# AST, Bytecode, and the Space In Between: An Exploration of Interpreter Design Tradeoffs

**Octave Larose, Michael Vollmer, and Stefan Marr**
University of Kent, Canterbury, United Kingdom

**ABSTRACT** Programming language interpreters usually interpret either an *abstract syntax tree* (AST) or *bytecode* (BC). An AST can be a tree of objects, which may be slow to interpret, while bytecode is a compact encoding that can give better run-time performance. However, defining the key characteristics of both interpreter designs is not straightforward. If the AST is encoded using an array instead of a pointer-based structure, is it still an AST interpreter? If bytecodes are represented as objects instead of compact bytes, is it still a bytecode interpreter?

In this paper, we explore the different design dimensions for the implementation of interpreters and discuss their tradeoffs and how they relate to AST and bytecode-based designs. They create a spectrum from *benefiting from the host language* to *minimizing distance with the target machine*. From the discussion, we derive guidelines for interpreter designs that enable implementers to navigate the tradeoff space between performance, engineering, tooling support, and language needs.

Finally, we discuss common optimizations that are independent of the program representation to demonstrate the performance of an optimized AST interpreter and bytecode interpreter implemented in Rust. While the bytecode interpreter is slightly faster, the difference is small, and we find the discussed optimizations to be more relevant for performance than the program representation, which we argue gives implementers more freedom in choosing a point in the design than generally assumed.

**KEYWORDS** Interpreters, Bytecode, Abstract-Syntax-Tree, Language Implementation, Comparison

## 1. Introduction

Interpreters are a lightweight and efficient way to implement a programming language, which many well-established languages, such as Bash, Java, JavaScript, Lua, PHP, Python, R, Ruby, and SQL, rely on. There are typically two approaches to interpreter implementation, which are identified by the used program representation: *abstract syntax tree* (AST) or *bytecode* (BC). An AST is a tree structure that roughly represents the syntax of the source code, mapping constructs such as control-flow statements, method calls, and arithmetic operations to a node of the corresponding type. In contrast, bytecode is a compact linearized representation, typically a sequence of instructions for a register or stack machine.

AST-based interpreters are often considered to be easier to implement at the cost of limited run-time performance and larger memory overhead. Since optimizing performance and minimizing memory use are often priorities for widely used language implementations, bytecode interpreters are commonly considered to be the most efficient design, with the majority of highly optimized interpreter implementations using bytecode rather than a tree structure.

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. We posit that AST and BC are not separate designs, but rather the ends of a spectrum.

**JOT reference format:**
Octave Larose, Michael Vollmer, and Stefan Marr. *AST, Bytecode, and the Space In Between: An Exploration of Interpreter Design Tradeoffs*. Journal of Object Technology. Vol. 25, No. 1, 2026. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2026.25.1.a15

In this paper, we give a novel characterization of interpreter designs as AST-like or BC-like when they lean more towards one end of the spectrum. We highlight the tension by exploring designs that are close to the midpoint of the spectrum. For this discussion, we assume the desire to implement a complex practice-oriented language, possibly using a system programming language. Based on this exploration of interpreter designs, we propose recommendations for interpreter designs considering different possible priorities of language implementers.

Even though performance often becomes the main design driver, we believe implementers have more freedom to recognize other tradeoffs such as engineering cost, tooling support, and language needs than generally assumed. To illustrate this, we compare the run-time performance of both an AST and a BC interpreter written in Rust, a system-level language that allows for direct memory access and fine-grained control over the execution. Both interpreters implement the same run-time optimizations, making their main difference their reliance on either AST or BC. With these interpreters reaching the performance of CPython 3.10, we find optimizations to be more important than the program representation. While the bytecode interpreter is indeed faster, the difference is small.

The contributions of this paper are as follows:

– a novel characterization of interpreter designs based on a spectrum from AST-like to BC-like, recognizing the benefits of either utilizing the host language, or moving closer to the target machine
– a set of recommendations for interpreter design based on these observations
– a comparison of the run-time performance of one optimized AST interpreter and one optimized bytecode interpreter, both written in Rust, to highlight the impact of program-representation-independent optimizations

## 2. Background

This section discusses common techniques for interpreters and AST and bytecode-based designs. Generally, we focus on the physical design, as it is seen by the underlying platform, and often implemented using a system programming language. Work such as the Gibbon compiler (Vollmer et al. 2017) can increase the abstraction level and connect a logical tree representation to a physical linearized one. We will discuss such approaches where relevant.

### 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 compiling to an intermediate representation, execute the representation themselves at the software level. In the literature, AST interpreters are often the most simple implementations. In the words of Abelson et al. (1996, sec. 4.1.7), we consider here only interpreters that are analyzing evaluators, and thus, the syntactic analysis is only performed once.

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]

However, interpreters themselves offer several advantages:

– They are faster to implement and e.g. allow us to experiment with language features faster.
– Their code representation can be more compact than native code, which can reduce memory usage.
– Startup is fast, since no translation to machine code needs to take place.
– They do not require dynamic native code generation, which can be disallowed for security reasons, making JIT compilation limited or impossible to rely on in some contexts.

