# Torrent: A Rust-based Emacs Lisp Compiler/Runtime

dwuggh

September 29, 2025

## Contents

**1 Compiler Architecture** **2**
1.1 background . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Bytecode VM . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

**2 GC design** **4**
2.1 Tracing GC vs Reference Counting GC . . . . . . . . . . . . . 4
2.2 rust implementations . . . . . . . . . . . . . . . . . . . . . . . 5

**3 Lisp Object Representation** **5**
3.1 objects are tagged pointer . . . . . . . . . . . . . . . . . . . . 5
3.2 ergomics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

**4 concurrency** **6**

**5 misceallous** **6**
5.1 String representation . . . . . . . . . . . . . . . . . . . . . . . 6
5.2 symbol representation and string interner . . . . . . . . . . . 6

**6 references** **6**

Firstly I want to say I'm not a professional computer science researcher nor a programmer, and I didn't take CS as my major, so I might get fundamental, conceptual errors because of my ignorance, please feel free to correct me when I'm wrong.

Secondly, as heavily inspired by rune, a fantastic project, I consider Torrent is the "anaphoric counterpart" for rune, that is, intentionally to adopt all the technologies that rune don't use. I don't want the design to be similar, it would be no point doing so. I want this project to explore more possibilities of a rust-based emacs lisp compiler, until we have a usable one; or at least I can use lisp to write configs for other systems, for example, niri.

## 1 Compiler Architecture

### 1.1 background

Most dynamic languages compile to bytecodes, then intepretes them in a bytecode VM. The most famous of this kind is java and its JVM. Bytecodes provide excellent cross-platform compatibility, as well as great startup time. Sometimes it is not fast enough, and we want the ability to compile the code into native machine codes. Mostly this is done by translating the bytecodes to platform-specific ISAs. However, chanllenges arrives for dynamic languages:

- symbols, functions and variables may subject to change at runtime, thus we cannot program a stable function call during codegen;
- we can not know the types of our variables in advance, this will prevent most instruction-level optimizations.
- ...

We can see that this problem arises because of language flexiblity. If we can "pin" the runtime, then we can provide type informations to the compiler, and generate fast code. This is the idea of Just-In-Time compilation, a.k.a JIT compilation. Fast JIT compilations offen do type specializations, function inlining, and many other techniques to make one routine as fast as possible. We also need a way to invalidate this routine when runtime envrionment is perturbed, this can be done by some guard checks. In terminaligy, we call it "optimization and de-optimization".

Highly optimized runtimes can also add another level of JIT, the most-optimized one, which can have significant increased compile time. This is called **Tier JIT**, google's v8 and mozilla's spidermonkey both adopt this approach.

There's also attempts to make AOT compilation fast(Serrano, Manuel, 2018). It would require many advanced static analysis.

## 1.2 Bytecode VM

there are 2 common VM type: **register VM** and **stack VM**. The stack VM is most common, JVM use a stack VM, as well as CPython and many others. Stack VM models operations to retrieve its arguments in a runtime stack. Register VM is like the physical CPU, it has a bunch of virtual registers, and operations can load&store from it instead of on stack. Still, it also use a stack for function callings. Lua 5.0 switched from stack VM to register VM(Ierusalimschy, Roberto and family=Figueiredo, given=Luiz Henrique, prefix=de, useprefix=true and Celes, Waldemar, ????) MoarVM. The 2 types of VM's opcode are inherently different, but can also easily translated from one to another. Researches(Shi, Yunhe, ????) also shows that although stack VM has a more dense bytecodes, register VM has fewer instructions, and generally runs faster.

Emacs lisp uses a stack VM, and Rune has a rust implementation for that. In torrent, I might add a register VM in the future. Right now, it plans to use a baseline JIT or AOT compiler to replace the bytecode compiler. There's an old yet interesting blog post(??, ????) that argues this. However, there aren't much languages take this approach. As far as I know, Julia does not have a bytecode intepreter; Julia only dumps LLVM's bitcode. From my short experience(writing a surface code simulator), its compile time is almost unbearable. And at that time Julia lacked many language features, lacked support for enum types, buggy reloading, etc. That's why I eventually turned to rust and re-write the whole simulation program.

