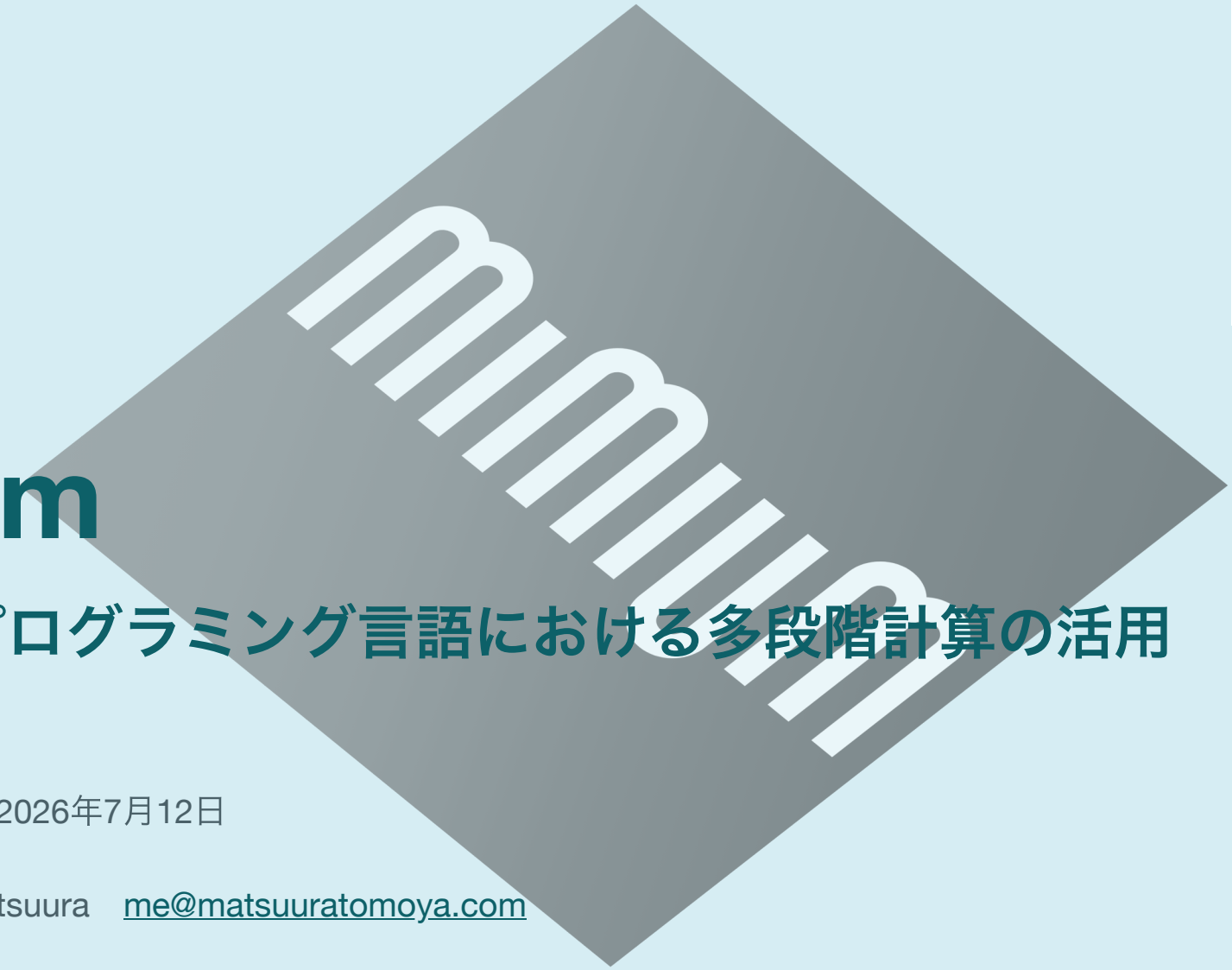


mimium

関数型音楽プログラミング言語における多段階計算の活用

関数型まつり 2026 — 2026年7月12日

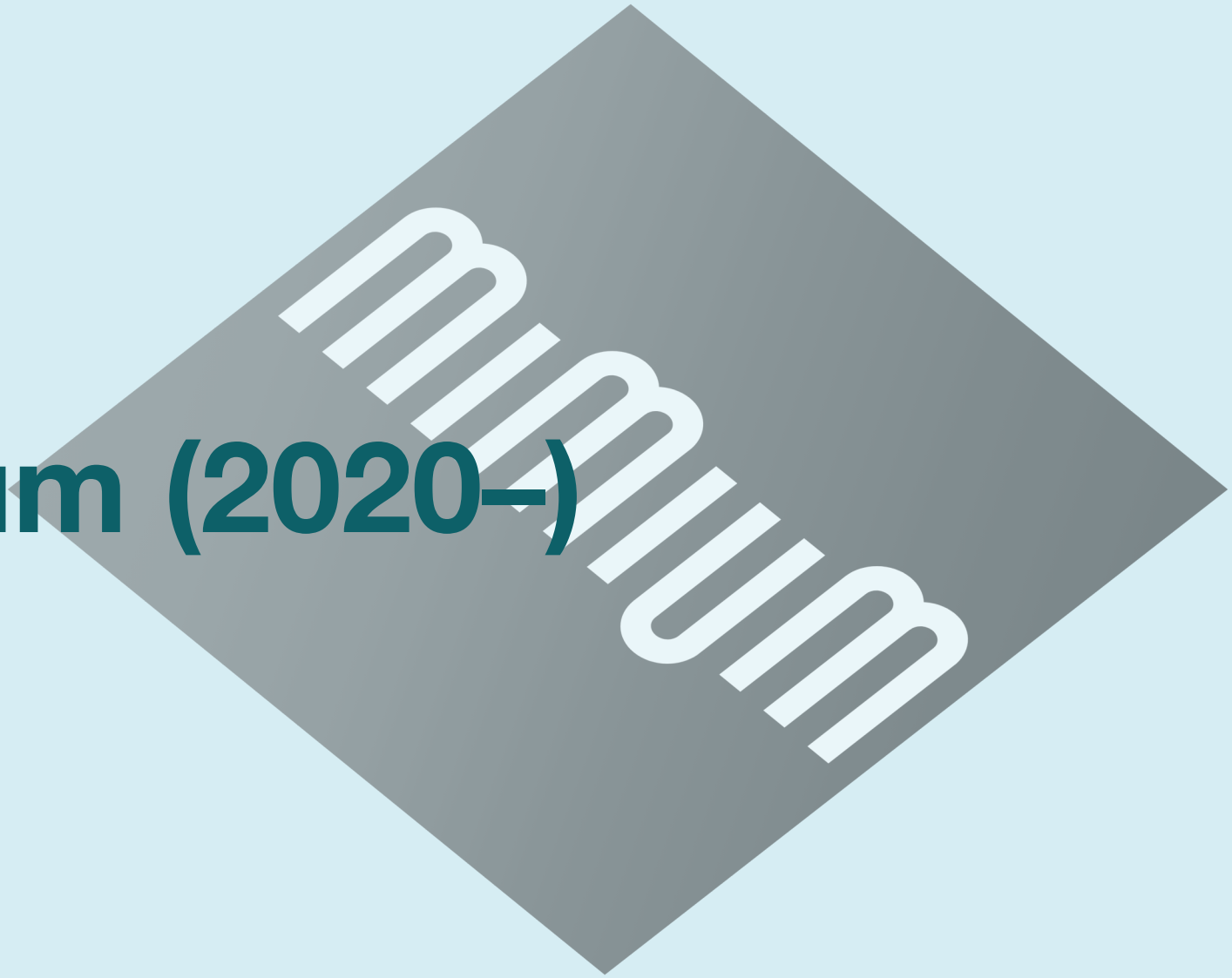
松浦知也 / Tomoya Matsuura　me@matsuuratomoya.com