This makes interpreters 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.

### 2.2. 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 implementing a language. For example, the node type for a source code element can implement an operation to realize its run-time semantics.

ASTs are not necessarily naive, one-to-one mappings of the program syntax to a tree representation either. They can deviate from the source, for instance to represent high-level concepts based on more basic ones or 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).

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 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 paper.

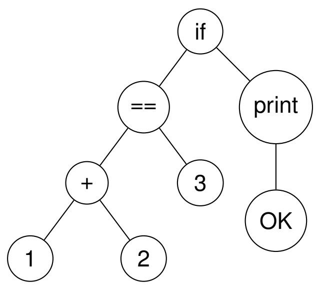
Figure 1a shows some simple code, and Figure 1b a simple AST built from this code. The root node `if` corresponds to the

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

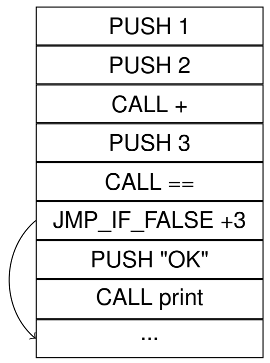
`if` statement, which has two children, one which represents the condition, and the other the body. Usually, execution is defined by a pre-order traversal of the tree: the root `if` node will be visited first, which will in turn check the condition by visiting the == node, which then visits the + node, 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 of 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.

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

**(a)** Source code of the example program



**(b)** As AST



```
PUSH 1
PUSH 2
CALL +
PUSH 3
CALL ==
JMP_IF_FALSE +3
PUSH "OK"
CALL print
...
```

**(c)** As Bytecode

**Figure 1** Example represented as AST and bytecode.

***2.2.1. 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 is useful for defining different sets of operations on the AST. One visitor can interpreter 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.

***2.2.2. Naturally-Recursive Execution*** An alternative implementation style in object-oriented languages foregoes the engineering benefits of the visitor pattern, possibly with some performance implications, and implements `execute` methods directly on the nodes. This is equivalent to Abelson et al. (1996)'s analyzing evaluators where the execution tree is composed from closures instead of objects. Figure 2a illustrates this interpreter variant. Here, the AST consists of an `AddNode` with two children, `Literal(1)` and `Literal(2)`. Execution still traverse 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. As mentioned above, in functional languages, this can be realized by using closures Abelson et al. (1996, sec. 4.1.7), which then ideally also benefit from tail-call optimizations in the host language.

This naturally-recursive style of interpreters has regained popularity with the GraalVM and Truffle projects (Würthinger et al. 2012, 2017), which started out by implementing a wide range of languages in this style on top of Java. The Truffle DSL (Humer et al. 2014a) further simplifies the implementation of interpreters in this style and makes it easier to benefit from meta-compilation in form of just-in-time compilation.

***2.2.3. Continuation-Passing Execution*** Another way to implement an AST interpreter is in a *continuation passing style* (CPS), a technique originating from Scheme (Reynolds 1972; Sussman & Steele Jr 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.

## 2.3. 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 a tree representation (an AST, though not meant for direct execution), then doing a pre-order traversal to linearize it into a sequence of instructions.

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 & Miranda (2016)

```rust
impl LiteralNode {
  fn new(value: Value) -> Self {
    Self { value }
  }
}

impl Node for LiteralNode {
  fn execute(&self, frame: &mut Frame) -> Value {
    self.value.clone()
  }
}

impl AddNode {
  fn new(child1: Node, child2: Node) -> Self {
    Self { child1, child2 }
  }
}

impl Node for AddNode {
  fn execute(&self, frame: &mut Frame) -> Value {
    self.child1.execute(frame) +
        self.child2.execute(frame)
  }
}

fn main() -> Value {
  tree = AddNode::new(Box::new(LiteralNode::new(1)),
                      Box::new(LiteralNode::new(2)));
  tree.execute(Frame())
}
```

**(a)** AST. The `Literal` nodes represent values and the `AddNode` adds the results of two subexpressions.

```rust
let bytecodes = [PUSH_LIT, 0, PUSH_LIT, 1, ADD];
let literals = [1, 2];
let mut stack = vec![];
let mut i = 0;

loop {
  bc = bytecodes[i];
  i += 1;

  match bc {
    PUSH_LIT => {
        let lit_idx = bytecodes[i];
        stack.push(literals[lit_idx]);
        i += 1;
    }
    ADD => {
      let (val2, val1) = (stack.pop(), stack.pop());
      stack.push(val1 + val2)
    }
    // ...
}
```

**(b)** BC. The bytecodes encode pushing literal values onto the stack, and then add them with an `ADD` instruction.

**Figure 2** Rust code, simplified slightly, for interpreters adding the integers 1 and 2, one AST-based and the other BC-based.

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.

Figure 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 "OK"` 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). Typically, register-based approaches are considered to be faster than stack-based approaches (Shi et al. 2008; Q. Zhang et al. 2022), because often they require fewer operations, which mitigates the bytecode dispatch overhead. Though, register-based bytecodes can be more complex, since they may require register allocation (Backus et al. 1957) when generating bytecode.

