

# mimum

## Multi-purpose Functional DSP Programming Language

ADC Japan — June 3, 2026

Tomoya Matsuura/松浦知也 [me@matsuuratomoya.com](mailto:me@matsuuratomoya.com)



# About Me

- 松浦知也  
Tomoya Matsuura
- 2019–2022: Kyushu University Graduate School of Design, Master's & Doctoral program
- 2022–2025: Tokyo University of the Arts, Art Media Center (AMC), Project Assistant Professor
- 2026–: Independent
- Research area: 音 楽 土 木 工  
**Civil Engineering of Music**



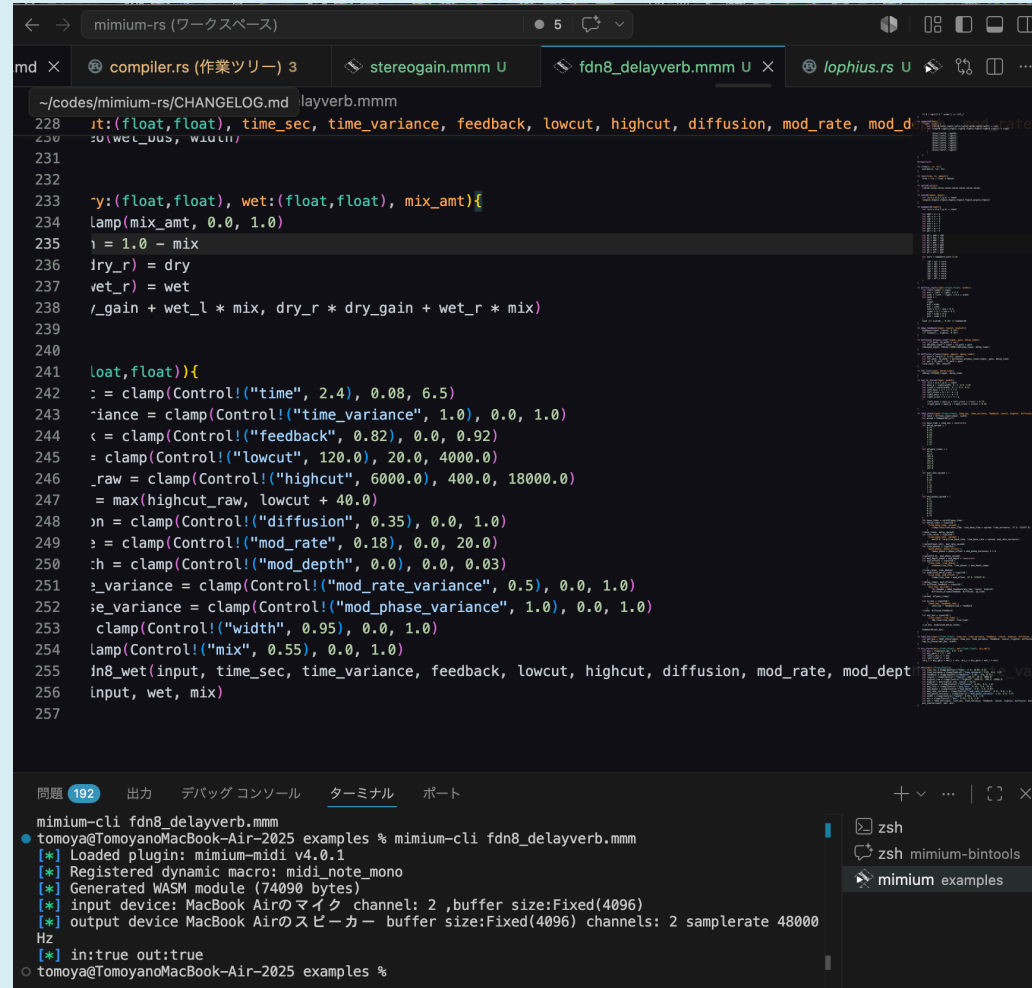
Overview

# mimium (2020–)

# What is mimium?

- **Minimal Musical medIUM** - also sounds like 耳 🧠 (Mimi:Ear)
- A functional programming language for signal processing, implementing minimum music-oriented features on top of a general-purpose language
- Developed in Rust
- Rust-like syntax
- Multiple Backends
  - Native Execution via JIT compilation through WASM(VSCode Extension with Language Server)
  - Runs in the browser (<https://mimium-org.github.io/mimium-web-editor/>)
  - Transpilation to Rust
  - VST/CLAP plugin (in progress)

# mimum in VS Code



```
~/codes/mimum-rs/CHANGELOG.md |layverb.mmm
228   t:(float,float), time_sec, time_variance, feedback, lowcut, highcut, diffusion, mod_rate, mod_d
229   su(wet_dus, width))
231
232
233   y:(float,float), wet:(float,float), mix_amt){
234   lamp(mix_amt, 0.0, 1.0)
235   l = 1.0 - mix
236   dry_r = dry
237   wet_r = wet
238   l_gain + wet_l * mix, dry_r * dry_gain + wet_r * mix)
239
240
241   loat,float){}
242   := clamp(Control!("time", 2.4), 0.08, 6.5)
243   iance = clamp(Control!("time_variance", 1.0), 0.0, 1.0)
244   < = clamp(Control!("feedback", 0.82), 0.0, 0.92)
245   = clamp(Control!("lowcut", 120.0), 20.0, 4000.0)
246   _raw = clamp(Control!("highcut", 6000.0), 400.0, 18000.0)
247   = max(highcut_raw, lowcut + 40.0)
248   on = clamp(Control!("diffusion", 0.35), 0.0, 1.0)
249   s = clamp(Control!("mod_rate", 0.18), 0.0, 20.0)
250   h = clamp(Control!("mod_depth", 0.0), 0.0, 0.03)
251   _variance = clamp(Control!("mod_rate_variance", 0.5), 0.0, 1.0)
252   se_variance = clamp(Control!("mod_phase_variance", 1.0), 0.0, 1.0)
253   clamp(Control!("width", 0.95), 0.0, 1.0)
254   lamp(Control!("mix", 0.55), 0.0, 1.0)
255   in8_wet(input, time_sec, time_variance, feedback, lowcut, highcut, diffusion, mod_rate, mod_dept
256   input, wet, mix)
257
```

```
mimum-cli fdn8_delayverb.mmm
tomoya@TomoyanoMacBook-Air-2025 examples % mimum-cli fdn8_delayverb.mmm
[*] Loaded plugin: mimum-midi v4.0.1
[*] Registered dynamic macro: midi_note_mono
[*] Generated WASM module (74090 bytes)
[*] input device: MacBook Airのマイク channel: 2 ,buffer size:Fixed(4096)
[*] output device MacBook Airのスピーカー buffer size:Fixed(4096) channels: 2 samplerate 48000
Hz
[*] in:true out:true
tomoya@TomoyanoMacBook-Air-2025 examples %
```

[Search "mimum" in the VS Code Marketplace.](#) The extension automatically downloads the runtime.