自己紹介

- 松浦知也 / Tomoya Matsuura
- 2019–2022: 九州大学大学院芸術工学府 修士・博士課程
- 2022–2025: 東京藝術大学 芸術情報センター(AMC) 特任助教
- 2026–: 独立（無職）
- 研究領域: **音楽土木工学**

Overview

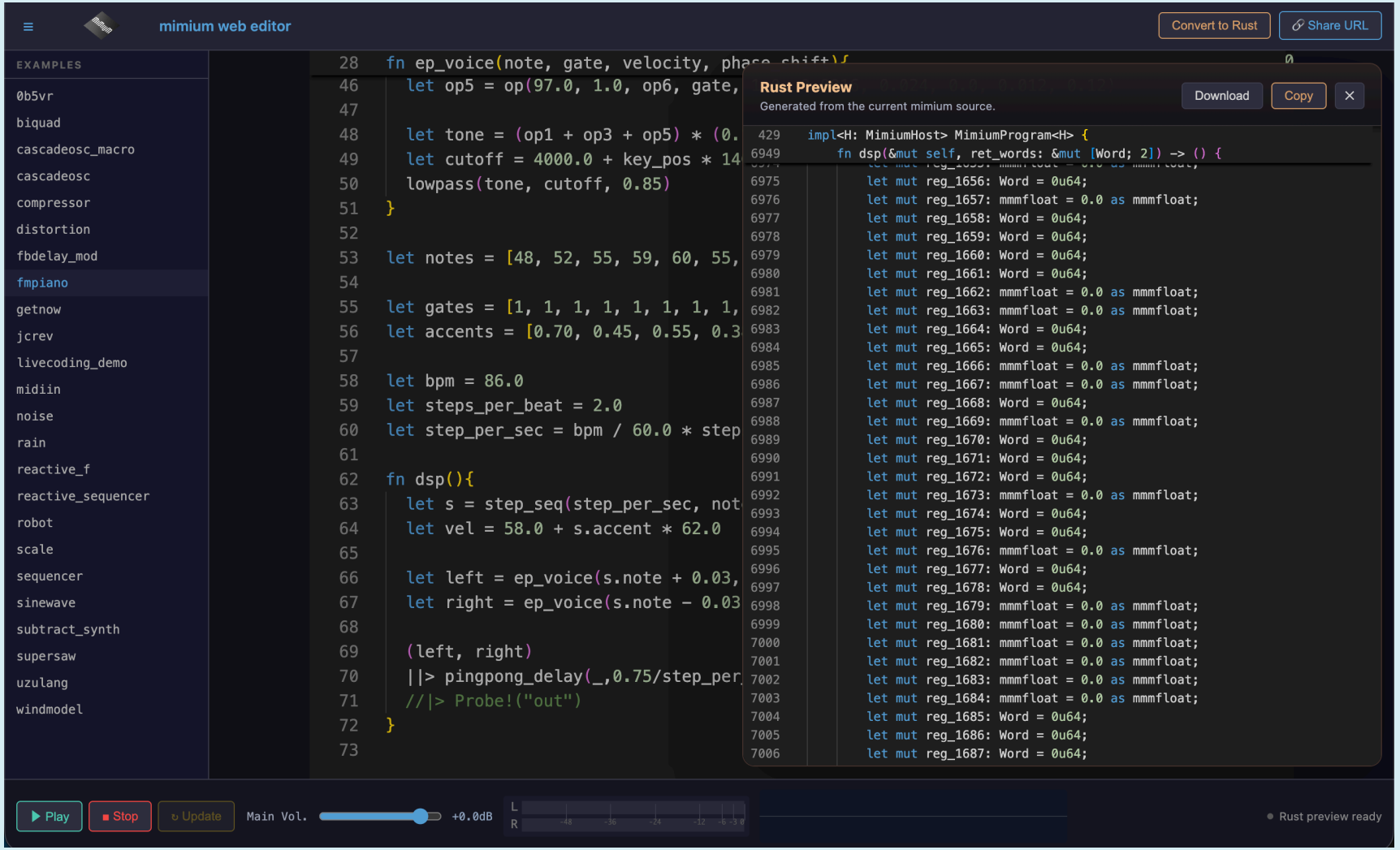
mimium (2020–)



mimiumとは?