Figure 2b shows 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. Efforts to reduce the cost of dispatch led to optimizations such as threaded code (Bell 1973), which duplicates the dispatch to the end of each bytecode handler to help the CPU's branch predictor, or superinstructions (Casey et al. 2007), which combine several bytecodes into one to reduce the dispatch overhead. When implementing dynamic languages, one can also make bytecodes more efficient by using run-time information and quickening (Brunthaler 2010a) to replace more generic bytecodes with specialized counterparts during run time that can then be more efficient.

## 2.4. Abstract Machines

There is a long history of abstract machines (Landin 1964; Krivine 2007) used for a wide range of purposes, including static analysis and other forms of formal reasoning (Van Horn & Might 2010) or perhaps as foundation for direct implementation (Warren 1983). Abstract machines such as SECD and Krivine's are known to directly realize the lambda calculus (Ager et al. 2003). Since we focus on techniques to implement today's rather complex languages, we skip over these machines and will only occasionally refer to them where they make good examples for specific design points of the larger design spectrum.

## 3. 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. However, we find that none of these traits is a clear indication that an interpreter is strictly AST or BC, since we highlight counterexamples for each of them.

### 3.1. Assumptions About Performance

AST interpreters are often criticized for their performance. Section 2.2.1 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 (Bell 1973; Casey et al. 2007), 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 2.2.2. 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 an AST designs are often criticized for doing *pointer chasing*, which limits their performance.

Thus, the less compact structure of 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.[2] 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 3.3.

### 3.2. Recursive versus Iterative

The AST interpreters discussed in Section 2 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, W. 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 recursively walking the tree structure, 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 2.2.3 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 key characteristic of an AST-based design. We give examples of non-recursive AST designs in Section 4.2, and of recursive BC-based interpreter in Section 4.1 for instance on top of Truffle (Humer & Bebić 2022; Marr & Ducasse 2015).

### 3.3. 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 on one byte for the opcode, and zero or more bytes to encode additional arguments (Goldberg & Robson 1983; Lindholm & Yellin 1999). CPython uses two bytes per bytecode, one for the opcode and one for an optional argument.[3]

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 (Würthinger et al. 2012; Humer et al. 2014a). 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 metacompilation systems, we confirmed this in earlier work (Larose et al. 2023) and found 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 interpreters. However, an AST can also

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

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

be represented as a compact structure, which we explore in Section 4.3.

### 3.4. 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 Truffle DSL.

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 4.2, 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 & Steele Jr (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."

### 3.5. 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 n.d.). 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 revealing, since its vagueness highlights that the design space for the interpreter representation may be more broad than a binary choice between AST and BC.

## 4. 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 each characteristic to be categorized definitively as either belonging to AST or BC interpreter designs. This section discusses designs that lie in between AST and BC.

### 4.1. 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 design choice is uncommon in classic bytecode interpreters, in our experience it helps meta-compilation systems such as RPython (Bolz et al. 2009) and the GraalVM (Würthinger et al. 2017) to produce better just-in-time-compiled code (Marr & Ducasse 2015), by more explicitly separating frame objects, i.e., activation records. For example, the SOM bytecode interpreters (Marr & Ducasse 2015; Larose et al. 2023) 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, which both helped the meta-compilation systems.

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.

### 4.2. 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 efficiently represent more complex control flow, as seen in Section 3.4. 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 (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 2.2.3, 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 & Friedman 1987; Sussman & Steele Jr 1975).

As mentioned in Section 4.1, 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.

### 4.3. A Bytecode-like AST Interpreter Design

As noted in Section 3.3, 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.[4] 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 4.2), a recursive traversal of the tree (Section 3.2) is not necessary. With such an interpreter design, we rely on a flat and compact data structure (Section 3.3) and can also avoid recursion for function application, which would allow us to handle non-local control flow (Section 3.4) efficiently. At this point, we have all traits typically associated with bytecode instead of AST interpreters.

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

### 4.4. 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 Truffle DSL (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.

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.

### 4.5. 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 bytecode-based specifically 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.

## 5. 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 *AST-like* and *BC-like*.

### 5.1. AST-like and BC-like: Host Language and Target Machine

We find it useful to conceptualize AST and bytecode interpreters on a spectrum. On one end, we have a AST interpreter, recursive and relying on virtual dispatch, e.g., with a visitor-based or naturally-recursive design (Section 2.2.1, Section 2.2.2). 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 (Section 2.3).

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. Therefore, we define AST-like as *benefiting from the host language* and BC-like as *minimizing distance with the target machine*.

Based on Section 3, we conclude that AST interpreters are commonly assumed to rely heavily on host language features. 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 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 (Bell 1973), 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.

## 5.2. 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), such that complex control flow can be represented more efficiently than using host language mechanisms (Section 3.4). One can then apply a pre-order transformation to rely on a compact tree structure and offset-based references (Section 3.3) 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 3.5), 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 4.3 that this design narrows the difference between AST and BC.

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 2.2.3 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.