# mimium on Web browser

The screenshot displays the 'mimium web editor' interface. On the left, a sidebar lists various audio processing examples such as '0b5vr', 'biquad', 'cascadeosc\_macro', 'cascadeosc', 'compressor', 'distortion', 'fbdelay\_mod', 'fmpiano', 'getnow', 'jcrev', 'livecoding\_demo', 'midiin', 'noise', 'rain', 'reactive\_f', 'reactive\_sequencer', 'robot', 'scale', 'sequencer', 'sinewave', 'subtract\_synth', 'supersaw', 'uzulang', and 'windmodel'. The main editor area shows Rust code for an audio processing function. The code includes a list of notes, gates, and accents, and a DSP function that processes audio samples. A 'Rust Preview' window on the right shows the compiled Rust code, which includes a struct definition for 'MimiumProgram' and a 'dsp' function that initializes and updates a set of registers. At the bottom, there is a control bar with 'Play', 'Stop', and 'Update' buttons, a volume slider set to '+0.0dB', and a stereo level indicator (L/R). A 'Rust preview ready' indicator is visible in the bottom right corner.

```
28 fn ep_voice(note, gate, velocity, phase_shift) {
46     let op5 = op(97.0, 1.0, op6, gate,
47
48     let tone = (op1 + op3 + op5) * (0.
49     let cutoff = 4000.0 + key_pos * 14
50     lowpass(tone, cutoff, 0.85)
51 }
52
53 let notes = [48, 52, 55, 59, 60, 55,
54
55 let gates = [1, 1, 1, 1, 1, 1, 1,
56 let accents = [0.70, 0.45, 0.55, 0.3
57
58 let bpm = 86.0
59 let steps_per_beat = 2.0
60 let step_per_sec = bpm / 60.0 * step
61
62 fn dsp(){
63     let s = step_seq(step_per_sec, not
64     let vel = 58.0 + s.accent * 62.0
65
66     let left = ep_voice(s.note + 0.03,
67     let right = ep_voice(s.note - 0.03
68
69     (left, right)
70     ||> pingpong_delay(_, 0.75/step_per
71     //> Probe! ("out")
72 }
73
```

```
429 impl<H: MimiumHost> MimiumProgram<H> {
6949     fn dsp(&mut self, ret_words: &mut [Word; 2]) -> () {
6975         let mut reg_1656: Word = 0u64;
6976         let mut reg_1657: mmmfloat = 0.0 as mmmfloat;
6977         let mut reg_1658: Word = 0u64;
6978         let mut reg_1659: Word = 0u64;
6979         let mut reg_1660: Word = 0u64;
6980         let mut reg_1661: Word = 0u64;
6981         let mut reg_1662: mmmfloat = 0.0 as mmmfloat;
6982         let mut reg_1663: mmmfloat = 0.0 as mmmfloat;
6983         let mut reg_1664: Word = 0u64;
6984         let mut reg_1665: Word = 0u64;
6985         let mut reg_1666: mmmfloat = 0.0 as mmmfloat;
6986         let mut reg_1667: mmmfloat = 0.0 as mmmfloat;
6987         let mut reg_1668: Word = 0u64;
6988         let mut reg_1669: mmmfloat = 0.0 as mmmfloat;
6989         let mut reg_1670: Word = 0u64;
6990         let mut reg_1671: Word = 0u64;
6991         let mut reg_1672: Word = 0u64;
6992         let mut reg_1673: mmmfloat = 0.0 as mmmfloat;
6993         let mut reg_1674: Word = 0u64;
6994         let mut reg_1675: Word = 0u64;
6995         let mut reg_1676: mmmfloat = 0.0 as mmmfloat;
6996         let mut reg_1677: Word = 0u64;
6997         let mut reg_1678: Word = 0u64;
6998         let mut reg_1679: mmmfloat = 0.0 as mmmfloat;
6999         let mut reg_1680: mmmfloat = 0.0 as mmmfloat;
7000         let mut reg_1681: mmmfloat = 0.0 as mmmfloat;
7001         let mut reg_1682: mmmfloat = 0.0 as mmmfloat;
7002         let mut reg_1683: mmmfloat = 0.0 as mmmfloat;
7003         let mut reg_1684: mmmfloat = 0.0 as mmmfloat;
7004         let mut reg_1685: Word = 0u64;
7005         let mut reg_1686: Word = 0u64;
7006         let mut reg_1687: Word = 0u64;
```

# mimum on VST/CLAP plugin(WIP)

The image shows a screenshot of a Max/MSP patch window titled "Untitled1 (unlocked)" and a separate window titled "Mimium CLAP Plugin".

The Max/MSP patch contains a "Mimium CLAP Plugin" object with a control panel. The control panel has a table with the following columns: #, Name, Slider, and Value. Below the table is a "live.gain~" object, a "0.0 dB" label, and a "dac~" object. A signal flow is shown with dashed yellow lines connecting the plugin's output to the "live.gain~" object, then to the "0.0 dB" label, and finally to the "dac~" object. A waveform display is also visible in the patch.

The "Mimium CLAP Plugin" window displays the following Pure Data code:

```
MIMIUM AUDIO PLUGIN
Live Coded Stereo Synth

1 let twopi = 6.283185307179586
2
3 fn phasor(freq: float ) {
4   (self + freq / samplerate) % 1.0
5 }
6
7 fn dsp() {
8   let left = sin(phasor(220.0) * twopi) * 0.18
9   let right = sin(phasor(440.0) * twopi) * 0.18
10  (left, right)
11 }
```

# Table of Contents

- Motivation and Language Design
- Code Examples
- Live Coding feature
- Runtime Implementation
- Plugin Extension & Embedding
- Performance & Benchmark
- Conclusion

