

# **Moodler: A Digital Modular Synthesiser with an Analogue User Interface**

Dan Piponi

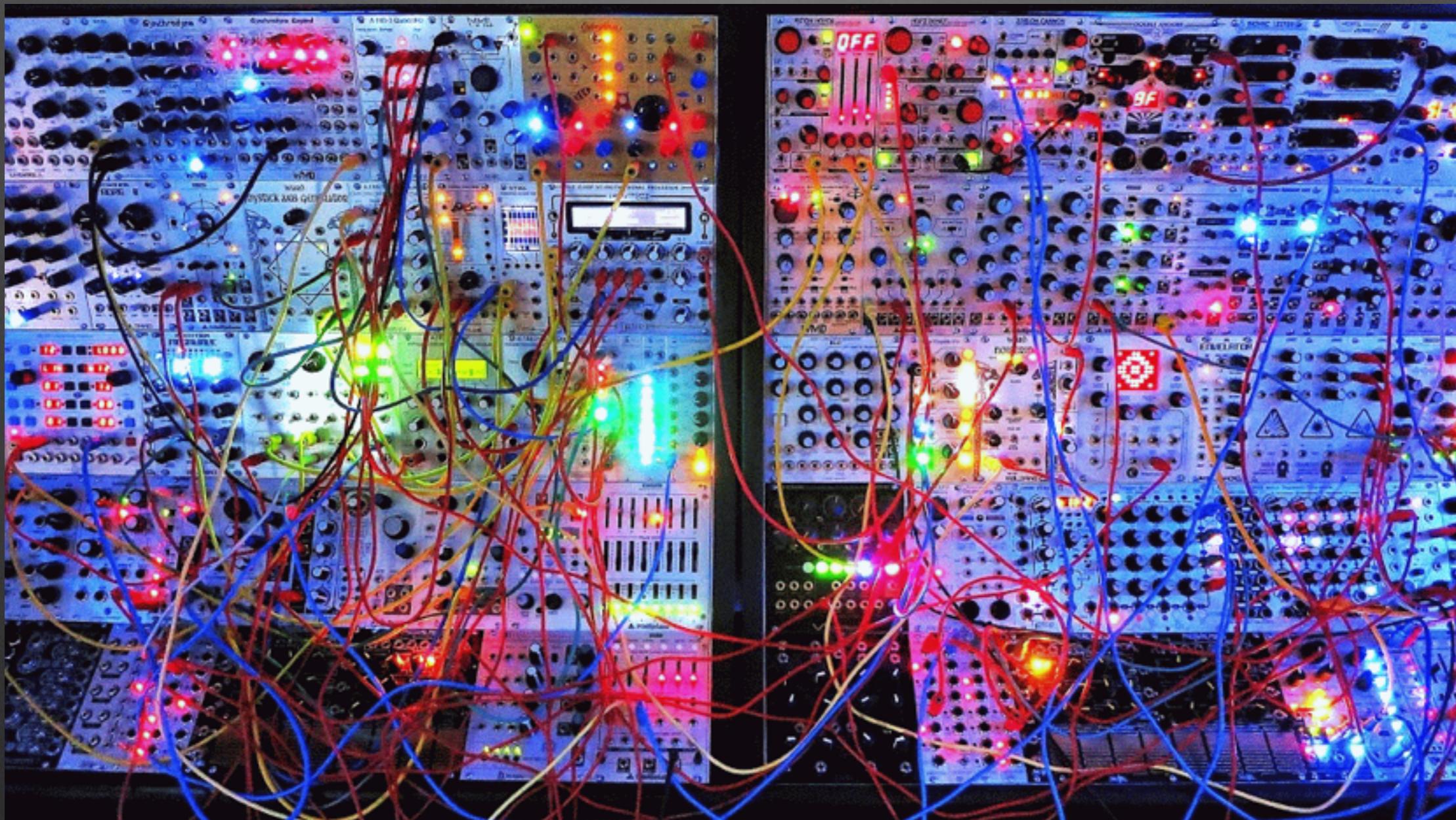
# Two starting points

Number 1:

LittleBits Synth Kit As a **Physically-embodied, Domain Specific Functional Programming Language**  
Noble, James and Jones, Timothy.