However in emacs lisp I think this is a good approach, nowadays most codes are native compiled anyway, and everyone seems to be pretty happy about it. Technically speaking, most emacs code will not change once ran, only a small subset of them would change, for example when some modes advicing functions, or us users debugging elisp, writing our configs, inventing new packages, etc. Still, most time we are likely to use emacs as an editor – because emacs is acidentally an editor, and editor, by definition, is a **tool** for editing files, not the editor itself. Anyway, we probably won't care much about compile speeds here.

In addition to the discussion above, I use cranelift for the codegen backend instead of LLVM. Cranelift provides great compile speeds due to its single IR architecture, and also provides fast enough execution speed(around 15% slower than LLVM). So I won't worry about compile speed at all.

Instead of bytecodes, I use a high-level IR to represent the elisp codes, that hardcode all special forms into syntax, and perform simple lexical binding analysis and closure analysis. I'll adopt more optimizations following

(Serrano, Manuel, 2018).

One interesting things can do is that we can actually serialize this HIR in form of lisp itself, so perhaps we can use elisp itself as the IR for elisp. I don't have much thoughts now: maybe this is useful, maybe its not. ChezScheme might use a similar idea: it does many optimizations using scheme itself, it is basically a scheme compiler written in scheme and achieves great performance. I didn't investigate more about it, maybe sometimes later, but this is definitely one great project worth to learn form

To summarize, we have the big picture for now:

Emacs lisp Source Code -> HIR -> Baseline JIT -> Optimizing JIT

only baseline JIT is naively implemented now.

## 2 GC design

GC theory is a huge topic for me, so I can only pick up what I read about. The main reference is the good GC handbook(Jones, Richard and Hosking, Antony and Moss, Eliot, 2023).

### 2.1 Tracing GC vs Reference Counting GC

In rune's design doc, the author said that reference counting is slow, this is actually not true. Theoicitally speaking, these 2 GC paradigm can be unified(Bacon, David F and Cheng, Perry and Rajan, V T, ????). Simply speaking, tracing GC traces the live objects while RC GC traces the dead objects. In practical RC GC has many caveats if implemented naively, but all of them can be overcomed(Jones, Richard and Hosking, Antony and Moss, Eliot, 2023). There mainly exists 2 problems:

1. loop detection is hard, especially to do it concurrently;
2. write barrier can cause mutator to have much overhead.

The first problem is solved in (Bacon, David F. and Rajan, V. T., 2001)(more citations needed) and several follow ups, described in (Jones, Richard and Hosking, Antony and Moss, Eliot, 2023). The second problem can be solved by using technique called **deferred reference counting** and **Coalesced reference counting**, especially the latter is super powerful. We also have LXR(Zhao, Wenyu and Blackburn, Stephen M. and McKinley, Kathryn S., 2022), the most recent high performance RC GC.

### 2.2 rust implementations

scheme-rs implements the bacon-rajan alorithm, where I directly copied its implementation in my repo. However, it does not adopt coalesced reference counting, and it uses tokio's channel to send mutation buffers immediately, which adds significant overhead. In test of Fibonacci sequence calculations, scheme-rs performs badly. For some reasons I cannot run flamegraph or perf when tokio is enabled, so I failed to gather perf data for now, but I believe the major factor is GC.

Recently the bacon-rajan algorithm is refined, by using a BFS instead of DFS(Giallorenzo, Saverio and Goretti, Francesco, 2025). The implementation is in rust. I haven't read this paper thoroughly though.

