Torrent: A Rust-based Emacs Lisp Compiler/Runtime

dwuggh

September 29, 2025

Contents

First, I want to say I am not a professional computer science researcher or programmer, and I did not major in CS. I may make fundamental, conceptual mistakes because of ignorance—please feel free to correct me when I am wrong.

Second, heavily inspired by Rune, a fantastic project, I consider **Torrent** the "anaphoric counterpart" to Rune—that is, it intentionally adopts technologies that Rune does not use. I do not want the design to be similar; that would defeat the purpose. This project aims to explore other possibilities for a Rust-based Emacs Lisp compiler until we have a usable one—or at least until I can use Lisp to write configs for other systems (for example, niri).

1 Compiler Architecture

1.1 Background

Many dynamic languages compile to bytecode and interpret it in a bytecode VM. The most famous example is Java and the JVM. Bytecode provides excellent cross-platform compatibility and fast startup. Sometimes that is not fast enough, and we want the ability to compile code to native machine instructions, usually by translating bytecode to platform-specific ISAs. However, dynamic languages face challenges:

- symbols, functions, and variables may change at runtime, so we cannot always generate stable calls;
- variable types are often unknown in advance, which blocks many instruction-level optimizations;
- …

These problems arise from language flexibility. If we can "pin" aspects of the runtime, we can provide type information to the compiler and generate faster code. This is the idea of **Just-In-Time** (JIT) compilation. Fast JITs often perform type specialization, function inlining, and other techniques to make hot paths fast. We also need a way to invalidate compiled code when assumptions no longer hold; guard checks support this **optimization and de-optimization** cycle.

Highly optimized runtimes often add an extra tier—a more aggressive JIT that spends more time compiling to get higher throughput. This **tiered JIT** approach is used by Google's V8 and Mozilla's SpiderMonkey.

There are also attempts to make AOT compilation fast [1], but they require substantial static analysis.

1.2 Bytecode VM

There are two common VM styles: **register VMs** and **stack VMs**. Stack VMs are most common; the JVM and CPython are examples. A stack VM retrieves operands from a runtime stack. A register VM, like a physical CPU, exposes a set of virtual

registers and operates on them (but still uses a stack for calls). Lua 5.0 switched from a stack VM to a register VM [2] MoarVM. The opcode sets differ, but translation between them is feasible. Research indicates that although stack VMs have denser bytecode, register VMs often execute fewer instructions and run faster [3].

Emacs Lisp uses a stack VM, and Rune provides a Rust implementation. In Torrent, I might add a register VM later. For now, the plan is to use a baseline JIT or AOT compiler instead of a bytecode interpreter. There is an old but interesting blog post arguing for this direction [4]. However, few languages take this approach. As far as I know, Julia does not have a bytecode interpreter; it emits LLVM bitcode. In my brief experience (writing a surface-code simulator), Julia's compilation time was nearly unbearable, and at the time it lacked several language features (e.g., enums) and had buggy reloading. I eventually rewrote the project in Rust.

For Emacs Lisp, this approach still seems promising. Today most code ends up native-compiled and users appear happy with that. Technically, most Emacs code does not change once loaded; only a small subset changes—for example, when modes advise functions, or when users debug Elisp, write configs, or develop packages. Most of the time, Emacs is used as an editor—a **tool** for editing files—so compile speed is less critical here.

For code generation I use Cranelift instead of LLVM. Cranelift's single-IR architecture yields excellent compile speed and sufficiently fast execution (around 15% slower than LLVM). I am therefore not concerned about compile time.

Rather than bytecode, I use a high-level IR (HIR) to represent Elisp. It encodes special forms directly and performs simple lexical-binding and closure analyses. I plan to adopt further optimizations following [1].

An intriguing option is to serialize HIR as Lisp itself—i.e., use Elisp as the IR. This might or might not be useful. ChezScheme may follow a similar idea: it performs many optimizations in Scheme; essentially a Scheme compiler written in Scheme that achieves great performance. I have not investigated this deeply yet, but it is certainly a project worth learning from.

To summarize, the current big picture is:

Emacs Lisp Source → HIR → Baseline JIT → Optimizing JIT

Only the baseline JIT has a naive implementation today.

2 GC Design

GC theory is vast; I can only summarize what I have read. My main reference is the GC handbook [5].

2.1 Tracing GC vs Reference Counting GC

Rune's design document claims reference counting is slow; that is not necessarily true. The two paradigms can be unified in theory [6]. In short, tracing GC finds **live** objects, while RC identifies **dead** ones. In practice, naive RC has pitfalls, but they can be addressed [5]. The main challenges are:

1. cycle detection, especially in concurrent settings;
2. write-barrier overhead on the mutator.

Cycle collection is addressed by [7] and follow-ups (see [5]). Barrier costs can be mitigated through **deferred reference counting** and **coalesced reference counting**; the latter is particularly powerful. We also have LXR [8], a recent high-performance RC GC.

2.2 Rust implementations

scheme-rs implements the Bacon–Rajan algorithm, which I initially copied. However, it does not adopt coalesced RC and uses Tokio channels to send mutation buffers immediately, adding overhead. On Fibonacci benchmarks, scheme-rs performs poorly. I could not run flamegraph or perf with Tokio enabled, so I lack data, but I suspect the GC is the major factor.

Recently, a BFS-based refinement of Bacon–Rajan was proposed [9], with a Rust implementation. I have not read the paper thoroughly yet.