# Two starting points

Number 2:



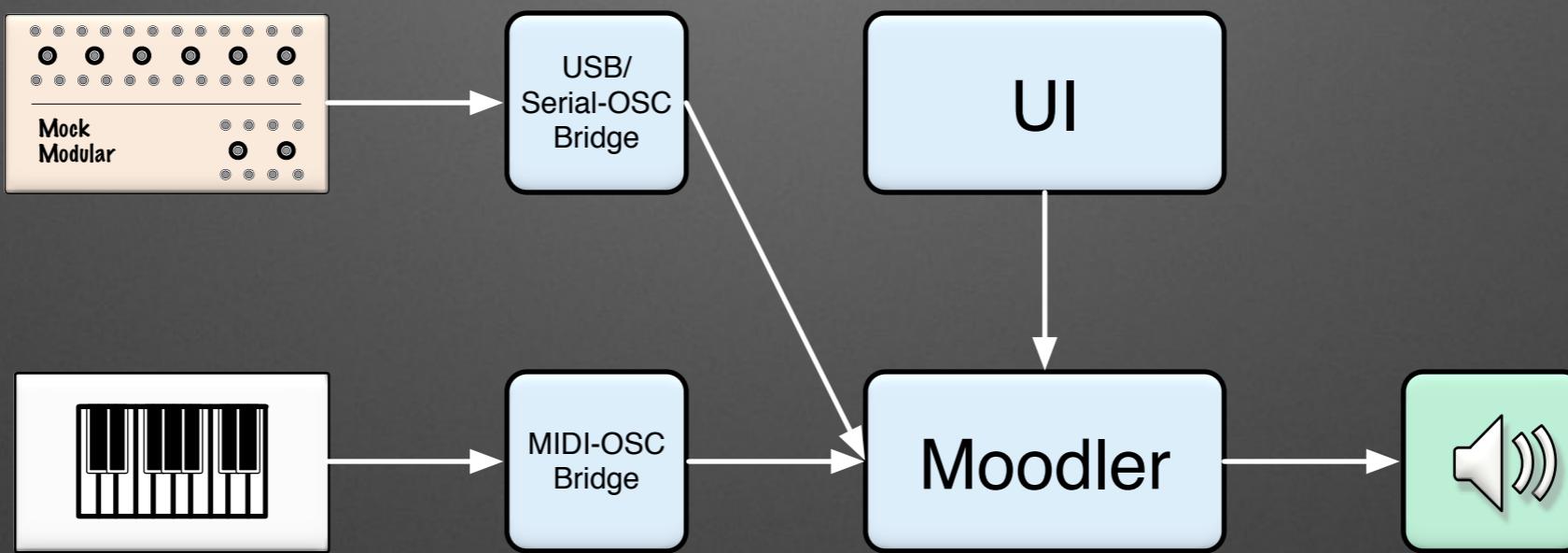
# Let's dive in...

- Standard synth example (`demo_test_full_synth`)
- Stand alone multisaw example (`test_demo_multisaw`)

# Let's dive in...

- Physical and virtual user interfaces
  - Cables and knobs
  - MIDI
  - OSC
- GUI written entirely in Haskell
- Back end written almost entirely in Haskell generating, compiling and linking C code on fly.