## 5.3. Advantages of AST-like

**Benefits of Utilizing the Host Language** Having defined AST-like to mean "benefiting from the host language", these interpreters tend to leverage the existing host language facilities, e.g., the host language call stack. This makes them easier to implement (Würthinger et al. 2012; Humer et al. 2014b; Kalibera et al. 2014; Basso et al. 2023), 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.

Moreover, BC-like designs work with a representation closer to that of machine code rather than source code, which can be 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 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, 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. 2013, 2017). 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[5] 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.

### 5.4. Advantages of BC-like

BC-like interpreters rely less on host language mechanisms. This can mean relying less on the host language stack, managing local control flow with jump offsets, or using threaded code. This distance with the host language implies greater engineering effort and lower maintainability.

Moving away from host-language mechanisms can also be beneficial when the guest language is limited by the host language. 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 and the guest-language stack is reified.

A more compact structure, e.g., a byte-based program representation, 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).[6]

## 6. Performance of AST and Bytecode

So far, we have discussed the design space between the classic AST and bytecode interpreter designs and highlighted that there are trade-offs. Performance is one of them, but there are tradeoffs in terms of engineering, tooling support, as well as matching the guest language semantics. In this section, we will however briefly look at the two extreme points and compare the performance of two optimized interpreters.

The goal here is to show that the choice of AST or bytecode is not as clear cut when it comes to performance, as often assumed. We demonstrate this by optimizing the interpreters with techniques that can be applied to both AST and BC-based interpreters to show that implementers have more freedom to choose a design based on other concerns than often assumed.

### 6.1. Our interpreters: SOMrs-AST and SOMrs-BC

We implemented two interpreters for the Simple Object Machine (SOM) (Haupt et al. 2010), a Smalltalk dialect. It is dynamically typed, has class inheritance, closures, and non-local returns. It shares many core features with languages such as Python and JavaScript, sharing key implementation challenges, but is less complex. Thus, implementing two interpreters is feasible with limited engineering resources.

[5] https://github.com/oracle/graal/blob/master/truffle/docs/bytecode_dsl/BytecodeDSL.md

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

Our first interpreter is a classic AST interpreter and the second is a classic BC interpreter. Thus, we stay close to the common AST and BC characteristics discussed in Section 3: our AST interpreter uses recursion to execute its pointer-based AST, and our bytecode interpreter uses a main dispatch loop to sequentially execute a compact, byte-based representation.

Both are part of the same codebase, share some core features, such as garbage collection (GC), and have received a similar amount of engineering effort. They are implemented in Rust, which we chose as it promises high performance while developers benefit from compile-time memory safety. Unfortunately, we had to forgo many of its safety guarantees for the sake of performance, e.g., to implement a garbage collector with MMTk (Lin et al. 2016).

Both interpreters use MMTk's semispace GC (Fenichel & Yochelson 1969) implementation. This has the benefit of being a relatively simple GC that allocates objects using a bump-pointer allocator, which makes allocation as fast and efficient as incrementing a pointer.

## 6.2. Implemented Optimizations

To see how the two end points of the spectrum behave, we focused on implementing optimizations that can be applied to both AST and bytecode interpreters. Thus, the following optimizations are present in both SOMrs-AST and SOMrs-BC, and are therefore orthogonal to the program representation. We selected these optimizations based on our previous experiences (Marr & Ducasse 2015; Larose et al. 2023), where we reported on the importance of different optimizations and their applicability to both interpreter designs.

**Inlining Control-Flow Structures and Lowering of Basic Operations** In SOM, everything is a method call, 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. Furthermore, we lower basic operations such as arithmetic operations, e.g. $+$, $-$, and $/$, or common string or array operations by implementing methods that are defined in SOM's core library with Rust in the interpreter.

**Direct Handling of Trivial Methods** Trivial methods, such as getters and setters, can avoid the call overhead to speedup execution. We detect such methods during parsing, implement them so that we can avoid allocating a frame object and directly perform the corresponding operation, e.g., reading the field of an object in case of a getter method.

**Monomorphic Inline Caching** To optimize the method dispatch, *inline caching* (Hölzle et al. 1991) 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, 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. 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 our own work (Larose et al. 2022, 2023), 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.

**NaN Boxing** We also implemented *NaN boxing* (Nystrom 2021), 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, including pointers, since they are using only 48 bits on 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`.

## 6.3. Results

**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. We use ReBench (Marr 2023) version 1.3 to configure the machine for benchmarking, reduce measurement noise, and to collect the results. We use the Are We Fast Yet benchmark suite (Marr et al. 2016). It includes five macro- and nine micro-benchmarks. It is designed for cross-language performance comparison and allows us to compare with CPython 3.10 and 3.13 to provide context with widely used bytecode interpreters. Thus, on CPython we run the Python versions of the benchmarks, and on som-rs the SOM versions. We run each benchmark 30 times for 1 iteration, and take the time of the iteration inside the harness.