[MMTK](https://mmtk.io)[MMTK) is a growing GC SDK in Rust. LXR [8] is implemented on a separate MMTK branch; there are many divergent commits, so merging will take time. MMTK already provides a solid platform; my long-term goal is to integrate with it.

3 Lisp Object Representation

3.1 Objects are tagged pointers

I follow Rune's approach: a shifting tagged pointer. Since most Elisp values are pointers and we are unlikely to do numeric-heavy work, the tagging scheme is not performance-critical. The current floating point implementation is incorrect; I am still deciding whether `f32` is sufficient. $\text{tagged}_{\text{ptr.rs}}$ provides the tagged-pointer interfaces; the abstraction seems sound.

3.2 Ergonomics

Rune handles moving GC with a clever technique using Rust lifetimes [10]. If I understand correctly, it pins objects on the stack during GC and relies on lifetime checks to prevent errors. In a JIT setting this no longer suffices; we need stack maps. Torrent uses a placeholder stack map for now while I investigate Cranelift's API. a question I asked and another discussion provide guidance on retrieving stack maps in Cranelift.

For moving collectors, a forwarding pointer (e.g., `Object::Indirect`) should suffice. I need to read more papers before committing; poor knowledge leads to poor design.

The inventory crate collects Rust subroutines, i.e., `#[defun]`-marked functions. Rune uses `build.rs` for this, which is more limited and harder to maintain. This idea is also borrowed from scheme-rs.

4 Concurrency

Rune's concurrency vision is described in this blog post. I currently know very little about concurrency, so I cannot offer strong opinions. Rune aims for Emacs compatibility; Torrent will feel free to diverge.

5 UI

In my vision, the new emacs UI should not be limited to 2D scenes and text buffers. But there aren't much choice for rust's GUI toolkit. vello and xilem is good and can be used, but vello uses compute shader pipeline, could be hard to integrate with other wgpu ecosystems. The viewmodel is proposed in this pull request, it obviously needs more careful considerations.

6 Miscellaneous

6.1 String representation

Rune has a great analysis of Emacs strings. My use cases are UTF-8 strings; the only reason for unibyte strings is passing binary data through FFI—better served by a dedicated type. An `Arc<String>` with `Arc::make_mut` should be adequate. We do not need to manage strings with the GC; Rust can manage them directly.

6.2 Symbols and interning

Torrent uses `lasso` as the string interner. For AOT compilation, we may need to serialize/deserialize its state.

The current symbol representation is messy. There are two types: `Ident` (a `lasso` tag) and `Symbol` (a symbol-table index plus a fallback tag). This is somewhat redundant; a more coherent design is needed.

7 References

References

[1] M. Serrano, "JavaScript AOT compilation," in *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, Boston MA USA: ACM, Oct. 24, 2018, pp. 50–63, ISBN: 978-1-4503-6030-2. DOI: 10.1145/3276945.3276950. Accessed: Sep. 12, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3276945.3276950.

[2] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "The Implementation of Lua 5.0,"

[3] Y. Shi, "Virtual Machine Showdown: Stack versus Registers,"

[4] "Using bytecode won't make your interpreter fast Dercuano," Accessed: Sep. 28, 2025. [Online]. Available: https://dercuano.github.io/notes/bytecodes-slow.html.

[5] R. Jones, A. Hosking, and E. Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*, 2nd ed. New York: Chapman and Hall/CRC, Jun. 1, 2023, ISBN: 978-1-003-27614-2. DOI: 10.1201/9781003276142. Accessed: Aug. 30, 2025. [Online]. Available: https://www.taylorfrancis.com/books/9781003276142.

[6] D. F. Bacon, P. Cheng, and V. T. Rajan, "A Unified Theory of Garbage Collection,"

[7] D. F. Bacon and V. T. Rajan, "Concurrent Cycle Collection in Reference Counted Systems," in *ECOOP 2001 — Object-Oriented Programming*, J. L. Knudsen, Ed., red. by G. Goos, J. Hartmanis, and J. Van Leeuwen, vol. 2072, Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 207–235, ISBN: 978-3-540-42206-8 978-3-540-45337-6. DOI: 10.1007/3-540-45337-7_12. Accessed: Aug. 30, 2025. [Online]. Available: https://link.springer.com/10.1007/3-540-45337-7_12.

[8] W. Zhao, S. M. Blackburn, and K. S. McKinley, "Low-Latency, High-Throughput Garbage Collection (Extended Version)," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, Comment: 17 pages, 7 Figures. This extends the original publication with an LBO analysis (Section 5.5), Jun. 9, 2022, pp. 76–91. DOI: 10.1145/3519939.3523440. arXiv: 2210.17175 [cs]. Accessed: Aug. 30, 2025. [Online]. Available: http://arxiv.org/abs/2210.17175.

[9] S. Giallorenzo and F. Goretti, "Breadth-first Cycle Collection Reference Counting: Theory and a Rust Smart Pointer Implementation," in *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*, Catania International Airport Catania Italy: ACM, Mar. 31, 2025, pp. 1412–1420, ISBN: 979-8-4007-0629-5. DOI: 10.1145/3672608.3707785. Accessed: Aug. 30, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3672608.3707785.

[10] "Implementing a safe garbage collector in Rust," Accessed: Sep. 28, 2025. [Online]. Available: https://coredumped.dev/2022/04/11/implementing-a-safe-garbage-collector-in-rust/.