- **Mi**nimal **M**usical med**IUM** — 耳👂も由来の一つ
- 汎用言語の上に最小限の音楽向け機能を実装した、信号処理のための関数型プログラミング言語
- Rustで開発 / Rust風の構文(意味論はさらに関数型寄り)
- 複数のバックエンド
  - WASMを介したJITコンパイルによるネイティブ実行(Language Server付きVS Code拡張)
  - ブラウザ上で動作 (https://mimium.org https://mimium-org.github.io/mimium-web-editor/)
  - Rustへのトランスパイル
  - VST/CLAPプラグイン(開発中)

mimium on Web Browser



本日の内容

- 動機と言語設計
- コア計算体系 λmmm とコード例
- **多段階計算(型付きマクロ)**
- **ライブコーディング機能とその実装**
- まとめ

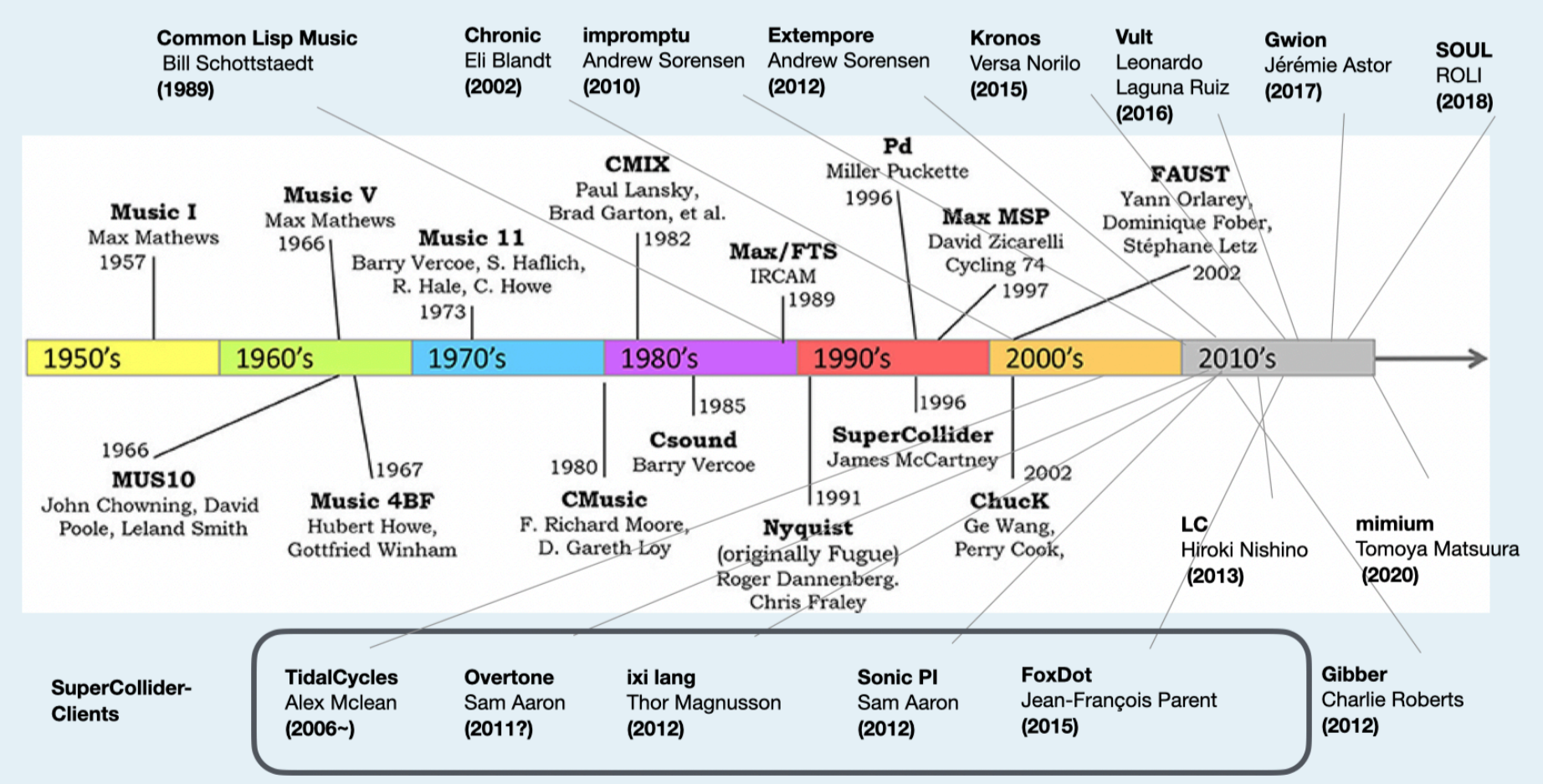
Design Concept

音楽プログラミング言語は
もっとシンプルでよいのではないか

Design Concept

音楽プログラミング言語は ~~もっとシンプルでよいのではないか~~ もっと普通のプログラミング言語に近い作り方にできないのか?

音楽プログラミング言語の歴史



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

なぜ新しいDSLを作るのか?

- 汎用言語では、本質的な音声処理の内容に加えて余計な記述が必要になる
  - メモリ管理
  - スレッド管理
  - ハードウェアの隠蔽
- そもそもその言語の表現力が十分に高ければ、そうした詳細はライブラリとして隠蔽できてほしい

"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

既存の汎用言語ではだめなのか?

- 残念ながら2020年代現在でも汎用言語のライブラリとして音声処理をするのは簡単ではない
  - C++やRustはハードウェア管理がユーザーコードに露出するので記述が煩雑
  - 一方でメモリ管理が隠蔽されている言語ではGCのタイミングをユーザーが制御で機内
- ライブコーディングのための、実行中コードのホットスワップをしたい

そこで：**汎用プログラミング言語だが音声処理を第一目的としたもの**を作るのはどうか

既存の言語はどうか

	 SuperCollider	ChucK	FAUST Faust
強み	記法の柔軟性	サンプル精度の制御	厳密な形式意味論
弱み	移植性が低い	UGenが第一級でない	exoticな記法
	巨大なコードベース	ライブコーディングでディレイ/リバーブが途切れる	ライブコーディング不可