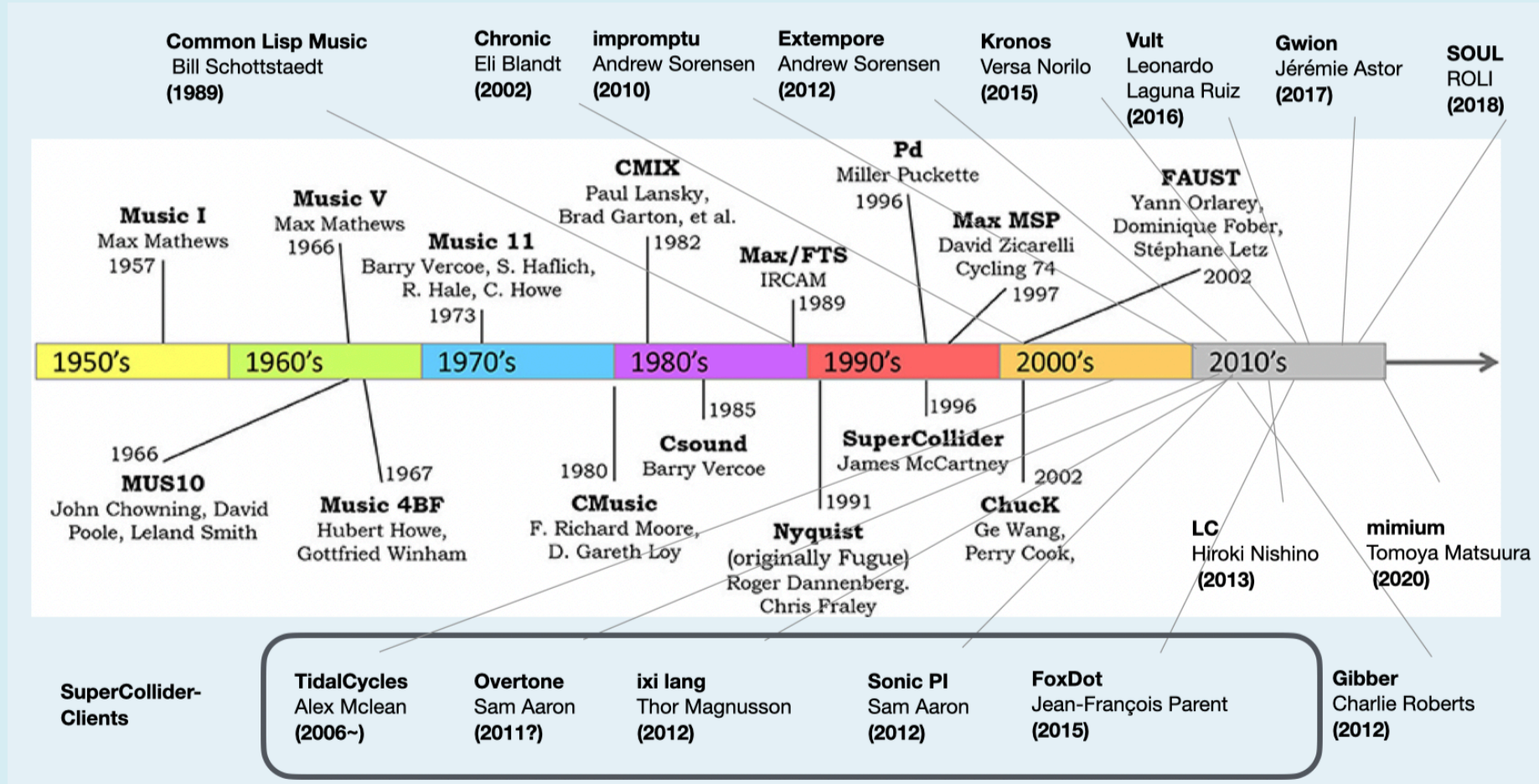
**Design Concept**

# **Sound programming languages should be simpler?**

Design Concept

**Sound programming languages  
should be ~~simpler~~ more like  
general programming  
language?**

# History of Computer Music Languages



Background: "Languages for Computer Music, Roger B. Dannenberg", *Frontiers in Digital Humanities*, 30 Nov. 2018, Matsuura added newer projects

# Why You Make a New Domain Specific Language?

- Because general-purpose language requires additional description in addition to essential audio processing content
  - Memory Management
  - Thread Management
  - Hardware Encapsulation
- But wait, if the language is expressive enough, can it hide those details as a library?

*"Is a specialized computer music language even necessary? In theory at least, I think not. The set of abstractions available in computer languages today are sufficient to build frameworks for conveniently expressing computer music.*

*Unfortunately, in practice, some pieces are missing in the implementations of languages available today. Often, the garbage collection is not performed in real time, and often argument passing is not very flexible. If lazy evaluation is not included, then implementing Patterns and Streams becomes more complicated."*



McCartney, James. "Rethinking the computer music language: SuperCollider." *Computer Music Journal* 26, no. 4 (2002): 61–68.  
<https://doi.org/10.1162/014892602320991383>

# Why Not Just Use an Existing General-Purpose Language?

- Unfortunately, as of the 2020s, no general-purpose language satisfies all of:
  - Detailed & Concise description of audio signal processing without care of hardware management
  - Controllable GC timing by the user
  - Hot-swap of running code for live coding

So, what if we design **general programming language but primarily designed for audio processing?**

# What About Existing Languages

	 <b>SuperCollider</b>	 <b>ChuckK</b>	<b>FAUST</b> <b>Faust</b>
<b>Strength</b>	Notation flexibility	Sample-accurate control	Strict formal semantics
<b>Weakness</b>	Less Portable	UGens not First-class	Exotic Notation
	Huge Codebase	Delay/reverb breaks on live-coding	Not Live-Codable

# Goal: A More Ordinary Programming Language

- Readable syntax similar to JS or Rust
- No special treatment of Unit Generators; no dedicated signal primitive type
  - With a set of primitive stateful operations, UGens can simply be function, not class
  - primitive operations: **Delay and Feedback**
- Mostly no GC for realtime safety

Language Design

# $\lambda_{\text{mmm}}$ — Core Calculus

# $\lambda_{\text{mmm}}$ : Based on Simply Typed, Call-by-Value Lambda Calculus

$$\begin{array}{l} \tau ::= \textit{unit} \\ | R \\ | I_n \quad n \in \mathbb{N} \\ | \tau \rightarrow \tau \end{array}$$

Types

$$\begin{array}{l} v ::= \textit{unit} \\ | R \quad R \in \mathbb{R} \\ | \textit{closure}(e, E) \end{array}$$

Values

$$\begin{array}{l} e_p ::= \textit{unit} \\ | R \quad R \in \mathbb{R} \\ e ::= e_p \\ | x \quad [\textit{var}] \\ | \lambda x. e \quad [\textit{abs}] \\ | \textit{let } x = e \textit{ in } e \quad [\textit{let}] \\ | e e \quad [\textit{app}] \\ | \textit{if}(e_c) e_t \textit{ else } e_e \quad [\textit{if}] \\ | \textit{delay}(n, e_{\textit{time}}, e_{\textit{bound}}) \quad [\textit{delay}] \\ | \textit{feed } x. e \quad [\textit{feed}] \end{array}$$

Terms

# 1-Pole Filter (Integrator)

```
fn onepole(x, ratio){  
    x*(1.0-ratio) + self*ratio  
}
```

- `self` is initialized to 0 and retrieves the return value of that function 1 sample ago
- Explicitly handles feedback connections — essential and often complex in signal processing
- Function is stateful: every samples returns different value, but its sequence is deterministic

# Biquad Filter

```
fn biquad_inner(x,a1,a2){
    let (ws, wss, _wsss) = self
    let w = x - a1*ws - a2*wss
    (w, ws, wss)
}
pub fn biquad(x,coeffs){
    let (a1,a2,b0,b1,b2) = coeffs;
    let (w,ws,wss) = biquad_inner(x,a1,a2);
    b0*w + b1*ws + b2*wss
}
```

