mimium

Multi-purpose Functional DSP Programming Language

ADC Japan — June 3, 2026

Tomoya Matsuura/松浦知也　me@matsuuratomoya.com

About Me

- Tomoya Matsuura (松浦知也)
- 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: **Civil Engineering of Music** (音楽土木工学)

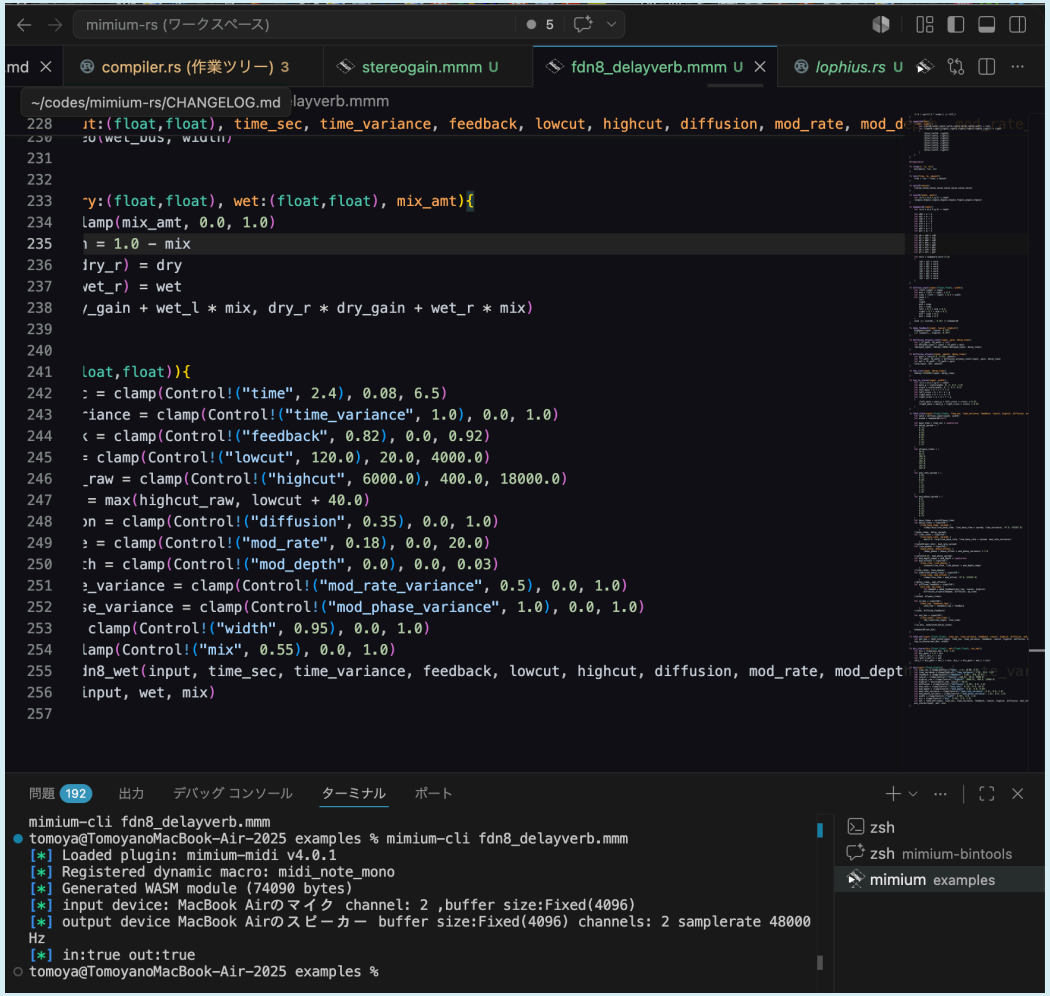
Overview

mimium (2020–)

What is mimium?

