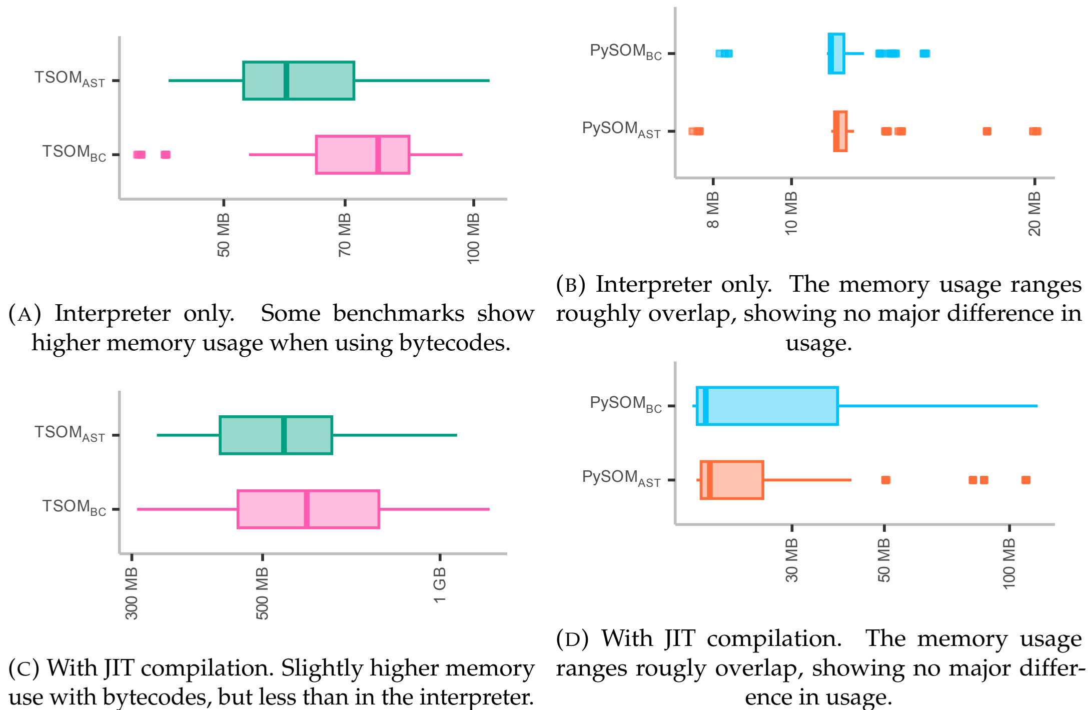


FIGURE 4.4: Performance of benchmark iterations over time for the PySOM interpreters. All data is normalized to the median of $\text{PySOM}_{\text{AST}}$ and shown with a logarithmic y-axis. While we see performance difference between $\text{PySOM}_{\text{AST}}$ and PySOM_{BC}, there does not seem to be a major difference in warmup.

the AST interpreter as baseline. This approach preserves the over-time variations, while having a consistent *goal* value of 1 (or better), that implementations would want to achieve.

The main question we try to answer here is whether there is a notable difference in warmup behavior between AST and bytecode interpreters. As can be expected from the peak performance results (see section 4.3.2), there are clear performance differences. However, we see neither for TSOM nor PySOM any systematic differences in warmup behavior. On TSOM_{BC} in fig. 4.3, we see Richards has a longer delay before reaching the peak performance, which may be due to Richards having a higher degree of polymorphism compared to most other benchmarks. However, none of the other benchmarks show it as pronounced. On PySOM in fig. 4.4, we may have more benchmarks that do not reach a similar performance, but this does not change after the shown 100 iterations either, and as such is not a warmup difference. A notable difference between $\text{PySOM}_{\text{AST}}$ and PySOM_{BC} is the CD benchmark being 10 times faster for $\text{PySOM}_{\text{AST}}$, which matches CD previously being an outlier in fig. 4.2. Overall, the PySOM interpreters achieve peak performance in a shorter period of time than the TSOM interpreters.



(A) Interpreter only. Some benchmarks show higher memory usage when using bytecodes.

(B) Interpreter only. The memory usage ranges roughly overlap, showing no major difference in usage.

(C) With JIT compilation. Slightly higher memory use with bytecodes, but less than in the interpreter.

(D) With JIT compilation. The memory usage ranges rougly overlap, showing no major difference in usage.

FIGURE 4.5: Maximum Resident Set Size (RSS) used by the interpreters, on a logarithmic scale.

Based on the available data, we cannot identify any systematic differences. Though, warmup issues are typically more pronounced on larger applications, and the Are We Fast Yet benchmarks are not large enough to trigger them.

4.3.4 Memory Usage

Besides run-time performance, memory use is the other major concern when it comes to program representation. As discussed in section 4.2.4, we assess the overall impact on memory use, as well as on the program representation itself.

Overall Memory Impact

Figure 4.5 shows the maximum Resident Set Size (RSS) used by the benchmarks. We show the data separately for TSOM and PySOM, each for the interpreter and with JIT compilation. The numbers of the interpreter are expected to show a stronger effect, if one is indeed there. The JIT compilers and their escape analysis may be able to reduce any additional allocation during optimization.

For TSOM, we can indeed see a difference in fig. 4.5a. The benchmarks on TSOM_{BC} seem to use more memory. Figure 4.5c shows a similar trend, but not as clearly.

For PySOM, if there is any effect, the AST interpreter might use slightly more memory. However, this is much less clear from the data.

When investigating the allocation profiles on TSOM, we found that the bytecode interpreters allocate a significantly larger number of `Long` and `Double` values, because the TruffleDSL's approach to boxing avoidance does not translate directly to bytecode interpreters. We also noticed an increase in the number of allocations of `Object[]` arrays. Since we need to reify the stack for our stack-based bytecode, each method activation allocates an extra `Object[]` array, which for some benchmarks leads to a significant increase in allocations. The last significant difference is the allocation of `Object[]` arrays in the bytecode loop to pass arguments to method calls. In the AST interpreter these allocations happen inside of `execute()` methods, which in some cases allows the compiler to already avoid the allocation even in the interpreter. However, in the bytecode interpreter, this allocation happens in the bytecode loop, before passing it on to a polymorphic call site. This prevents compilers from avoiding the allocation. Thus, on top of GraalVM, the AST interpreters may have some benefits in terms of memory usage, because boxing avoidance and code structure can lead to fewer allocations.

On PySOM, we do not see such effects because we do not have statement-level boxing avoidance and the SOM method calling convention does not require an array allocation for calls with fewer than four arguments, which are the most common ones.

While investigating the allocation behavior in detail, we also looked at precise memory allocation behavior. Though, the results were too similar to include and with garbage collection in the mix, maximum RSS seemed to be the more practically relevant metric.

Memory Usage for Program Representation

As detailed in section 4.2.4, we approximate the differences in memory used by the program representation by using 1 to 100 copies of the SOM unit tests. One version of the benchmark executes the unit tests, covering every line, and thus, allocates memory to store profiling information, polymorphic inline caches, etc. However,

it of course also allocates memory for the necessary run-time objects. Though, the run-time objects are garbage-collected almost immediately, since the unit tests do not hold onto them. Thus, the program representation dominates memory use more and more with an increasing number of copies of the unit tests. A second version of this benchmark merely loads the code. Thus, memory is allocated to represent the program, but neither run-time objects nor profiling data is allocated, providing a lower bound for code that is not executed.

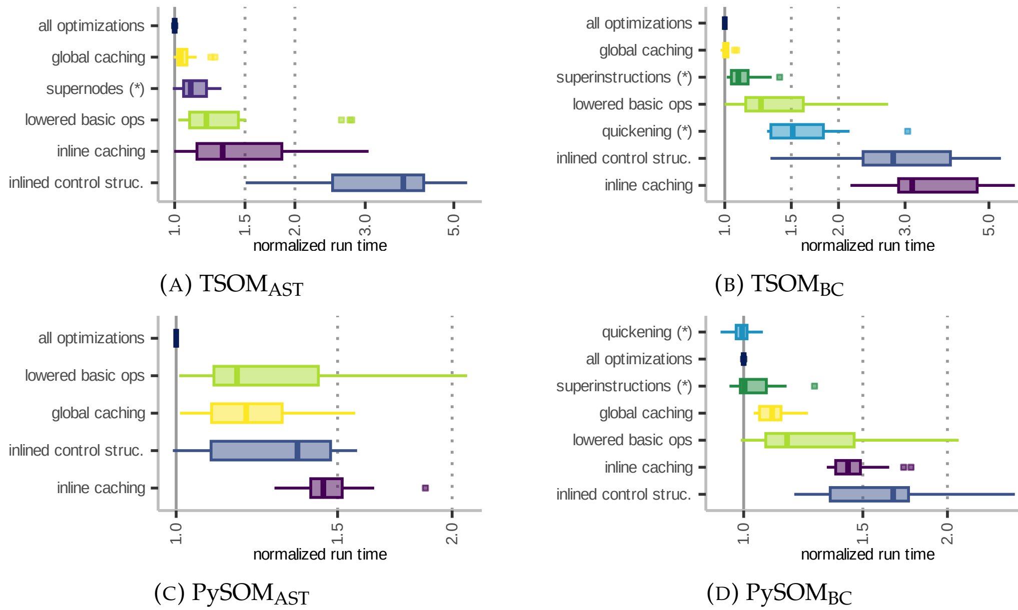
With this approach, the AST on PySOM requires about 2.55× more memory than the bytecode when it is not executed and about 2.30× when it is executed. For TSOM, we see 2.89× more memory use for the AST when we load the code but do not execute it, and only 1.21× more when we load and execute it. As mentioned in section 4.1.6, the bytecode interpreter reused the node objects of the AST interpreter to benefit from the profiling they provide. The size of these nodes is comparably large, and can explain the increase in memory use during execution in the TSOM_{BC} interpreter, which we do not see in PySOM_{BC}, because there we do no use this technique. Thus, with increased run-time profiling, the difference between bytecode and AST interpreters in terms of memory use further narrows.

4.3.5 Impact of Optimizations

This section assesses the impact of our optimizations from section 4.1.3 to better understand how they interact with the choice of program representation. This analysis is intended to help language implementers to prioritize optimizations.

Impact on Interpreter Performance

Figure 4.6 shows the impact of the various optimizations on interpreter performance, without JIT compilation. It shows how the run time increases when specific optimizations are removed. Inlining of control structures (see section 4.1.3) and inline caching (see section 4.1.3) have the highest impact overall. On TSOM_{AST}, removing the inlining of control structures increases the median run time by 3.74× (min. 1.51×, max. 5.40×). On PySOM_{BC}, its impact is lower but still the most important one at 1.66× (min. 1.19×, max. 2.52×).



(A) TSOM_{AST} (B) TSOM_{BC}

(C) $\text{PySOM}_{\text{AST}}$ (D) PySOM_{BC}

FIGURE 4.6: Performance impact of removing individual optimizations on the interpreter-only speed of the fully optimized interpreters. Run time normalized to the implementation with all optimizations, on a logarithmic scale. Optimizations marked with (*) are specific to AST or bytecode interpreters. Overall, inlining of control structures, inline caching, and lowering of basic operations give the highest benefit.

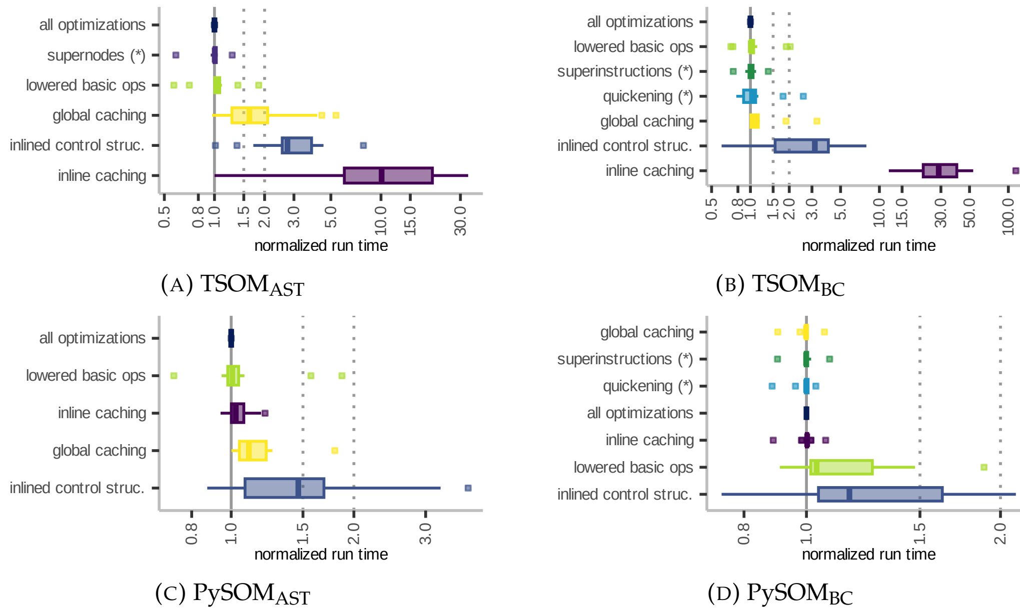
On TSOM_{BC} and $\text{PySOM}_{\text{AST}}$, inline caching is more important. On TSOM_{BC} it gives 3.13× (min. 2.15×, max. 5.86×) and on $\text{PySOM}_{\text{AST}}$ 1.45× (min. 1.28×, max. 1.87×).

As can be seen with the difference between PySOM and TSOM, inlining is more beneficial when the cost of method activation is high, which is the case for TSOM.

The caching of globals is not of high importance in the TSOM interpreters, but will be more important when we look at just-in-time-compiled performance.

Supernodes and superinstructions give roughly similar benefits on TSOM_{AST} and TSOM_{BC} respectively, which is to be expected, since they reduce the dispatch overhead between operations and open up a similar optimization potential. However, they are highly targeted and thus only give a modest gain.

When optimizing interpreter performance, we would recommend to start with inline caching, which is the most widely applicable optimization and highly important for large-scale applications. Afterwards, lowering of basic operations will give good benefits and can target specific benchmarks of interests. Finally, inlining of control structures will also be important for interpreter performance. However, it can come with significant implementation complexity.



(A) TSOM_{AST} (B) TSOM_{BC}

(C) $\text{PySOM}_{\text{AST}}$ (D) PySOM_{BC}

FIGURE 4.7: Performance impact of individual optimizations on the peak performance of the fully optimized interpreters. Run time normalized to the implementation with all optimizations, on a logarithmic scale. Optimizations marked with (*) are specific to AST or bytecode interpreters. Inline caching gives the highest benefit on TSOM, because there it is needed for inlining across SOM methods. PySOM benefits more from inlining of control structures.

Impact on Peak Performance

Before looking into the impact of specific optimizations on peak performance, we can observe in fig. 4.7 that some optimizations are less important for it than others. Furthermore, one difference to the impact on interpreter performance is that caching of globals makes a good difference on TSOM for specific benchmarks, because with it the compiler can assume the value to be a constant, which enables it to optimize more. Also on TSOM, the lowering of basic operations, supernodes, superinstructions, and even quickening have a less noticeable impact, because the compiler can perform similar optimizations.

Similar to the interpreter performance, inline caching is important on TSOM. Removing it from TSOM_{AST} results in 10.09× (min. 1.00×, max. 33.39×) increase in run time. On TSOM_{BC}, it results in 28.96× (min. 11.83×, max. 114.38×), making it the most important optimization. This is because inline caching enables the GraalVM's partial evaluator to inline across SOM methods, which is important to enable subsequent optimizations. This is in stark contrast to PySOM, where we see a difference of 1.03× (min. 0.94×, max. 1.21×) on $\text{PySOM}_{\text{AST}}$, and 1.00× (min. 0.89×, max. 1.07×), on

PySOM$_{BC}$. Since RPython uses meta-tracing, inline caching is not needed to enable inlining.

On PySOM, the most important optimization is the inlining of control structures. On PySOM$_{AST}$ it results in 1.46× (min. 0.87×, max. 3.81×) on PySOM$_{AST}$, and on PySOM$_{BC}$ it gives 1.17× (min. 0.74×, max. 2.11×). The inlining helps to reduce the overall trace length and makes optimization opportunities usually more evident to the compiler. However, on top of PySOM$_{BC}$ we do see also some benchmarks slowing down when applying the optimization. In the cases we observed, the inlining in the bytecode leads to the compiler not having as precise information on the lifetime of variables, which can lead to extra boxing. Here, we notice that the AST structure and the use of the host-language stack communicates such details more precisely. The reified stack in the bytecode interpreter on the other hand can lead to such details being obscured. This effect is also noticeable on TSOM$_{BC}$ where removing inlining of control structures results in 3.16× (min. 0.60×, max. 7.93×). increase of run time. While generally beneficial, there are still some slowdowns observable.

These results show that optimizations have a larger performance impact on our Graal-based interpreters, as seen from it ranging from 0.5x to 30x/100x on TSOM compared to only going up to 2x/3x on our PySOM interpreters. This may attest to Graal-based virtual machines requiring more engineering effort, i.e. optimizations, to achieve good run-time performance, compared to RPython-based virtual machines (Marr et al. 2015).

Overall, we conclude from this analysis that there is a stronger difference between meta-tracing and partial evaluation than the choice of program representation. However, it remains to be noted that in some cases the explicit reified stack in a bytecode interpreter can obscure variable lifetimes, which makes it harder for a meta-compiler to yield optimal code.

4.4 Result Discussion

In the following discussion, we will reflect on the results, their generalizability, the limitations of this study, and our perception of engineering differences.

4.4.1 Results and their Generalizability

Results Our results show that in the context of meta-compilation systems, there is no overwhelming evidence to only consider bytecode-based interpreters, as one would expect based on folklore. Instead, we have found that naturally-recursive AST interpreters can be competitive with BC interpreters, validating our original hypothesis ("Can AST interpreters be as performant as BC interpreters?") for meta-compilation-based systems.

Specifically, in section 4.3.1 we have seen that the pure interpreter performance of AST interpreters can be better than that of bytecode-based interpreters. The peak performance, seen in section 4.3.2, is also generally on par. Currently, the memory use is perhaps the most significant difference. There are some cases where the interpreter structure of bytecode interpreters has drawbacks. Here, techniques such as multi-level quickening (Brunthaler 2021) may provide solutions. On the other hand, it is still true that bytecodes are significantly more compact (see section 4.3.4) than ASTs. Especially for established languages with large codebases, this can be a dominating factor. For new languages, perhaps in the context of research, small applications, or scripting, AST interpreters do not only seem to be *good enough*, but may have benefits over bytecode interpreters.

By challenging the assumption that bytecode interpreters always outperform AST interpreters, we also implicitly question whether bytecode interpreters do always outperform other hybrid designs, e.g. serialized tree-based interpreters (section 3.3.3), or if we might observe similar results as in this chapter. This is outside of the scope of this work, though, as this would require the implementation and maintenance of many more interpreters, thus be an untenable amount of engineering effort.

Generalizability to Other Languages and Meta-Compilation Systems We previously argued in section 2.7 that we believe our results to generalize to other dynamic languages beyond SOM, since it shares many implementation challenges with widely used dynamic languages, e.g. Python, Ruby, or JavaScript. Therefore, we believe that our results in this chapter are transferable to other language implementations on the studied systems.