- `self` is polymorphic (becomes as same as return type)

# Feedback Delay

```
fn fbdelay(input, time, fb, mix){  
    input * mix + 1.0 - mix * delay(40001.0, input + self * fb, time)  
}
```

- `delay` is a built-in function. Three arguments: maximum delay size, input signal, delay length
- max delay size must be compile-time constant

# Sinewave Oscillator

```
use math::*
fn phasor_zero(freq) {
    (self + freq/samplerate)%1.0
}
pub fn phasor(freq,phase_shift=0) {
    (phasor_zero(freq) + phase_shift)%1.0
}
pub fn sinwave(freq,phase=0) {
    phasor(freq,phase)*2.0*PI() |> sin
}
```

- `samplerate` is a runtime-defined environment variable
- Pipe `|>` is notation for writing `a(b)` as `b |> a` — pairs well with functional signal dataflow

Key Insight

**UGen = Function,  
so higher-order functions  
enable meta-operation to the  
processors**

# Using Higher-Order Functions: SuperSaw

```
#stage(macro)
let detune_table = [128,-128,408,-417,704,720]
let numbers = detune_table |> len |> lift
let detunepitches = map(|x| x / (2^7), detune_table)
fn supersaw() {
  let init = `|freq, detune| { saw(freq, 0) / $numbers }
  foldl(detunepitches, init, |elem, acc| {
    `|freq, detune| {
      let f = freq + $(elem|>lift) * detune
      ($acc) (freq, detune) + saw(f, 0) / $numbers
    }
  })
}
```

# 多 段 階 計 算 Multi-stage Computation (Typed Macro)

- Quote(`) and Splice(\$ ) in lisp, with type safety
  - Type checking & inference is done before the macro is expanded
- Compile time computation can be extended in various way
- For instance, if text parsing program is executed in compile-time computation?
  - Embed **another DSL on mimium!**

# uzulang(mini-notation) DSL on mimum

```
use core::*
use mininotation::*
use osc::*
use env::adsr
use delay::stereo_delay
use filter::lowpass
fn dsp(){
    let note = run_note!(mini("60 <62 72> <[64 [74 72] 65] [65 62]> ~ 69"))
    ||> legato(0.2,_, 1)
    saw(note.val-12 |> midi_to_hz, 0.0)
    * adsr({attack = 0.001,release=0.1,sustain=0.9,gate=note.gate})
    ||> lowpass(_, ((sinwave(0.1,0)+1)*500+400), 4)
}
```

script inside `mini` function is embedded DSL on mimum

Unique Features

# Live Coding

# Live coding in mimium

- In mimium native CLI, when user update the code and save, the sound are automatically updated to newer code
- The tail of feedback delay or reverbration are kept naturally on update
- However, it does not modify existing runtime. Instead it compile new source code from zero and **swap entire virtual machine** while copying state variable as it can
- "Static" Live Coding

```

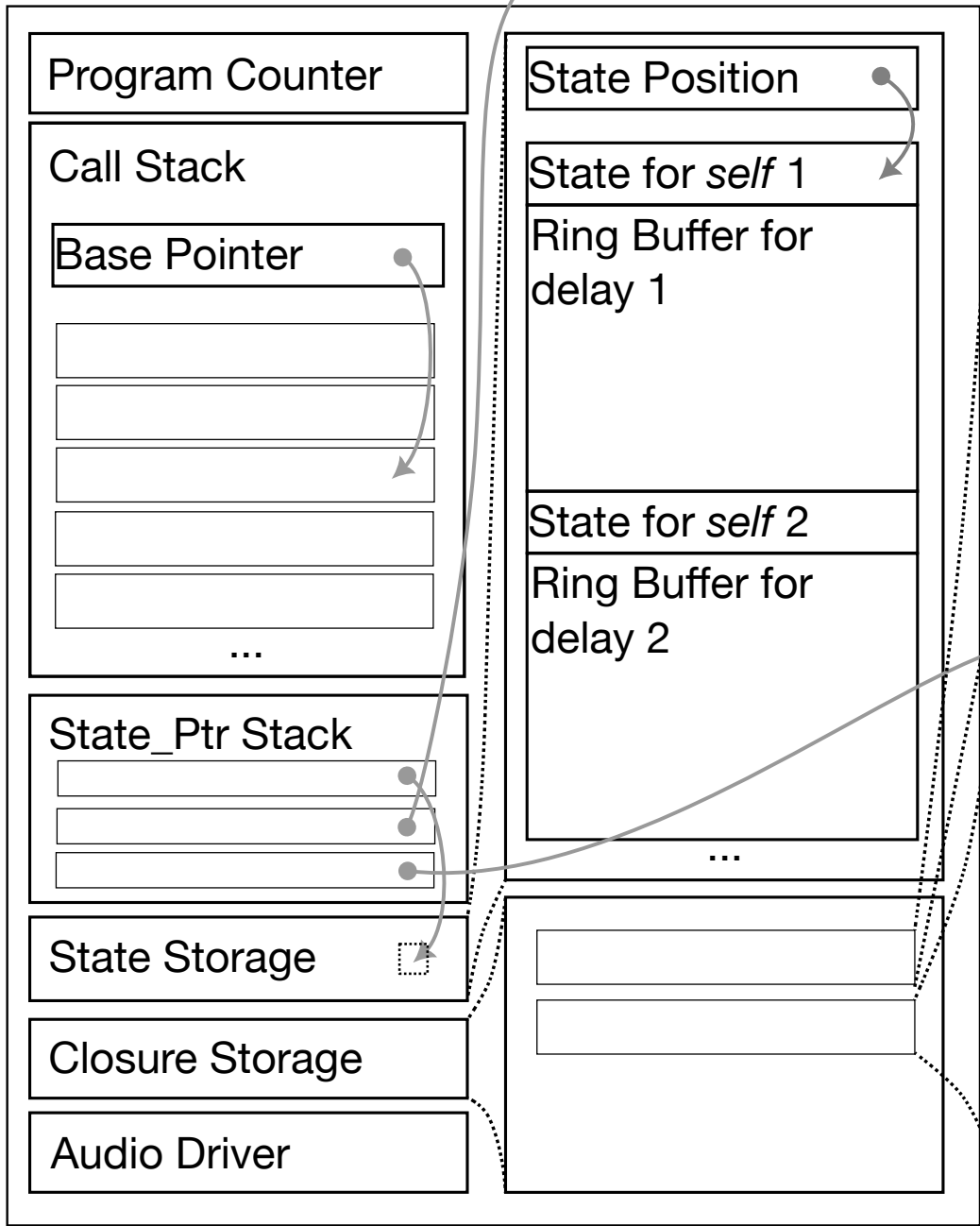
use ...
let notes = [60,62,64,67,72]
fn melody(cps){
    let l = notes |> len
    let phase = phasor(cps/(l-1),0)
    let i = phase * l |> floor
    let freq = notes[i] |> midi_to_hz
    let gate = {gate = (phase*1%1) < 0.2} |> adsr
    let ff = (sinwave(1,0)+1)*1000+200
    saw(freq,0)
    ||> lowpass(_,ff,4)
    ||> _ * gate
}
let cps = 2
fn dsp(){
    melody(cps)
    |> tostereo
    ||> pingpong_delay(_,0.5,0.3,1.0,0.5)
}