目標: もっと「普通の」プログラミング言語っぽく

- JSやRustに近い読みやすい構文
- Unit Generatorを特別扱いしない — 信号専用のプリミティブ型を持たない
  - 最小限のステートフルなプリミティブ演算があれば、UGenはクラスではなくただの関数になる
  - プリミティブ演算: **DelayとFeedback**
  - これはすでにFaustが証明している
- リアルタイム安全性のため、基本的にGCなし(ほぼ全ての方がコピー渡し、関数や配列など一部の型は参照カウント)

方針：ラムダ計算をベースにしつつ、Faustと同等の記述ができるような関数型言語

Language Design

λ_{mmm} — コア計算体系

λ_{mmm}: 単純型付き・値呼びラムダ計算の拡張

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

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

単極フィルタ(積分器)

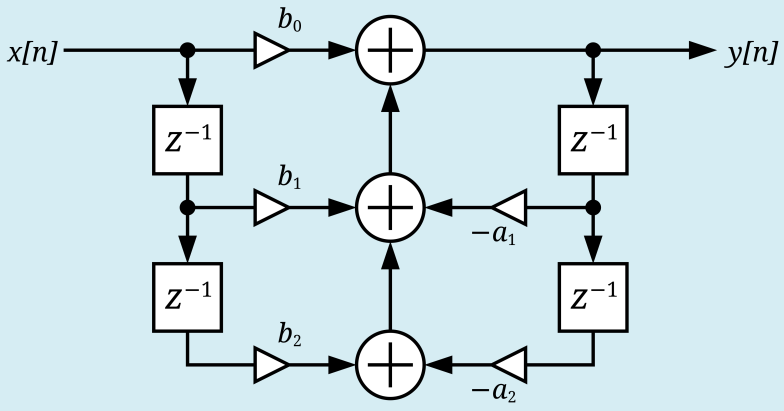
```
fn onepole(x, ratio){
    x*(1.0-ratio) + self*ratio
}
```

$$let\ onepole\ =\ \lambda x.\ \lambda ratio.\ feed\ y.\ x * (1.0 - ratio) + y * ratio\ in\ \ldots$$

- `self`は0で初期化され、その関数の1サンプル前の返り値を参照する
- 信号処理で本質的かつ複雑になりがちなフィードバック結線を明示的に扱える
- 関数はステートフル: 毎サンプル異なる値を返すが、その列は決定的
  - 時系列的に見れば参照透過といえる

BiQuadフィルタ

```
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`は多相的(その関数の返り値型と同じ型になる)

フィードバックディレイ

```
fn fbdelay(input, time, fb, mix){
  input * mix + delay(40001, input + self * fb, time) * (1 - mix)
}
```

- `delay`は組み込み関数。引数は3つ: 最大ディレイ長、入力信号、ディレイ長
- 最大ディレイ長はコンパイル時定数でなければならない（ここを可変にしたければ後述のマクロで解決）

サイン波オシレータ

```
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`はランタイムが定義する環境変数
- パイプ`|>`は`a(b)`を`b |> a`と書く記法 — 関数型的な信号のデータフロー記述と相性が良い

Key Insight

UGen = 関数
高階関数による
プロセッサへのメタ操作

高階関数の活用: 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
        }
    })
}
```

Typed Macro

多段階計算
― 型付きマクロの計算体系

多段階計算(Multi-stage Computation)とは

- 「コードを生成するコード」を言語内で**型安全に**書くための体系
- Lispのquasiquote/unquoteの型付き版と考えると分かりやすい
  - Quote(`` ` ``): 式を評価せず「コード値」として扱う
  - Splice(`$`): コード値をコードの中に埋め込む
- MetaML [Taha & Sheard 1997] に始まる型付き多段階言語の系譜
- 実用的には、MetaOCamlやScala3以降のマクロ、SaTysFiなど

λ_{Mmmm}: 多段階計算への拡張

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

Types

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

Values

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

Terms

ステージ付きの型付け規則

型判断にステージ n を明示: $\Gamma \vdash^n e : \tau$　(λmmmの型に $\mathrm{Code}\,\tau$ を追加)

$$\frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^{n} \ `e : \mathrm{Code}\,\tau}\ [\mathrm{quote}] \qquad \frac{\Gamma \vdash^{n} e : \mathrm{Code}\,\tau}{\Gamma \vdash^{n+1} \ \$e : \tau}\ [\mathrm{splice}]$$

$$\frac{x : \tau @ n \in \Gamma}{\Gamma \vdash^{n} x : \tau}\ [\mathrm{var}] \qquad \frac{\Gamma \vdash^{n} e : R}{\Gamma \vdash^{n} \ \mathrm{lift}\ e : \mathrm{Code}\ R}\ [\mathrm{lift}]$$

- $x : \tau @ n$ = 環境Γ中で「変数xはステージnで型τを持つ」という束縛
- Quoteの中身は「1つ先のステージ」の式として型付けされる
- 変数は束縛されたステージでのみ参照でき、クロスステージ参照は`lift`で明示

mimiumのステージ構成

- ステージ0 = マクロ(コンパイル時) / ステージ1 = 実行時(信号処理)
- `#stage(macro/main)`アノテーションで命令型っぽくステージを切り替え（実際にはクォートとスプライスの糖衣構文）
- コンパイル時に任意の計算(ループ・再帰・データ構造操作)を実行できる
- `foo!(bar)`はマクロ呼び出し`$(foo(bar))`の糖衣構文

```
#stage(macro)
let numbers = detune_table |> len |> lift  // ステージ0で計算し、コードへ持ち上げ
fn supersaw(){
    `|freq,detune|{ saw(freq,0) / $numbers } // 引用の中はステージ1の式
    ...
}
```

マクロの実行 — Code Combinatorの応用

- マクロ展開時、クォート式は**AST(コード値)を組み立てる操作の呼び出し列**に変換される(スプライスも消える)
  - Lispのquasiquoteが`cons`/`list`に展開されることの型付き版 = **Code Combinator**