Expanding the scope beyond the guest language, we can then ask whether our results would generalize to other meta-compilation systems. We believe this to be possible, as we purposefully explored our research questions on top of two very different meta-compilation systems. On the one hand, there is the difference in compilation approach. While RPython uses trace-based compilation, GraalVM uses method-based compilation, both having very different trade-offs in terms of performance and engineering characteristics (Marr et al. 2015). On the other hand, they differ drastically in the degree of support they provide and the sophistication of the runtime system. While RPython uses a minimalistic approach with a few annotations, GraalVM uses the Truffle framework and TruffleDSL to enforce a much more strict way how languages are to be implemented. Similarly, the RPython system has a relatively straightforward incremental garbage collector, while GraalVM can use the HotSpot JVM and many of its garbage collectors. Since RPython compiles to C, our RPython interpreters are closer to system-language interpreters albeit with the restrictions of RPython. Thus, both systems arguably differ significantly from each other, which means our results may generalize to other meta-compilation systems, or at least, to meta-compilation systems that are similar to GraalVM or RPython.

The results suggest that there may be further room to improve meta-compilation systems for AST as well as bytecode interpreters. AST interpreters could benefit from techniques for more compact representations that more closely resemble bytecodes (Koparkar et al. 2021; Pape 2021; Pape et al. 2017; Vollmer et al. 2017), thus leaning more towards BC-like designs for the sake of performance (as we previously discussed in section 3.4.4). Of course, taking the other perspective, support for bytecode interpreters could be further improved for instance with support for boxing avoidance as provided by the TruffleDSL (Humer et al. 2014a), multi-level quickening (see section 2.3.4) or threaded code (see section 2.3.5). In short, meta-compilation systems could be engineered to have fewer limitations in the design of the interpreter, such that they could get closer to the degree of control, and greater performance, available to system-level interpreters.

Generalizability to System-level Interpreters From the results in this chapter alone, it is unclear whether our findings may generalize to interpreters implemented in

system-level languages. Generally, we assume these languages to have more control over data layout and control flow, which results in a different performance trade-off space, and may therefore yield entirely different results.

So, we make system-level interpreters the focus of the next chapter. While this chapter focused on the domain of meta-compilation, we will next broaden the design context by focusing on interpreters written in a system-level language, namely Rust. Based on the findings in this chapter, we will see if Rust-based AST interpreters bring as many engineering benefits in terms of ease of implementation, if they can also rival BC interpreters in terms of run-time performance, if AST also requires more memory, but also whether that extra memory usage is somewhat balanced out by the size of run-time information. We will also see whether the semantics of Rust influences the design of Rust-based interpreters by encouraging more AST-like, or BC-like design decisions.

This domain of custom-built systems is different from that of meta-compilation systems, which do not reach such high levels of performance. While relying on a system-level language is beneficial performance-wise, it can induce a very large amount of engineering effort, which can be practical for industry projects such as V8 and JavaScriptCore VMs and even research projects such as the Wizard WebAssembly interpreter (Titzer 2022), but is less suited to many other projects. For meta-compilation systems, a key focus and benefit is rather the overall reduction of engineering effort, compared to manually implementing one or several JIT compiler tiers, GC implementation, custom tooling, and more. Another point of note is that meta-compilation systems have restrictions to the type of optimizations that are supported, as well as the code patterns the compilers are optimized for. For instance, bytecode interpreters currently cannot immediately benefit from optimizations such as threaded code, and while work is underway (Izawa et al. 2025, 2022) to enable language implementers to benefit from more such optimizations, it seems unlikely that one will have the same freedom as for completely custom-built language implementations.

4.4.2 Limitations of this Study

As discussed in section 4.1.6, there are differences between the interpreters and their optimizations. These differences may have an impact on the accuracy and precision of our comparisons. While we believe that the most significant differences are related to the different underlying systems, i.e., GraalVM and RPython, there are also differences in the degree of optimization applied to AST and bytecode interpreters. Overall, we believe that we reached a similar level of optimization on all systems, enabling us to draw conclusions from this study. However for completeness, we will reiterate the main differences as limitations.

Boxing Avoidance One of the major differences between TSOM_{AST} and TSOM_{BC} is the degree of boxing avoidance realized in the interpreter. The AST interpreter is able to do boxing avoidance for full statements, relying on the tree structure, which specializes itself to code that avoids boxing of `long` and `double` values. In the bytecode interpreter, this is not currently possible in the same way. The linear structure of bytecode does not provide the context for this optimization and the stack being reified as an `Object[]` array leads to boxing between bytecodes. Using more quickening (Brunthaler 2010a, 2021) and an additional stack for unboxed values may help to improve this and reduce some of the memory overhead seen in section 4.3.4. However, we also saw the cost of the reified stack itself in the memory cost. Thus, boxing is not the only factor here, as the overall structure of bytecode interpreters also had an impact. We believe this difference may only lead us to slightly overestimating the difference between TSOM_{AST} and TSOM_{BC} on interpreter-only performance.

In the context of the overall study, we also believe this difference does not have a major impact since neither $\text{PySOM}_{\text{AST}}$ nor PySOM_{BC} do boxing avoidance, and thus, we have a consistent comparison along this axis.

Degree of Run-time Profiling The TSOM interpreters both have a high degree of run-time profiling. This includes profiling the activated specialization in nodes as well as active branch and value profiling. Both TSOM_{AST} and TSOM_{BC} do almost the same amount of profiling. There are only minor differences caused by nodes that are only used in TSOM_{AST} since they are not applicable to the bytecode interpreter,

because the operations are expressed as bytecodes. Given that the peak performance is fairly similar (see section 4.3.2), we believe there is no fundamental issue.

PySOM does not do the same style of run-time profiling. While it does self-optimization and quickening in the interpreter, meta-tracing uses concrete values from the trace, which means we do not need to collect profiling information in the interpreter.

Bytecode-level Optimizations and Supernodes Bytecodes and superinstructions were identified in a step-wise process based on profiles of bytecode sequences and run-time profiles of benchmarks. As such, the ones implemented are the ones that gave the highest likelihood for performance improvements. We used the same approach on AST nodes, identifying common patterns to construct supernodes.

We could have further expanded the bytecode set and increased the number of implemented supernodes. Similarly, we could continue to identify further opportunities for bytecode quickening and also add supernodes to $\text{PySOM}_{\text{AST}}$. Given that $\text{PySOM}_{\text{AST}}$ already outperforms PySOM_{BC} in terms of interpreter performance, any impact on the overall conclusions is likely not to be significant. We believe that overall, further performance gains on the used benchmarks would be minimal, because the overall benefit of these optimizations is limited as we saw from the results in fig. 4.6.

Our experience with larger codebases also suggests that any performance improvement potential is thwarted by the cost of method calls, which seem to dominate larger applications in the interpreter.

4.5 Related Work

We are not the first to compare AST and bytecode interpreters. Most closely related to our work on top of meta-compilation systems is work by Niephaus et al. (2018) and Kalibera et al. (2014). Niephaus et al. (2018) implement GraalSqueak and compare an AST with a bytecode-inspired interpreter on top of GraalVM. GraalSqueak is also a Smalltalk, which they evaluate with a prime number sieve and Fibonacci benchmark. However, what they call bytecodes does not leverage the benefits of a compact representation, because each of their “bytecodes” is represented by a node object

which requires at least a Java header, typically 16 bytes, and a parent pointer of 4 or 8 bytes. Our work here uses a bytecode encoding where each instruction is represented with one or two bytes (see section 4.1.2). Furthermore, we compare the interpreters not just on top of GraalVM, but also RPython, and use the more comprehensive Are We Fast Yet benchmarks. Many of the mentioned GraalVM-specific optimizations are common to languages using the Truffle framework, and we apply them, too.

Kalibera et al. (2014) implemented FastR, an AST interpreter for a subset of the R language, and noted that they achieved better performance than the GNU R bytecode interpreter. While it is another data point suggesting the potential of AST interpreters, it is a comparison between worlds, i.e., across different systems and implementation languages with many confounding factors. Our comparison removes these factors and can examine the differences between program representations, since we have AST and bytecode interpreters in the same system. That being said, we apply many of the techniques they recommend in Kalibera et al. (2014) section 4. For instance, we apply optimizations as soon as possible on AST and bytecode to gain performance. We utilize aggressive specializations, cache lookups, and use storage strategies to specialize data structures to reach state-of-the-art performance.

Körner et al. (2021) compared the performance of AST and bytecode interpreters in the context of Prolog, and found AST interpreters to be more efficient in many cases, also noting their ease of implementation compared to bytecode. One aspect here is that Prolog is more amenable to being represented as an AST due to its recursive execution model, and attests to the strength of the more natural representation of a tree when compared to the linear nature of bytecode. These types of thoughts are not new. D'Hondt (2008) asked whether we should consider bytecodes to be an atavism, however, without a clear answer with respect to performance.

In the context of meta-compilation systems, the difference between meta-tracing and partial evaluation has been studied before (Marr et al. 2015). While we also use GraalVM and RPython as the only practical meta-compilation systems, this work focused on the question of program representation.

4.6 Conclusion

Meta-compilation systems enable us to implement languages based on interpreters, while components such as just-in-time compilation and garbage collection are provided by the system. This makes interpreters a key part of these systems, and makes meta-compilation a key space in which to analyze interpreter design. Language implementers benefit not just from reduced engineering effort, but also all the know-how around interpreters, and part of this folklore is that bytecode interpreters are faster and more memory efficient than AST interpreters.

In this chapter, we showed that bytecode interpreters are not necessarily more efficient than AST interpreters on top of meta-compilation systems. Instead, AST interpreters are on par, and in some cases even faster than bytecode interpreters, *with and without* meta-compilation. We showed this based on PySOM and TSOM, for which we built both an AST and bytecode interpreter on top of both RPython and GraalVM. We evaluated performance and memory use based on the Are We Fast Yet benchmarks (Marr et al. 2016). Therefore in this context, AST-like interpreters benefiting from their proximity with the host language may be able to do so without a performance cost.

While we found that both AST and bytecode interpreters can be on par performance-wise, we could also confirm that bytecodes are more memory-efficient than AST nodes, as expected from BC-like interpreters relying on a compact representation, instead of AST-like interpreters using sparse host language objects. However, as soon as run-time profiling information comes into the mix, the difference becomes smaller. Furthermore, with the structure of bytecode interpreters, they may need more run-time memory allocations, which can have a negative impact on overall memory use. We also investigated the performance impact of common optimizations, which can help language implementers to prioritize their implementation.

We believe our results show that AST interpreters should not be so readily discounted when it comes to implementing efficient and fast language implementations. While bytecodes can be more compact, AST interpreters have engineering benefits such as direct utilization of the host-language stack, more localized and encapsulated state, and their tree structure facilitates early optimizations that can make them a

suitable choice for some implementations.

We have seen that meta-compilation frameworks can put constraints on some design decisions, or encourage others, in our case pushing BC interpreters a bit towards being AST-like. So, meta-compilation systems could also benefit from further improvements to provide more flexibility. Using techniques proposed by Koparkar et al. (2021), Pape et al. (2017), Pape (2021), and Vollmer et al. (2017) may enable more compact representations of ASTs that can still be interpreted and benefit from self-optimization. Truffle's Bytecode DSL (*Bytecode DSL* 2025; Humer et al. 2022), which generates BC interpreters on top of Truffle, is also very promising. It aims to make the implementation of bytecode interpreters on top of GraalVM as convenient as AST interpreters, could solve the issues we observed around boxing avoidance and may enable the integration of multi-level quickening (Brunthaler 2021).

While we were not able to measure warmup differences with the benchmarks we used, we know that bytecode interpreters can suffer from longer compilation times on top of GraalVM. The partial evaluation system will first unroll the bytecode loop and duplicate it as many times as a method has bytecodes, and inline all called methods aggressively. This results in huge compiler graphs and is a known limitation with the current system. Future work should investigate how to effectively interleave classic optimizations with partial evaluation to minimize the compiler graph and the corresponding compilation time.

GraalVM is currently also limited in the optimizations it can apply to bytecode interpreters since the GraalIR only supports structured control flow. Thus, creative solutions to e.g. enable effective threaded interpretation would be needed.

Our work so far was strictly focused on interpreters in the context of meta-compilation, as opposed to interpreters written in system-level languages, e.g. C, C++ or Rust. Therefore, our results may or may not generalize outside of meta-compilation. The following chapter thus aims to answer this question by focusing on comparing two optimized interpreters, one AST-based and one BC-based, both written using the Rust language. We focus on their performance, memory usage, but also how the implementation effort compares to our interpreters on top of meta-compilation systems.

Chapter 5

Interpreters in Rust

5.1 Introduction

Many interpreters are written in system-level languages, e.g. C, C++ or Rust, to achieve the best possible performance. These languages provide control over memory layout, and offer high performance, at the cost of more engineering effort.

The previous chapter focused on AST interpreters and BC interpreters in the context of meta-compilation systems, written in Java and RPython, and being based on a meta-compilation framework that imposed certain restrictions and requirements in the overall design of our interpreters. This chapter focuses instead on interpreters written in Rust (see section 2.6), a language which offers high performance and some safety guarantees. Within the context of meta-compilation, we found AST interpreters could rival BC interpreters in terms of run-time performance and memory usage. In this chapter, we perform similar experiments within our optimized, Rust-based interpreters, to see if these results also extend outside of the realm of meta-compilation, for interpreters written in system-level languages.

We describe our two Rust-based interpreters for the SOM language (see section 2.2), SOMrs-AST and SOMrs-BC[1]. Unlike TSOM and PySOM in the previous chapter, the SOMrs interpreters are *pure*, i.e., not based on meta-compilation systems (see section 2.5). Though like our meta-compilation-based interpreters, both SOMrs-AST and SOMrs-BC are designed based on the design dimensions we associated with AST-like and BC-like designs in section 3.2. We focus on garbage collection and how we implemented it in both runtime systems, as we previously did not need to

[1] https://github.com/OctaveLarose/som-rs

implement it for our intrpreters based on meta-compilation, and as it had an impact on interpreter design decisions, especially in our AST interpreter. We then assess the run-time performance of our interpreters, and their memory usage, both in general and isolating the memory needed solely for the program representation.

The contributions of this chapter are:

1. the implementation of SOMrs-AST and SOMrs-BC, both optimized interpreters written in the Rust language, and a discussion of their design.
2. an analysis showing that optimized BC interpreters outperform AST interpreters in terms of run-time performance when both are written in system-level languages, though that the gap is not as large as one may assume.
3. a discussion on the design of AST interpreters on top of Rust, and the associated challenges with certain AST-like traits, especially when paired with efficient garbage collection.

5.2 Overview of the SOMrs Implementations

Both interpreters are implemented in Rust, execute the SOM language, and were designed with similar priorities as we have seen in section 4.1, though not strictly identical:

- being strongly AST-like and BC-like, to enable comparisons between interpreters on both ends of our design spectrum.
- run-time performance.
- similar overall designs with our meta-compilation-based interpreters, to enable potential comparisons with them.

As with the previous chapter, our interpreters are designed to be as strongly AST-like and BC-like as possible. We justify labeling our interpreters as such in the next section, where we associate each interpreter with AST/BC design dimensions defined in section 3.5.

Run-time performance is a key focus in our interpreters also, as it is a core part of the appeal of using a system-level language such as Rust. It is therefore of high

interest to language implementations written in this language, and written in system-level languages as a whole. Therefore, our interpreters strive to be fast, though they have received less overall engineering effort than our interpreters described in the previous chapter.

Additionally, we kept the structure of the SOMrs interpreters similar to that of TSOM/PySOM, with the goal of enabling comparisons between both. Though, comparisons are limited in terms of run-time performance, since our meta-compilation-based interpreters have received more optimization effort overall, while SOMrs had to dedicate some of its engineering effort e.g. to implement garbage collection. We can, however, roughly compare interpreters in terms of memory usage (see section 5.5.2).

5.2.1 High-level Overview

We implemented SOMrs using the Rust language, which we chose as it promises high performance, as well as give us full control over the code representation and the design of the interpreter. The language also provides compile-time memory safety guarantees (see section 2.6.1), which help catch bugs ahead of time by imposing constraints, which we found to help with the implementation and maintainability of our interpreters. Though, while these guarantees were especially useful to develop the initial versions of SOMrs, we gradually did away with these guarantees in some areas for the sake of performance using the `unsafe` keyword, most notably to integrate efficient garbage collection, which we later see in section 5.3.

SOMrs-AST is designed to be strongly *AST-like* and SOMrs-BC is designed to be strongly *BC-like*. Thus, we stay close to the common AST and BC characteristics discussed in section 3.5, which we list for both of our interpreters.

Both are part of the same codebase, share some core features, such as garbage collection, and both have received a similar amount of engineering effort. Our interpreters implement the same optimizations, or implement optimizations that have a clear counterpart in the other interpreter. Since all optimizations are shared, we can meaningfully compare the performance of both interpreters, and assume that the most important difference between both is their choice of program representation, being AST or BC. We expand on all implemented optimizations in section 5.2.5.

5.2.2 AST

```
1 pub enum AstExpression {
2     GlobalRead(Box<GlobalNode>),
3     LocalVarRead(u8),
4     NonLocalVarRead(u8, u8),
5     ArgRead(u8, u8),
6     FieldRead(u8),
7     LocalVarWrite(u8, Box<AstExpression>),
8     NonLocalVarWrite(u8, u8, Box<AstExpression>),
9     LocalExit(Box<AstExpression>),
10    InlinedCall(Box<InlinedNode>),
11    UnaryDispatch(Box<AstUnaryDispatch>),
12    BinaryDispatch(Box<AstBinaryDispatch>),
13    ...
14 }
```

LISTING 5.1: Part of the AST representation used in SOMrs-AST, stored in a Rust `enum` type.

Non-Compact Representation Listing 5.1 shows part of its AST representation. The AST used by SOMrs is similar to that of TSOM and PySOM, as previously seen in section 4.1.1. We describe the AST nodes in full in appendix C.1.

Our AST nodes are defined in an `AstExpression` enum. In Rust, `enum` instances are laid out in memory such that for any enumeration item, no matter which variant it is, its size will be that of the largest variant, plus additional storage for the tag of the item[2]. This means that AST nodes corresponding to expressions, in SOMrs-AST, do not have variable size. In our case, every expression needs 24 bytes in total, as the largest variant needs 16 bytes and Rust needs an additional byte to keep the tag of the enum, with this tag needing to be aligned on an 8-byte boundary for alignment purposes (on a 64-bit machine). This is in contrast to TSOM and PySOM, which relied on Java and Python objects respectively, which have object headers, alignment requirements, as well as size that depends on their object fields, thus requiring more memory overall to store their representation.

SOMrs-AST relies on a non-compact representation. As we can see in listing 5.1, AST nodes are stored and accessed with `Box<T>`, i.e. pointers into the Rust heap. Although SOMrs makes frequent use of pointers directly into our garbage collection heap `Gc<T>` (as we will see in section 5.3.2), we instead use `Box<T>` pointers because

[2]The Rust compiler can also occasionally require no additional storage for the tag, if it can store it in unused bits in the data type itself.

we store the AST on the Rust heap. We discuss this decision not to store the AST on our GC heap in section 5.3.3. Garbage collection was, generally speaking, more difficult to implement within SOMrs-AST than SOMrs-BC.

The AST representation is more compact than that of TSOM/PySOM in that relying on a Rust `enum` means every AST node has the same size, therefore that consecutive AST nodes can easily be stored next to one another in memory. Therefore, each method has a `Vec<AstExpression>` which stores expressions consecutively in memory, but expressions refer to subexpressions using the `Box<T>` type, meaning those subexpressions could be anywhere in memory. So, our representation is relatively non-compact overall.

Recursive Design SOMrs-AST has a naturally-recursive design, as described in section 2.1.1, matching the design of our meta-compilation-based AST interpreters. AST nodes implement the `Evaluate` trait and its associated `evaluate` method in which the behavior of the node is specified, mirroring "`execute`" in TSOM and PySOM. In our meta-compilation-based AST interpreters, calls to this "`evaluate`" method dynamically dispatch to the correct implementation depending on the caller. However, SOMrs-AST does not rely on dynamic dispatch, while it is supported by Rust (using the `dyn` keyword), and instead dispatching an `AstExpression` is done with a `match` statement with branches for each possible expression variant. This is a trait that we previously categorized as BC-like (see section 3.5), which we used in this case as we believe it to be an idiomatic way to implement different AST nodes in Rust.