```

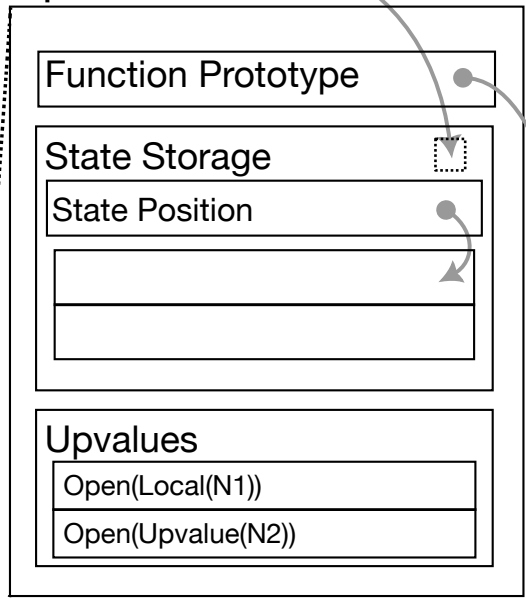
Implementation

# Runtime Representation

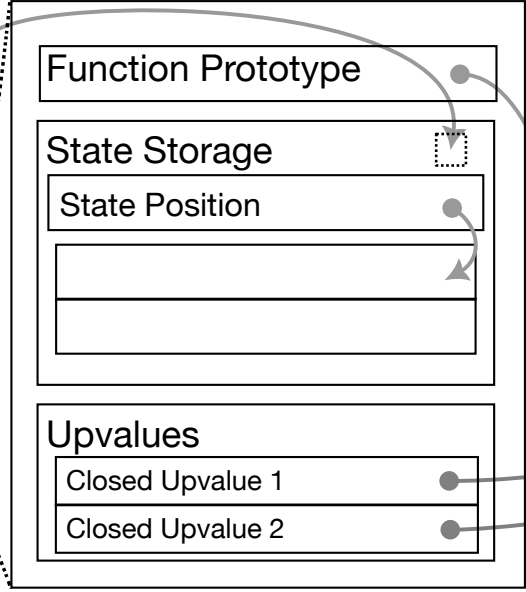
# Virtual Machine



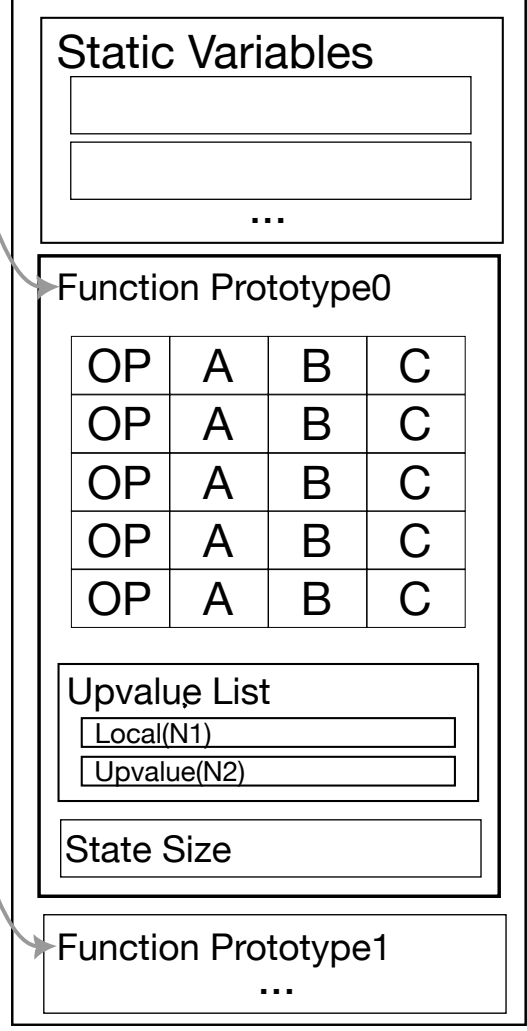
# Open Closure



# Escaped Closure



# Program



Somewhere on the Heap Memory  
(Maybe Shared with other closures)

# 相 对 移 動 ポ イ ン タ モ デ ル

## Relatively Moving Pointer Model

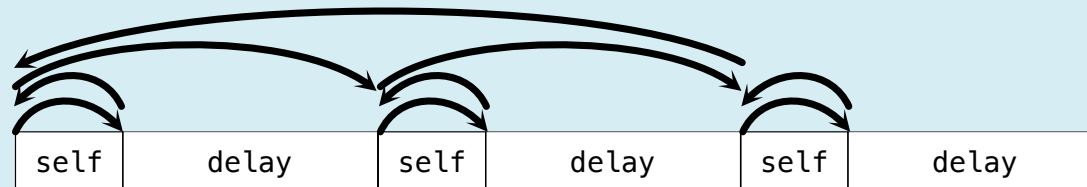
```
fn fbdelay(input, time, fb, mix){
    input * mix + 1.0 - mix * delay(40001.0, input + self * fb, time)
}

fn dsp(input){
    input
    ||> fbdelay(_, 2000, 0.9, 0.5)
    ||> fbdelay(_, 3000, 0.7, 0.5)
    ||> fbdelay(_, 5000, 0.5, 0.2)
}
```

- `a ||> foo(_, b, c)` is equivalent to `foo(a, b, c)`
- Three `fbdelay` should be interpreted as different instances

```
fn fbdelay(input, time, fb, mix){
  ...
  let s = get_self();
  shift_state_position(1);
  ...
  update_ringbuffer(...);
  shift_state_position(-1);
  ...
  let ret_value = ...
  set_self(ret_value);
  ret_value
}
```

```
fn dsp(input){
  let a = fbdelay(input, 2000, 0.9, 0.5);
  shift_state_position(40004);
  let b = fbdelay(a, 3000, 0.7, 0.5);
  shift_state_position(40004);
  let c = fbdelay(b, 5000, 0.5, 0.2);
  shift_state_position(-80008);
  c
}
```



# RMP model and Live Coding

- States are aligned to the order of function call tree
- **Tree data can be structurally compared**
  - 最 長 共 通 部 分 列 Longest Common Subsequence (LCS) algorithm (similar to React)
- Take the diff of root `dsp` function call tree between 2 version
- Generate "Patches" for copying available state data into newer state storage