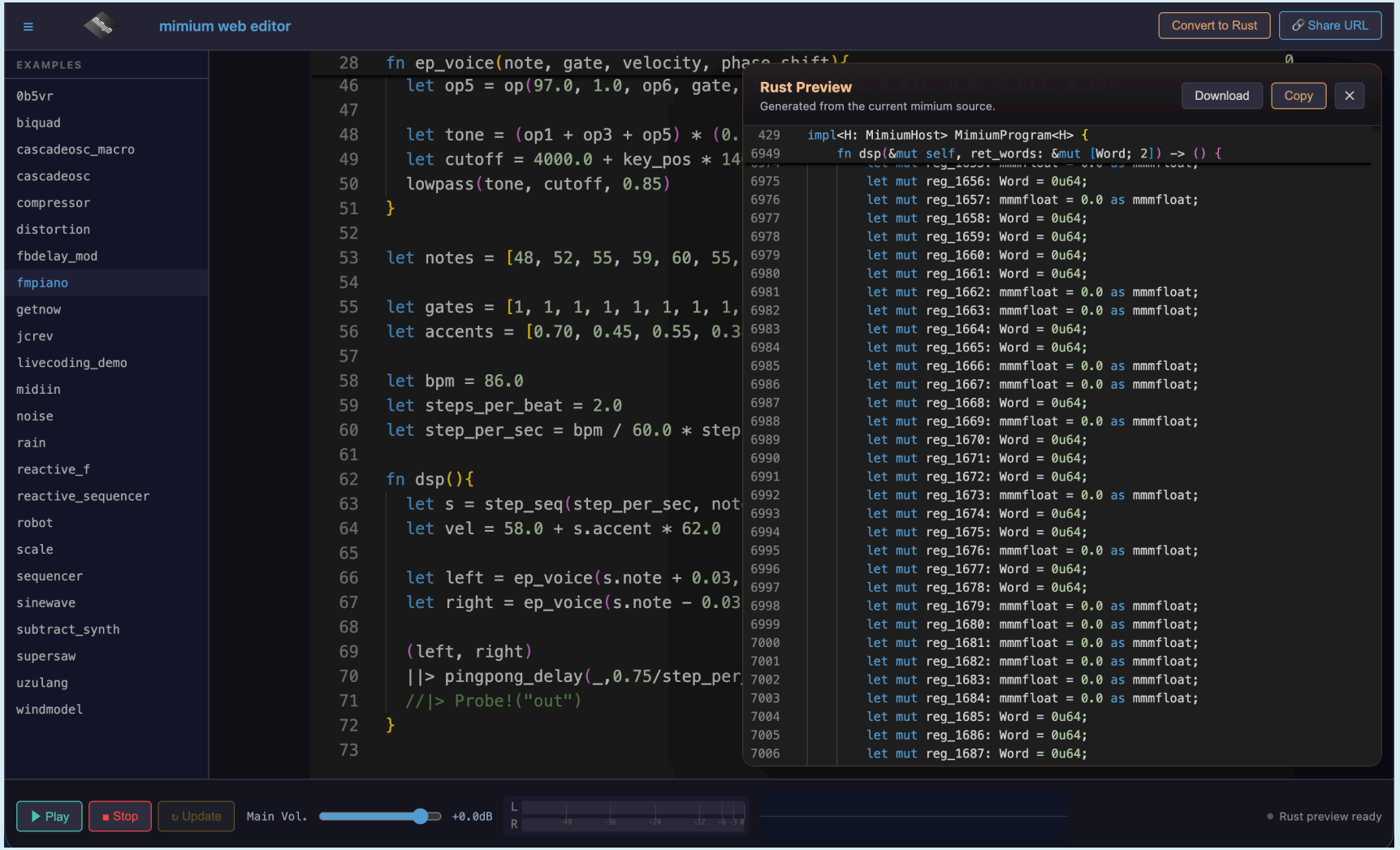
- **Mi**nimal **M**usical med**IUM** - also sounds like 耳👂(Mimi:Ear)
- A functional programming language for signal processing, implementing minimum music-oriented features on top of a general-purpose language
- Developed in Rust
- Rust-like syntax
- Multiple Backends
  - Native Execution via JIT compilation 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)

mimium in VS Code



Search "mimium" in the VS Code Marketplace. The extension automatically downloads the runtime.

mimium on Web browser



mimium on VST/CLAP plugin(WIP)