Exceptions For Non-Local Control Flow While Rust does not feature an exception type, SOMrs relies on its own exception-like type to handle non-local control flow. Each operation returns an enum `Return`, which can be a normal return, or a non-local return, both wrapping values. A non-local return also wraps a target scope, and so caller methods have their own "exception handlers" where they check that the return value of the previous statement was or was not a non-local return, and propagate the non-local return to the correct target scope if needed.

5.2.3 BC

```
1 pub enum Bytecode {
2     Pop,
3     PushLocal(u8),
4     PushNonLocal(u8, u8),
5     PushArg(u8),
6     PushNonLocalArg(u8, u8),
7     PushField(u8),
8     PushBlock(u8),
9     PushGlobal(u8),
10    WriteLocal(u8, u8),
11    ReturnLocal,
12    Jump(u16),
13    JumpBackward(u16),
14    JumpOnTrue(u16),
15    Send1(Interned),
16    Send2(Interned),
17    Send3(Interned),
18    SendN(Interned),
19    ...
20 }
```

LISTING 5.2: Part of the SOMrs-BC bytecode set.

Bytecode Set SOMrs-BC is a stack-based bytecode interpreter, with a bytecode set intentionally designed to roughly match that of TSOM and PySOM, as we described in section 4.1.2 and show in full in appendix B.2. We show the full bytecode set for SOMrs-BC itself in appendix C.2.

Listing 5.2 shows part of the bytecode set. As with AST nodes, we represent bytecode as a Rust enum, meaning that the size of a single bytecode is fixed, being equal to the size of the largest variant plus storage for the tag. Therefore, while a `PushLocal(u8)` variant could be stored using only two bytes of storage (one for the tag and one for its argument), Rust instead uses 4 bytes to represent it, as it does for each bytecode. This means our bytecode set is not as compact as it could possibly be, since several bytecode variants are given extra bytes that they make no use of. We come back to this trade-off between engineering and performance in section 5.6.2.

Purely Iterative Design Bytecode is dispatched with a main bytecode loop. We do not rely on a recursive design, as first described in section 3.3.1, but instead a non-reentrant bytecode interpreter design, with the bytecode loop function invoked once and used for all methods throughout the execution. This is in contrast with

TSOM and PySOM which relied on a guest-method recursive bytecode loop function, which we argued in section 4.1.2 was an AST-like design in some aspects, as meta-compilation encourages designs that leverage host language mechanisms. SOMrs-BC, instead, relies on what we find to be a more BC-like design.

No Exceptions For Non-Local Control Flow Our other bytecode interpreters rely on the host language stack for their execution, but SOMrs-BC has a purely iterative design. It thus also has its own stack, and therefore, any non-local control flow can be handled by modifying this stack, popping frames until the correct scope is reached.

5.2.4 Differences with Interpreters Based on Meta-Compilation

As SOMrs relies on the Rust language and is completely outside the context of meta-compilation, its design has several differences with TSOM and PySOM. This section already established some differences between these interpreters and SOMrs in terms of the runtime representation, such as how AST/BC is laid out in memory and how both interpreters handle dispatching to the next instruction. Here, we describe differences stemming from being outside of the context of meta-compilation.

Less Profiling Information An important difference between SOMrs and our meta-compilation-based interpreters is that since our Rust-based interpreters feature no JIT compilation, they do not need to store profiling information for the sake of efficient dynamic code generation. They thus do not need to e.g. increment invocation counters to report hot methods, and instead only store run-time information that benefits the performance of the interpreter itself, such as inline caches. This is likely to save memory through needing less storage for profiling information, and to improve run-time performance by avoiding spending cycles to store and update this profiling information.

Calling Convention As we described in section 4.1.6, TSOM also has to allocate object arrays for method calls due to the calling convention enforced by GraalVM. PySOM did not enforce any calling convention, and SOMrs similarly has no such restrictions. This freedom is why SOMrs mirrors the decision originally taken by

PySOM to specialize method dispatches based on one, two, or three arguments, as seen in listing 5.1 with the `[Unary|Binary|...]Dispatch` nodes, and in listing 5.2 with the `Send[1-2-...]` bytecode variants, with otherwise an N-ary dispatch node/bytecode which incurs an additional allocation.

Object Boxing Both TSOM and PySOM rely on object boxing to some extent. While TSOM leveraged the TruffleDSL to implement specializations that avoided a large amount of boxing, all values are boxed in PySOM. SOMrs instead relies on NaN boxing/NaN tagging (which we later see in section 5.2.5) to represent values, which was not an option for our meta-compilation-based VMs since neither Graal nor RPython support tagging schemes. This means that the value is not boxed on the heap, but encoded within a 64-bit floating point number. The type of the value has to be checked prior to being used, and that its type has to be extracted from the floating point value, but these operations are cheap bitwise calculations. SOMrs relying on NaN boxing is also why it does not rely on storage strategies and an object model like that of TSOM/PySOM (see section 4.1.4), since boxing avoidance is far less beneficial to SOMrs.

5.2.5 Implemented Optimizations

We focused on implementing optimizations that can be applied to both AST and BC interpreters, so that we could meaningfully compare the performance of both interpreters. Thus, the following optimizations are present in both SOMrs-AST and SOMrs-BC, and are therefore orthogonal to the program representation.

Apart from *NaN boxing*, these optimizations were previously described in section 2.3, and the majority match optimizations present in meta-compilation systems, described in section 4.1.3 and section 4.1.5. We define those shared optimizations again here briefly since SOMrs implements several of them differently than TSOM/PySOM, e.g. SOMrs uses a different number of cache entries for inline caching.

Inlining Control-Flow Structures and Lowering of Basic Operations In SOM, everything is a message send, which is a flexible and elegant model, but comes at the cost of performance. Therefore, reducing the number of method calls is essential

for good performance. Since control-flow operations are defined as methods, e.g., `#whileFalse:`, we inline such methods to avoid the call overhead and represent control flow directly.

The inlined constructs are methods previously mentioned in section 2.3.2, namely `#ifTrue:`, `#ifFalse:`, `#ifTrue:ifFalse:`, matching `nil`-specific methods e.g. `#ifNil:`, `#whileTrue:`, `#whileFalse:`; `#and:`, `#or:`, and `#to:do:`. The inlining decisions taken mirror those of TSOM/PySOM, and the only difference is that SOMrs also inlines the `#to:do` method, which was only done in TSOM_{AST}/$\text{PySOM}_{\text{AST}}$.

Unlike our AST interpreters based on meta-compilation, we do not directly replace standard arithmetic operation function calls such as + and >= with specialized AST nodes directly when parsing. Though, we directly implement them in both interpreters as primitive functions, alongside e.g. common string or array operations, and so they are special-cased to achieve better performance nonetheless.

Trivial Methods SOMrs directly handles trivial methods (see section 2.3.3), avoiding the allocation of a stack frame for simple methods such as getters and setters. Both SOMrs-AST and SOMrs-BC handle all four trivial method types which we defined in Chapter 2. In practice, this means every method in both of our interpreters is categorized as either a normal method, primitive method, or trivial method.

Monomorphic Inline Caching To optimize the method dispatch, *inline caching* (see section 2.3.2) stores lookup results at a call site. After a successful method lookup, the receiver class and the result is cached at that call site and thus avoids the need for subsequent lookups when executing the call site with the same receiver class again. Since our benchmarks are overwhelmingly monomorphic (see appendix A.1), as are the majority of call sites in industrial benchmarks for dynamic languages (Kaleba et al. 2022), storing only one cache entry per call site is sufficient for SOMrs. We previously used polymorphic inline caches, but did not see noticeable performance benefits, and thus, decided to simplify the implementation.

Supernodes and Superinstructions Based on the work of Casey et al. (2007) and Larose et al. (2023, 2022) we added specialized and coarser bytecodes and AST nodes

for common operations typically referred to as superinstructions or supernodes. For example, we added the `PUSH_1` bytecode to push a 1 onto the stack, and the `IncLocal` AST node to increment a given local variable by 1. We implement less superinstructions and supernodes overall than our interpreters based on meta-compilation, though, as we see in appendix C.

NaN Boxing We implemented *NaN boxing* (Leijen 2022), a technique that encodes data by using not-a-number bit patterns of IEEE 754 64bit floating numbers. With this technique, NaN doubles can be used to encode a type tag and a value. Crucially, there is room for pointers, since they require only 48 bits on current 64-bit systems. This allows us to represent doubles without boxing, i.e., without allocating a wrapper object, as well as 32bit integers, `true`, `false`, `nil`.

A key benefit of NaN boxing is that if every number type is a double by default, then no unboxing is necessary for double types, which can be read directly from the value. This is not the case in SOM, which therefore does not benefit from this advantage of NaN boxing. However, when we later implemented various object tagging schemes instead of NaN boxing, we did not observe a significant performance improvement. Therefore, the versions of SOMrs analyzed in this chapter rely on NaN boxing and not an alternative type of tagging.

5.3 Garbage Collection in SOMrs

In Chapter 4, Garbage Collection (GC) was already provided to our interpreters by the meta-compilation frameworks themselves. This is not the case for SOMrs, which needs to implement its own garbage collector(s) or rely on an existing GC framework.

Automatic garbage collection in Rust is idiomatically handled using naive reference counting (see section 2.5.1), with the `Rc<T>` type. This type allows for multiple ownership, by keeping a reference count to the number of live references, and safely dropping the value when this counter reaches zero. To allow for safe mutable access, it can be paired with the `RefCell<T>` which dynamically checks borrow rules, therefore data `T` can have multiple owners that can mutate it using a `Rc<RefCell<T>>` type. This representation is useful for SOMrs, since e.g. a `Rc<RefCell<Class>>` can

be shared among several `Instances` of this class, such that any instance can read but also write to static class fields.

However, this approach leads to issues, most notably difficulties handling cycles, but also lower execution speeds. Pointer access is ubiquitous, yet accessing a value here requires checking that the value is not already mutably borrowed, and to increment the reference count. After using any such pointer type and having it be dropped when exiting the current scope, the reference count gets decremented and then checked to potentially free the value. Moreover, naive reference counting cannot easily deal with cycles, which can cause memory leaks.

For these reasons, and while it is an idiomatic way to implement garbage collection in a Rust-based VM, we chose not to rely on the `Rc<T>` type. To reduce the overall engineering effort needed while achieving good performance, we instead implement GC by relying on the MMTk framework, which we see first in section 5.3.1. We then describe our approach in section 5.3.2, and since efficient GC can be difficult to correctly implement in Rust, we expand on these challenges in section 5.3.3.

5.3.1 MMTk: Memory Management ToolKit

While not directly provided by the host language, there are several ways to implement garbage collection within Rust, which we described several of in section 2.6.4. For SOMrs, we chose to rely on MMTk (Blackburn et al. 2004), which stands for *Memory Management ToolKit*. It is a framework for implementing garbage collectors, which is being implemented in Rust (Lin et al. 2016).

MMTk was chosen for the following reasons:

- its stability, having existing sturdy bindings for OpenJDK[3] and V8[4], among other industry virtual machines.
- the promise of high performance, an essential argument for our interpreters as mentioned at the start of section 5.2.
- saving engineering effort from not having to implement GC strategies ourselves.

[3] https://github.com/mmtk/mmtk-openjdk
[4] https://github.com/mmtk/mmtk-v8

- the ability to easily switch between different GC strategies, to have the possibility to switch to a more effective one in the future.
- being written in Rust itself, thus being easier to integrate with our Rust-based virtual machine by avoiding designing a foreign function interface.

It is not a Rust-specific solution, as MMTk is designed to be usable in any language implementation. SOMrs must ensure correctness itself, e.g. properly report all reachable values when scanning the heap, or ensure that the world is properly stopped during collections.

5.3.2 Overview of our Garbage Collection Approach

Our pointer type is a mutable raw pointer directly into the garbage collection heap, called `Gc<T>`. The pointer that it refers to is mutable, as opposed to an immutable `*const T`, since we need to have the ability to modify the data in memory.

SOMrs works with both a *mark-and-sweep* collector and a *semispace* collector (see section 2.5.2), with both implementations provided by the MMTk framework. Mark-and-sweep (McCarthy 1960) is a simpler, non-moving approach to GC, which was the initial GC approach for SOMrs, and with which compatibility has been kept. Though, for performance reasons, the default GC plan for both SOMrs virtual machines is a semispace approach (Fenichel et al. 1969). The reasoning behind this choice is that allocation is cheaper with a semispace approach, as it relies on a bump pointer allocator, i.e. requesting N bytes cheaply allocates by increasing a pointer into the GC heap by N. That is because memory is guaranteed to be compact since all de-allocations happen as data gets moved and compacted to the new heap, therefore a de-allocation can never create a gap in memory. Meanwhile, mark-and-sweep never moves or compacts data, so de-allocations can lead to memory gaps, and so any allocation requires scanning the heap for a large enough space available in memory.

Faster allocation is key in our benchmarks since collections are not as common as they would be for larger, longer-running applications. Given that our systems are interpreters and do not feature JIT compilation, they are usually meant to run for shorter amounts of time, therefore not run long enough to trigger many collections.

Given the flexibility of MMTk, we could easily switch to a more efficient garbage collection algorithm in the future, e.g. Immix (Blackburn et al. 2008), should we want to further improve the performance of our runtime systems. Though since our current priority is allocation speed and not maximizing throughput, we find the performance of our GC sufficient for the purposes of this work.

5.3.3 Ensuring Soundness With Moving GC

As seen in the previous subsection, SOMrs can rely on GC approaches that move data when collecting, with collections being possibly instigated from various parts of our virtual machines. This potential moving can be at odds with the Rust type system if we do not properly inform it that specific data can be moved, which can lead to undefined behavior. In this section, we explain how such bugs may occur, and how we designed the SOMrs virtual machines to avoid these issues.

Example of Unsoundness from Moving GC

Rust often assumes that variables cannot get moved and *could not possibly have been moved*, and optimizes based on this knowledge. These invariants indeed hold in regular, safe Rust code, or when using non-moving GC e.g. mark-sweep. However, this is not the case with moving GC, such as with our semispace approach.

```
1 // Note: universe.current_frame is a known GC root
2 let prev_frame: &mut Gc<Frame> = &mut universe.current_frame;
3
4 // Requesting memory may trigger GC
5 let mut frame_ptr: Gc<Frame> = gc_interface.request_memory(SIZE);
6
7 // ERROR: prev_frame may be invalid if GC was triggered!
8 frame_ptr.initialize_frame(method, prev_frame)
```

LISTING 5.3: Example of unexpected undefined behavior from moving GC, which relies on `unsafe` Rust code.

Listing 5.3 shows an example of unsound code. Frames store references to the previous frame, in order to be able to access non-local variables. Therefore in the first line, we get a reference to the current frame, which will be the “previous frame” for the new frame. We then request memory to create a new frame object, which as with all memory requests, may trigger a collection and therefore move objects. Though

there should normally be no issues in this case, since the `prev_frame` is a reference to a `Gc<Frame>` that we store in `universe`, a structure that is used to report GC roots, and therefore the value is not isolated on the Rust stack. Finally, we use `prev_frame` to initialize the new frame object (as well as its associated `method`).

This leads to rare bugs where if GC triggers in this case, `prev_frame` still points to the old heap, having seemingly not been correctly adapted by GC despite having been reported as a root. Such an issue leads to the new frame getting initialized with invalid data (e.g. storing an invalid pointer to its associated method), and execution may or may not fail at some later point because of this incorrect behavior.

This is explained by the "frozen reference property"[5], meaning that the Rust compiler may perform optimizations assuming that the variable referenced cannot have changed, leading to incorrect behavior. Moving GC is implemented using `unsafe` code, yet the Rust compiler optimizes based on the knowledge that aliasing rules are respected and not circumvented using `unsafe`, which we indirectly did in this context with GC moving the `prev_frame` variable. Therefore, undefined behavior can be introduced through a mismatch between the expectations of the compiler and `unsafe` code breaking those expectations.

To limit such issues, throughout our VMs, we inform the Rust optimizer of which variables should be considered to have potentially moved. This can be done with the compiler hint `std::hint::black_box`, which tells the compiler not to perform optimistic optimizations on a given variable, and so assume that e.g. it may have been moved. Though, the stronger solution is for us to rely on the `UnsafeCell<T>`[6] type, which allows for interior mutability, i.e. an `&UnsafeCell<T>` may point to data that is being mutated, without introducing undefined behavior.

Therefore, whenever GC may occur, we can ensure that the compiler is informed of any potential variables that may have been moved, to avoid issues as we described in this section. This highlights that our GC implementation circumvents the Rust type system, and so that we have to manually ensure that any GC API call does not introduce undefined behavior, without Rust safety guarantees aiding in this process.

[5] https://gist.github.com/Manishearth/70856e2f01e18935681c
[6] https://doc.rust-lang.org/std/cell/struct.UnsafeCell.html

Though, this solution still allows multiple references to be created, which can lead to unsound behavior. We designed our interpreters such that the high-level structure minimizes the chances of such issues occurring, and though we cannot be certain that we have tracked every possible point of contention we have taken a lot of care to limit the possibility of such bugs, as we see in the next sections. Both SOMrs interpreters can run every benchmark in our suite, and more importantly pass every test in our SOM test suite. Our careful design and extensive testing leads us to believe that our runtime systems are sound, but these potential issues with our approach may be harder to manage if adapted to work within larger VMs, with many more potential GC triggers needing to be manually tracked.

Primitives Only Relying On The Universe On a Need-to-use Basis

Both SOMrs virtual machines feature a variety of primitives, to perform operations ranging from adding values to concatenating strings. Several of these primitives can trigger garbage collection, however. Invoking `Class::#new` will naturally need to allocate memory for a new instance of a class, but even simply adding two integers may possibly require allocating a big integer object on the heap. To avoid issues such as those described in section 5.3.3, primitives maintain invariants that allow us to avoid similar unsoundness bugs by design.

Both interpreters rely on a `Universe` Rust class for access to global state and operations that impact said global state, such as directing the flow of execution by requesting the creation of a new frame, or making requests to the garbage collector in general. Therefore, all primitives were previously given access to this `Universe` class to be able to request allocations at will.

However, primitives take arguments which we want to remain on the AST or BC stack, such that they are reachable for the GC. This stack is global state, which leads to an issue of ownership, since dispatching a primitive means fetching its arguments through global state, but primitives themselves need a reference to this global state if they require it e.g. to request memory.

A possible fix is by using `unsafe`, which can let us duplicate references. Dispatching any primitive can then be handled by fetching a reference to the arguments stored

on the stack, duplicating the reference so that the stack and its associated global state are no longer deemed by the Rust compiler to be already borrowed, and then both arguments and global state can be passed as arguments to the primitives. Theoretically, since primitives only take *references* to values as arguments, and not arguments directly, the referenced arguments should get correctly updated after moving GC and the references should then point to correct, updated values. However, it would lead to subtle bugs in some primitives, where the references to the arguments would not get correctly updated, in similar compiler errors as described in section 5.3.3. Moreover, the stack is a Rust vector, which Rust may theoretically reallocate if too many values are added to it by some stack-intensive primitive operation, e.g. the `Object»#perform:` primitive, which allows indirectly calling any method on the receiver object. Therefore, any duplicated reference may rarely be incorrect, and so this behavior unsound.

To address this issue, primitives are divided into two distinct types: those which never need access to the `Universe` or global state to function, and those which do. Primitives take in either the arguments they fetched from the stack, or a reference to global state, but never both. For primitives which never need access to global state, as part of dispatching them, their arguments are fetched and their types checked, and the primitive is then invoked as a function which takes in just these arguments. This has the major benefit of providing an easy way to identify which functions can never trigger GC, therefore cannot lead to GC bugs.