**Run-time Performance** We compare SOMrs-AST and SOMrs-BC to two versions of CPython. CPython has received many optimizations in its past few releases, significantly improving performance, which makes version 3.13, the latest release, a good baseline. Figure 3 shows the performance of all systems. SOMrs-BC is second in terms of median run-time performance over all benchmarks at 1.75x. CPython 3.10 has a median run-time performance of 1.84x and SOMrs-AST is at 1.99x. Both som-rs interpreters have similar performance ranges, with 0.78x to 2.64x for SOMrs-BC, and 1.17x to 2.80x for SOMrs-AST. Thus, they are roughly similar in performance, but the AST interpreter is slightly slower.

We believe these results show that our som-rs 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. Arguably, it is a level of performance that can be reached with only modest effort by many custom language implementations, and thus, is a performance level representative for language implementations supported only by limited engineering resources.

While these results are not expected to generalize, they suggest that the choice between AST and bytecode may not be as

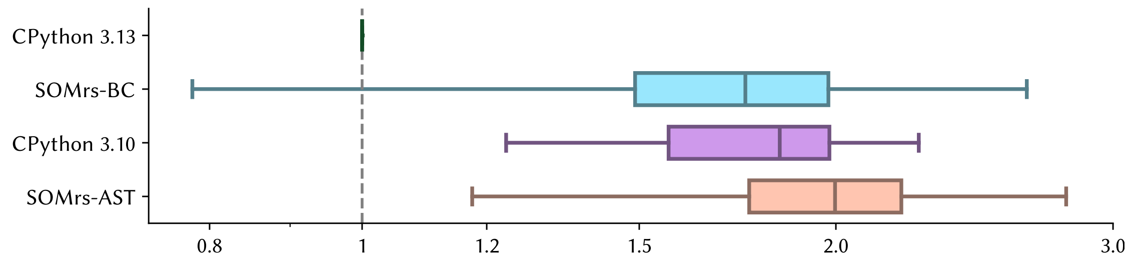


**Figure 3** Interpreter run-time performance of the Are We Fast Yet benchmarks, with CPython 3.13 as baseline, using a logarithmic x-axis. SOMrs-AST and SOMrs-BC are at a similar level, and are in the same ballpark as CPython 3.10.

important as careful optimizations. While further optimizations of the som-rs 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.

### 6.4. Future Optimizations Opportunities

Several optimizations could be implemented, making our interpreters go from optimized to highly optimized. Of these optimizations, many were originally conceived for BC interpreters, but we find that they can also apply to AST-like interpreters. This highlights how comparing highly optimized interpreters is trickier than it might first appear: we quickly run into the key issue that this work aims to address, that the distinction between AST and BC can be unclear or unhelpful.

An example of a classic BC interpreter optimization is threaded interpretation (Bell 1973), which SOMrs-BC could benefit from. We omitted it since it does not have a clear equivalent for a recursive AST interpreter. Though, when using a more compact AST representation, SOMrs-AST could possibly benefit from it, too.

Similarly, given that SOMrs-BC is a stack-based bytecode interpreter, it may also benefit from caching top-of-stack values (Ertl 1995). A classic AST interpreter could not easily mirror this optimization, since closeness with the host language often implies relying entirely on the host language stack. However, as described in Section 4.2, it is possible to design an AST interpreter with its own custom stack, thus enabling the implementation of a similar caching technique.

Top-of-stack caching would bring the design of SOMrs-BC closer to that of a register-based bytecode interpreter, which may have performance benefits (Shi et al. 2008; Q. Zhang et al. 2022). Although SOMrs-AST could also be designed to rely on a set of registers, it would be a further departure still from a classic AST design, i.e. a simpler design with a tree of nodes linked by pointers. This is also why we do not make the AST structure more compact, as described in Section 4.3, although it would likely give a performance benefit.

Finally, one relatively simple optimization missing from SOMrs-AST is relying on closures when interpreting the AST (Abelson et al. 1996, sec. 4.1.7), which may give a performance benefit at little cost. This could further narrow the performance gap between SOMrs-AST and SOMrs-BC with little effort, which would further attest to the performance of our AST interpreter.

## 7. Recommendations for Interpreter Design

Based on the observations in Section 5.3 and Section 5.4, we propose recommendations for interpreter design.

**Limited Engineering Resources: Utilize Host Language** 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. By leveraging the host language stack by adopting a recursive interpreter structure, and relying on host language objects for the interpreter representation, the need for custom tooling remains minimal and avoids an increasing maintenance burden.

**Good Performance: AST Interpreters can Remain an Option** If good run-time performance is important, interpreter optimizations are essential. In our experience, even AST interpreters can achieve it, while also benefiting from the host language. Specifically, a recursive AST interpreter benefits from many classic optimizations and can yield good performance.

**Best Performance: Bytecode Interpreters are Likely Preferable** If very good performance is important, one will likely need to use designs close to the target machine. A more optimized representation for the best possible cache locality, such as highly compact bytecode, is still likely to yield superior performance in most cases. However, reducing the distance with the target machine can significantly increase the engineering effort, since it requires tooling to be implemented and maintained.

**Minimizing Memory Use: Either Option Can Give A Compact Representation** If memory use is a main concern, e.g. if the interpreter is targeting resource-constrained systems, using a compact program representation is desirable. While bytecodes are a common choice, ASTs can also be represented in compact formats (see Section 4.3). This helps minimizing memory for programs. Though often, optimizing the representation of objects and other metadata is even more important and independent of the program representation (Ugawa et al. 2022).