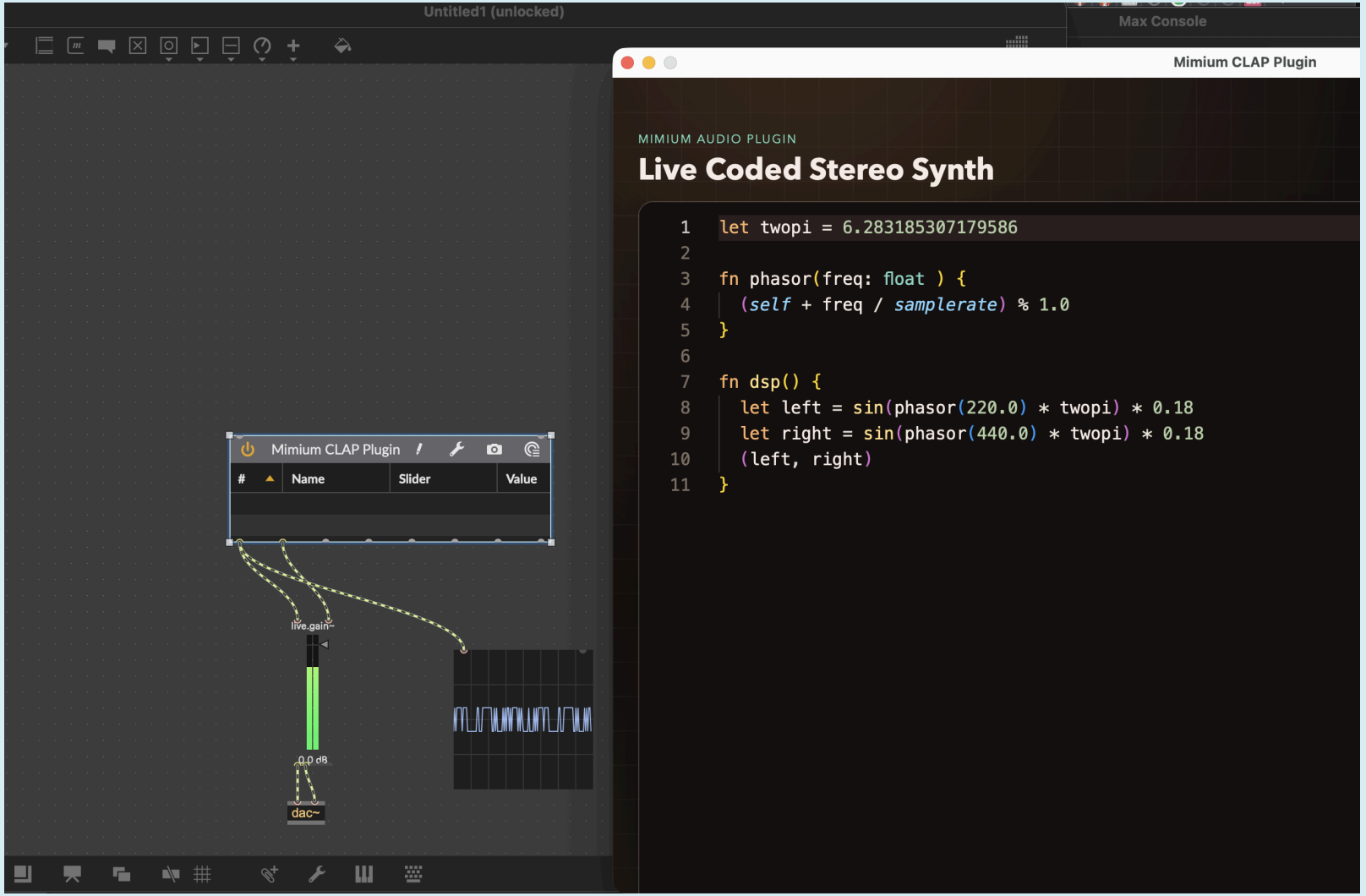


Table of Contents

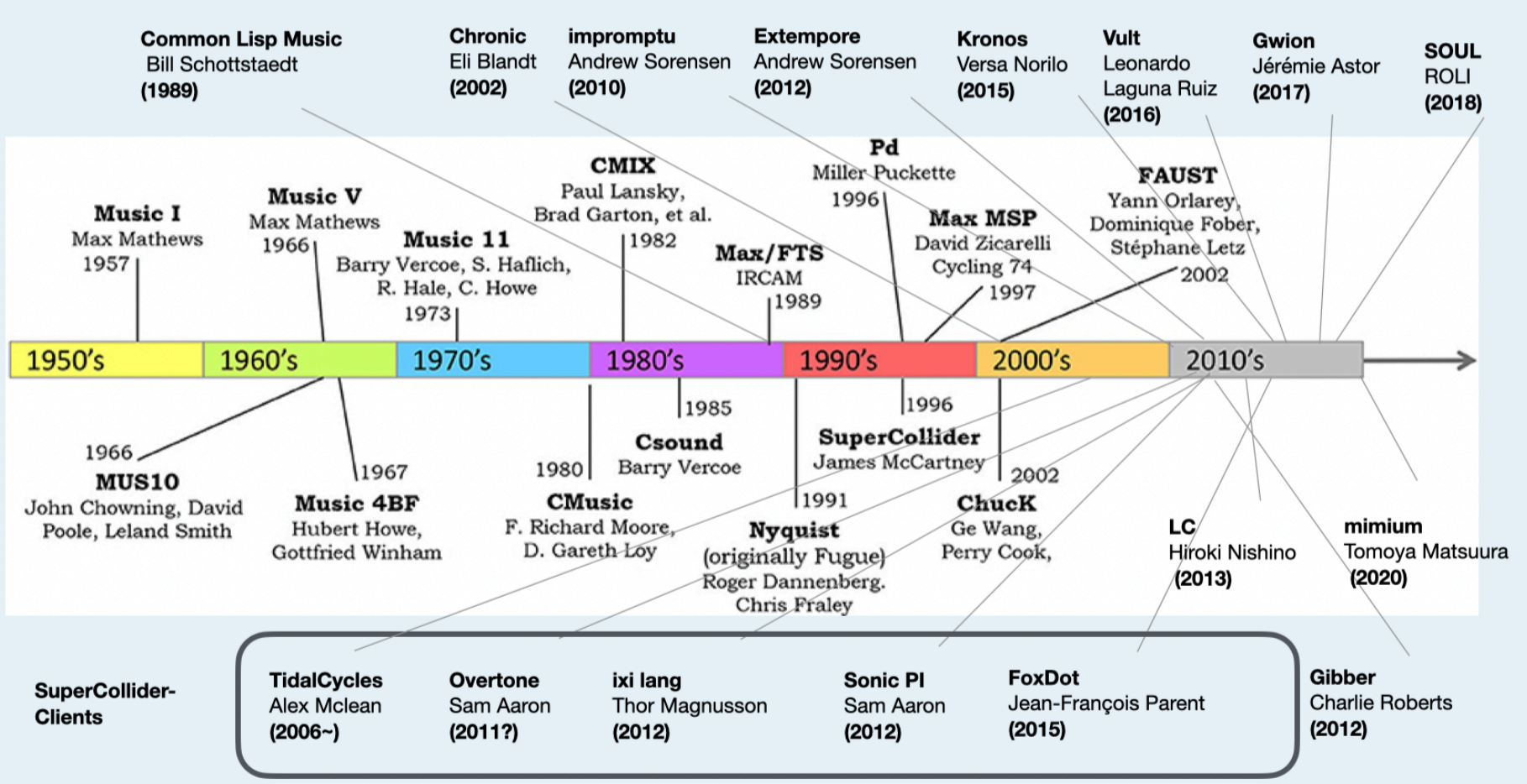
Design Concept

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

	 SuperCollider	 ChucK	FAUST Faust
Strength	Notation flexibility	Sample-accurate control	Strict formal semantics
Weakness	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

λ_{mmm} — Core Calculus

λ_{mmm}: Based on Simply Typed, Call-by-Value Lambda Calculus

単 純 型 付 き 値 呼 び ラ ム ダ 計 算

$$
\begin{array}{rll}
\tau ::= & unit & \\
| & R & \\
| & I_n & n \in \mathbb{N} \\
| & \tau \to \tau &
\end{array}
$$

Types

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

Values