Every other primitive does not directly get provided its arguments, and is instead tasked to fetch and verify them from the global stack themselves. These primitives maintain the invariant that no value is left unreachable on the Rust stack, prior to any arbitrary code execution that may rarely trigger GC.

This dispatch mechanism leads to some added complexity, but avoids the unsoundness we would get from duplicating references. Moreover, this helps maintain our virtual machines by limiting the space where GC bugs can originate. This is an example of us designing our interpreters to avoid potential undefined behavior despite our use of the `unsafe` keyword, and this can be seen as a high-level mirroring of Rust ownership rules, which avoid bugs by limiting the amount of live references to data, while we limit the amount of live references to shared global state.

AST: Finding Roots, and the Host Language Stack

A common GC problem is that of finding roots. For live objects not to be missed by GC, the implementation must ensure that all root objects are reachable and reported prior to any GC phase. This led to issues regarding SOMrs-AST and its extensive reliance on the Rust stack, to be AST-like as first defined in section 3.4, which SOMrs-BC did not share given its non-recursive and non-reentrant design.

```
1 impl Evaluate for BinaryDispatchNode {
2     fn evaluate(&mut self, universe: &mut Universe) -> Return {
3         let receiver = self.receiver.evaluate(universe);
4
5         // /!\ if this call triggers GC...
6         // ...the contents of "receiver" did not get moved!
7         let arg = self.arg.evaluate(universe);
8
9         self.dispatch(self.method_name, receiver, arg, universe)
10    }
11 }
```

LISTING 5.4: Example breaking SOMrs-AST code, where we miss the `receiver` root since it is isolated on the Rust stack.

Listing 5.4 highlights this issue. When evaluating a `BinaryDispatchNode`, dispatching must be done after evaluating the value of the receiver and the argument. Since both can be any expression or block, evaluating them may trigger GC, especially if they represent complex long-running operations. If this happens, this can lead to invalid behavior: if after evaluating `receiver`, we evaluate `arg` and this triggers GC, then `receiver` was not reported to the GC and may point to data which has now gotten collected.

Our solution instead was to have SOMrs-AST rely on a custom value stack, which shadows part of the Rust stack. Whenever we would leave a value stranded on the Rust stack and not reachable by GC, we do not rely on the Rust stack but instead push it to our custom global stack, and we then report each GC pointer within it as a root. Values are removed from the stack when they are no longer used by the VM.

Adding and removing values to our custom stack has an unfortunate performance cost. However, an unexpected benefit is that this helps improve method dispatching, which is highly common. Since our stack is a `Vec<Value>` type, its values are stored contiguously. When dispatching to a new method, we need to allocate a new `Frame`

object in which the arguments of the method must be copied. And since the arguments are already stored on our stack contiguously, which matches their expected memory layout in `Frame` objects, we can call `Vec::drain` to efficiently remove the arguments from the stack while giving us a slice that we can copy to the `Frame` object. Moreover, we may be able to get further speedups by never moving arguments off of this global stack into the frame, and instead directly accessing/modifying them, though we may then require additional bookkeeping to ensure we can correctly clean the stack in the event of a non-local return.

Despite this optimization mitigating the performance loss, reifying the language stack in this way is in no way an ideal solution. However, we found it to be necessary to ensure that SOMrs-AST properly interacts with GC if it is to keep its recursive design.

Interestingly, this makes SOMrs-AST a stack-based AST interpreter, while one would typically associate using stacks with non-recursive, e.g. more BC-like designs. This indeed makes our AST interpreter amenable to be refactored to be more BC-like, since by no longer relying on the host language stack for temporary values, we could also more easily refactor the interpreter to not rely on the host language stack for control flow, and no longer use a recursive structure.

AST: Storing Nodes on the Rust Heap

As seen in section 5.2.2, SOMrs-AST stores its AST representation on the Rust heap instead of our GC heap. Edges between AST nodes do not rely on our `Gc<T>` pointer type, but on the Rust `Box<T>` type, the standard pointer type for data on the Rust heap. We implemented a version of SOMrs-AST which did not rely on `Box<T>` at all, and stored all the AST nodes on our GC heap instead. Unfortunately, this led to major issues with moving GC invalidating pointers to some AST nodes.

We use the code in Listing 5.5 to highlight the problem. `AstBody` is one of the most common AST nodes, which stores a sequence of nodes, and is used by methods/blocks to execute all their expressions in order. This code works as expected if the AST nodes are stored on the Rust heap. If they are stored on the GC heap instead, but using our non-moving GC implementation, we also do not run into issues. However,

```
1 impl Evaluate for AstBody {
2     fn evaluate(&mut self, universe: &mut Universe) -> Return {
3         let mut last_value = Value::NIL;
4         for expr in &mut self.exprs {
5             last_value = expr.evaluate(universe);
6         }
7         Return::Local(last_value)
8     }
9 }
```

LISTING 5.5: The `evaluate` method in SOMrs-AST for any sequence of nodes, invoked for every block and method.

storing nodes on the GC heap while using moving GC breaks this implementation. Since nodes are GC objects, they get moved whenever a collection triggers, and since `evaluate` method calls can execute any AST node, they may always be assumed to potentially trigger GC. If this happens in this example, then `self` itself would point to an incorrect, non-moved version of the `AstBody` node, `self.exprs` would become invalid and SOMrs-AST then has incorrect behavior.

We may manually inspect the Rust stack ourselves to track and move all node references, as previously described (see section 5.6.1), but this remains a complicated and likely error-prone solution. The easiest solution may be to report all these `self` values as roots, for each `AstBody`, in another stack, on top of the one previously described in the previous section. Then the next expression in the sequence would be accessed by not accessing `self`, but the value of `self` we stored on our own stack, to ensure we do not access an incorrect value. Though, this would likely induce major slowdowns.

On another note, this design decision would make our AST interpreter partly rely on its own reified stack for control flow. To avoid slowdowns, this would encourage refactoring the overall design to not be recursive, and so shifting the interpreter towards being more BC-like. Alongside the stack we already use to store temporary values that we saw in the previous section, one can see how using Rust while trying to get a well-performing recursive design, leads to pressure to switch to a non-recursive, more BC-like design, from the host language itself.

Because of these issues, we opt not to store the AST on the GC heap at all. We do not find this decision to be limiting in terms of performance, and we did not observe a clear benefit on our non-moving GC implementation.

Self-Referential Frames

```
1 pub struct Frame {
2     pub previous_frame: Gc<Frame>,
3
4     pub associated_method: Gc<Method>,
5
6     // To know where to resume execution when returning to parent frames.
7     pub bytecode_idx: u16,
8
9     // Arguments and local variables would idiomatically be stored in vectors,
          though we do not. (see current section)
10    // pub args: Vec<Value>,
11    // pub locals: Vec<Value>
12 }
```

LISTING 5.6: A example of a frame in SOMrs-BC. Frames in SOMrs-AST are almost identical, though e.g. do not need to store a bytecode index.

The most accessed object in both our virtual machines are language frames, which are created for each normal (i.e. non-primitive, not specialized) SOM method call. Listing 5.6 shows an example. Each SOMrs-BC frame contains an `associated_method` field, storing a pointer to the method associated with the current frame, used to access the bytecode for the current method or literal values. Additionally, the current bytecode index is stored when creating new frames, such that when they are finished executing, execution can resume in the parent context at the right index. This leads to frames requiring slightly more memory in SOMrs-BC compared to SOMrs-AST (see section 5.5.2)

Frames in both the AST and BC interpreter contain a reference to the previous frame, as well as arguments and local variables for the scope associated with this frame. However, the number of arguments and locals associated with each frame depends on what scope it represents. There can be an arbitrarily large amount of arguments or local variables in SOM (though often capped at 255), and our virtual machines must account for these possibilities.

The idiomatic way to handle this in Rust is to use one or several vector types: `Vec<Value>`. Prior to managing our own GC heap, a SOMrs `Frame` had to contain a vector for these values, and creating a new frame thus involved an additional invocation, that of a vector on the Rust heap. Vectors can always grow in size, however, which is a non-requirement in this case. Given any scope, we always expect the same number of arguments and the same number of local variables, which

was determined by our SOM parser and compiler. Fixed-size arrays may then appear better suited to our purposes, but their size has to be determined at compile-time, while we only determine this size during runtime, during our compilation phase from SOM to AST/BC.

Using GC, or more specifically being in charge of our own allocations, helps us solve this problem by letting us change the memory layout of our `Frame` objects. Frames are allocated with extra memory of a variable size N, with N being the size of a `Value` type times the total number of argument and local variables. Therefore, instead of storing them on the Rust heap with a `Vec<Value>` type, we can store them directly inline for any frame and ensuring each frame incurs a single allocation and not two.

This design meets the Rust type system halfway. A `Frame` is still a Rust struct, and we can access all of its data normally, e.g. reading the bytecode index for the current frame is done using `frame.bytecode_idx`. Reading arguments and locals, however, is done through known offsets from a pointer to the frame, and the Rust type system can not know that these values are actually part of the frame without issue.

5.4 Methodology to Compare the SOMrs Interpreters

Our methodology for SOMrs is largely based on section 4.2. Its goal is to characterize the performance trade-offs of AST and BC interpreters in the context of Rust, focusing on run-time performance and memory usage.

Since SOMrs does no JIT compilation, we do no analysis for peak performance and warmup performance. Furthermore, we did not assess the individual performance impact of our optimizations. When developing SOMrs, we relied on our performance results for interpreter performance in TSOM/PySOM to prioritize which optimizations to implement, and when doing so, we observed similar performance benefits as expected based on our previous findings. Therefore, we did not deem a similar examination of the impact of optimizations for SOMrs to warrant the engineering effort, as we expect it would yield largely similar results as we already found in section 4.3.1.

5.4.1 Interpreter Performance

Run-time performance in SOMrs is only assessed through interpreter performance, as we do no JIT compilation. We compile both interpreters to stand-alone binaries, while enabling several flags for optimal performance. This includes setting the optimization level to the maximum available (`opt-level = 3`), disabling parallel building to ensure LLVM generates optimal code (`codegen-units = 1`), enabling link-time optimization (`lto = yes`), and minimizing the amount of stack information held onto by disabling unwinding the stack in case of an error (`panic = "abort"`). Though, other than changing the optimization level, we found these flags to be of minimal performance benefit.

5.4.2 Memory Usage

As with our interpreters on top of meta-compilation systems, we expect our AST interpreter to have some memory overhead compared to our BC interpreter, since our bytecode is a more compact representation than our AST nodes. Though, the SOMrs interpreters collect far less profiling information, since there is no JIT compilation involved, which is likely to impact the results by having both systems require less memory overall.

Because of the complexity of the underlying meta-compilation systems, our experiments in Chapter 4 relied on the maximum resident set size (see section 4.2.4), which is not perfectly precise but an overapproximation of memory use. Using the same metric on SOMrs did not yield interesting results, since our implementation pre-allocates a fixed size for the GC heap, therefore the maximum RSS for most benchmarks was generally the size of the GC heap. But since we have more control over garbage collection than TSOM/PySOM, we can more accurately track all allocations within SOMrs, and so report exactly how many bytes were requested throughout run time.

To calculate the overall impact of the program representation on memory use, we keep a global memory counter which gets incremented by the memory requested for each allocation. This allows us to also track precisely what is being allocated (e.g. a frame, method, instance, ...) since requesting memory requires specifying

the data structure for which it is required, to aid in debugging and maintaining our interpreters. This is especially useful to track the impact of the program representation itself, isolating types that correspond to methods, classes, literal values, and any type that is used to represent the program. Though as AST expressions are stored on the Rust heap (see section 5.2.2), we also manually track these Rust heap allocations and increase the memory usage based on their sizes.

5.4.3 Experimental Setup

All experiments ran on Ubuntu 22.04.5 (kernel 5.15.0-130), with two 6-core Intel Xeon E5-2620 CPU at 2.40GHz and 16GB RAM. This is the same machine we used for our experiments on meta-compilation-based interpreters in section 4.3, though with a more up-to-date Ubuntu and kernel version, a change which we do not believe has meaningfully affected their performance.

We use ReBench (Marr 2023) version 1.3 to set up the machine for benchmarking, e.g. by minimizing measurement noise, and to collect the results. We use the Are We Fast Yet benchmark suite (see section 2.7), to allow helpful cross-language comparisons. This enables us to compare the performance of our SOMrs interpreters with widely used bytecode-based virtual machines, such as Node.js, Java and CPython. We use two different versions of CPython, namely CPython 3.10 and CPython 3.13, since CPython has received many optimizations in its past few releases, significantly improving performance.

As with our interpreters in Chapter 4, we measure interpreter performance, by invoking benchmarks 100 times each, and with the benchmark harness measuring how long one iteration takes.

5.5 Results of Comparing our SOMrs AST and BC Interpreters

In this section, we compare our two SOMrs interpreters in terms of run-time performance and memory usage. We follow the methodology previously defined in section 5.4.

5.5.1 Run-time Performance

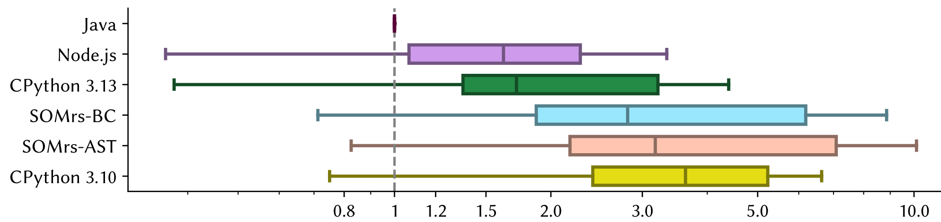


FIGURE 5.1: Interpreter run-time performance of the Are We Fast Yet benchmarks, on a logarithmic scale, with Java as baseline. SOMrs-AST and SOMrs-BC are roughly at the same level, and are in the same ballpark as CPython 3.10.

We compare SOMrs-AST and SOMrs-BC to other interpreters in terms of run-time performance, with Java as a baseline. Figure 5.1 shows the performance of all systems. After Java, Node.js is first with a median slowdown of 1.69× compared to Java, then CPython 3.13 with 1.72×. We then have SOMrs-BC and SOMrs-AST, with median slowdowns of 2.81× and 3.18× respectively, then followed by CPython 3.10 with 3.63×.

Both SOMrs interpreters have similar performance ranges, with 0.71× to 8.86× for SOMrs-BC, and 0.83× to 10.11× for SOMrs-AST. Thus, they are roughly similar in performance, but the AST interpreter is a bit slower, around 16%. This performance difference can be observed in our results, and is more pronounced than with our meta-compilation interpreters (see section 4.3.1) but we do not find it to be large enough to make our AST interpreter entirely outclassed by its BC counterpart.

We believe these results show that our SOMrs implementations reached a level of optimization that makes them suitable for a wide range of use cases where just-in-time compilation is not needed. They can both struggle to reach the performance of highly optimized implementations like Java, which have received far more engineering effort, but also occasionally outperform them. Arguably, it is a level of performance that can be reached with only modest effort by many custom language implementations, and thus, it is a performance level that is representative for language implementations with limited engineering resources.

Our BC interpreter outperforming its AST counterpart may be explained in part by SOMrs-AST relying on the host language stack for its execution, while also having to keep its own stack to track all GC roots. But since the performance difference is

minor, while these results are not expected to generalize, they suggest that the choice between AST and bytecode may not be as important as careful optimizations. While further optimizations of the SOMrs implementations may increase the difference in performance, each language implementation project needs to weigh the tradeoffs between performance and maintainability, where for instance tooling plays a major role.

To address our original hypothesis: *can AST interpreters be as performant as BC interpreters?*, we find that when relying on pure, Rust-based interpreters, our AST interpreter is outperformed by its BC counterpart. Though, the performance gap is not wide enough for SOMrs-BC to fully outclass SOMrs-AST.

To compare SOMrs with our interpreters in Chapter 4, we also compare the performance of all our interpreters in appendix C.3

5.5.2 Memory Usage

As discussed in section 5.4.2, we assess the overall memory usage by aggregating the amounts of memory needed for all allocations, and then isolate the amount of memory associated with allocations that are needed to store the program representation itself.

Total allocations

We calculate how many bytes are allocated in total for each benchmark by both SOMrs-AST and SOMrs-BC. This includes all bytes for the program representation, but also any structure needed at run time, such as language frames or class instances.

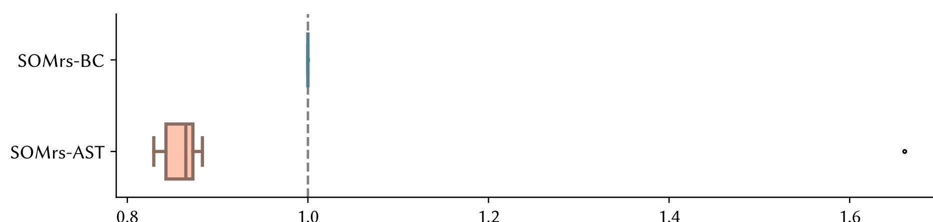


FIGURE 5.2: Total number of bytes allocated, with SOMrs-BC as a baseline.

Figure 5.2 shows our results. For all but one benchmark, SOMrs-AST required far less memory overall throughout execution, with a median of 0.50x less memory allocated, down to a minimum of 0.45x. These results are not explained by the size

of the program representation, but by the amount, and size, of structures allocated during run time. Instances are allocated at run time, yet have the exact same size and memory layout for both SOMrs implementations. However, frames require 8 more bytes in SOMrs-BC, as they store a pointer to their associated method to access e.g. literal values for the current context, while SOMrs-AST stores this information in AST nodes directly. And frames are by far the most allocated structure, since every guest language method invocation (non primitive, non trivial method, and which was not inlined) requires a new frame. Therefore, our AST interpreter needs less heap memory overall, especially for longer running benchmarks.

This memory usage difference being caused by frames is shown by the fact that the only benchmark where the AST needed 1.66x more memory is Mandelbrot, which consists of nested loops that all get inlined, as well as primitive method calls. Therefore, few stack frames are allocated during execution. This shows that if the frame size was smaller in our BC interpreter, so that it matched the size of a SOMrs-AST frame, our BC interpreter would require less heap memory than its AST counterpart. But in practice, since AST interpreters can typically store information on the host language stack instead on a reified stack, they may sometimes require less heap memory than BC interpreters, despite the program representation itself often being less memory-efficient.

5.5.3 Program Representation Size

On top of calculating the overall memory usage of our interpreters, we also isolate the memory usage needed for the program representation itself.

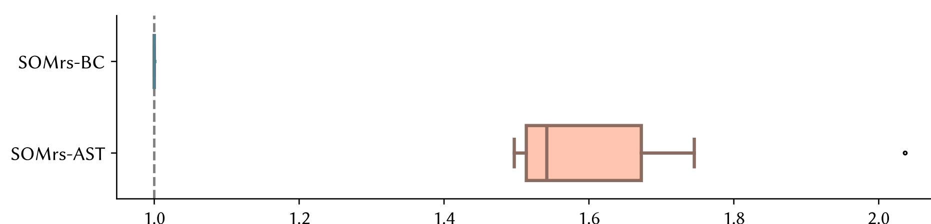


FIGURE 5.3: Comparison of the memory needed by both SOMrs interpreters to represent the program itself, with SOMrs-BC as the baseline. Higher is worse.

Our results can be seen in fig. 5.3. For every benchmark, representing the AST required more memory than representing the BC. SOMrs-AST needed a median of

1.54× more memory than SOMrs-BC, with a minimum of 1.49× and a maximum of 2.03×. Therefore, in our SOMrs interpreters, relying on an AST structure always requires much more memory than relying on a BC structure.