```
`{ saw(f, 0) }
// ↓ 型検査後の変換(イメージ)
mkapp(mkvar("saw"), [mkvar("f"), mklit(0)])
```

- 変換後はクォートを含まないただのプログラム（組み立て操作の実体は何でもよい）
- mimiumでは**AST操作をVMの組み込みプリミティブとして追加**するだけ（マクロは**実行時と同じレジスタマシンVM**で普通のプログラムとして走るので専用のインタプリタ不要）

Cf. Kiselyov, "MetaOCaml: Ten Years Later" (SCP, 2026) / Kameyama, Kiselyov & Shan, "Combinators for Impure yet Hygienic Code Generation" (2015)

型付きマクロの何がうれしいのか

- **型検査・型推論がマクロ展開の前に完了する**
  - 展開後のコードが謎の構文エラー・型エラーを起こさない
  - Cプリプロセッサや、Faustのマクロ的機能でよく知られた問題を回避
- コンパイル時計算はさまざまな方向に拡張できる
- たとえば、テキストをパースするプログラムをコンパイル時に実行したら?
  - mimiumの上にさらに**別のDSLを埋め込める**!

uzulang(mini-notation) — mimium上のDSL

```
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)
}
```

`mini`関数の中の文字列は、mimium上に埋め込まれたDSL。パーサコンビネータ自体がmimiumで書かれている

Unique Features

ライブコーディング

mimiumのライブコーディング

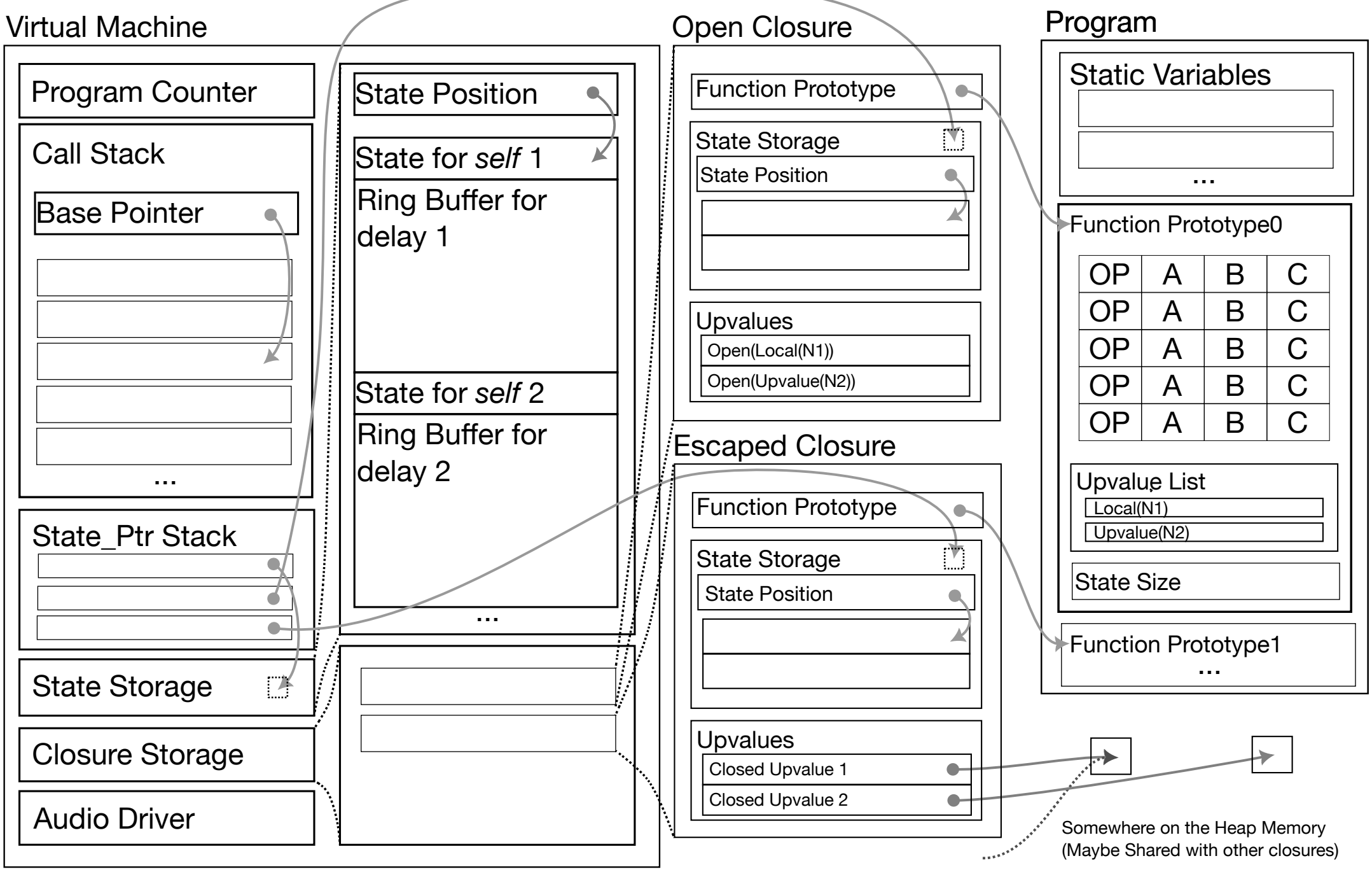
- ネイティブ版(CLI / VS Code拡張)では、実行中にコードを編集して保存するだけで音が更新される
- フィードバックディレイやリバーブの残響は、更新をまたいで自然に保持される
- ただし既存のランタイムを変更するのではなく、新しいソースコードをゼロからコンパイルし、状態変数を可能な限りコピーしながら**仮想マシンを丸ごと入れ替える**
- 「静的」ライブコーディング

```
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

ランタイムの内部表現


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)

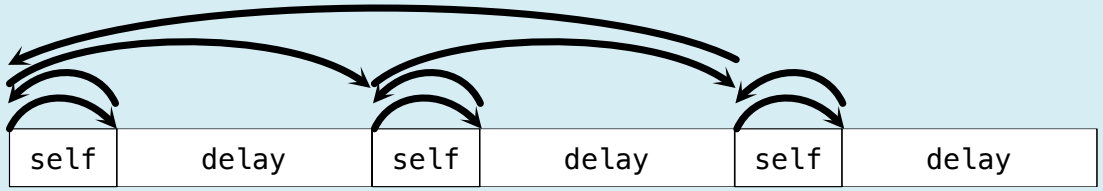
相対移動ポインタ（RMP）モデル

```
fn fbdelay(input, time, fb, mix){
    input * mix + delay(40001, input + self * fb, time) * (1 - mix)
}
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)` はコンパイル時に展開されるパイプと部分適用
  - `$(a |> |x| `foo($x,b,c))` の糖衣構文：すなわち `foo(a,b,c)`
- 3つの `fbdelay` はそれぞれ別のインスタンスとして解釈される必要がある