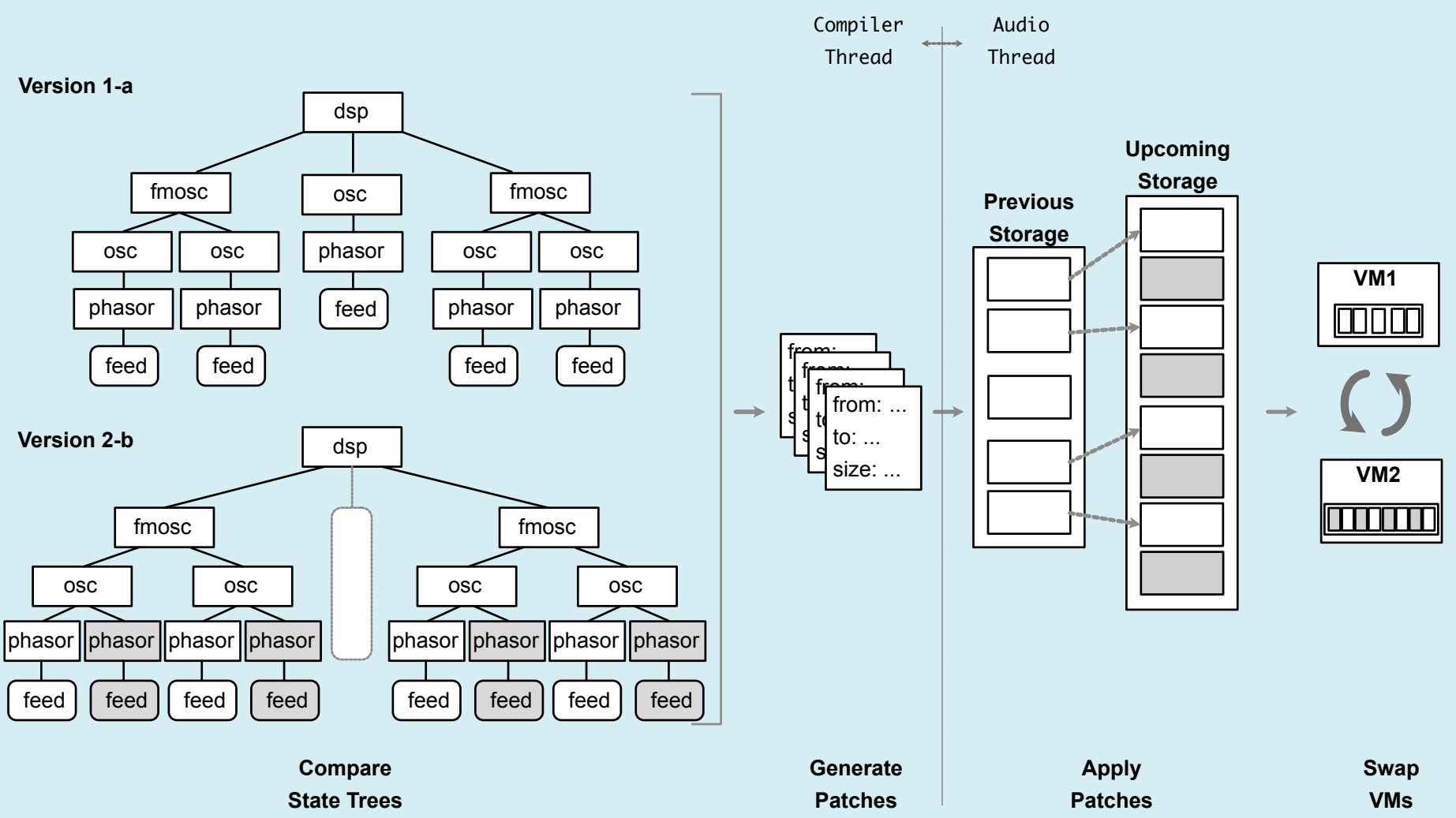
# RMP model and Live Coding

```
State ::= Feed (Type)
       | Mem (Type)
       | Delay (Type, Max_Time)
       | DirectFnCall (Vec (State))
```

- Reduced version of call tree can be derived
- Recursive function call can be removed from this tree because it generates closure (kind of instance of function), and state operation for closure is done on the closure instance itself, so the tree traversal must stop

# Code Example

```
fn phasor(freq) {
  (self+(1/freq))%1.0
}
fn osc(freq) {
-  phasor(freq * 2 * PI |> sin //case a
+  phasor(freq+(phasor(freq/10))) * 2 * PI |> sin //case b
}
fn fmosc(freq,rate) {
  osc(freq + osc(rate))
}
fn dsp() {
-  fmosc(440,10) + osc(880) + fmosc(1320,10)//case 1
+  fmosc(440,10) + fmosc(1320,10)//case 2
}
```



Each of "Patch" is like memcpy operation

# Static Live coding

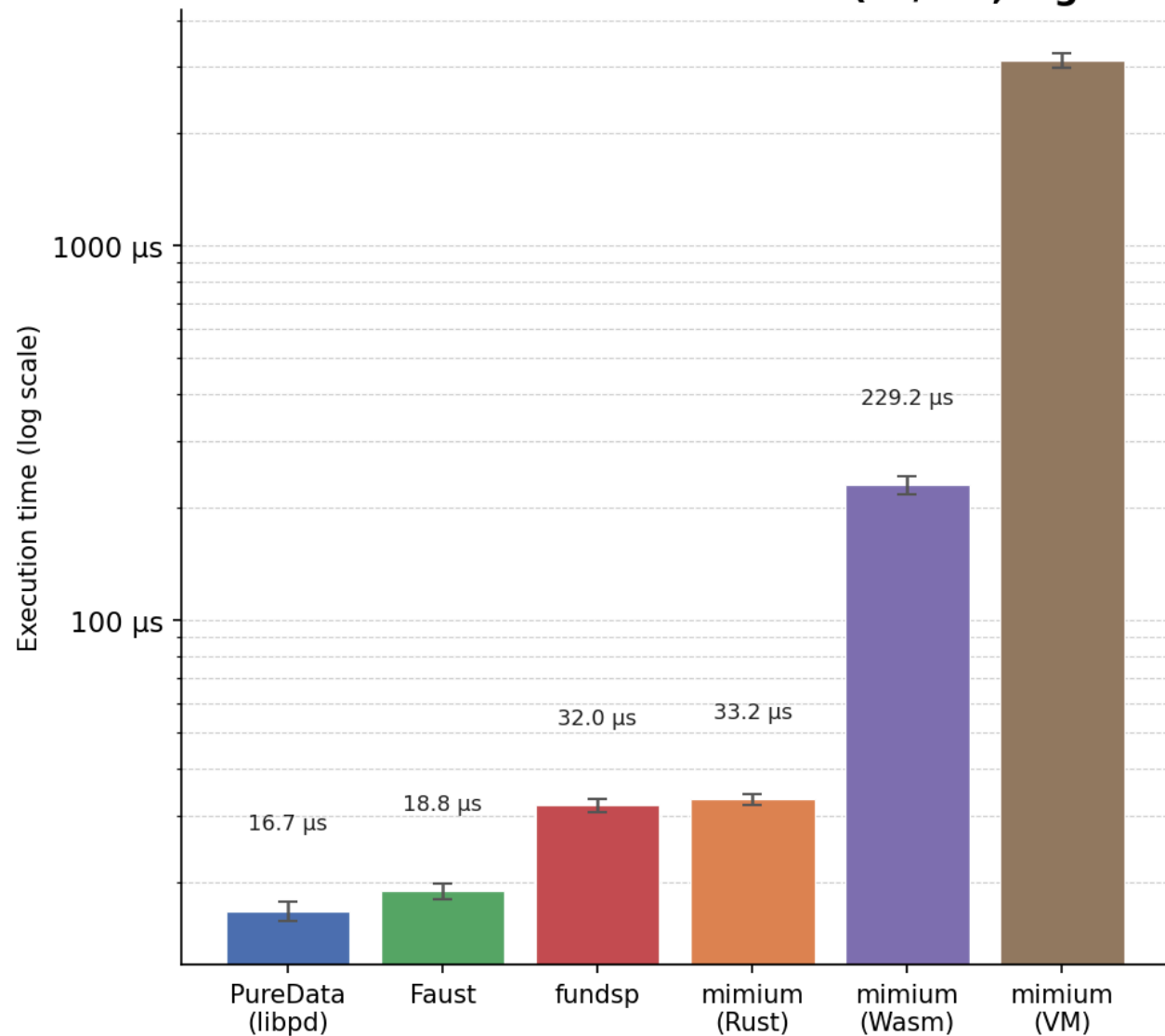
- Operation done solely with comparison of text source without integration of editor
- Does not block audio thread until applying patch
- Block duration depends on **structural difference of the code**, not the entire code size
- Keep simplicity of the runtime implementation - transpiler still works without affects
- User can focus on **current algorithm, not the runtime state**