These results are in line with our results for the total number of bytes allocated, where our benchmark that allocated very few frames saw the AST require a similarly larger amount of memory. They are also in line with those obtained in section 4.3.4, in which the memory use for the BC interpreter was lower than that of the AST, which has a less compact program representation. However, for SOMrs, the observed amount of memory needed may be lower than what we observed in TSOM and PySOM. This is not a strict comparison, since SOMrs calculates the precise amount of bytes needed to represent the program, while our meta-compilation systems calculated the maximum heap size requested by the language implementations when executing programs which loaded large amounts of code, such that the run time was dominated by the cost of the program representation. Regardless, both measures represent the gap in the memory needed to represent a program for an AST and a BC interpreter, though SOMrs is more precise.

For TSOM and PySOM, while the difference narrowed during the execution itself due to the addition of profiling information, when the program was not executed, the AST on PySOM needed 2.55× more memory than its BC counterpart, while on TSOM that difference was 2.89×. In comparison, SOMrs-AST only needs a median of 1.54× more than its BC counterpart. This may be explained by the fact that the AST is represented using Rust structures as opposed to Java/RPython objects, in which case every object also requires some additional metadata, such as for object headers and parent class pointers.

5.6 SOMrs: Interpreter Design Discussion

In this section, we expand on the design of both SOMrs interpreters. We discuss several design decisions in both SOMrs-AST and SOMrs-BC, focusing particularly on how the majority of these decisions were shaped and influenced by the usage of the Rust language.

We first talk about SOMrs-AST, and how the Rust language clashed with our intended design.

5.6.1 Design Mismatch: Rust and AST-Like Interpreters

A key advantage of AST interpreters is the reduction of overall engineering effort (see section 3.4.3), from relying on host language facilities. SOMrs-AST does rely on the host language stack and host language mechanisms, e.g. a common node trait `Evaluate`, but we find that while Rust provides many facilities that AST interpreters can leverage, closeness with the Rust language can be detrimental. Indeed, we find this to be a design mismatch between Rust and naturally recursive AST interpreter, a problem similar to semantic mismatch (see section 3.4.4), which refers to a mismatch between the host language and guest language. In this case, it corresponds to a mismatch between the host language and certain design decisions, which are AST-like in our case.

Mutable Tree Structures in Rust

Rust ownership rules (see section 2.6.1) can make it difficult to work with tree-like data structures. A tree is made up of nodes with edges between them, with each node typically represented by a host language object. For example, in SOMrs-AST, a `VarWriteNode` may hold a reference to an `AdditionNode`, which represents the expression to be written to the local variable in question.

This is easily implementable in Rust, but can cause issues if the tree structure can be mutated during run time, and when execution is recursive. If the tree must be mutable, then executing a method requires borrowing it mutably. However, if guest code decides to invoke a guest method recursively, then invoking that method a second time will require borrowing the method object mutably again, which goes against ownership rules. This happens because the mutable borrow for the first invocation of the method was never relinquished, as a consequence of our naturally recursive AST interpreter relying heavily on the host language stack. Invoking a method is done by calling `evaluate` for that method, and so borrowing it mutably,

with that borrow not relinquished until this method is done executing, hence recursion breaking execution.

There are several potential solutions to this problem, but the most efficient and least likely to induce additional runtime costs may be to rework the interpreter to not rely on the host language stack for its execution. Though, we consciously tried to avoid such a design since we previously defined recursive execution as a characteristic of AST-like interpreters (see section 3.5), and so SOMrs-AST was designed to be naturally recursive, like TSOM_{AST} and $\text{PySOM}_{\text{AST}}$. We solved this problem through our `unsafe` garbage collection API, where several mutable references are made possible, and the caller is in charge of ensuring soundness (see section 5.3.3). This solution is clearly not idiomatic Rust, however, and recursive AST interpreters struggle with other aspects of GC, such as finding and reporting roots.

Finding GC Roots on the Rust Stack

Since SOMrs-AST is designed to be strongly AST-like, it relies on the host language stack, and so often stores temporary variables on it. However, as we described in section 5.3.3, we find this design to be ill-suited to Rust when using any GC implementation that may need to scan the host language stack, and we therefore engineered the interpreter to have its own custom stack.

A possible alternative solution would be to use `unsafe` and inline assembly (using Rust's `asm!` macro) to get a pointer to the Rust stack and manually report GC heap pointers that we find in it. This would also involve spilling the contents of all registers to the Rust stack when collection is triggered, since they may also be storing GC pointers. While it may solve this issue, this is a non-idiomatic approach which would involve having different ASM instructions depending on the architecture, and which would be very likely to introduce new bugs from the added complexity.

So, since we avoid the Rust stack and reify our own, a non-recursive interpreter design becomes fewer design changes away. Therefore, the Rust language not providing mechanisms to easily scan the host language stack becomes a mismatch with optimized AST-like interpreters that rely on recursion, and is an example of the host language encouraging a more BC-like approach. It should also be noted that most

system-level languages do not provide stack scanning mechanisms, and so that this problem is not exclusive to the Rust language.

Serialized AST Interpreter

As we first saw in section 3.3.3, it is possible to implement an interpreter that relies on a serialized tree structure, i.e. a data structure where nodes are contiguous in memory, and access one another not using pointers but using relative offsets. It is a beneficial representation in terms of run-time performance, since it provides better cache locality and representing edges with offsets requires less memory than pointers do.

This optimization was considered for SOMrs-AST, since as seen in section 5.2.2, the majority of AST nodes have the same size, and as a result many of them are already stored contiguously in memory. More specifically, every root expression in any method or block is stored contiguously with other root expressions, while they all contain pointers to the nodes for their subexpressions. We could instead store even those subexpressions contiguously in memory with the root expressions, and rely on indices into this node list to represent edges, without incurring large amounts of engineering effort. In fact, it is already likely to be the most idiomatic way to represent mutable trees in Rust, being nearly identical to an arena (Goregaokar 2021). We would be relying on a special case of arena, since they do not typically impose any specific order for their stored elements, while we would likely strictly store our AST nodes in a pre-order serialization order. This would keep nodes that execute one after the other close in memory, leading to better cache locality, and this would allow us to store no relative offset if the next node to execute is the next one in memory.

We did not implement it since as we established in section 3.4, while this representation is used to model an AST, we consider it to be a prime example of leading to an interpreter design that lies somewhere ambiguously between AST and BC. Both SOMrs interpreters would then rely on a compact representation, and the difference between an AST node and a bytecode would become quite narrow, while we would need to start relying on a main AST node dispatch loop as we do in SOMrs-BC.

We expect that there would be a performance gain from this change, though we are unsure whether it would be major or minor, since we have not implemented any interpreter that relies on a serialized tree structure. Gibbon (Vollmer et al. 2017) achieved over 2x speedup on some benchmarks by relying on serialized trees, but these benchmarks were dominated by the cost of tree traversal, while our benchmarks are not large enough to generate extremely large ASTs.

5.6.2 BC Interpreter Design in Rust

As BC-like designs intentionally distance themselves from the host language, we did not find a design mismatch between our BC interpreter and Rust as in our AST interpreter. This section focuses instead on design decisions that were taken, or not taken, in SOMrs-BC.

Bytecodes Not Variable Width

As seen in section 5.2.3, each bytecode is represented as a variant of a Rust `enum`, meaning our bytecode set is not variable width and so not as compact as it could possibly be. This means that each bytecode has the same size in memory, and since some bytecodes require more information than others (e.g. a `PushLocal(u8)` should need less memory than a `Jump(u16)`), most bytecodes are bigger than they could be. The main benefit of this design decision is seen in the bytecode dispatch loop, which can then be represented with a switch statement over all possible bytecode. Using Rust, dispatching to the correct bytecode handler is done idiomatically thanks to the language checking the type of the bytecode for us, and the payload of each bytecode gets automatically extracted into named variables, leading to far less boilerplate code. It also has the benefit that Rust enumerations are easily represented by most debuggers, and can easily be printed to standard output in a human readable way. Its engineering benefits extend to the fact that when adding new bytecodes, their proper integration into the VM will be checked by the type system, since the interpreter will not compile until every switch statement checking the type of a bytecode accounts for this new possibility, which includes the main dispatch loop but also e.g. optimizing bytecode passes or disassembler.

In contrast, TSOM, PySOM and many other VMs rely on a variable-width encoding with raw bytes. We experimented with moving away from an enumeration to instead adopt a similar design, but we did not notice performance benefits from doing so. This means the memory saved does not seem to be a meaningful difference on our benchmark set, and therefore, we kept the idiomatic solution.

Threaded Code

While threaded code (see section 2.3.5) can yield good performance, SOMrs-BC does not implement it. Our concern was that we could not identify a matching optimization for AST interpreters , where instruction dispatch is usually handled using host-language mechanisms, and have therefore less control over instruction dispatch. However, as mentioned in section 5.2.2, SOMrs-AST has some degree of control over its instruction dispatch, as it does not rely on dynamic dispatch, and executing AST nodes is instead done using a match/case statement for each possible node, similarly to how one would typically dispatch bytecode. Therefore, we believe that it would not take a significant refactoring effort to modify SOMrs-AST to implement a main AST dispatch loop, and to make it rely on threaded code by making AST nodes directly jump to the body of the next node.

This design decision would then make our AST interpreter far more BC-like, since there would be little difference between the described AST dispatch loop and a classic BC dispatch loop. As such, threaded code was implemented in neither interpreter, as implementing it only in SOMrs-BC may give it an unfair performance edge over the AST which would prevent meaningful performance comparisons, and implementing it also in SOMrs-AST would make its design too BC-like. Though, the performance benefits may not be major in the case of a SOM interpreter, as we previously explained in section 2.3.5.

5.7 Conclusion

By writing interpreters in a system-level language such as Rust, one can write interpreters with high execution speeds, which is highly appealing for many language projects, especially when run-time performance is a key concern. As this thesis aims

to analyze the interpreter design space, we identify this as an important area to focus on.

While bytecode interpreters are often deemed to be the better choice to implement an interpreter that achieves excellent run-time performance, we did not know whether AST interpreters could rival BC interpreters in this context, as we observed to be the case in the previous chapter on meta-compilation systems, despite commonly held assumptions about AST interpreters. Prior to this chapter, we also did not know if the engineering benefits of AST-like interpreters, which rely extensively on the host language, extend to a language such as Rust where the language itself imposes constraints that may clash with efficient AST interpreter designs.

To answer these questions, we have implemented one strongly AST-like interpreter and one strongly BC-like interpreter, namely SOMrs-AST and SOMrs-BC, and compared their run-time performance and memory usage.

We found that our SOMrs interpreters both achieve acceptable performance, being on the level of CPython 3.10, though Python has more complex semantics than SOM. Under these conditions, our BC interpreter outperforms our AST interpreter, though the difference is not large. We did find, though, that Rust-based AST interpreters came with several engineering challenges when paired with a naturally recursive design and garbage collection that is not naive reference counting, such as difficulty making the AST mutable to add inline caches, or properly tracking and reporting all GC roots.

Our results show that AST-like interpreters can be implemented on top of a system-level language while being reasonably efficient, though that on top of Rust, the host language encourages pushing the design of such an interpreter towards becoming more BC-like. As with meta-compilation pushing BC designs slightly closer to being AST-like, we find that using the Rust language encourages AST interpreters to adopt more BC-like traits.

Both interpreters could still be pushed further in terms of performance, though improving the performance of our AST interpreter should be done while not straying too much towards a BC-like design. These results highlight potential mismatches between the host language and the chosen interpreter design, where the host language may be a bad fit because of the constraints that it imposes, or the facilities that it does

not provide. Such mismatches may be interesting to explore in the future with other hybrid designs on top of Rust, or with different host languages.

Chapter 6

Interpreter Design Recommendations

This chapter synthesizes our findings by providing recommendations for interpreter design, based on our exploration of the interpreter design space and on our experiments. With this, we aim to help language implementers by giving recommendations based on their needs and requirements, and based on the design space analysis which we performed throughout this work.

We organize these recommendations on a chapter basis, starting with high-level recommendations, based on Chapter 3, then recommendations for interpreters based on meta-compilation, based on chapter 4, and for Rust-based interpreters, based on chapter 5.

To make these recommendations easier to refer to and use, we summarize them at the end of this chapter in a table.

6.1 High-level Recommendations

Based on our observations from Chapter 3, we propose high-level recommendations for interpreter design. We talk about hybrid approaches as overviewed in section 3.3, advantages of AST-like compared to BC-like as seen in section 3.4.3, as well as AST-like interpreters benefiting from their non compact representation, as mentioned in the same section 3.4.3.

6.1.1 Consider Hybrid Approaches

We gave an overview of hybrid approaches in section 3.3. Our aim was to highlight that the design space was not limited to a binary choice between an AST design and a BC design. Rather, one can design a variety of interpreters that lie somewhere in-between. This may be beneficial for some implementations, which for instance strive for good performance but have limited means in terms of engineering effort, and so largely benefit from a degree of reliance on host language mechanisms.

Hybrid approaches can be beneficial in a variety of contexts. For instance, if the interpreter is targeting resource-constrained systems, then low memory use is a key concern and using a compact program representation is then desirable. Our AST representations required more memory in both TSOM/PySOM (see section 4.3.4) and SOMrs (see section 5.5.3), therefore, we find that bytecode requires less memory and may thus be the natural choice. But a hybrid approach may be used to instead rely on a compact AST format (see section 3.3.3), which may be a valuable choice to reduce the memory needed to store the representation, while still benefiting from relative proximity with the host language.

Alternatively, hybrid approaches may be a desirable choice because of design requirements or constraints that come with some runtime systems, depending on the domain in which they are designed, the host language relied upon or the guest language implemented. For instance, we talked in section 3.3.4 about some BC-like interpreters built on top of the Truffle framework, and so interpreters that leverage this framework may benefit from adopting hybrid designs.

6.1.2 Limited Engineering Resources: Utilize Host Language

As we mentioned in section 3.4.4, if large amounts of engineering effort are available, a BC-like approach is likely preferable to achieve the best possible performance since the interpreter can get ever closer to the target machine. Though as argued in section 3.4.3, if the engineering effort that can be dedicated to a language implementation is limited, it can be very beneficial to stay close to the host language.

For instance, by adopting a recursive interpreter structure and so leveraging the host language stack, the need for custom tooling remains minimal. In contrast, if the

stack is reified, code for stack printing or visualization becomes essential to enable debugging, where we can rely on the standard tooling with the host-language stack. Similarly, relying on host language objects for the interpreter representation means one can often rely on existing mechanisms provided by the host language to easily print and inspect their contents. Such re-use of existing mechanisms sidesteps a maintenance burden, as any such additional debugging tools must be managed by the language implementer.

Utilizing the host language also reduces engineering effort by reducing overall complexity. For instance, by reifying the stack, we add the complexity of reasoning about a stack machine, which is likely to imply a tail of bugs around stack balancing issues. Given that managing complexity is essential to software engineering, and that maintaining software can cost a large amount of time and resources, reliance on the host language is beneficial in ways that should not be underestimated.

6.2 Recommendations for Interpreters Based On Meta-Compilation

Based on our findings and on our experience designing our interpreters in meta-compilation systems, we give a set of recommendations for interpreters based on meta-compilation systems. We base them on our results regarding interpreter performance in section 4.3.1, peak performance in section 4.3.2, and observations on the benefits of non-compact AST representation, partly based on section 4.3.5.

6.2.1 Interpreter Performance: Both AST and BC Viable

We find our AST interpreters to be competitive with our BC interpreters in terms of interpreter performance. Based on this, we believe AST interpreters to be a viable design choice on top of meta-compilation systems, especially given their engineering benefits, as seen earlier in section 6.1.2. These results suggest that regardless of the interpreter design, good performance can be achieved, which may generalize to hybrid approaches leaning more towards the middle of our design spectrum defined in section 3.4.

Though, this advantage may be due to meta-compilation systems struggling with the code structure of a bytecode interpreter, and if so it may shift based on future

changes in meta-compilation systems. For instance, Graal-based runtime systems may now benefit from automatically generate BC interpreters (*Bytecode DSL* 2025; Humer et al. 2022), and more opportunities for boxing avoidance in these interpreters, which we found TSOM_{BC} to currently struggle with (see section 4.4.2).

6.2.2 Peak Performance: Likely Orthogonal to AST or BC

A core part of the appeal of meta-compilation is automatically benefiting from JIT compilation from an interpreter. Therefore, one may ask whether the design of the interpreter has an impact on the best performance achieved thanks to the generated native code.

When evaluating the peak performance achieved by JIT compilation in our runtime systems (see section 4.3.2), we did not see a clear advantage for either our AST or BC interpreters. This suggests that regardless of the chosen interpreter design, good peak performance can be achieved. It is possible that such an advantage for a certain interpreter design exists and that it was not identifiable in our experiments, since the peak performance achieved by our TSOM-based systems was rather similar, and the performance of our PySOM-based systems suffered from a few highly recursive benchmarks.

Additionally, one may ask whether an interpreter design that is more hybrid would be able to perhaps yield better performance than our interpreters designed to be AST-like and BC-like. We indeed pushed our BC interpreters slightly towards AST-like designs, since a recursive BC interpreter design was mandated by Truffle, and we found such a design to also yield better peak performance than a purely iterative one with RPython (see section 4.1.2). This benefit stemmed from more clearly informing the meta-compilation system of what code belongs to each guest language method, and thus providing better information on which to base optimizations. However, we cannot think of other changes to the design of our interpreters that could report yet more useful information about the running program to the meta-compilation system, and so at present we do not think there is such a hybrid design that would increase peak performance much further.

In conclusion, our results thus far find no clear edge for any particular interpreter design in terms of peak performance.

6.2.3 Non-Compact Representation: Benefits Dynamic Rewritng and Localizing Profiling Information

Our AST in both TSOM and PySOM is a non-compact representation, with each AST node being a distinct language object that encapsulates a specific operation. As we first noted in section 3.4.3, compared to a compact representation, using a tree-like, pointer-based representation provides additional flexibility, such as the fact that nodes naturally localize state. This can help reasoning, but it can also be advantageous in other ways.

For instance, we found in section 4.3.5 that with meta-tracing, inlining could reduce the peak performance achieved by our BC interpreter in some cases, since this led to the compiler having less precise information about the lifetime of variables, and so having to do extra boxing. In comparison, the AST kept more information about each variable through its sparse structure and localized state, avoiding this issue.

For meta-compilation systems which need profiling to generate code at runtime, a non-compact representation also has the natural benefit of localizing profiling information. Given an AST node, we can easily store profiling information specific to it inline or with a clear reference to out-of-line data. In contrast, profiling information in a bytecode-based interpreter needs to be stored in data structures that are accessed by a wide range of different bytecodes and thus, the interaction pattern is more complex and requires global instead of local invariants to be maintained.

Another benefit of a non-compact representation is that it can be useful, as mentioned in section 3.4.3, when efficient handling of dynamic rewriting is a major requirement. Compared with a compact representation, one can easily replace nodes in the tree with more specialized nodes when profiling information reports this operation to be valuable (Würthinger et al. 2012), or potentially collapse a subtree into a single node. In contrast, if the representation is too compact, inserting elements different from the original set is more difficult. If it is the same size as the original

data or smaller, the replacement is not impactful in terms of memory. Though if it is larger, it cannot fit where the original did, and one needs to rewrite the sequence as a whole, or use solutions such as jump tables or side tables that incur a minor run time cost for the indirection. Dynamic rewriting is not incompatible with BC-like implementations, e.g. with bytecode quickening (Brunthaler 2010a,b), but it is more cumbersome than with a less compact representation.