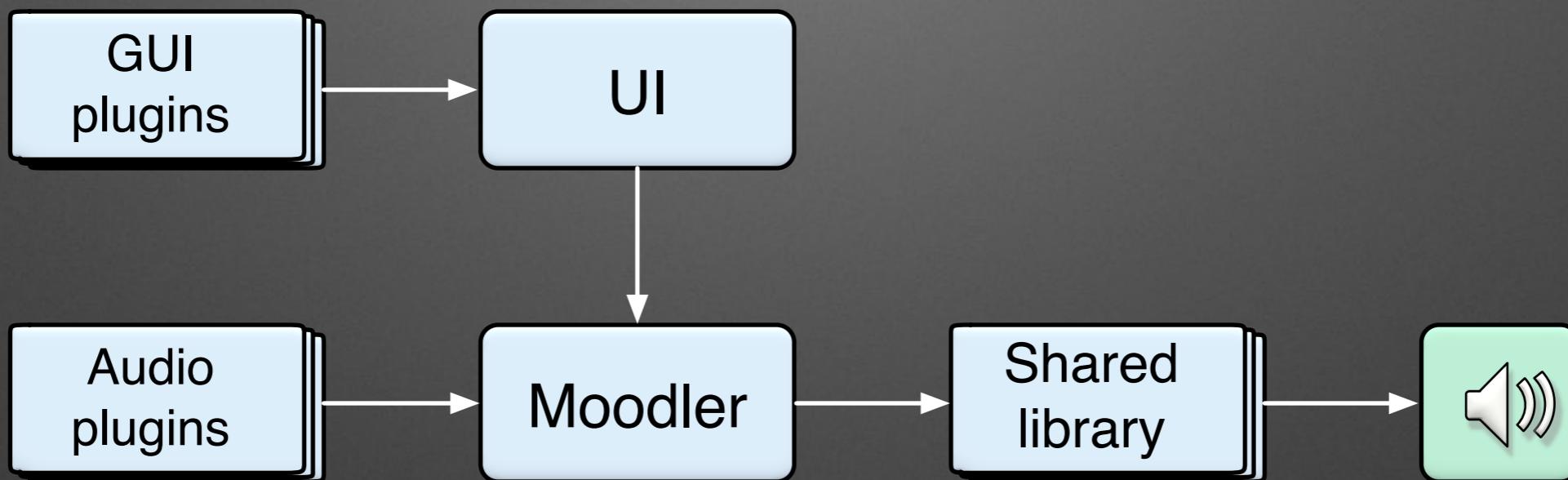
# Overview



# Decisions, decisions...

- Want to exploit existing libraries
- Want graphic user interface
- Want to talk various protocols: USB/Serial, MIDI
- Want fast generation and compilation of fast code