```
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モデルとライブコーディング

- 状態は関数呼び出し木の順序に沿って並んでいる
- **木構造データは構造的に比較できる**
  - 最長共通部分列(LCS)アルゴリズム(ReactのVirtual DOM差分と同様)
- 2つのバージョン間で、ルートの`dsp`関数の呼び出し木の差分を取る
- 使い回せる状態データを新しいState Storageへコピーする「パッチ」を生成

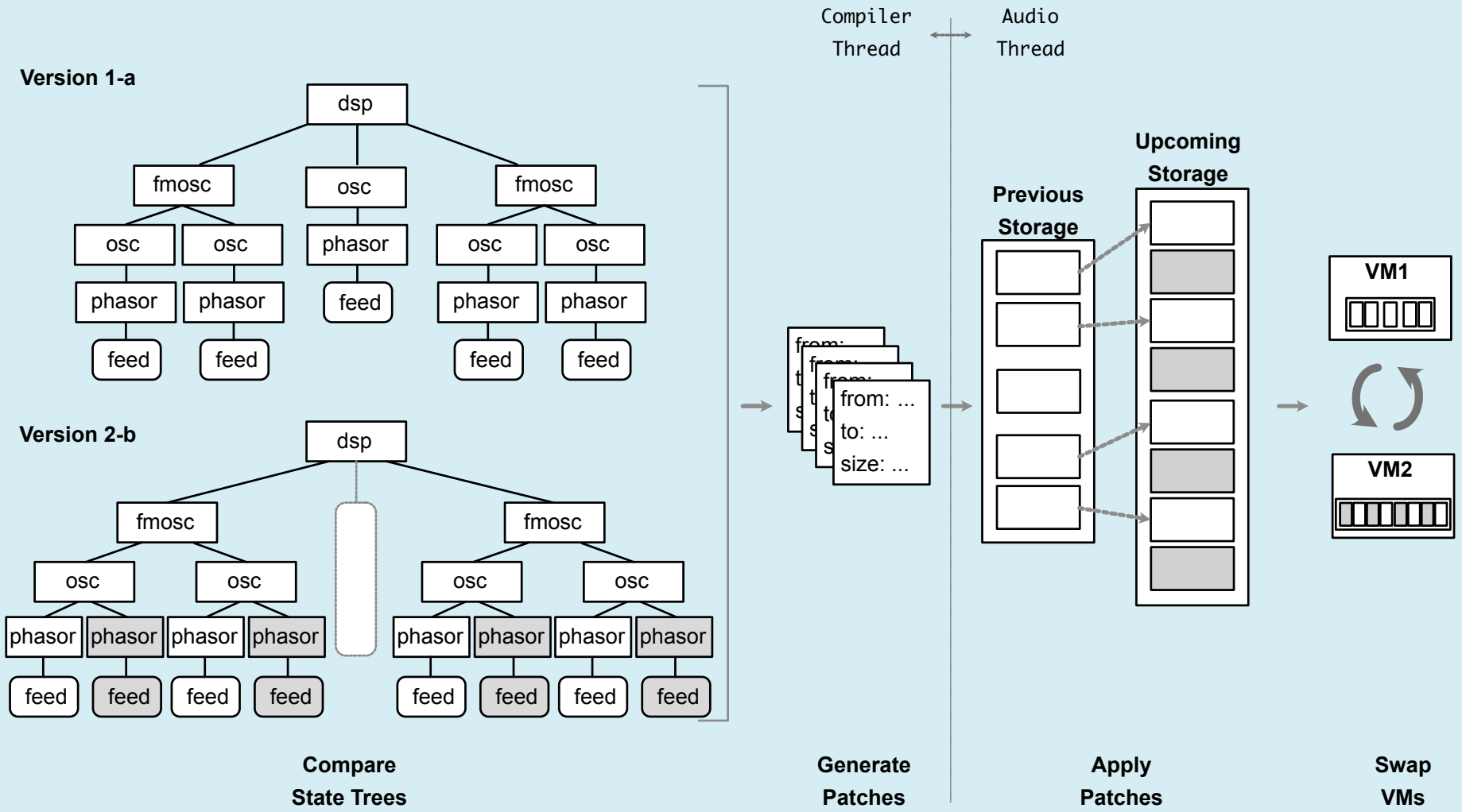
RMPモデルとライブコーディング

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

- 呼び出し木の縮約版が導出できる
- 再帰関数の呼び出しはこの木から除外できる。再帰・高階関数はクロージャ(関数のインスタンスのようなもの)を生成し、クロージャの状態操作はインスタンス自身の上で行われるため、木の走査は必ず停止する