Measurement

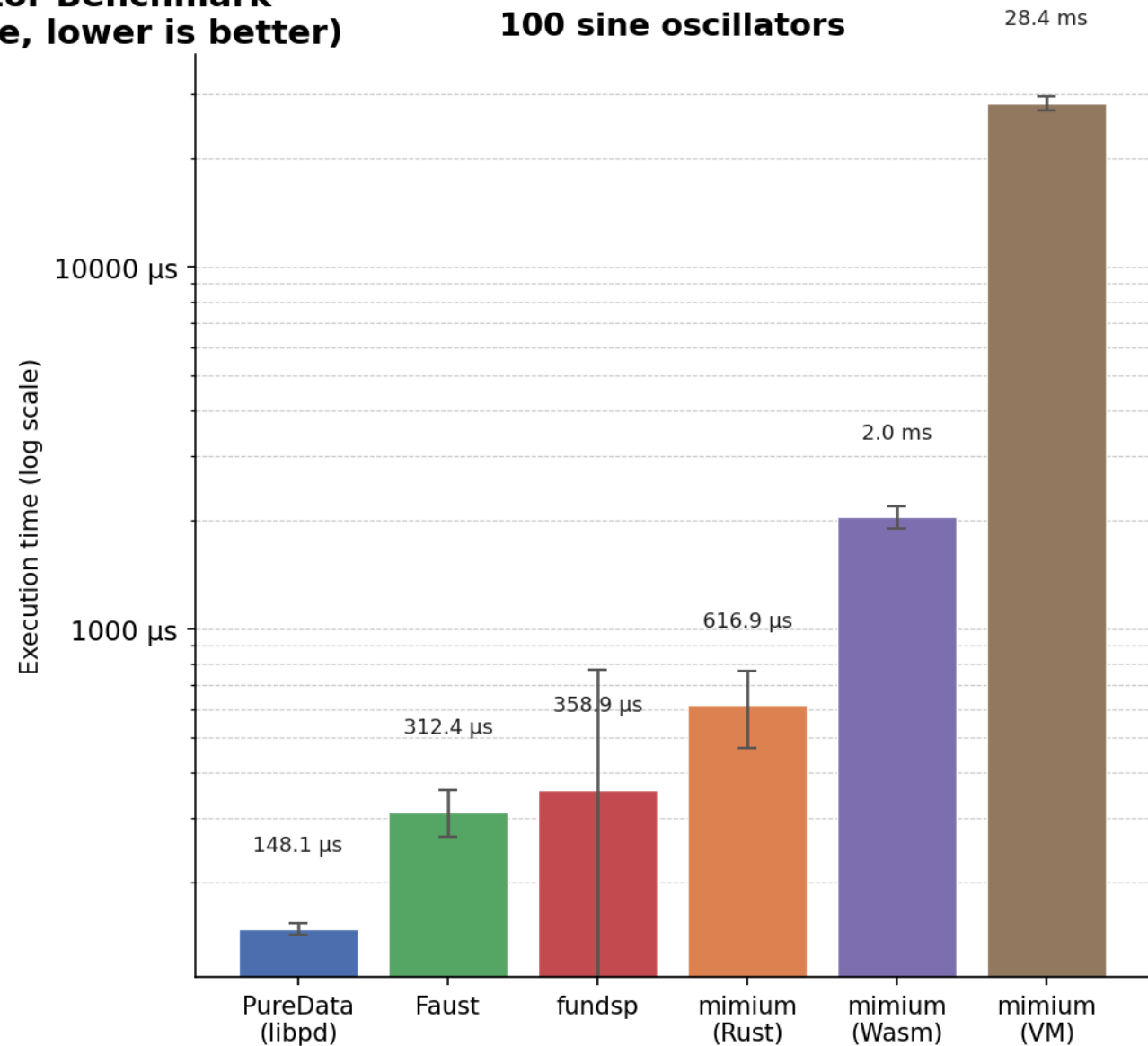
# What about Performance?