**Mismatch Between Host and Guest Language Needs Consideration** A language with complex control-flow mechanisms may be hard to implement efficiently with a recursive interpreter design, as seen in Section 3.4. 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.

**Conclusion** When choosing a representation, and overall designing an interpreter, it is not a binary choice between AST and bytecode. Instead, there is a large space of design concerns that need to be considered. Often relying on the host language has engineering benefits but moving away from it may yield more opportunities for performance optimizations. Different implementation projects have different needs and different amounts of engineering resources available. Though, most projects are likely well positioned somewhere in the middle of our conceptualized design space, striving for good performance, but without the need to leverage the target machine as much as possible.

## 8. 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 other papers compare the run-time performance of AST and bytecode interpreters. In earlier work (Larose et al. 2023), we investigated the performance in the context of metacompilation systems where one implements an interpreter and the runtime system provides JIT compilation. Within the context of metacompilation, we found 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 papers 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. 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.

## 9. Conclusion

In this paper, 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 also showed the run-time performance of an AST interpreter and a BC interpreter implemented in Rust, which both received the same amount of engineering effort. In this context, we focused on interpreter optimizations that are independent of the chosen interpreter representation. We find that given a moderate amount of engineering effort, our BC interpreter outperforms our AST interpreter by only a small margin, highlighting that interpreter optimizations matter far more for run-time performance than the design choice of relying on an AST or BC structure. However, for both interpreters additional optimizations would be possible. Thus, the performance difference could potentially grow with further optimizations. Nonetheless, we believe that this shows that language implementers have more freedom than usually assumed to choose a suitable point on the design spectrum, making the right tradeoffs between performance, engineering effort, tooling support, and match with the implemented language.

Based on these observations and experiments, we have provided guidelines for interpreter design depending on the needs and means of programming language implementers. 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 tradeoffs. In the end, there is a fundamental push-and-pull better run-time performance and the cost of engineering, which is a classic software problem extending far beyond interpreters.

We hope this work contributes to the conversation on interpreter designs and implementation strategies and future work explores more points in the design space. After all, implementing interpreters can be complex, and as Titzer (2022) notes, "writing a highly efficient interpreter remains a black art."

### Acknowledgments

We thank Nicolas Polomack for his contributions to the som-rs implementations. This work was supported by a studentship from the Engineering and Physical Sciences Research Council (2619006) and a gift from Oracle.

## References

Abelson, H., Sussman, G. J., & with Julie Sussman. (1996). *Structure and Interpretation of Computer Programs* (second ed.). MIT Press; McGraw-Hill. Hardcover.

Ager, M. S., Biernacki, D., Danvy, O., & Midtgaard, J. (2003, 8 27). A Functional Correspondence between Evaluators and Abstract Machines. In *Proceedings of the 5th acm sigplan international conference on principles and practice of declaritive programming* (pp. 8–19). ACM. doi: 10.1145/888251.888254

Backus, J. W., Beeber, R. J., Best, S., Goldberg, R., Haibt, L. M., Herrick, H. L., … Nutt, R. (1957). The FORTRAN Automatic Coding System. In *Papers presented at the february 26-28, 1957, western joint computer conference: Techniques for reliability* (p. 188–198). ACM. doi: 10.1145/1455567.1455599

Basso, M., Bonetta, D., & Binder, W. (2023). Automatically generated supernodes for ast interpreters improve virtual-machine performance. In C. D. Roover, B. Rumpe, & A. Shaikhha (Eds.), *Proceedings of the 22nd acm sigplan international conference on generative programming: Concepts and experiences* (pp. 1–13). ACM. doi: 10.1145/3624007.3624050

Bell, J. R. (1973, June). Threaded Code. *Communications of the ACM*, *16*(6), 370–372. doi: 10.1145/362248.362270

Bolz, C. F., Cuni, A., Fijalkowski, M., & Rigo, A. (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* (pp. 18–25). ACM. doi: 10.1145/1565824.1565827

Brunthaler, S. (2010a, October). Efficient interpretation using quickening. In *Proceedings of the 6th symposium on dynamic languages* (pp. 1–14). ACM. doi: 10.1145/1899661.1869633

Brunthaler, S. (2010b). Inline caching meets quickening. In T. D'Hondt (Ed.), *Ecoop 2010 – object-oriented programming* (Vol. 6183, pp. 429–451). Springer. doi: 10.1007/978-3-642-14107-2_21

Bush, W. R., Samples, A. D., Ungar, D., & Hilfinger, P. N. (1987). Compiling Smalltalk-80 to a RISC. In *Proceedings of the second international conference on architectural support for programming languages and operating systems* (pp. 112–116). ACM. doi: 10.1145/36206.36192

Béra, C., & Miranda, E. (2016, August 23). A bytecode set for adaptive optimizations. In *Proceedings of the 8th edition of the international workshop on smalltalk technologies.* Prague, Czech Republic.