コード例

```
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
}
```



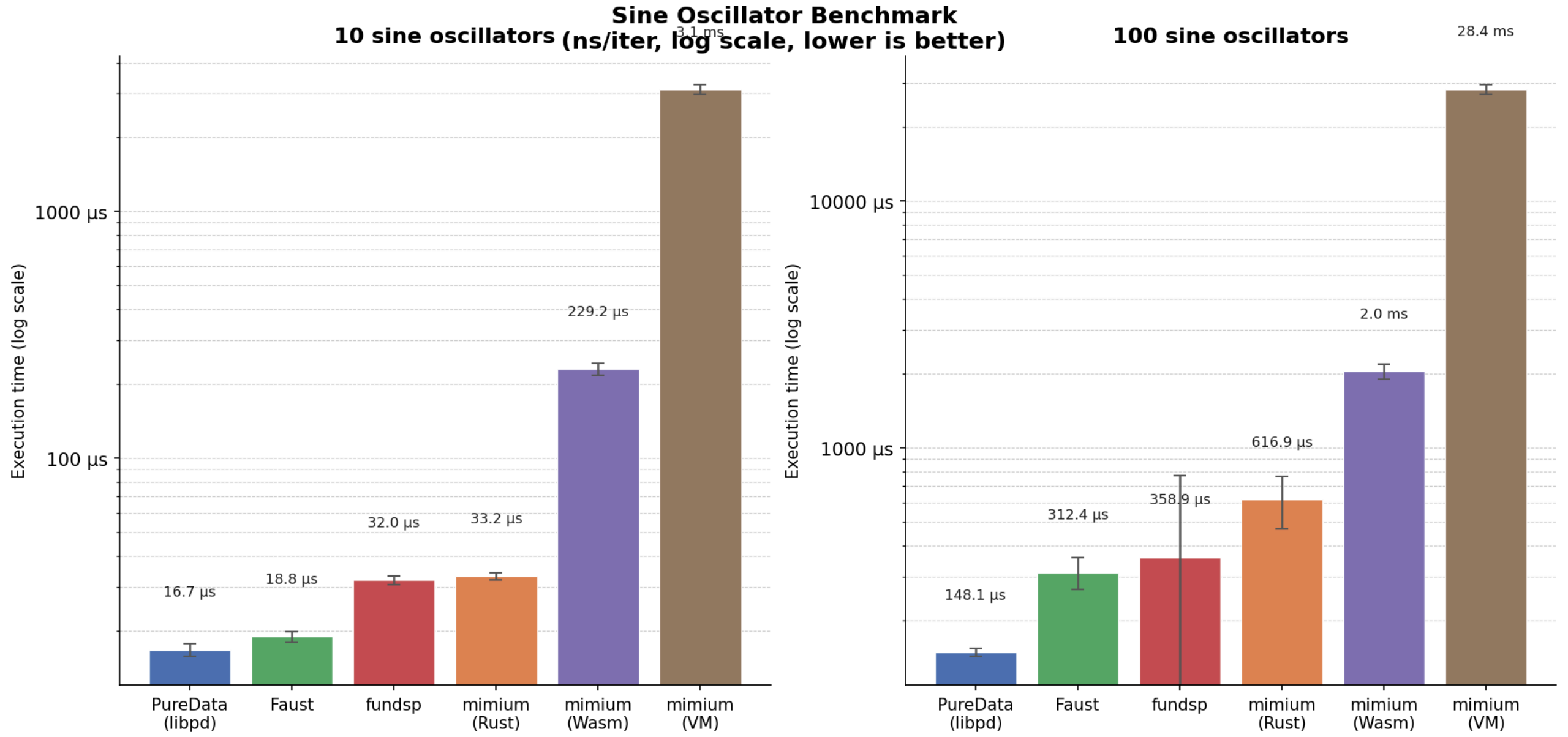
それぞれの「パッチ」はmemcpyのような操作（コピー元、コピー先、データサイズ）