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

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 mimium

```
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 mimium

Unique Features

Live Coding

Live coding in mimium

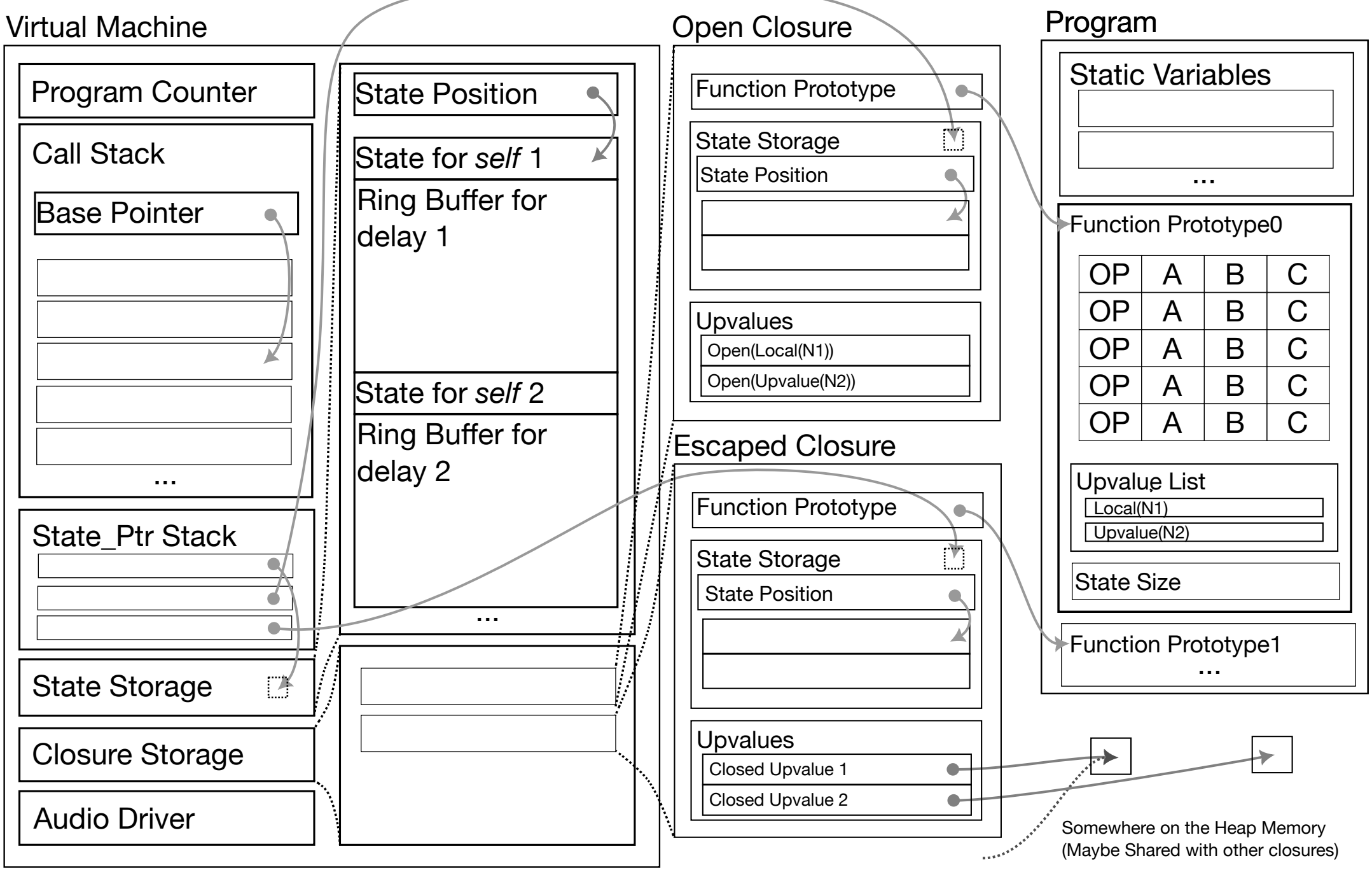
- In mimium native CLI, when user update the code and save, the sound are automatically updated to newer code
- The tail of feedback delay or reverbration are kept naturally on update
- However, it does not modify existing runtime. Instead it compile new source code from zero and **swap entire virtual 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*l%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)
}
```