Casey, K., Ertl, M. A., & Gregg, D. (2007). Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters. *ACM Trans. Program. Lang. Syst.*, *29*(6), 37. doi: 10.1145/1286821.1286828

D'Hondt, T. (2008). Are Bytecodes an Atavism? In R. Hirschfeld & K. Rose (Eds.), *Self-sustaining systems* (Vol. 5146, pp. 140–155). Springer. doi: 10.1007/978-3-540-89275-5_8

Dybvig, R. K. (1987). *Three implementation models for scheme* (Unpublished doctoral dissertation). USA. (UMI Order No. GAX87-22287)

Ertl, M. A. (1995). Stack caching for interpreters. In *Proceedings of the acm sigplan 1995 conference on programming language design and implementation* (pp. 315–327). ACM. doi: 10.1145/207110.207165

Felleisen, M., & Friedman, D. P. (1987). Control operators, the secd-machine, and the λ-calculus. In M. Wirsing (Ed.), *Formal description of programming concepts* (p. 193-222). North-Holland.

Fenichel, R. R., & Yochelson, J. C. (1969). A lisp garbage-collector for virtual-memory computer systems. *Communications of the ACM*, *12*(11), 611–612.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (1st ed.). Addison-Wesley Professional.

Goldberg, A., & Robson, D. (1983). *Smalltalk-80: The language and its implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Haupt, M., Hirschfeld, R., Pape, T., Gabrysiak, G., Marr, S., Bergmann, A., … Krahn, R. (2010, June 26–30). The som family: Virtual machines for teaching and research. In *Proceedings of the 15th annual conference on innovation and technology in computer science education* (pp. 18–22). ACM. doi: 10.1145/1822090.1822098

Humer, C., & Bebić, N. (2022). *Operation dsl: How we learned to stop worrying and love bytecodes again.* Retrieved from https://2022.ecoop.org/details/truffle-2022/3/Operation-DSL-How-We-Learned-to-Stop-Worrying-and-Love-Bytecodes-again (Truffle Workshop '22, Presentation)

Humer, C., Wimmer, C., Wirth, C., Wöß, A., & Würthinger, T. (2014a). A Domain-Specific Language for Building Self-Optimizing AST Interpreters. In *Proceedings of the 13th international conference on generative programming: Concepts and experiences* (pp. 123–132). ACM. doi: 10.1145/2658761.2658776

Humer, C., Wimmer, C., Wirth, C., Wöß, A., & Würthinger, T. (2014b, September). A domain-specific language for building self-optimizing ast interpreters. *SIGPLAN Not.*, *50*(3), 123–132. doi: 10.1145/2775053.2658776

Hölzle, U., Chambers, C., & Ungar, D. (1991). Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Ecoop '91: European conference on object-oriented programming* (Vol. 512, pp. 21–38). Springer. doi: 10.1007/BFb0057013

Kaleba, S., Larose, O., Jones, R., & Marr, S. (2022, December 7). Who You Gonna Call: Analyzing the Run-time Call-Site Behavior of Ruby Applications. In *Proceedings of the 18th symposium on dynamic languages* (p. 14). ACM. doi: 10.1145/3563834.3567538

Kalibera, T., Maj, P., Morandat, F., & Vitek, J. (2014, March). A Fast Abstract Syntax Tree Interpreter for R. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (pp. 89–102). ACM. doi: 10.1145/2576195.2576205

Körner, P., Schneider, D., & Leuschel, M. (2021, September).

On the Performance of Bytecode Interpreters in Prolog. In *Functional and constraint logic programming: 28th international workshop, wflp 2020* (Vol. 12560, pp. 41–56). Springer. doi: 10.1007/978-3-030-75333-7_3

Krivine, J.-L. (2007, September 1). A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, *20*(3), 199–207. doi: 10.1007/s10990-007-9018-9

Landin, P. J. (1964, January 1). The Mechanical Evaluation of Expressions. *The Computer Journal*, *6*(4), 308–320. doi: 10.1093/COMJNL/6.4.308

Larose, O., Kaleba, S., Burchell, H., & Marr, S. (2023, October). Ast vs. bytecode: Interpreters in the age of meta-compilation. *Proceedings of the ACM on Programming Languages*, *7*(OOPSLA2), 318–346. doi: 10.1145/3622808

Larose, O., Kaleba, S., & Marr, S. (2022, March). *Less Is More: Merging AST Nodes To Optimize Interpreters.*

Lin, Y., Blackburn, S. M., Hosking, A. L., & Norrish, M. (2016). Rust as a Language for High Performance GC Implementation. In *Proceedings of the 2016 acm sigplan international symposium on memory management* (pp. 89–98). ACM. doi: 10.1145/2926697.2926707

Lindholm, T., & Yellin, F. (1999). *The java virtual machine specification* (2nd ed.). Addison-Wesley Longman, Amsterdam.