The MMTK is a growing GC SDK written in rust. LXR(Zhao, Wenyu and Blackburn, Stephen M. and McKinley, Kathryn S., 2022) implements their work in a separate branch in MMTK, however their are thousands divergent commits, so it will take time to merge. MMTK looks promising now and already provides a solid GC platform, my ultimate goal is to integrate with MMTK some day.

## 3 Lisp Object Representation

### 3.1 objects are tagged pointer

I took rune's approach here, a shifting tagged pointer. Since most objects in elisp are pointers, and we probably won't do scientific computations in elisp, so tagging method is really irrelevent to performance. Currently floating numbers implementation is wrong, I'm still thinking whether f32 is enough. $\text{tagged}_{\text{ptr.rs}}$ provides interfaces for tagged pointer, I think its abstraction is good.

### 3.2 ergomics

In rune, to deal with moving GC, the author gives a innovative solution that ulitizes rust's lifetime system(??, 2022). If I don't get it wrong(I hope), This system pins objects on stack to not move during GC, and provide static checks(the lifetime system) to prevent errors from happening. In the JIT context, this method no longer works? and we have to use the stack map. in Torrent I use a fake stackmap for now, as I'm investigating cranelift's API. a question I ask and another discussion provides good guideline for retrieving stack maps in cranelift.

For moving GCs, I believe a forwarding pointer with types like Object::Indirect will do the trick. I think I need to look more papers to decide what to do, a poor knowledge base only lead to poor designs.

the inventory crate is used to collect the rust subroutes, i.e. those *#[defun]* marked functions. In Rune, this is done by utilizing build.rs, which is limited and harder to maintain. This idea is also stealed from scheme-rs.

## 4 concurrency

Rune's design is described in this blog post. I must admit I know basically nothing about concurrent for now, so I cannot give valuable opinions. Rune aim to be compatible with emacs, then I shall discover new approaches in Torrent, by not compatible with emacs(lol).

## 5 misceallous

### 5.1 String representation

Rune has a great analysis on emacs string representation. However my only use cases are utf-8 strings, the only usage of unibyte-string is to pass binary data through FFI, but why not design a new type then. Naively I recon a Arc<String> with Arc::make$_{\mathrm{mut}}$ might do the job, we don't need to manage strings with our GC smart pointer; they can be managed by rust directly.

### 5.2 symbol representation and string interner

Torrent uses lasso as the string interner. To do AOT compilation, we may need to serialize/deserialize its data.

The symbol representation is messy. There are 2 types, Ident and Symbol: Ident are just a lasso's tag, and Symbol is the combination of symbol table's index and a tag for fallback. This design is somewhat redundant, and I need more thorough design of symbol and ident.

## 6 references

Bacon, David F and Cheng, Perry and Rajan, V T (). *A Unied Theory of Garbage Collection.*

Bacon, David F. and Rajan, V. T. (2001). *Concurrent Cycle Collection in Reference Counted Systems*, Springer Berlin Heidelberg.

Giallorenzo, Saverio and Goretti, Francesco (2025). *Breadth-First Cycle Collection Reference Counting: Theory and a Rust Smart Pointer Implementation*, ACM.

Ierusalimschy, Roberto and family=Figueiredo, given=Luiz Henrique, prefix=de, useprefix=true and Celes, Waldemar (). *The Implementation of Lua 5.0.*

Jones, Richard and Hosking, Antony and Moss, Eliot (2023). *The Garbage Collection Handbook: The Art of Automatic Memory Management*, Chapman and Hall/CRC.

(). *Using Bytecode Wont Make Your Interpreter Fast  Dercuano.*

(2022). *Implementing a Safe Garbage Collector in Rust.*

Serrano, Manuel (2018). *JavaScript AOT Compilation*, ACM.

Shi, Yunhe (). *Virtual Machine Showdown: Stack versus Registers.*

Zhao, Wenyu and Blackburn, Stephen M. and McKinley, Kathryn S. (2022). *Low-Latency, High-Throughput Garbage Collection (Extended Version).*