examples/livecoding-demo.mmm

Implementation

Runtime Representation


Virtual Machine
Program Counter
Call Stack
Base Pointer
...
State_Ptr Stack
State Storage
Closure Storage
Audio Driver
State Position
State for self 1
Ring Buffer for delay 1
State for self 2
Ring Buffer for delay 2
...
Open Closure
Function Prototype
State Storage
State Position
Upvalues
Open(Local(N1))
Open(Upvalue(N2))
Escaped Closure
Function Prototype
State Storage
State Position
Upvalues
Closed Upvalue 1
Closed Upvalue 2
Program
Static Variables
...
Function Prototype0
OP A B C
OP A B C
OP A B C
OP A B C
OP A B C
Upvalue List
Local(N1)
Upvalue(N2)
State Size
Function Prototype1
...
Somewhere on the Heap Memory
(Maybe Shared with other closures)

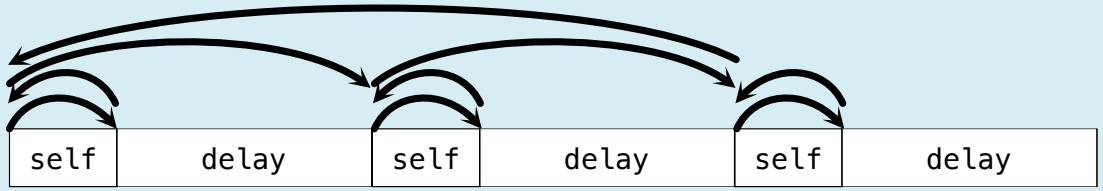
相 対 移 動 ポ イ ン タ モ デ ル
Relatively Moving Pointer Model

```
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 intepreted 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

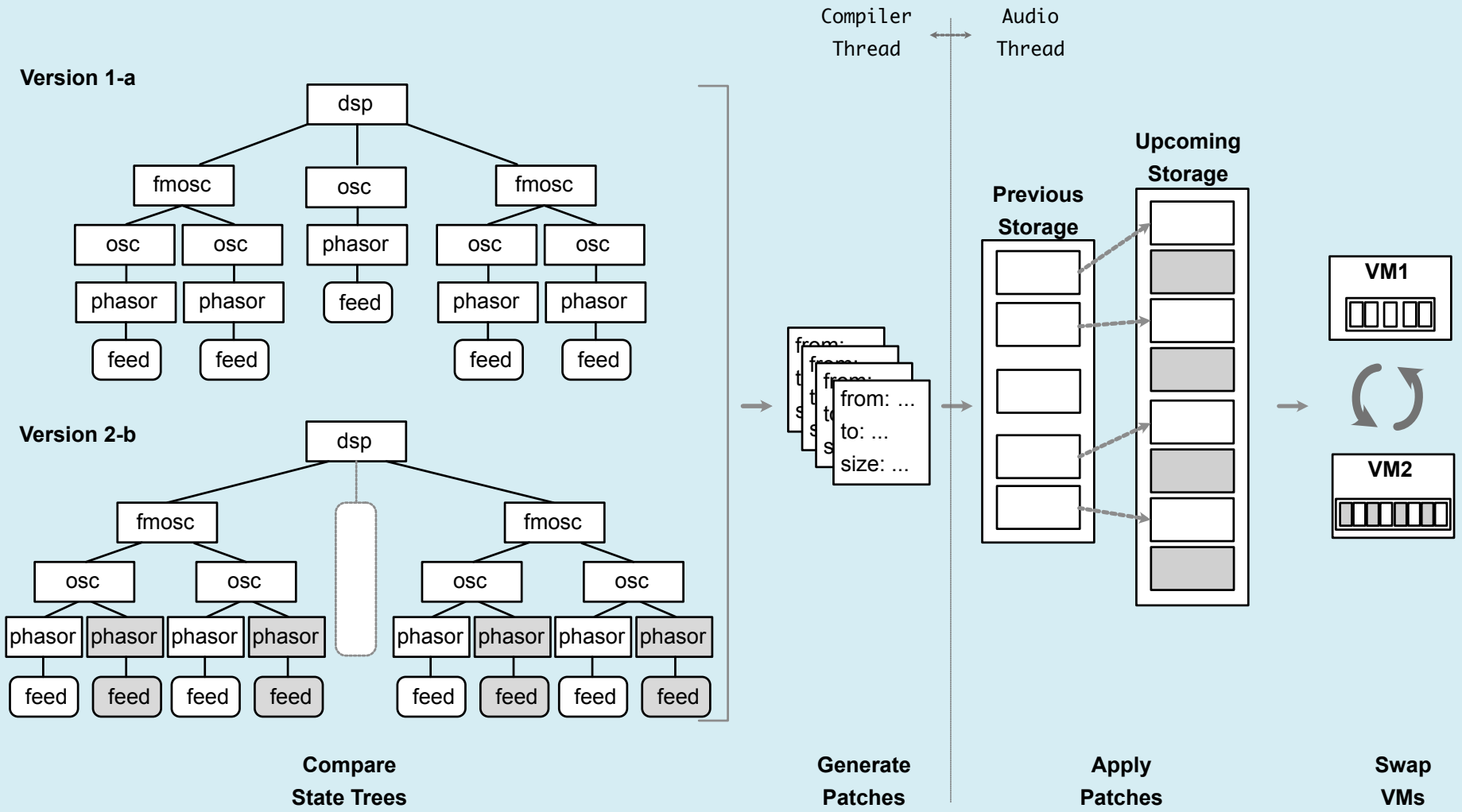
RMP model and Live Coding