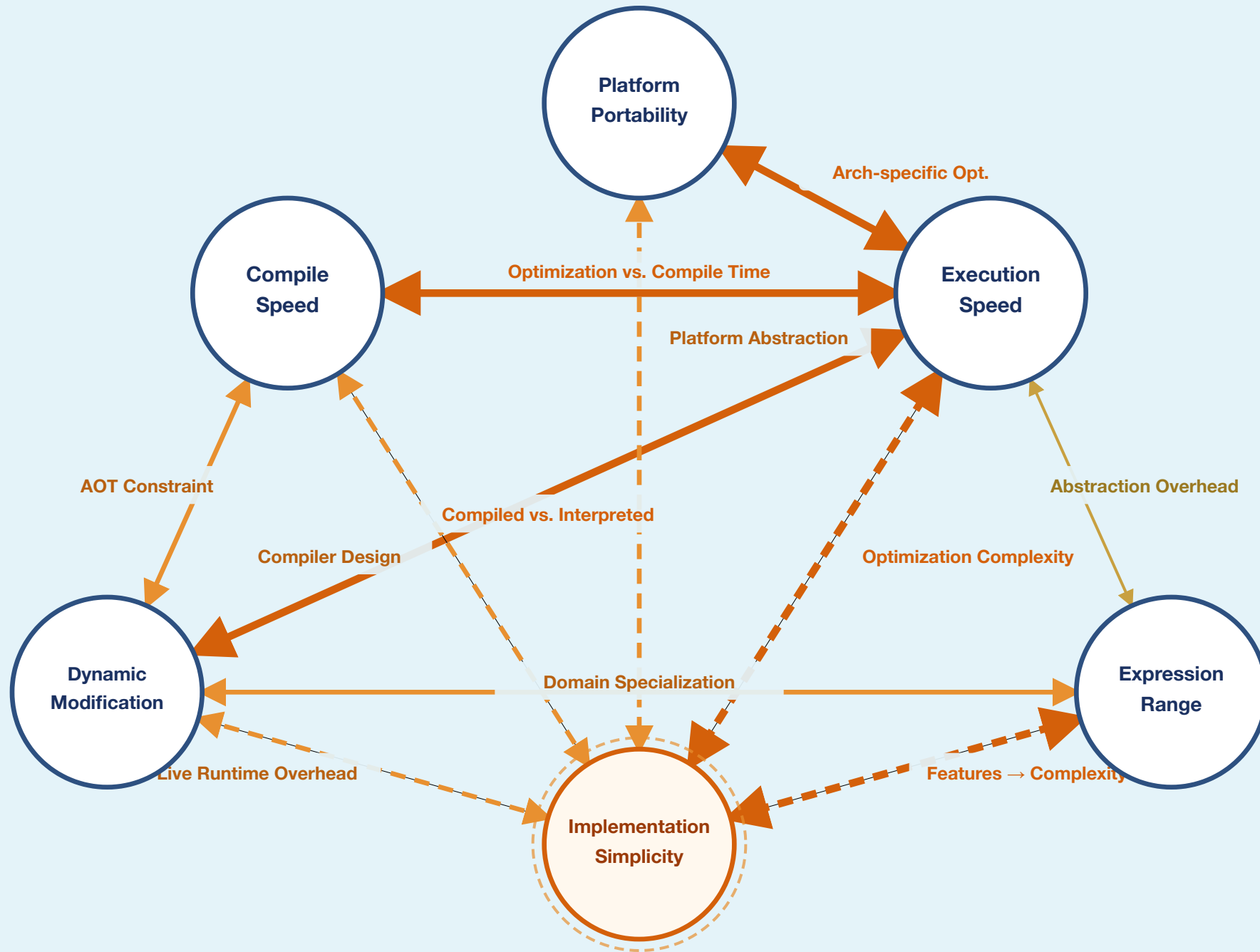
### Sine Oscillator Benchmark

10 sine oscillators (ns/iter, log scale, lower is better)



### 100 sine oscillators





Extension

# Plugin System

# Plugin System

- mimium can call external Rust functions through plugin system
- Native extensions like MIDI, Audio File Sampler, GUI system are implemented with this manner
- Even audio driver(`samplerate` and `now` literals) and basic scheduler are implemented with plugin system
- The plugin developer can define their own plugin through `SystemPlugin` trait and `mimum_export_plugin!` macro. (Dynamically loadable)

# Audio File Sampler Example

```
fn counter() {  
    self+1.0  
}  
  
//add -1.0 offset so that the counter start from 0 at t=0  
fn dsp() {  
    let sampler = Sampler_mono!("../tests/assets/konnichiwa.aif")  
    let player = sampler.player  
    let len = sampler.length  
    (counter()-1.0)%len  
    |> player  
    |> Probe!("out")  
}
```

`foo!(bar)` is the shorthand for macro invocation `$(foo(bar))`

# Audio File Sampler Example

```
fn counter() {
    self+1.0
}
//add -1.0 offset so that the counter start from 0 at t=0
fn dsp() {
    let sampler = _get_sampler(0) //0 is internal id
    let player = sampler.player
    let len = sampler.length
    (counter()-1.0)%len
    |> player
    |> _probe_intercept(0) //0 is internal id too
}
```

After the macro expansion

```

#[derive(Default)]
pub struct SamplerPlugin {
    sample_buffers: Vec<Arc<Vec<f64>>>,
    sample_namemap: HashMap<String, usize>,
}

impl SamplerPlugin {
    ...
    #[mimum_plugin_macro]
    pub fn make_sampler_mono(&mut self, v: &[(Value, TypeNodeId)]) -> Value {
        ...
        Value::Code(Self::sampler_record_expr(idx, sample_length))
    }
    #[mimum_plugin_fn]
    pub fn get_sampler(&mut self, pos: f64, file_idx: f64) -> f64 {
        let idx = file_idx as usize;
        let samples = self.sample_buffers.get(idx);
        match samples {
            Some(vec) => interpolate_vec(vec, pos),
            None => {
                mimum_lang::log::error!("Invalid file index: {idx}");
                0.0
            }
        }
    }
}

```

```
mimum_export_plugin! {
  plugin_type: SamplerPlugin,
  plugin_name: "mimum-symphonia",
  plugin_author: "mimum-org",
  capabilities: {
    has_audio_worker: false,
    has_macros: true,
    has_runtime_functions: true,
  },
  runtime_functions: [
    ("__get_sampler", get_sampler),
  ],
  macro_functions: [
    ("Sampler_mono", make_sampler_mono),
  ],
  type_infos: [
    { name: "Sampler_mono",
      sig: SamplerPlugin::sampler_mono_signature(), stage: 0 },
    { name: "__get_sampler",
      ty_expr: function!(vec![numeric!(), numeric!()], numeric!()),
      stage: 1 },
  ],
}
```

**Conclusion**

# **Future Directions**

# Reflection on the Language Design

- Multi-purpose: Programming language as a exploration tools without pre-determined role, not just a problem-solving tool
- It's becoming useful tool, at least for me
- There are more spaces to explore in the intersection of the PL theory and DSP theory
- Agentic coding is especially powerfull for language development, let's try make new one

# Future development

- More development of Rust & other transpilers (including microcontroller target)
- Improving module system, and package manager?
- Sophisticated polymorphism (maybe type class?)
- Better GUI integration and plugin system
- Development of semantics for more macro temporal structure (composition)

# Summary

- mimium is a **functional music programming language** based on  $\lambda$ mmm
- UGens are first-class functions — enabling higher-order composition
- Through type-safe macro, user can create another DSL on top of it
- Static live coding through a structural comparison of the codes
- Available at <https://mimium.org/> / <https://github.com/tomoyanonymous/mimium-rs>

# Thank You

Tomoya Matsuura/松浦知也

[me@matsuuratomoya.com](mailto:me@matsuuratomoya.com)

<https://matsuuratomoya.com>

<https://mimum.org/>

<https://github.com/tomoyanonymou/mimum-rs>



mimum