Marr, S. (2023, August). *ReBench: Execute and Document Benchmarks Reproducibly.* GitHub. (Version 1.2 https://github.com/smarr/ReBench/) doi: 10.5281/zenodo.1311762

Marr, S., Daloze, B., & Mössenböck, H. (2016, November 1). Cross-Language Compiler Benchmarking—Are We Fast Yet? In *Proceedings of the 12th symposium on dynamic languages* (pp. 120–131). ACM. doi: 10.1145/2989225.2989232

Marr, S., & Ducasse, S. (2015, October). 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* (pp. 821–839). ACM. doi: 10.1145/2814270.2814275

Midtgaard, J., Ramsey, N., & Larsen, B. (2013, 9 16). Engineering Definitional Interpreters. In *Proceedings of the 15th symposium on principles and practice of declarative programming* (pp. 121–132). ACM. doi: 10.1145/2505879.2505894

Niephaus, F., Felgentreff, T., & Hirschfeld, R. (2018, July). 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* (pp. 30–35). ACM. doi: 10.1145/3242947.3242948

Nystrom, R. (2021). *Crafting Interpreters*. Genever Benning. Retrieved from https://craftinginterpreters.com/

Reynolds, J. C. (1972, 8 1). Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the acm annual conference - volume 2* (pp. 717–740). ACM. doi: 10.1145/800194.805852

Richards, M. (1969). Bcpl: a tool for compiler writing and system programming. In *Proceedings of the may 14-16, 1969, spring joint computer conference* (Vol. 34, pp. 557–566). ACM. doi: 10.1145/1476793.1476880

Rigger, M., Grimmer, M., Wimmer, C., Würthinger, T., & Mössenböck, H. (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., Marr, S., Homer, M., & Noble, J. (2019, July 15). Transient typechecks are (almost) free. In *33rd european conference on object-oriented programming* (Vol. 134, pp. 5:1–5:28). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.ECOOP.2019.5

Rossberg, A. (Ed.). (n.d.). *WebAssembly Core Specification.* Retrieved from 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* (pp. 158–159). ACM. doi: 10.1145/1094855.1094912

Seaton, C. (2016, 10). Ast specialisation and partial evaluation for easy high-performance metaprogramming. In *Workshop on meta-programming techniques and reflection.* Retrieved from https://chrisseaton.com/rubytruffle/meta16/meta16-ruby.pdf

Shi, Y., Casey, K., Ertl, M. A., & Gregg, D. (2008, January). Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization*, *4*(4), 1–36. doi: 10.1145/1328195.1328197

Sussman, G. J., & Steele Jr, G. L. (1975, December). *Scheme: An interpreter for extended lambda calculus* (Tech. Rep. No. AIM-349). Massachusetts Institute of Technology.

Titzer, B. L. (2022, October). A Fast In-Place Interpreter for WebAssembly. *Proceedings of the ACM on Programming Languages*, *6*(OOPSLA2), 646–672. doi: 10.1145/3563311

Ugawa, T., Marr, S., & Jones, R. (2022, December 5). Profile guided offline optimization of hidden class graphs for javascript vms in embedded systems. In *Proceedings of the 14th acm sigplan international workshop on virtual machines and intermediate languages* (p. 11). ACM. doi: 10.1145/3563838.3567678

Van Horn, D., & Might, M. (2010, 9 27). Abstracting Abstract Machines. In *Proceedings of the 15th acm sigplan international conference on functional programming* (pp. 51–62). Association for Computing Machinery. doi: 10.1145/1863543.1863553

Vollmer, M., Spall, S., Chamith, B., Sakka, L., Koparkar, C., Kulkarni, M., … Newton, R. R. (2017). Compiling Tree Transforms to Operate on Packed Representations. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)* (Vol. 74, pp. 26:1–26:29). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.ECOOP.2017.26

Warren, D. H. (1983, October). *An Abstract Prolog Instruction Set* (Tech. Rep. No. Technical Note 309). Menlo Park, CA, USA: SRI International, Artificial Intelligence Center, Computer Science and Technology Division. Retrieved from https://www.sri.com/wp-content/uploads/2021/12/641.pdf

Würthinger, T., Wimmer, C., Humer, C., Wöß, A., Stadler, L., Seaton, C., … Grimmer, M. (2017). Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th acm sigplan conference on programming language design and implementation* (pp. 662–676). ACM. doi: 10.1145/3062341.3062381

Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., … Wolczko, M. (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* (pp. 187–204). ACM. doi: 10.1145/2509578.2509581

Würthinger, T., Wöß, A., Stadler, L., Duboscq, G., Simon, D., & Wimmer, C. (2012, October). Self-optimizing ast interpreters. In *Proceedings of the 8th dynamic languages symposium* (pp. 73–82). ACM. doi: 10.1145/2384577.2384587

Zhang, Q., Xu, L., & Xu, B. (2022, December). RegCPython: A Register-based Python Interpreter for Better Performance. *ACM Transactions on Architecture and Code Optimization*, *20*(1), 1–25. doi: 10.1145/3568973

Zhang, W., Larsen, P., Brunthaler, S., & Franz, M. (2014). Accelerating Iterators in Optimizing AST Interpreters. In *Proceedings of the 2014 acm international conference on object oriented programming systems languages & applications* (pp. 727–743). ACM. doi: 10.1145/2660193.2660223