```
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 stops

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
}
```



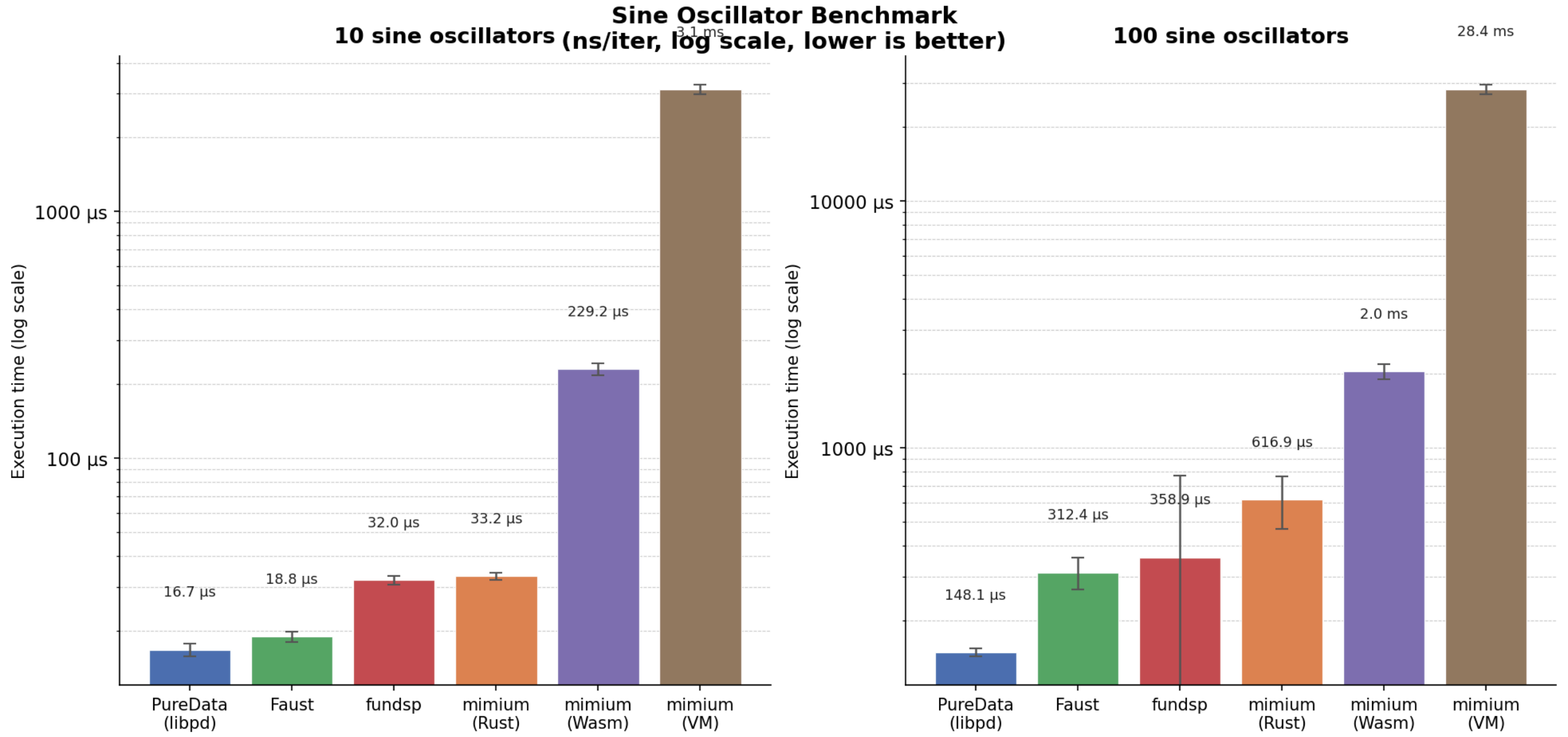
Each of "Patch" is like memcpy operation

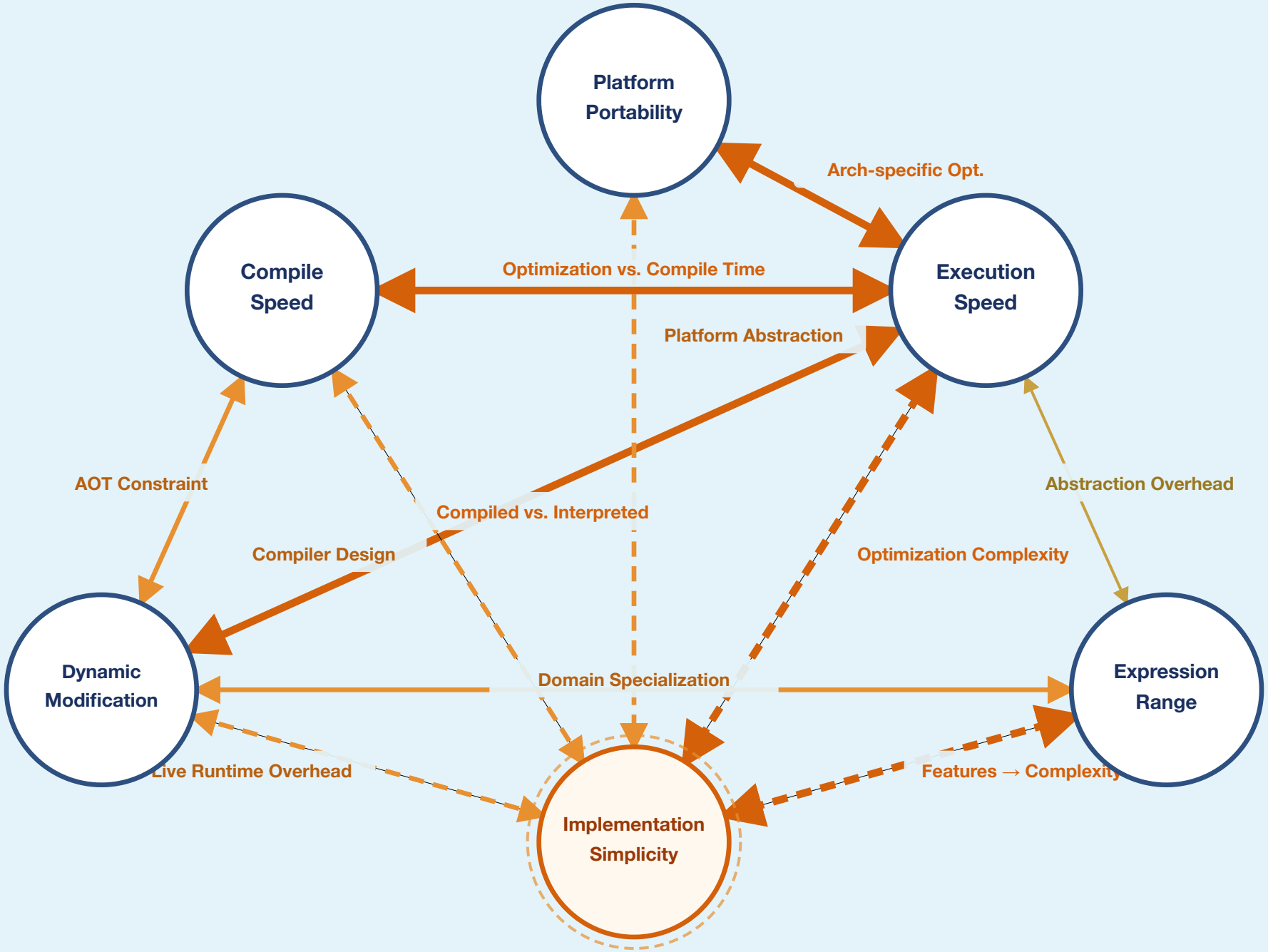
Static Live coding

- 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
(ns/iter, log scale, lower is better)
10 sine oscillators
100 sine oscillators
Execution time (log scale)
1000 µs
100 µs
10000 µs
16.7 µs
18.8 µs
32.0 µs
33.2 µs
229.2 µs
3.1 ms
148.1 µs
312.4 µs
358.9 µs
616.9 µs
2.0 ms
28.4 ms
PureData (libpd)
Faust
fundsp
mimium (Rust)
mimium (Wasm)
mimium (VM)


Platform Portability
Arch-specific Opt.
Compile Speed
Optimization vs. Compile Time
Execution Speed
Platform Abstraction
AOT Constraint
Compiled vs. Interpreted
Abstraction Overhead
Compiler Design
Optimization Complexity
Dynamic Modification
Domain Specialization
Expression Range
Live Runtime Overhead
Features → Complexity
Implementation Simplicity

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 `mimium_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