# Code Structure



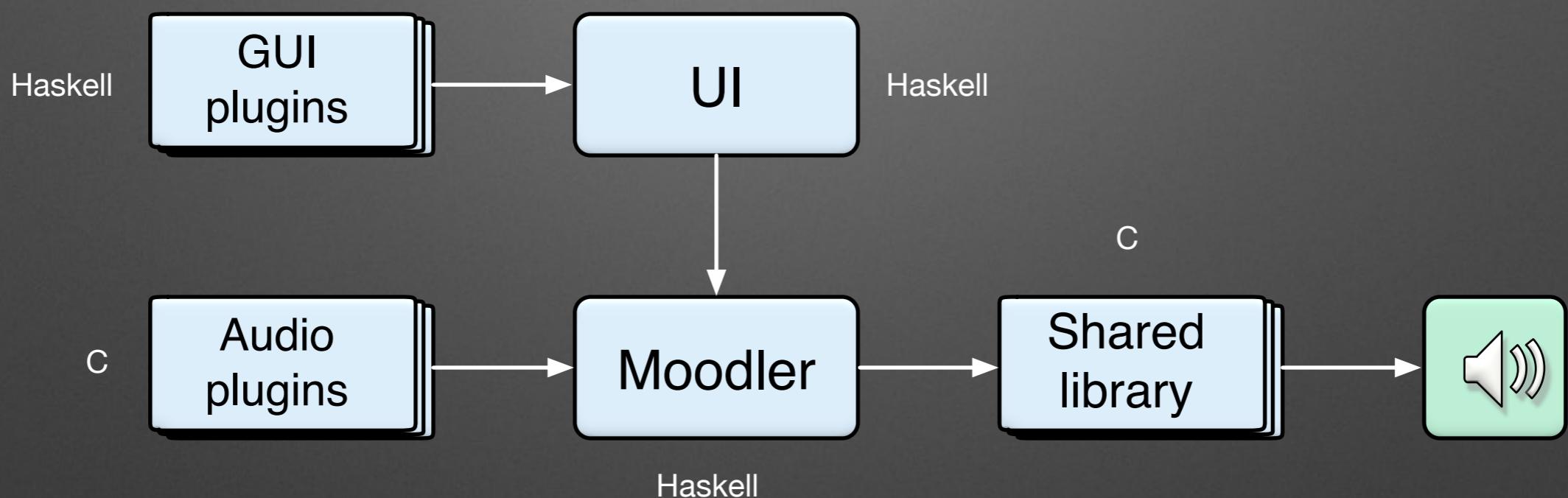
# Haskell

- I can program in it
- Great fit for code generation, FFI, dlopen, C parser and representation, GHC accessible through library.
- Don't quite trust it for devices other than network. But OSC rests on TCP/IP so delegate MIDI and USB/Serial to external applications and audio to C.
- Not convinced by existing GUI offerings but I don't mind drawing everything myself: gloss. Not perfect fit but does what it promises well.

# C

- I can program in it
- If I used LLVM I'd still need to write a compiler to generate LLVM. I think of clang as an API to generate LLVM.
- Plenty of existing C audio code to borrow.
- I want to eventually generate standalone but hackable code for microcontrollers.

# Code Structure



# .msl Plugins

```
double result;

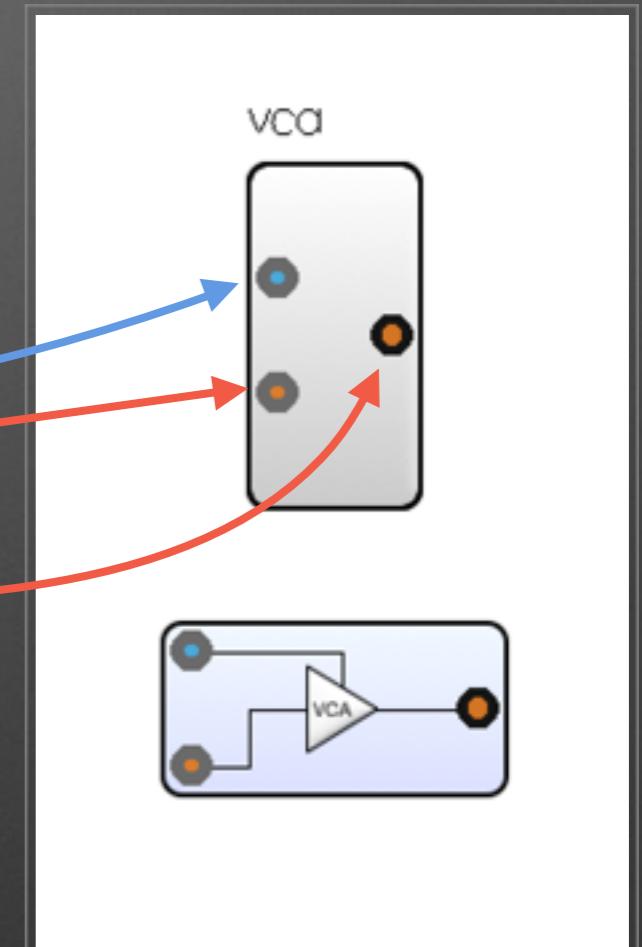
void init() { }

void fini() { }

inline void exec(in __attribute__((normal(1.0)))
                  control cv,
                  in sample signal,
                  out sample result) {
    result = cv*signal;
}
```