This edge in peak performance in $\text{PySOM}_{\text{AST}}$ compared to PySOM_{BC} might be limited to the specific meta-compilation system which our interpreters relied upon. But the advantages regarding dynamic rewriting and profiling extend to any runtime systems that do dynamic rewriting and/or dynamically generate native code, and so neither benefit is limited solely to meta-compilation systems. Though both of these benefits were especially noticeable while working on our meta-compilation-based interpreters, in which self-optimization and profiling are ubiquitous and vital to good performance.

6.3 Recommendations for Rust-based Interpreters

Based on our experience implementing SOMrs, we provide guidelines for designing interpreters in Rust. We base them on our experimental results in section 5.5.1 and our discussion of AST-like designs on top of Rust in section 5.6.1.

6.3.1 Best Performance: Bytecode Interpreters are Likely Preferable

Our bytecode interpreter outperformed our AST interpreter on top of Rust (see section 5.5.1). This is not an unexpected result, since if very good performance is important, one likely needs to use a design close to the target machine (see section 3.4.4). A more optimized representation for the best possible cache locality, such as highly compact bytecode, is thus likely to yield superior performance in most cases. However, reducing the distance with the target machine can significantly increase the engineering effort (see section 6.1.2), e.g. by requiring tooling to be implemented and maintained, which we did find to be the case with SOMrs-BC.

We found implementing a BC interpreter in Rust to be unsurprising in that the main challenges we encountered were typical for BC interpreters, e.g. stack misalignment issues, ensuring correct bytecode generation, and overall added maintenance effort compared to its AST counterpart. In large part, BC-specific challenges did not come from Rust typesystem rules, therefore matched the experience one would get when relying on different host languages. That is because by designing SOMrs-BC to be strongly BC-like, it purposefully does not heavily leverage host language facilities. Though, SOMrs-BC did rely on the host language in that our bytecodes are represented using a Rust `enum` instead of raw bytes, as seen in section 5.2.3. This made debugging and implementing tooling easier, as well as making the bytecode loop slightly simpler to implement and a bit more readable. We did not observe a performance cost from not relying on raw bytes, though we are not positive that this finding would generalize outside of our benchmark set.

6.3.2 Good Performance: AST Interpreters can Remain an Option

While we found our Rust-based BC interpreter to outperform our AST interpreter, our AST interpreter still performed relatively well (see section 5.5.1). We also found that it required less memory overall (see section 5.5.2). While neither interpreter was optimized to the utmost, both had implemented many optimizations, which by themselves can push the performance of both systems to achieve acceptable performance.

Therefore, if one wishes to benefit from the host language to reduce engineering effort, but also strives to achieve good run-time performance, an AST design can still perform well. One could then imagine gradually transitioning to a more and more BC-like design to achieve further performance still, if improving the performance of the interpreter becomes of higher priority as time goes on, at the cost of benefiting less and less from host language mechanisms. Though, benefiting less from Rust mechanisms may not be so large an issue because of the mismatch between Rust and AST interpreter designs, which we address next.

6.3.3 Recommendation for AST: Non-Recursive Design

For the reasons previously mentioned in section 5.6.1, if one wishes to implement an optimized AST interpreter, we discourage making its execution rely on recursion. While recursive designs performed well for both our AST and BC meta-compilation-based interpreters (see section 4.1), it is less amenable to a Rust-based AST interpreter that is concerned about run-time performance. Specifically, Rust ownership rules are a mismatch with mutable tree structures (see section 5.6.1), which make implementing major optimizations such as inline caching difficult. Moreover, when implementing garbage collection, it becomes necessary to ensure no values are left unreachable on the Rust stack (see section 5.3.3), which Rust provides no mechanisms to aid with, with the potential exception of inline assembly.

With a non-recursive AST design, the AST can easily be represented as one or several node arenas, which makes it easy to mutate parts of the tree at will. And since the interpreter then does not rely heavily on the host language stack, efficient GC can be more easily implemented. Such a design would arguably lean towards BC-like, as it would imply the design of a main dispatch loop akin to that of a classic bytecode interpreter, as well as possibly encourage storing the AST as a linear sequence, and enable other optimizations such as relying on a serialized representation (see section 3.3.3). This then does highlight the issue that an efficient, strongly AST-like interpreter in Rust is difficult to implement, since there is a mismatch between the host language mechanisms provided by Rust and the design of AST interpreters, namely issues with mutating trees and garbage collection. So, we find that when using the Rust language, designing interpreters to be more BC-like appears to be more valuable.

6.3.4 Mismatch With Host Language Needs Consideration

When implementing an interpreter, we previously highlighted that there may be potential semantic mismatches between the guest language and the choice of host language. For instance, a language with complex control-flow mechanisms may be hard to implement efficiently with a recursive interpreter design, as seen in section 3.2.4. If the host language cannot efficiently model, e.g. exceptions or continuations, it may

be necessary to move the interpreter design away from the host language and for instance reify frames. Therefore, the differences between host and guest language should be carefully considered when designing an interpreter.

Our experience with SOMrs-AST, as described in the previous section, also highlights mismatches between the host language and the design of the interpreter. The host language may impose some constraints, or not make certain features available, that make difficult the implementation of some interpreter designs. For instance, SOMrs-AST struggled with the constraints imposed by Rust ownership rules, but also the Rust language not providing mechanisms to easily scan the host language stack, leading to issues with development.

An analysis of potential mismatches, semantic or otherwise, may be interesting future work. This could be listing out features provided by a host language that may aid or hinder developing an interpreter using it, e.g. how easy garbage collection is to implement on top of any given language. Or, focusing on language choices, analyzing how host languages features interact with the implementation of guest languages. One could imagine deriving a map of the interpreter design space with regards to this aspect, and drawing conclusions from which host languages are most suited to which guest languages. Though, this is outside of the scope of this thesis, in which we focused our engineering effort on pushing the performance of a relatively small amount of optimized interpreters.

6.4 Conclusion

This thesis focuses on analyzing the interpreter implementation space, which is quite broad as any interpreter can be designed in a variety of ways. As a consequence, when implementing an interpreter, it can be difficult to identify what design decisions are beneficial or should be prioritized. Therefore, this chapter focused on synthesizing several findings from our work into actionable recommendations based on the needs and requirements of a given runtime system.

We summarize the recommendations given in this chapter in table 6.1. We highlight that instead of adopting a binary view of interpreter design with just AST and BC interpreters, there is a large space of hybrid designs that may be considered,

TABLE 6.1: Summary table for our design recommendations in this chapter.

Requirement	Recommendation
Choosing an interpreter design	Consider hybrid approaches (section 6.1.1)
Limited engineering resources	Utilize host language (section 6.1.2)
Prioritizes performance	BC-like design (section 6.1.2)
Meta-Comp.: Good interpreter performance	Both AST and BC designs viable (section 6.2.1)
Meta-Comp.: Good peak performance	Likely orthogonal to AST or BC (section 6.2.2)
Does frequent dynamic rewriting / Wants benefits of profiling info more localized	Non-Compact Repr. (section 6.2.3)
Best interpreter performance	BC-like design (section 6.3.1)
Good interpreter performance	AST design still performs well (section 6.3.2)
Choosing an interpreter design/host language	Consider mismatch with host language e.g. Rust and recursive designs (section 6.3.3, section 6.3.4)

and that may be valuable for different language implementations. In case of limited engineering resources, leveraging the host language can be highly beneficial, though moving away from it may yield more opportunities for performance optimizations.

On top of meta-compilation systems, we find that AST interpreters can be competitive with BC interpreters, both in terms of interpreter performance and peak performance. We expect that these findings would generalize to hybrid interpreter designs with a similar amount of optimization effort, given that they also report sufficient information to meta-compilation systems to achieve good peak performance. We also highlight that a non-compact, more AST-like representation can have several benefits on runtime systems that do frequent dynamic rewriting and perform large amounts of profiling, as is the case with our interpreters based on meta-compilation.

When designing interpreters in a system-level language, we find that bytecode designs provide better performance, as the literature reports. Though, designing a strongly AST-like interpreter can still achieve good performance and still benefit from the host language, which can be valuable to some language implementations. While an AST-like design has engineering benefts still, we did find Rust to be an example of a mismatch between host language and interpreter design, namely a strongly AST-like one. Indeed, the host language mechanisms provided by Rust can be at odds

with a recursive, tree-based design, especially with efficient garbage collection, and thus we recommend circumventing such issues by relying on a more BC-like design, such as a classic bytecode interpreter or e.g. a serialized AST interpreter. In general, this highlights that implementers must consider unexpected mismatches between the host language and their chosen interpreter design.

These recommendations may hopefully assist language implementers in constructing interpreters that fit their needs. They are applicable to strongly AST-like or BC-like designs, but also hybrid interpreter designs which mix AST-like and BC-like characteristics. We base these guidelines on interpreters for the SOM language, which is highly dynamic and shares many implementation and optimization challenges with other dynamic languages, e.g. Python, Ruby or JavaScript, and so we expect our results to generalize to other interpreters for dynamic languages.

Chapter 7

Conclusion

This thesis focused on programming language interpreters, their various potential designs, and the trade-offs associated with different design decisions. We also identified design dimensions associated with several different interpreters. To conclude this thesis, we summarize our findings and address our research questions.

Firstly: *what are key design dimensions that define the implementation space of programming language interpreters?* To define the interpreter implementation space, one may be tempted to rely on a simple binary model that only considers AST and BC interpreters. In Chapter 3, we find this not to be sufficient, since there are many hybrid approaches that lie somewhere in between AST and BC, and so cannot be as easily categorized. To encourage reasoning about this design space as a spectrum, we coined the terms *AST-like* and *BC-like*, to help describe interpreters closer to either of those designs, or which are ambiguously AST-like and BC-like. We identified a key difference between them as AST-like interpreters being *close to the host language*, as in they leverage host language mechanisms to help reduce engineering effort, and BC-like interpreters instead having the goal to *minimize the distance with the target machine*, as in try to generate code that runs as efficiently as possible, sacrificing the benefits of the host language, and requiring more engineering effort to increase run-time performance. From these terms, we extract key design dimensions to reason about interpreter design (see section 3.5), such as AST-like designs typically using recursion while BC-like designs avoid the host language stack, or AST-like designs relying on a sparse, non-linear program representation that relies on host language objects, while BC-like designs tend to rely on a compact, linear, byte-based representation. Chapter 3 provided a high-level overview of the interpreter design space as a

spctrum, contrasting staying close to the host language for engineering benefits, or separating itself from it to strive for better performance, which we highlight through our list of design dimensions.

After establishing the high-level context, we focused on interpreters defined within more specific design domains. We asked: *how does meta-compilation influence the design and performance of interpreters?* Meta-compilation is a valuable area in which to discuss interpreter design, since when writing an interpreter, one automatically benefits from JIT compilation and so largely improves performance of the runtime system at little cost. We found that in this case, AST interpreters could rival BC interpreters in terms of both interpreter performance and peak performance. We posit that because of the influence of the meta-compilation frameworks, the designs of our BC interpreters are encouraged to use host language mechanisms in ways that make them slightly more AST-like. Though we find the design of our BC interpreters to be suited to their underlying meta-compilation frameworks in terms of dynamically generating efficient code, as peak performance is similarly high in both our AST and BC interpreters.

In terms of memory usage, we found AST interpreters to require more memory than BC interpreters, due to their less compact representation. However, since such systems need to perform large amounts of run-time profiling, both designs require a large amount of additional memory to store this information, and so the difference in terms of memory usage narrows during execution.

We focused on the performance of interpreters based on existing meta-compilation frameworks, as well as the individual performance impact of various optimizations in terms of interpreter performance and peak performance. We provided a methodology for evaluating and assessing meta-compilation frameworks, which may be used by future projects.

To address our research question, we find that meta-compilation can influence the design of interpreters as to make BC interpreters lean more towards AST-like, and make AST interpreters competitive with BC interpreters in terms of run-time performance and peak performance.

Afterwards, we focused on system-level languages, of which a core part of their appeal is the performance granted by the amount of control that they provide. Therefore, we asked: *How does using the Rust language influence the design and performance of interpreters?* We found that as expected, a language such as Rust granted us enough control over the implementation to perform more optimizations than meta-complation systems did allow, e.g. NaN boxing. Unlike with meta-compilation systems, we found our BC interpreter to outperform our AST interpreter, as literature reports. However, the performance difference was not as major as one may expect, showing that optimized AST interpreters can have performance that is not vastly outperformed by that of an optimized BC interpreter. With regards to implementation effort, implementing a BC design did not cause major issues, but implementing an optimized AST interpreter in Rust was complicated due to mismatch between Rust and recursive, tree-based interpreters. This problem is far from unsolvable, but would likely require shifting the structure of our AST interpreter to be closer to a BC-like design, which we purposefully avoided doing in this work as we strived to compare strongly AST-like and strongly BC-like interpreters.

As with the previous research question, we find that design decisions can shift closer to AST-like or BC-like depending on what the interpreter is based upon, be it a a host language or a meta-compilation framework. Here, we find that the influence of the Rust language on interpreter design makes it more difficult overall to implement an AST-like design, and so more beneficial to focus on a BC-like design.

We provided recommendations based on our high-level overview, and the domains in which this thesis focused, which we synthesized in Chapter 6. We encourage the use of hybrid approaches if they may fit the needs of language implementers, and to rely on systems with AST-like design decisions for their engineering benefits, and BC-like for their performance benefits. For meta-compilation systems, we find both strongly AST-like and BC-like designs to be viable in terms of interpreter performance and peak performance, and so that language implementers may have relatively large amounts of freedom in this context. Within meta-compilation systems, we also find that a non-compact representation worked especially well to account for frequent dynamic rewriting. Finally, we recommend BC-like designs when writing interpreters

for the Rust language, though an AST-like design may still perform well. Overall, we recommend considering potential mismatches between the host language and the guest language, or the host language and specific interpreter designs, i.e. recursive interpreters in Rust.

7.1 Future Work

Future work may investigate the performance and design of different interpreters than we did in this thesis, such as interpreters that are not as strongly AST-like and BC-like as possible, but rather hybrid designs as seen in section 3.3. Indeed, we do not have results on what the performance of these various designs would be in the context of meta-compilation or when using Rust, nor potential matches and mismatches with those technologies. Such experiments could highlight benefits and disadvantages of specific arrangements of AST-like and BC-like design decisions, which may be highly valuable to implementers considering relying on a specific hybrid design.

Since we found that meta-compilation frameworks and the Rust language influence interpreter designs so as to lean closer to AST-like or BC-like, further research may also analyze interpreters based on different host languages. We find that certain host language mechanisms can encourage or discourage certain design decisions, but we based analyses of beneficial and detrimental design decisions for AST-like and BC-like designs on a select few runtime systems. So, this amount of interpreters could be increased, and our results therefore expanded upon. From such an analysis, one might also possibly define additional key design dimensions that were not relevant to our interpreters, and so potentially missing from our work.

Based on our key design dimensions and our model of interpreter design, one may envision interpreter frameworks designed fully with AST-like and BC-like design decisions in mind. Such frameworks may have the explicit goal to always enable closeness with the underlying host language, and so enable AST-like design decisions, i.e. leverage from native debugging, GC, etc., similar to the capabilities of meta-compilation frameworks. Alternatively, they may ensure that one is always able to be close to the target machine and so to create BC-like designs, i.e. allow for threaded

interpretation, or inline assembly. One may also improve meta-compilation frameworks to better support strongly BC-like interpreters (*Bytecode DSL* 2025; Humer et al. 2022) , or perhaps investigate ways to engineer the Rust language to be more amenable to AST interpreters.

Appendix A

Background (Chapter 2)

A.1 Overview of the Are We Fast Yet benchmarks

The descriptions and metrics in this section were taken from the Are We Fast Yet Github repository[1], which also contains additional metrics[2] about the benchmark set, such as observed polymorphism, allocations, array accesses, and more.

Table A.1 gives code metrics about each benchmark, to specify how large these benchmarks are and how macro-benchmarks differ from micro-benchmarks in terms of size. We show the number of lines of code executed at least once, the number of classes of which at least one method was executed, and the number of methods executed at least once. Additionally, we show *per iteration methods*, which are methods executed during each iteration, to not count methods only needed to set up or tear down each benchmark.

Table A.2 provides metrics about the observed polymorphism in every benchmark. For every call site, we provide the number of observed different receivers, as well as provide the total number of calls on each call site. We can see that our benchmark set is predominantly monomorphic, i.e. that there is only one possible receiver for most call sites, and that there are far fewer calls on polymorphic call sites overall.

[1] `https://github.com/smarr/are-we-fast-yet`
[2] `https://github.com/smarr/are-we-fast-yet/blob/master/docs/metrics.md`

TABLE A.1: Code Size Metrics for all Are We Fast Yet benchmarks.

	Executed Lines	Classes	Executed Methods	Per Iteration Methods
CD	356	16	43	41
DeltaBlue	387	20	99	75
Havlak	421	18	110	87
Json	232	14	56	56
Richards	279	12	47	47
Bounce	42	5	11	11
List	30	2	9	9
Mandelbrot	39	0	2	2
NBody	105	3	14	14
Permute	33	3	13	13
Queens	36	3	13	13
Sieve	22	3	9	9
Storage	23	4	10	10
Towers	42	2	12	12

TABLE A.2: Receiver polymorphism for the Are We Fast Yet benchmarks, i.e. number of potential receivers at any call site.

	Num. Rcvrs	Call Sites	Calls
CD	1	166	75,892,212
	2	3	23,151,272
DeltaBlue	1	171	6,219,974
	2	8	480,130
	3	8	3,576,416
	4	7	840,146
Havlak	1	256	79,581,122
	2	3	2,720,556
Json	1	163	32,115,418
	5	2	678,200
Richards	1	108	95,521,018
	2	2	10,000
Bounce	1	31	17,736,018
List	1	28	17,271,018
Mandelbrot	1	10	16
NBody	1	38	8,500,102
Permute	1	25	51,982,018
Queens	1	28	29,088,018
Sieve	1	19	60,018
Storage	1	23	32,774,018
Towers	1	27	39,390,018

Appendix B

Meta-Compilation Systems (Chapter 4)

B.1 Abstract Syntax Tree Nodes

The PySOM and TSOM AST interpreters differ significantly in the number of different AST nodes they provide and use. Since the Truffle framework and TruffleDSL strongly encourage to use nodes, the TSOM interpreter comes with many more nodes. On the other hand, the PySOM interpreter often relies on simple functions, for instance to implement primitives.

In table B.1 we give a general overview that is based on TSOM. However, we omit any nodes that are not used as true AST nodes but are merely additional implementation details. This includes helper nodes, and nodes that realize built-in functions but are never used at the AST level.

For brevity, node variants will use the notation `[Optional]` to indicate optional name parts. Thus the entry `[Non]LocalArgument` represents two nodes, one for local and one for non-local argument accesses. With `(Read|Write)` we denote alternative name parts. `[Uninit]` node variants typically need to resolve some detail on first execution, but then rewrite themselves to a version that caches the initialization result.

B.2 Bytecode Set

The SOM bytecode set was originally very minimal with 16 instructions.[1] For our purposes, we expanded it based on static and dynamic bytecode frequency. On the one hand, we wanted to encode programs as compact as possible, which means we selected bytecodes, typical arguments, and bytecode sequences based on the static frequency in which they appeared in our benchmarks. On the other hand, we selected them based on the dynamic frequency, i.e., based on how often they were executed. The resulting bytecode set can be seen in table B.2.

Note, the bytecode sets in TSOM and PySOM use different bytecode variants for accessing frames and method activations, i.e., sends, since the two implementations differ as discussed in section 4.1.6.

[1] https://github.com/SOM-st/som-java/blob/master/src/som/interpreter/Bytecodes.java

TABLE B.1: SOM AST Nodes including Supernodes and Variants.