```rust
#[derive(Default)]
pub struct SamplerPlugin {
    sample_buffers: Vec<Arc<Vec<f64>>>,
    sample_namemap: HashMap<String, usize>,
}
impl SamplerPlugin {
    ...
    #[mimium_plugin_macro]
    pub fn make_sampler_mono(&mut self, v: &[(Value, TypeNodeId)]) -> Value {
        ...
        Value::Code(Self::sampler_record_expr(idx, sample_length))
    }
    #[mimium_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 => {
                mimium_lang::log::error!("Invalid file index: {idx}");
                0.0
            }
        }
    }
```

```
mimium_export_plugin! {
    plugin_type: SamplerPlugin,
    plugin_name: "mimium-symphonia",
    plugin_author: "mimium-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

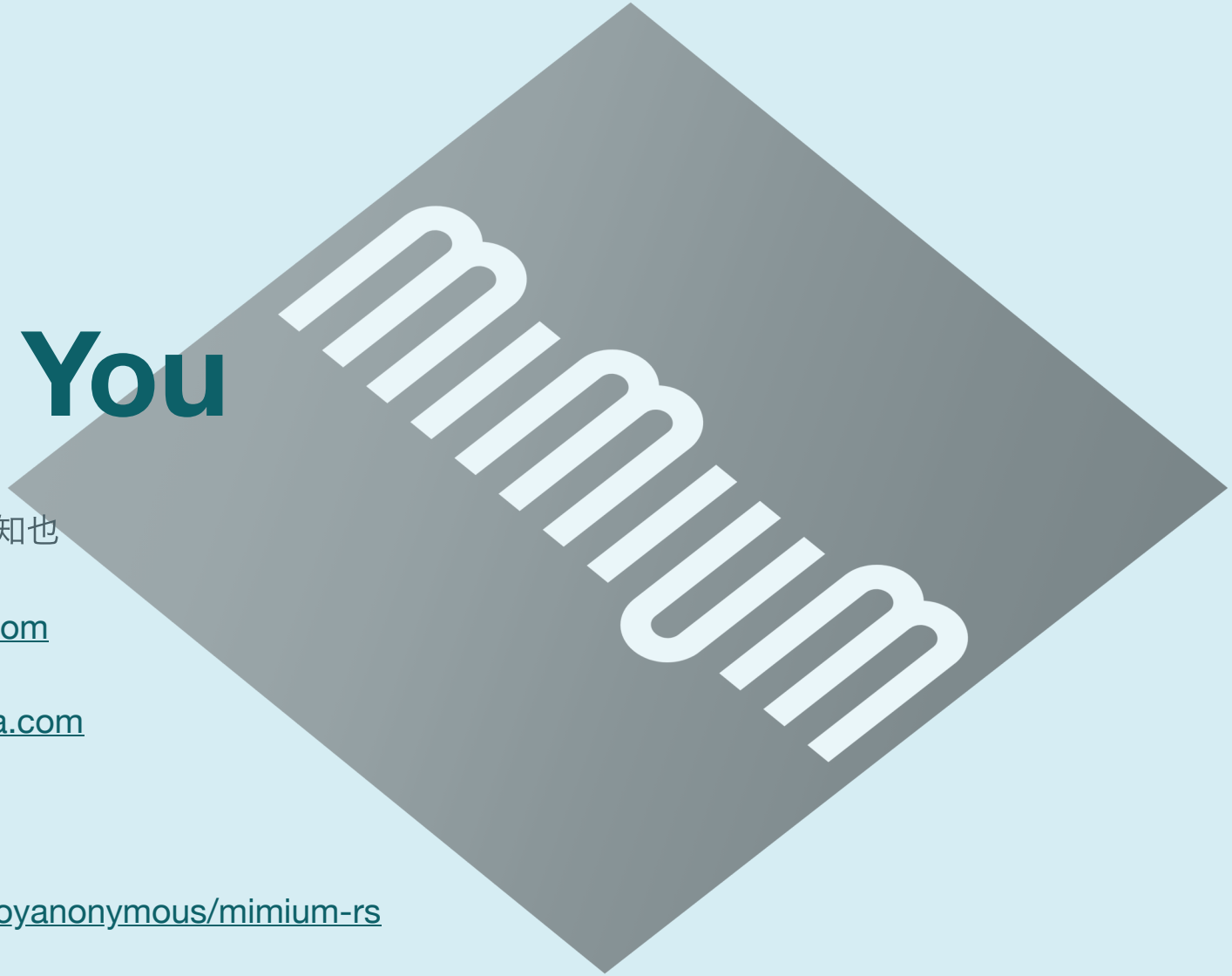
Future development

- 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 λmmm
- 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

https://matsuuratomoya.com

https://mimium.org/

https://github.com/tomoyanonymous/mimium-rs