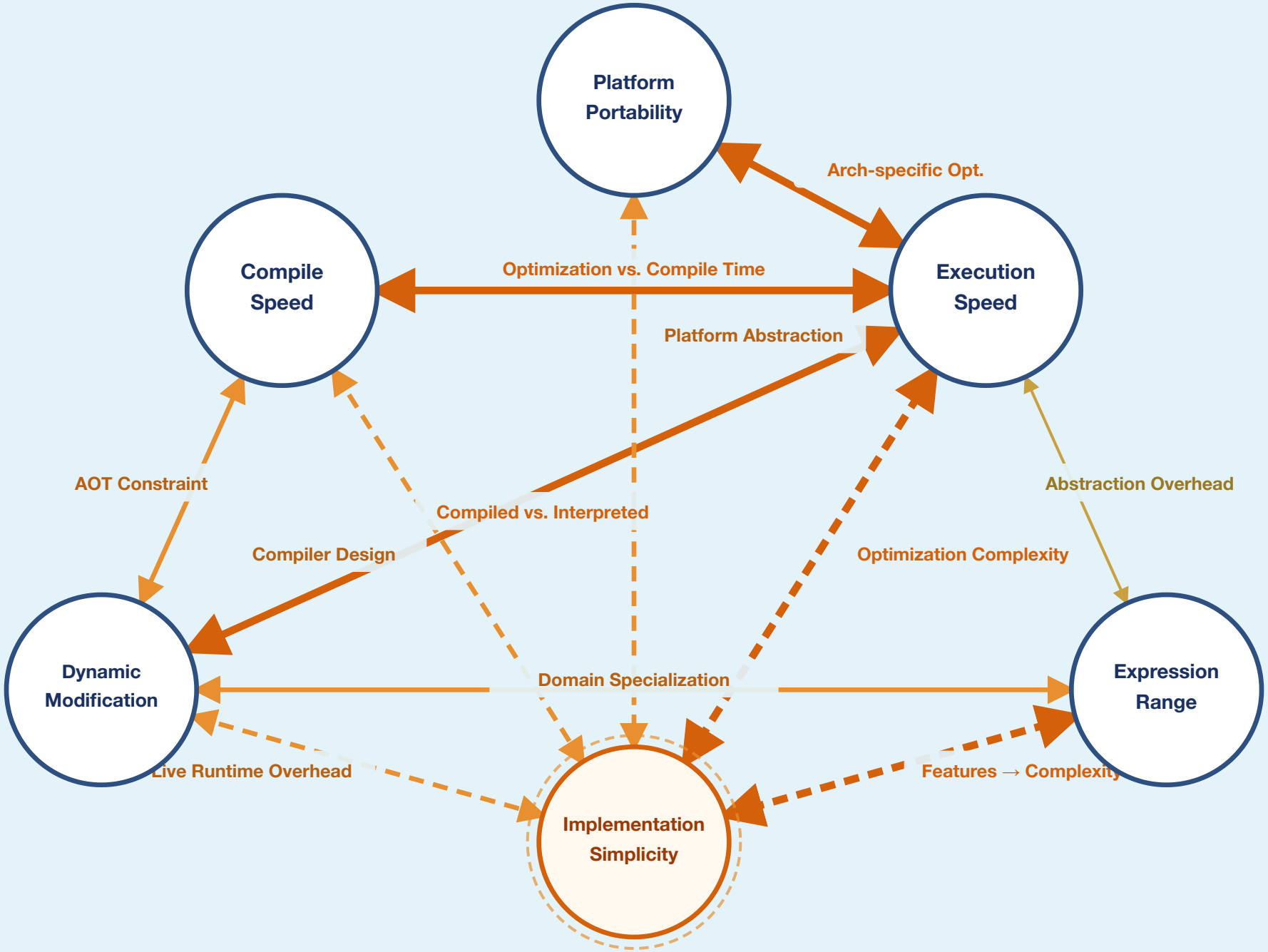
静的ライブコーディング

- エディタとの連携なしに、テキストソースの比較だけで完結する
- パッチ適用の瞬間までオーディオスレッドをブロックしない
- ブロック時間はコード全体のサイズに関係なく、あくまで**コードの構造的な差分**に依存する
- ランタイム実装の単純さを保てる（トランスパイラもそのまま動く）
- ユーザーは**ランタイムの状態ではなく、いま書かれているアルゴリズム**に集中できる

Measurement

パフォーマンスはどうなのか?


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

Conclusion

今後の展望

言語デザインの振り返り

- Multi-purpose: あらかじめ役割が決められた問題解決の道具ではなく、探索の道具としてのプログラミング言語
- 少なくとも自分にとっては、実用的な道具になりつつある
- プログラミング言語理論とDSP理論の交差点には、まだ探索の余地がたくさんある
- Agentic Codingは言語開発と特に相性が良い

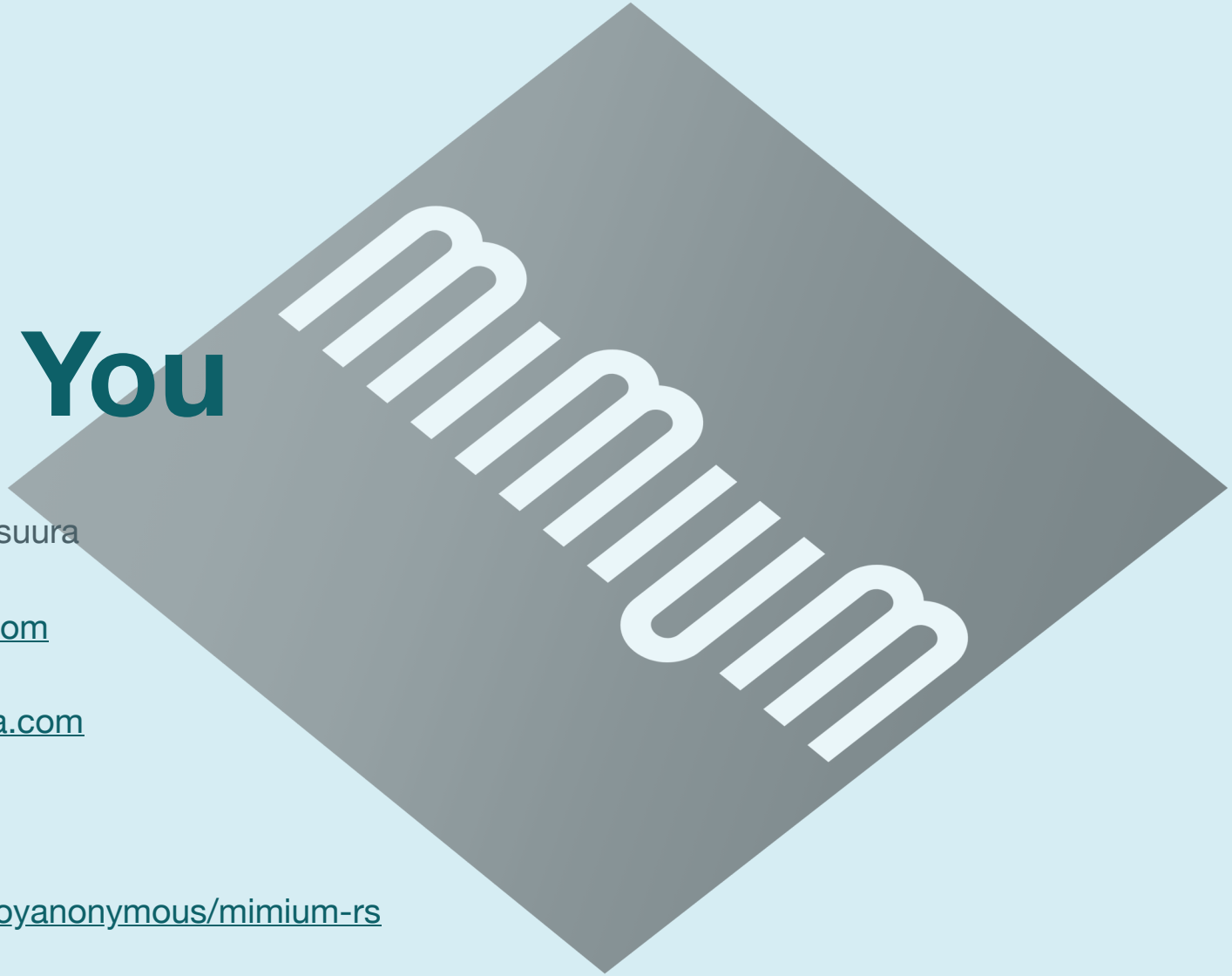
今後の開発

- Rustおよび他言語へのトランスパイラの発展(マイコンターゲットを含む)
- モジュールシステムの改善、パッケージマネージャ?
- より洗練された多相性(型クラス?)
- GUI統合とプラグインシステムの改善
- よりマクロな時間構造(作曲)のための意味論の開発

まとめ

- mimiumはλmmmに基づく**関数型音楽プログラミング言語**
- UGenは第一級の関数 — 高階関数による合成が可能
- 多段階計算に基づく型安全なマクロで、上に別のDSLを構築できる
- コードの構造的比較による静的ライブコーディング
- Available at https://mimium.org/ / https://github.com/tomoyanonymous/mimium-rs

Thank You



松浦知也 / Tomoya Matsuura

me@matsuuratomoya.com

https://matsuuratomoya.com

https://mimium.org/

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