<table>
<tr><th>Name of AST Node</th><th>Description</th></tr>
<tr><td colspan="2">Fine-grained Nodes</td></tr>
<tr><td><code>Sequence</code></td><td>list of expressions, evaluated in order</td></tr>
<tr><td><code>[Non]LocalArgument(Read|Write)</code></td><td>read or write of argument</td></tr>
<tr><td><code>[Non]LocalVariable(Read|Write)</code></td><td>read or write of variable</td></tr>
<tr><td><code>Field(Read|Write)</code></td><td>read or write of an object field</td></tr>
<tr><td><code>(Uninit|Cached|True|False|Nil)Global</code></td><td>read a (specialized) global</td></tr>
<tr><td><code>UninitMessageSend</code></td><td>lookup and activate method,
rewrites itself, posssibly to built-in operation</td></tr>
<tr><td><code>GenericMessageSend</code></td><td>lookup and activate method, generic case</td></tr>
<tr><td><code>ReturnNonLocal</code></td><td>unwind stack to outer lexical method</td></tr>
<tr><td><code>CatchNonLocalReturn</code></td><td>catch non-local and return from method</td></tr>
<tr><td><code>(Int|Double|Generic)Literal</code></td><td>evaluate to literal value</td></tr>
<tr><td><code>Block(WithContext)</code></td><td>instantiate block, possibly with lexical context</td></tr>
<tr><td colspan="2">Basic Operation Nodes</td></tr>
<tr><td><code>(+|-|*|/|//|%|rem|<=|<|>|>=|=|==</code>
<code><>|~=|&|^|cos|sin|sqrt|abs|</code>
<code><<< | >>> | && | #and: |</code>
<code>|| | #or: | not)</code></td><td>various basic operations</td></tr>
<tr><td><code>New</code></td><td>instantiate object or array</td></tr>
<tr><td><code>#at: | #at:put: | #putAll:</code></td><td>read, write, and overwrite all for containers</td></tr>
<tr><td><code>#value | #value: | #value:with:</code></td><td>operations to activate blocks</td></tr>
<tr><td colspan="2">Nodes with Inlined Blocks</td></tr>
<tr><td><code>Inlined(And|Or|If|IfElse|</code>
<code>IfNil|IfNotNil|</code>
<code>Do|To|ToBy|DownTo|While)</code></td><td>inlines literal block arguments and realize
the control structure</td></tr>
<tr><td><code>ReturnLocal</code></td><td>return from method, with value of expression</td></tr>
<tr><td colspan="2">Supernodes</td></tr>
<tr><td><code>LiteralStringEquals</code></td><td>compare value to literal string</td></tr>
<tr><td><code>[Non]LocalFieldStringEquals</code></td><td>compare field to literal string</td></tr>
<tr><td><code>(Unary|Binary|Ternary|Quat)ArgSend</code></td><td>read arg, send method with 1-4 parameters</td></tr>
<tr><td><code>[Uninit]IncField</code></td><td>increment a field value</td></tr>
<tr><td><code>Inc[Non]LocalVariable</code></td><td>increment a variable based on subexpression</td></tr>
<tr><td><code>IntInc[Non]LocalVariable</code></td><td>increment a variable by a constant</td></tr>
<tr><td><code>IntInc</code></td><td>increment a value by a constant</td></tr>
<tr><td><code>(<|>=)Int</code></td><td>compare value to integer constant</td></tr>
<tr><td><code>LocalArg(<|>=)Int</code></td><td>compare argument to integer constant</td></tr>
<tr><td><code>[Non]LocalVarReadUnaryMsgWrite</code></td><td>read var, send unary message, write back result</td></tr>
<tr><td><code>[Non]LocalVarReadSquare</code></td><td>read var, multiply with itself</td></tr>
<tr><td><code>[Non]LocalVarReadSquareWrite</code></td><td>read var, multiply with itself, write back result</td></tr>
</table>

TABLE B.2: SOM Bytecode Set with all Superinstructions and Quickening Variants.

Bytecode	Description
Bytecodes and Variants Encoding Common Arguments	
HALT	exit interpreter loop
DUP	duplicate top of stack
PUSH_LOCAL [0|1|2]	push local, with explicit index byte, or bytecode variant explicit index is encoded as 1 byte
PUSH_ARG [0|1|2]	push argument, with explicit index byte, or bytecode variant
PUSH_FIELD [0|1]	push field value, with explicit index byte, or bytecode variant
PUSH_BLOCK	instantiate block, from index in literal array
PUSH_BLOCK_NO_CTX	instantiate block but do not set outer lexical scope
PUSH_CONSTANT [0|1|2]	push constant from literal array
PUSH_[0|1|NIL]	push zero integer, one integer, or nil object
PUSH_GLOBAL	push global, explicit index byte into literal array for name of global for lookup. Quicken bytecode, caching global association object
POP	pop top of stack
POP_LOCAL [0|1|2]	pop top of stack, store into local
POP_ARGUMENT	pop top of stack, store into argument
POP_FIELD [0|1]	pop top of stack, store into field
SEND	lookup and activate method, but quicken bytecode based on number of arguments
SUPER_SEND	activate method in lexical superclass
RETURN_LOCAL	return, use top of stack as result
RETURN_NON_LOCAL	return method that instantiate current block, unwind stack as necessary
Superinstructions	
RETURN_SELF	return, use `self`/`this` as result
RETURN_FIELD_0|1|2	return, use field read of `self` as result
INC	increment the top-of-stack value by one
DEC	decrement the top-of-stack value by one
INC_FIELD	increment field
INC_FIELD_PUSH	increment field and push result on stack
Control-flow Bytecodes for Block Inlining	
JUMP[2]	relative jump, with unsigned 1 or 2 byte offset
JUMP[2]_ON_TRUE_TOP_NIL	jump if top of stack is `true` and set top to `nil`, otherwise pop
JUMP[2]_ON_FALSE_TOP_NIL	jump if top of stack is `false` and set top to `nil`, otherwise pop
JUMP[2]_ON_TRUE_POP	jump if top of stack is `true`, always pop top of stack
JUMP[2]_ON_FALSE_POP	jump if top of stack is `true`, always pop top of stack
JUMP[2]_BACKWARDS	relative backwards jump, with unsigned 1 or 2 byte offset
Quickened Bytecodes	
Q_PUSH_GLOBAL	push global by reading from cached association object
Q_SEND [1|2|3]	lookup with polymorphic inline cache, and activate method variants to specialize for 1-3 arguments

Appendix C

Rust-based Interpreters (Chapter 5)

C.1 Abstract Syntax Tree Nodes

We show the AST nodes of SOMrs-AST in table C.1. As with appendix B.1, node variants use the notation `[Optional]` to indicate optional name parts.

SOMrs-AST has a smaller node set than TSOM_{AST}, e.g. not having nodes for arithmetic operations, and implementing less supernodes. Like $\text{PySOM}_{\text{AST}}$, the absence of the Truffle framework did not encourage using a great amount of nodes, and so it often relies on primitive functions instead. Though since the project is less mature than our meta-compilation-based interpreters, it also received less engineering effort overall, and so has more room for optimizations through custom, specialized nodes.

C.2 Bytecode Set

We show the SOMrs-BC bytecode set in full in table C.2. It takes inspiration from the bytecode set of TSOM and PySOM, to facilitate comparison with both of these interpreters. However, it does include some changes.

To efficiently inline `to:do:` (which TSOM and PySOM do not inline), we introduce the `Dup2` and `JumpIfGreater` bytecodes.

The SOM core library was modified to add the methods `ifNil:`, `ifNotNil:`, `ifNil:ifNotNil:` and `ifNotNil:ifNil:`. Our benchmark set was lightly modified to rely on these methods, turning e.g. method calls from `nil ifTrue:` to simply `ifNil:` for convenience. SOMrs-BC inlines all of these methods, thus our bytecode set thus also contains jump bytecodes related to these operations, which all check if

TABLE C.1: SOMrs-AST Nodes including Supernodes and Variants.

Name of AST Node	Description
Fine-grained Nodes	
`[Non]LocalVar(Read|Write)`	read or write of variable
`Arg(Read|Write)`	read or write of argument, local or non-local
`Field(Read|Write)`	read or write of an object field
`Global`	read a (specialized) global, likely using cache
`NAryDispatch`	lookup and activate method, possibly using cache
`SuperDispatch`	lookup and activate method from superclass
`ReturnLocal`	return to outer lexical method
`ReturnNonLocal`	unwind stack to outer lexical method
`Literal`	evaluate to literal value
`Block`	instantiate block with lexical context
Nodes with Inlined Blocks	
`Inlined(And|Or|If|IfElse| IfNil|IfNotNil| While|ToDo)`	inlines literal block arguments and realize the control structure
Supernodes	
`(Unary|Binary|Ternary)Dispatch`	read arg, send method with 1-3 parameters
`IncLocal`	increment a local variable by 1
`DecLocal`	decrement a local variable by 1

the top-of-stack value is `nil` and act accordingly. TSOM and PySOM now also inline these methods and implement similar jump bytecodes, but the versions we evaluated in Chapter 4 predate this change to the core library, and therefore these bytecodes were not yet present in their bytecode set. It should be noted that these changes to SOM itself, and to our benchmark set to rely on these methods, had no noticeable performance impact on any of our interpreters.

SOMrs-BC also omits some bytecodes which did not yield, or which we did not expect to yield, performance benefits, such as `PUSH_BLOCK_NO_CTX`, `RETURN_FIELD_0|1|2`, `INC_FIELD` or `INC_FIELD_PUSH`, and also discarded the `HALT` bytecode. Moreover, several of our bytecodes do not have specialized versions, e.g. with `PUSH_LOCAL [0|1|2]` being instead a generic `PopLocal(u8)` bytecode.

Finally, we do not have quickened bytecodes like `Q_PUSH_GLOBAL` or `Q_SEND`, as while the equivalent bytecodes in SOMrs verify whether there is associated cache and act accordingly, they do not rewrite themselves.

C.3 Performance comparison between SOMrs and TSOM/PySOM

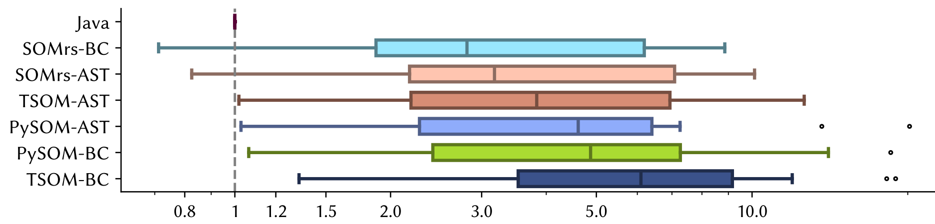


FIGURE C.1: Interpreter run-time performance of the Are We Fast Yet benchmarks, on a logarithmic scale, with Java as baseline. We compare both SOMrs interpreters, TSOM interpreters and PySOM interpreters.

In Figure C.1, we conflate the results of fig. 4.1 and fig. 5.1 to show the performance of SOMrs and TSOM/PySOM, side by side. Comparing median run times, SOMrs-BC is first after Java, with a median slowdown of 2.81×, then followed by SOMrs-AST and a median slowdown of 3.18×. It is then followed by $TSOM_{AST}$ with 3.83×, then $PySOM_{AST}$ and $PySOM_{BC}$ with similar median slowdowns of 4.61× and 4.87× respectively, then finally $TSOM_{BC}$ with 6.10×.

These results show that SOMrs outperforms TSOM and PySOM, as expected from relying on a system-level language and a “pure” interpreter (see section 2.5). Though, it does not vastly outperform them, as it received less engineering effort overall than our meta-compilation-based interpreters.

TABLE C.2: SOMrs-BC Bytecode Set.

Bytecode	Description
Bytecodes and Variants Encoding Common Arguments	
`Dup`	duplicate top of stack
`Dup2`	duplicate second value from top of stack, only used when inlining `to:do:`
`PushLocal(u8)`	push local, with explicit index byte
`PushNonLocal(u8, u8)`	push non local, with explicit index byte and relative scope byte
`PushArg(u8)`	push argument, with explicit index byte
`PushNonLocalArg(u8, u8)`	push argument, with explicit index byte and relative scope byte
`PushField(u8)`	push field value, with explicit index byte
`PushBlock(u8)`	instantiate block, from index in literal array
`PushConstant(u8)`	push constant, from index in literal array
`PushGlobal(u8)`	push global, from index in literal array for name of global for lookup. Caches global association object
`Push0|Push1|PushNil`	push zero integer, one integer, or nil object
`Pop`	pop top of stack
`PopLocal(u8, u8)`	pop top of stack, store into local, with explicit index byte and relative scope byte
`PopArg(u8, u8)`	pop top of stack, store into argument, with explicit index byte and relative scope byte
`PopField(u8)`	pop top of stack, store into field
`[Send1|Send2|Send3|SendN](u16)`	lookup and activate method, based on number of arguments, and using inline cache if present
`SuperSend(u16)`	activate method in lexical superclass
`ReturnLocal`	return, use top of stack as result
`ReturnNonLocal(u8)`	return method that instantiate current block, unwind stack as necessary based on relative scope index
Superinstructions	
`ReturnSelf`	return, use `self/this` as result
`Inc`	increment the top-of-stack value by one
`Dec`	decrement the top-of-stack value by one
Control-flow Bytecodes for Block Inlining	
`Jump(u16)`	relative jump, with unsigned 2 byte offset
`JumpOnTrueTopNil(u16)`	jump if top of stack is `true` and set top to `nil`, otherwise pop
`JumpOnFalseTopNil(u16)`	jump if top of stack is `false` and set top to `nil`, otherwise pop
`JumpOnTruePop(u16)`	jump if top of stack is `true`, always pop top of stack
`JumpOnFalsePop(u16)`	jump if top of stack is `true`, always pop top of stack
`JumpBackward(u16)`	relative backwards jump, with unsigned 2 byte offset
`JumpOnNilTopTop(u16)`	jump if top of stack is `nil`, otherwise pop
`JumpOnNotNilTopTop(u16)`	jump if top of stack is not `nil`, otherwise pop
`JumpOnNilPop(u16)`	jump if top of stack is `nil`, always pop
`JumpOnNotNilPop(u16)`	jump if top of stack is not `nil`, always pop
`JumpIfGreater(u16)`	jump if top of stack is greater than second value from top of stack, pop both if so

Bibliography

Abelson, H. and G. J. Sussman (July 1996). *Structure and interpretation of computer programs*. second. MIT Press ; McGraw-Hill. ISBN: 0262011530. URL: http://mitpress.mit.edu/sicp/full-text/book/book.html.

Aronson, S. (2018). *shifgrethor*. URL: https://github.com/withoutboats/shifgrethor.

Aycock, J. (June 2003). "A Brief History of Just-In-Time". In: *ACM Comput. Surv.* 35.2, pp. 97–113. ISSN: 0360-0300. DOI: 10.1145/857076.857077.

Bacon, D. F., P. Cheng, and V. T. Rajan (2004). "A unified theory of garbage collection." In: *OOPSLA*. Ed. by J. M. Vlissides and D. C. Schmidt. ACM, pp. 50–68. ISBN: 1-58113-831-8. URL: http://dblp.uni-trier.de/db/conf/oopsla/oopsla2004p.html#BaconCR04.

Bala, V., E. Duesterwald, and S. Banerjia (May 2000). "Dynamo: A Transparent Dynamic Optimization System". In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. PLDI '00. ACM, pp. 1–12. ISBN: 1-58113-199-2. DOI: 10.1145/358438.349303.

Barrett, E., C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt (Oct. 2017). "Virtual Machine Warmup Blows Hot and Cold". In: *Proc. ACM Program. Lang.* 1.OOPSLA, 52:1–52:27. ISSN: 2475-1421. DOI: 10.1145/3133876.

Basso, M., D. Bonetta, and W. Binder (2023). "Automatically Generated Supernodes for AST Interpreters Improve Virtual-Machine Performance." In: *GPCE*. Ed. by C. D. Roover, B. Rumpe, and A. Shaikhha. ACM, pp. 1–13. URL: http://dblp.uni-trier.de/db/conf/gpce/gpce2023.html#BassoBB23.

Bell, J. R. (June 1973). "Threaded Code". In: *Communications of the ACM* 16.6, pp. 370–372. ISSN: 0001-0782. DOI: 10.1145/362248.362270.

Béra, C. and E. Miranda (Aug. 2016). "A bytecode set for adaptive optimizations". In: *Proceedings of the 8th Edition of the International Workshop on Smalltalk Technologies*.

IWST'16. Prague, Czech Republic. URL: https://rmod.inria.fr/archives/papers/Bera14a-IWST-BytecodeSet.pdf.

Blackburn, S. M., P. Cheng, and K. S. McKinley (2004). "Oil and Water? High Performance Garbage Collection in Java with MMTk". In: *ICSE*. Ed. by A. Finkelstein, J. Estublier, and D. S. Rosenblum. IEEE Computer Society, pp. 137–146. ISBN: 0-7695-2163-0. URL: http://dblp.uni-trier.de/db/conf/icse/icse2004.html#BlackburnCM04.

Blackburn, S. M. and K. S. McKinley (2008). "Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance". In: *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. Tucson, AZ, USA: ACM, pp. 22–32. ISBN: 978-1-59593-860-2. DOI: http://doi.acm.org/10.1145/1375581.1375586. URL: http://portal.acm.org/citation.cfm?id=1375581.1375586.

Bolz, C. F., A. Cuni, M. Fijalkowski, and A. Rigo (2009). "Tracing the Meta-level: PyPy's Tracing JIT Compiler". In: *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ICOOOLPS '09. ACM, pp. 18–25. ISBN: 978-1-60558-541-3. DOI: 10.1145/1565824.1565827.

Bolz, C. F., L. Diekmann, and L. Tratt (2013a). "Storage Strategies for Collections in Dynamically Typed Languages". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. ACM, pp. 167–182. ISBN: 978-1-4503-2374-1. DOI: 10.1145/2509136.2509531.

Bolz, C. F. and L. Tratt (2013b). "The Impact of Meta-Tracing on VM Design and Implementation". In: *Science of Computer Programming* 98, pp. 408–421. ISSN: 0167-6423. DOI: 10.1016/j.scico.2013.02.001.

Brunthaler, S. (Oct. 2010a). "Efficient Interpretation Using Quickening". In: *Proceedings of the 6th Symposium on Dynamic Languages*. DLS'10 12. ACM, pp. 1–14. DOI: 10.1145/1899661.1869633.

Brunthaler, S. (2010b). "Inline Caching Meets Quickening". In: *ECOOP 2010 – Object-Oriented Programming*. Ed. by T. D'Hondt. Vol. 6183. Springer, pp. 429–451. ISBN: 978-3-642-14107-2. DOI: 10.1007/978-3-642-14107-2_21.

Brunthaler, S. (Feb. 2011). “Purely Interpretative Optimizations”. PhD thesis. TU Wien.

Brunthaler, S. (2021). “Multi-Level Quickening: Ten Years Later”. In: *CoRR* abs/2109.02958. arXiv: 2109.02958. URL: https://arxiv.org/abs/2109.02958.

Bush, W. R., A. D. Samples, D. M. Ungar, and P. N. Hilfinger (1987). “Compiling Smalltalk-80 to a RISC.” In: *ASPLOS*. Ed. by R. H. Katz and M. Freeman. SIGARCH Computer Architecture News 15(5), SIGOPS Operating System Review 21(4), SIGPLAN Notices 22(10). ACM Press, pp. 112–116. ISBN: 0-8186-0805-6. URL: http://dblp.uni-trier.de/db/conf/asplos/asplos87.html#BushSUH87.

Bytecode DSL (Mar. 2025). Released alongside Truffle 24.2. https://github.com/oracle/graal/tree/master/truffle/src/com.oracle.truffle.dsl.processor/src/com/oracle/truffle/dsl/processor/bytecode. Oracle Corporation.