# .msl Plugins

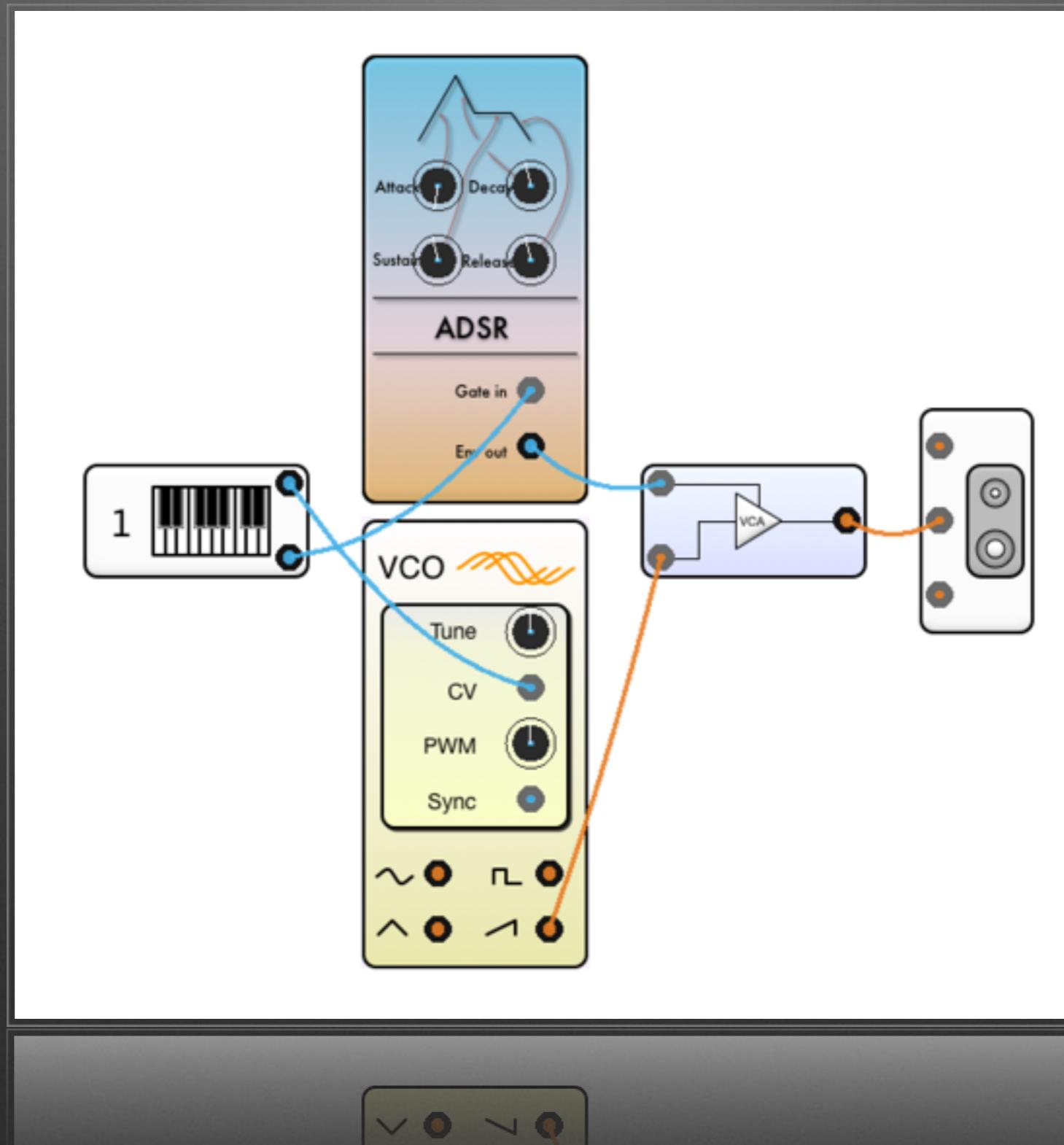
```
double result;  
  
void init() { }  
  
void fini() { }  
  
inline void exec(in __attribute__((normal(1.0)))  
                 control cv, __  
                 in sample signal,  
                 out sample result) {  
    result = cv*signal;  
}
```



# .hs Plugins

```
do
  plane <- currentPlane
  p <- mouse
  panel <- container' "panel_2x1.png" p (Inside plane)
  lab <- label' "vca" (p+(-36.0,84.0)) (Outside panel)
  name <- new' "vca"
  inp <- plugin' (name ! "cv") (p+(-24,24)) (Outside panel)
  setColour inp "#control"
  inp <- plugin' (name ! "signal") (p+(-24,-24)) (Outside panel)
  setColour inp "#sample"
  out <- plugout' (name ! "result") (p+(24,0)) (Outside panel)
  setColour out "#sample"
  recompile
  return ()
```

# A minimal synth



# Generated C code

```
void execute(struct State * state, double * buffer)
{
    for (int i = 0; i < 256; ++i)
    {
        state->id5.result = state->input13.result;
        state->id12.result = state->input19.result;
        state->sum21.result = state->id5.result + state->id12.result;
        state->id7.result = 0;
        audio_saw_exec(state->sum21.result,
                        state->id7.result,
                        &state->audio_saw1);
        state->id10.result = state->audio_saw1.result;
        adsr_exec(state->input15.result,
                  state->input16.result,
                  state->input18.result,
                  state->input17.result,
                  state->input20.result,
                  &state->adsr0);
        state->vca22.result = state->adsr0.result * state->id10.result;
        buffer[2 * i] = state->vca22.result + 0;
        buffer[2 * i + 1] = state->vca22.result + 0;
    }
}
```

# From .msl...

```
double last_up;
double last_down;
double multiplier_up;
double multiplier_down;
double result;

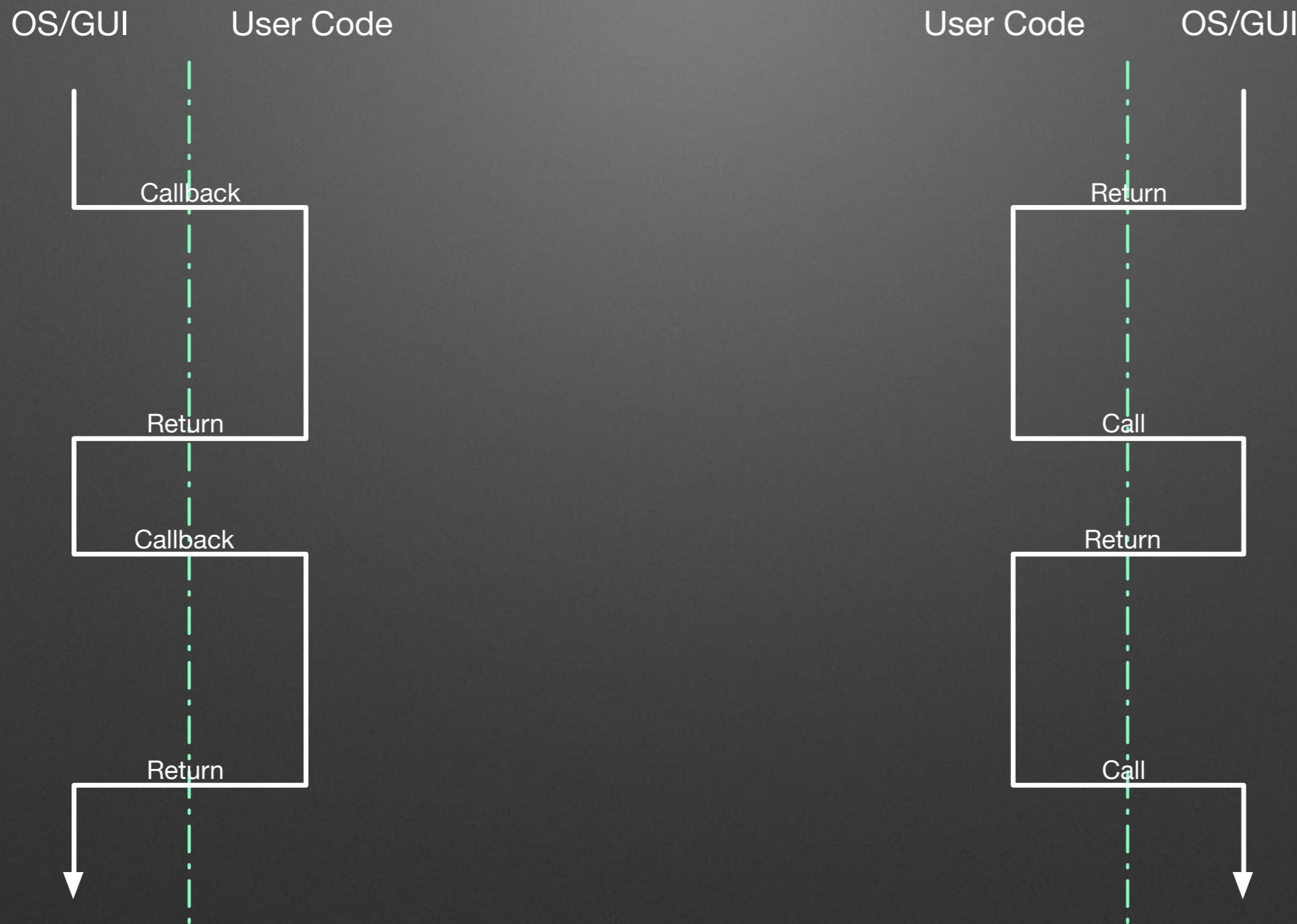
void init() {
    last_up = -1.0;
    last_down = -1.0;
}

void exec(in control decay_up, in control decay_down,
          in control input, out control result) {
    if (result > input) {
        if (decay_down != last_down) {
            multiplier_down = exp(-dt/max(0.001, decay_down));
        }
        result = input+multiplier_down*(result-input);
        last_down = decay_down;
    } else if (result < input) {
        if (decay_up != last_up) {
            multiplier_up = exp(-dt/max(0.001, decay_up));
        }
        result = input-multiplier_up*(input-result);
        last_up = decay_up;
    }
}
```