Casey, K., M. A. Ertl, and D. Gregg (2007). “Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters”. In: *ACM Trans. Program. Lang. Syst.* 29.6, p. 37. ISSN: 0164-0925. DOI: 10.1145/1286821.1286828.

Chaitin, G. J., M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein (1981). “Register allocation via coloring”. In: *Computer languages* 6.1, pp. 47–57.

Chambers, C., D. Ungar, and E. Lee (Oct. 1989). “An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes”. In: *Proceedings on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA. ACM, pp. 49–70. ISBN: 0-89791-333-7. DOI: 10.1145/74878.74884.

Collins, G. E. (1960). “A method for overlapping and erasure of lists”. In: *Communications of the ACM* 3.12, pp. 655–657.

D’Hondt, T. (2008). “Are Bytecodes an Atavism?” In: *Self-Sustaining Systems*. Ed. by R. Hirschfeld and K. Rose. Vol. 5146. Lecture Notes in Computer Science. Springer, pp. 140–155. ISBN: 978-3-540-89274-8. DOI: 10.1007/978-3-540-89275-5_8.

Dybvig, R. K. (1987). “Three implementation models for scheme”. UMI Order No. GAX87-22287. PhD thesis. USA.

Ertl, M. A. (1995). “Stack Caching for Interpreters”. In: *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*.

PLDI'95. ACM, pp. 315–327. DOI: 10.1145/207110.207165. URL: http://www.complang.tuwien.ac.at/papers/ertl95pldi.ps.gz.

Ertl, M. A. and D. Gregg (Nov. 2003). "The Structure and Performance of Efficient Interpreters". In: *Journal of Instruction-Level Parallelism*. J of ILP'03 5, pp. 1–25. URL: http://www.jilp.org/vol5/v5paper12.pdf.

Felleisen, M. and D. P. Friedman (1987). "Control operators, the SECD-machine, and the lambda-calculus." In: *Formal Description of Programming Concepts*. Ed. by M. Wirsing. North-Holland, pp. 193–222. ISBN: 0-444-70253-9. URL: http://dblp.uni-trier.de/db/conf/ifip2/fdpc1987.html#FelleisenF87.

Fenichel, R. R. and J. C. Yochelson (1969). "A LISP Garbage-Collector for Virtual-Memory Computer Systems". In: *Communications of the ACM* 12.11, pp. 611–612. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/363269.363280. URL: http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=363280.

Franz, M. and T. Kistler (Dec. 1997). "Slim binaries". In: *Commun. ACM* 40.12, pp. 87–94. ISSN: 0001-0782. DOI: 10.1145/265563.265576. URL: https://doi.org/10.1145/265563.265576.

Futamura, Y. (1999, Originally published in 1971). "Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler". In: *Higher-Order and Symbolic Computation* 12.4, pp. 381–391. ISSN: 1388-3690. DOI: 10.1023/A:1010095604496.

Gal, A., C. W. Probst, and M. Franz (2006). "HotpathVM: An Effective JIT Compiler for Resource-constrained Devices". In: *Proceedings of the 2nd International Conference on Virtual Execution Environments*. VEE'06. ACM, pp. 144–153. ISBN: 1-59593-332-6. DOI: 10.1145/1134760.1134780.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (Oct. 1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st. Addison-Wesley Professional. ISBN: 978-0201633610.

Goldberg, A. and D. Robson (1983). *Smalltalk-80: The Language and its Implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201113716.

Goregaokar, M. (2015). *rust-gc*. URL: https://github.com/Manishearth/rust-gc/.

Goregaokar, M. (2021). *Arenas in Rust*. URL: https://manishearth.github.io/blog/2021/03/15/arenas-in-rust/.

Grimmer, M., R. Schatz, C. Seaton, T. Würthinger, M. Luján, and H. Mössenböck (June 2018). "Cross-Language Interoperability in a Multi-Language Runtime". In: *ACM Transactions on Programming Languages and Systems* 40.2, pp. 1–43. DOI: 10.1145/3201898.

Haupt, M., R. Hirschfeld, T. Pape, G. Gabrysiak, S. Marr, A. Bergmann, A. Heise, M. Kleine, and R. Krahn (June 2010). "The SOM Family: Virtual Machines for Teaching and Research". In: *Proceedings of the 15th Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE'10. ACM, pp. 18–22. ISBN: 978-1-60558-729-5. DOI: 10.1145/1822090.1822098.

Hölzle, U., C. Chambers, and D. Ungar (1991). "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches". In: *ECOOP '91: European Conference on Object-Oriented Programming*. Vol. 512. LNCS. Springer, pp. 21–38. ISBN: 3-540-54262-0. DOI: 10.1007/BFb0057013.

Hughes, J. and L. Tratt (2025). "Garbage Collection for Rust: The Finalizer Frontier". In: *Proceedings of the ACM on Programming Languages* 9.OOPSLA2, pp. 3588–3614.

Humer, C. and N. Bebić (2022). *Operation DSL: How We Learned to Stop Worrying and Love Bytecodes again.* Truffle Workshop '22, Presentation. URL: https://2022.ecoop.org/details/truffle-2022/3/Operation-DSL-How-We-Learned-to-Stop-Worrying-and-Love-Bytecodes-again.

Humer, C., C. Wimmer, C. Wirth, A. Wöß, and T. Würthinger (2014a). "A Domain-Specific Language for Building Self-Optimizing AST Interpreters". In: *Proceedings of the 13th International Conference on Generative Programming: Concepts and Experiences*. GPCE '14. ACM, pp. 123–132. ISBN: 978-1-4503-3161-6. DOI: 10.1145/2658761.2658776.

Humer, C., C. Wimmer, C. Wirth, A. Wöß, and T. Würthinger (Sept. 2014b). "A domain-specific language for building self-optimizing AST interpreters". In: *SIGPLAN Not.* 50.3, pp. 123–132. ISSN: 0362-1340. DOI: 10.1145/2775053.2658776. URL: https://doi.org/10.1145/2775053.2658776.

Izawa, Y., H. Masuhara, and C. F. Bolz-Tereick (2025). "A Lightweight Method for Generating Multi-Tier JIT Compilation Virtual Machine in a Meta-Tracing Compiler Framework." In: *ECOOP*. Ed. by J. Aldrich and A. Silva. Vol. 333. LIPIcs. Schloss

Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:29. ISBN: 978-3-95977-373-7. URL: http://dblp.uni-trier.de/db/conf/ecoop/ecoop2025.html#IzawaMB25.

Izawa, Y., H. Masuhara, C. F. Bolz-Tereick, and Y. Cong (2022). "Threaded Code Generation with a Meta-Tracing JIT Compiler". In: *Journal of Object Technology* 21.2, 2:1–11. ISSN: 1660-1769. DOI: 10.5381/jot.2022.21.2.a1.

Kaleba, S., O. Larose, R. Jones, and S. Marr (Dec. 2022). "Who You Gonna Call: Analyzing the Run-time Call-Site Behavior of Ruby Applications". In: *Proceedings of the 18th Symposium on Dynamic Languages*. DLS'22. ACM, p. 14. DOI: 10.1145/3563834.3567538.

Kalibera, T., P. Maj, F. Morandat, and J. Vitek (Mar. 2014). "A Fast Abstract Syntax Tree Interpreter for R". In: *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE'14. ACM, pp. 89–102. ISBN: 978-1-4503-2764-0. DOI: 10.1145/2576195.2576205.

Klabnik, S. and C. Nichols (2023). *The Rust programming language*. No Starch Press.

Koparkar, C., M. Rainey, M. Vollmer, M. Kulkarni, and R. R. Newton (Aug. 2021). "Efficient Tree-Traversals: Reconciling Parallelism and Dense Data Representations". In: *Proceedings of the ACM on Programming Languages* 5.ICFP, pp. 1–29. DOI: 10.1145/3473596.

Körner, P., D. Schneider, and M. Leuschel (Sept. 2021). "On the Performance of Bytecode Interpreters in Prolog". In: *Functional and Constraint Logic Programming: 28th International Workshop, WFLP 2020*. Vol. 12560. WFLP'20. Springer, pp. 41–56. ISBN: 978-3-030-75333-7. DOI: 10.1007/978-3-030-75333-7_3.

Landin, P. J. (1964). "The Mechanical Evaluation of Expressions." In: *Comput. J.* 6.4, pp. 308–320. URL: http://dblp.uni-trier.de/db/journals/cj/cj6.html#Landin64.

Larose, O. (June 2025). "Adding garbage collection to our Rust-based interpreters with MMTk". In: *Workshop on Modern Language Runtimes, Ecosystems, and VMs*. MoreVMs'25.

Larose, O., S. Kaleba, H. Burchell, and S. Marr (Oct. 2023). "AST vs. Bytecode: Interpreters in the Age of Meta-Compilation". In: *Proceedings of the ACM on Programming Languages*. OOPSLA'23 7.OOPSLA2, pp. 318–346. ISSN: 2475-1421. DOI: 10.1145/3622808.

Larose, O., S. Kaleba, and S. Marr (Mar. 2022). *Less Is More: Merging AST Nodes To Optimize Interpreters*.

Larose, O., M. Vollmer, and S. Marr (2025). "AST, Bytecode, and the Space In Between: An Exploration of Interpreter Design Tradeoffs". In: *The Journal of Object Technology*. ICOOOLPS'25 24.3.

Leijen, D. (July 2022). *What About the Integer Numbers? Fast Arithmetic with Tagged Integers – A Plea for Hardware Support*. Tech. rep. MSR-TR-2022-17. Presented at ML language workshop 2022 (co-located with ICFP'22). Microsoft. URL: https://www.microsoft.com/en-us/research/publication/what-about-the-integer-numbers-fast-arithmetic-with-tagged-integers-a-plea-for-hardware-support/.

Lin, Y., S. M. Blackburn, A. L. Hosking, and M. Norrish (2016). "Rust as a language for high performance GC implementation." In: *ISMM*. Ed. by C. H. Flood and E. Z. Zhang. ACM, pp. 89–98. ISBN: 978-1-4503-4317-6. URL: http://dblp.uni-trier.de/db/conf/iwmm/ismm2016.html#LinBHN16.

Lindholm, T. and F. Yellin (Apr. 1999). *The Java Virtual Machine Specification*. 2nd ed. Addison-Wesley Longman, Amsterdam, p. 496. ISBN: 0201432943. URL: http://www.amazon.de/Java-Virtual-Machine-Specification-Addison-Wesley/dp/0201432943%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0201432943.

Marr, S. (Aug. 2023). *ReBench: Execute and Document Benchmarks Reproducibly.* Version 1.2 https://github.com/smarr/ReBench/. Open Source Software. DOI: 10.5281/zenodo.1311762.

Marr, S., B. Daloze, and H. Mössenböck (Nov. 2016). "Cross-Language Compiler Benchmarking—Are We Fast Yet?" In: *Proceedings of the 12th Symposium on Dynamic Languages*. DLS'16. ACM, pp. 120–131. ISBN: 978-1-4503-4445-6. DOI: 10.1145/2989225.2989232.

Marr, S. and S. Ducasse (Oct. 2015). "Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches for Self-Optimizing Interpreters". In: *Proceedings of the 2015 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '15. ACM, pp. 821–839. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2814270.2814275.

McCarthy, J. (1960). “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”. In: *Communications of the ACM* 3.4, pp. 184–195. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/367177.367199. URL: http://portal.acm.org/citation.cfm?id=367199.

Midtgaard, J., N. Ramsey, and B. Larsen (Sept. 2013). “Engineering Definitional Interpreters”. In: *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*. PPDP ’13. Madrid, Spain: ACM, pp. 121–132. ISBN: 978-1-4503-2154-9. DOI: 10.1145/2505879.2505894.

Niephaus, F., T. Felgentreff, and R. Hirschfeld (July 2018). “GraalSqueak: A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework”. In: *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS’18. ACM, pp. 30–35. DOI: 10.1145/3242947.3242948.

Nystrom, R. (July 2021). *Crafting Interpreters*. Genever Benning, p. 639. ISBN: 978-0990582939. URL: https://craftinginterpreters.com/.

Ottoni, G. and B. Liu (Mar. 2021). “HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, pp. 340–350. DOI: 10.1109/CGO51591.2021.9370314. URL: https://doi.ieeecomputersociety.org/10.1109/CGO51591.2021.9370314.

Pape, T. (2021). “Efficient Compound Values in Virtual Machines”. Doctoral Thesis. Universität Potsdam, p. 242. DOI: 10.25932/publishup-49913.

Pape, T., C. Bolz, and R. Hirschfeld (2017). “Adaptive Just-in-time Value Class Optimization for Lowering Memory Consumption and Improving Execution Time Performance”. In: *Science of Computer Programming* 140. Object-Oriented Programming and Systems (OOPS 2015), pp. 17–29. ISSN: 0167-6423. DOI: 10.1016/j.scico.2016.08.003.

Pape, T., T. Felgentreff, R. Hirschfeld, A. Gulenko, and C. F. Bolz (2015). “Language-independent Storage Strategies for tracing-JIT-based Virtual Machines”. In: *Proceedings of the 11th Symposium on Dynamic Languages*. DLS 2015. ACM, pp. 104–113. ISBN: 978-1-4503-3690-1. DOI: 10.1145/2816707.2816716.

Pichler, C., B. Urban-Forster, P. Li, R. Schatz, and H. Mössenböck (2025). "Fast and Compact: Reducing Size of AOT-Compiled Java Code without Sacrificing Performance". In: *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. MPLR '25. Singapore, Singapore: Association for Computing Machinery, pp. 12–22. ISBN: 9798400721496. DOI: 10.1145/3759426.3760976. URL: https://doi.org/10.1145/3759426.3760976.

Piumarta, I. and F. Riccardi (1998). "Optimizing Direct-threaded Code by Selective Inlining". In: *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation - PLDI '98*. ACM, pp. 291–300. DOI: 10.1145/277650.277743.

Proebsting, T. A. (1995). "Optimizing an ANSI C interpreter with superoperators". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*. ACM Press, pp. 322–332. DOI: 10.1145/199448.199526.

Qunaibit, M., S. Brunthaler, Y. Na, S. Volckaert, and M. Franz (2018). "Accelerating Dynamically-Typed Languages on Heterogeneous Platforms Using Guards Optimization". In: *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Vol. 109. ECOOP'18. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 16:1–16:29. ISBN: 978-3-95977-079-8. DOI: 10.4230/LIPIcs.ECOOP.2018.16. URL: http://drops.dagstuhl.de/opus/volltexte/2018/9221.

Richards, M. (1969). "BCPL: a tool for compiler writing and system programming." In: *AFIPS Spring Joint Computing Conference*. Ed. by H. W. Fuller. Vol. 34. AFIPS Conference Proceedings. AFIPS Press, pp. 557–566. ISBN: 978-1-4503-7902-1. URL: http://dblp.uni-trier.de/db/conf/afips/afips69s.html#Richards69.

Rigger, M., M. Grimmer, C. Wimmer, T. Würthinger, and H. Mössenböck (2016). "Bringing low-level languages to the JVM: Efficient execution of LLVM IR on Truffle". In: *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*, pp. 6–15.

Roberts, R., S. Marr, M. Homer, and J. Noble (July 2019). "Transient Typechecks are (Almost) Free". In: *33rd European Conference on Object-Oriented Programming*. Vol. 134. ECOOP'19 5. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 5:1–5:28. ISBN: 978-3-95977-111-5. DOI: 10.4230/LIPIcs.ECOOP.2019.5.

Rohou, E., B. N. Swamy, and A. Seznec (2015). "Branch Prediction and the Performance of Interpreters: Don't Trust Folklore". In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '15. San Francisco, California: IEEE Computer Society, pp. 103–114. ISBN: 978-1-4799-8161-8. URL: https://hal.inria.fr/hal-00911146/document.

WebAssembly Core Specification (Apr. 19, 2022). Version 2.0. W3C. URL: https://www.w3.org/TR/wasm-core-2/.

Sasada, K. (2005). "YARV: yet another RubyVM: innovating the ruby interpreter". In: *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA '05. San Diego, CA, USA: ACM, pp. 158–159. ISBN: 1-59593-193-7. DOI: 10.1145/1094855.1094912. URL: http://doi.acm.org/10.1145/1094855.1094912.

Seaton, C. (Oct. 2016). "AST specialisation and partial evaluation for easy high-performance metaprogramming". In: *Workshop on Meta-Programming Techniques and Reflection*. URL: https://chrisseaton.com/rubytruffle/meta16/meta16-ruby.pdf.

Shi, Y., K. Casey, M. A. Ertl, and D. Gregg (Jan. 2008). "Virtual Machine Showdown: Stack Versus Registers". In: *ACM Transactions on Architecture and Code Optimization*. TACO'08 4.4, pp. 1–36. ISSN: 1544-3566. DOI: 10.1145/1328195.1328197. URL: https://doi.org/10.1145/1328195.1328197.

shredder (2019). URL: https://github.com/Others/shredder.

Sussman, G. J. and G. L. Steele Jr (Dec. 1975). *SCHEME: An Interpreter for Extended Lambda Calculus*. Tech. rep. AIM-349. Massachusetts Institute of Technology.

Titzer, B. L. (Oct. 2022). "A Fast In-Place Interpreter for WebAssembly". In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2, pp. 646–672. DOI: 10.1145/3563311.

Van de Vanter, M. L., C. Seaton, M. Haupt, C. Humer, and T. Würthinger (Mar. 2018). "Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools". In: *Programming Journal* 2.3, p. 30. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2018/2/14.

Vollmer, M., S. Spall, B. Chamith, L. Sakka, C. Koparkar, M. Kulkarni, S. Tobin-Hochstadt, and R. R. Newton (2017). "Compiling Tree Transforms to Operate on

Packed Representations". In: *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Vol. 74. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 26:1–26:29. ISBN: 978-3-95977-035-4. DOI: 10.4230/LIPIcs.ECOOP.2017.26.

West, C. (2024). *gc-arena*. URL: https://github.com/kyren/gc-arena.

Wimmer, C., C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger (Oct. 2019). "Initialize Once, Start Fast: Application Initialization at Build Time". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA, pp. 1–29. DOI: 10.1145/3360610.

Wöß, A., C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck (2014). "An Object Storage Model for the Truffle Language Implementation Framework". In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '14. ACM, pp. 133–144. ISBN: 978-1-4503-2926-2. DOI: 10.1145/2647508.2647517.

Würthinger, T., C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer (2017). "Practical Partial Evaluation for High-performance Dynamic Language Runtimes". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI'17. ACM, pp. 662–676. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062381.

Würthinger, T., C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. 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*. Onward!'13. ACM, pp. 187–204. ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2509581.

Würthinger, T., A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer (Oct. 2012). "Self-Optimizing AST Interpreters". In: *Proceedings of the 8th Dynamic Languages Symposium*. DLS'12. ACM, pp. 73–82. ISBN: 978-1-4503-1564-7. DOI: 10.1145/2384577.2384587.

Zhang, Q., L. Xu, and B. Xu (Dec. 2022). "RegCPython: A Register-based Python Interpreter for Better Performance". In: *ACM Transactions on Architecture and Code Optimization* 20.1, pp. 1–25. DOI: 10.1145/3568973.

Zhang, W., P. Larsen, S. Brunthaler, and M. Franz (2014). “Accelerating Iterators in Optimizing AST Interpreters”. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA ’14. Portland, Oregon, USA: ACM, pp. 727–743. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660223.

Zhao, W., S. M. Blackburn, and K. S. McKinley (June 2022). “Low-Latency, High-Throughput Garbage Collection”. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI’22. San Diego, CA, USA: ACM, pp. 76–91. ISBN: 9781450392655. DOI: 10.1145/3519939.3523440.