# ...to C

```
void vactroid_exec(double decay_up,
                   double decay_down,
                   double input,
                   struct vactroid * vactroid)
{
    if (vactroid->result > input)
    {
        if (decay_down != vactroid->last_down)
        {
            vactroid->multiplier_down = exp(-dt / max(0.001, decay_down));
        }
        vactroid->result = input + vactroid->multiplier_down * (vactroid->result - input);
        vactroid->last_down = decay_down;
    }
    else if (vactroid->result < input)
    {
        if (decay_up != vactroid->last_up)
        {
            vactroid->multiplier_up = exp(-dt / max(0.001, decay_up));
        }
        vactroid->result = input - vactroid->multiplier_up * (input - vactroid->result);
        vactroid->last_up = decay_up;
    }
}
```

# UI Control Flow



# Reinversion of Control

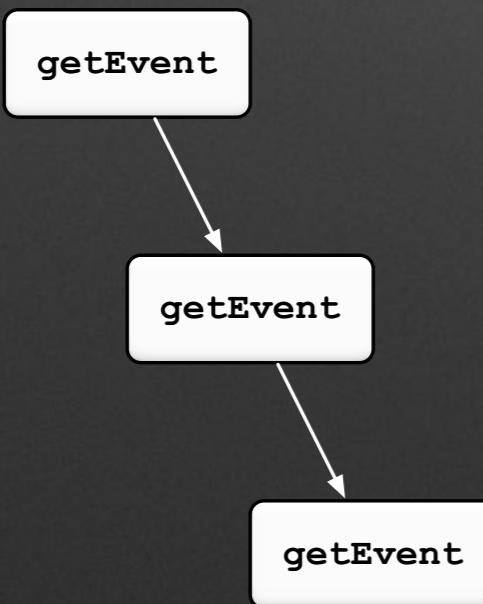
## Two approaches

```
do  
  e <- getEvent  
  case e of ...  
    ...do stuff ...
```



```
do  
  e <- getEvent  
  case e of ...  
    ...do stuff ...
```

This continuation is established as a callback and control is relinquished to GUI

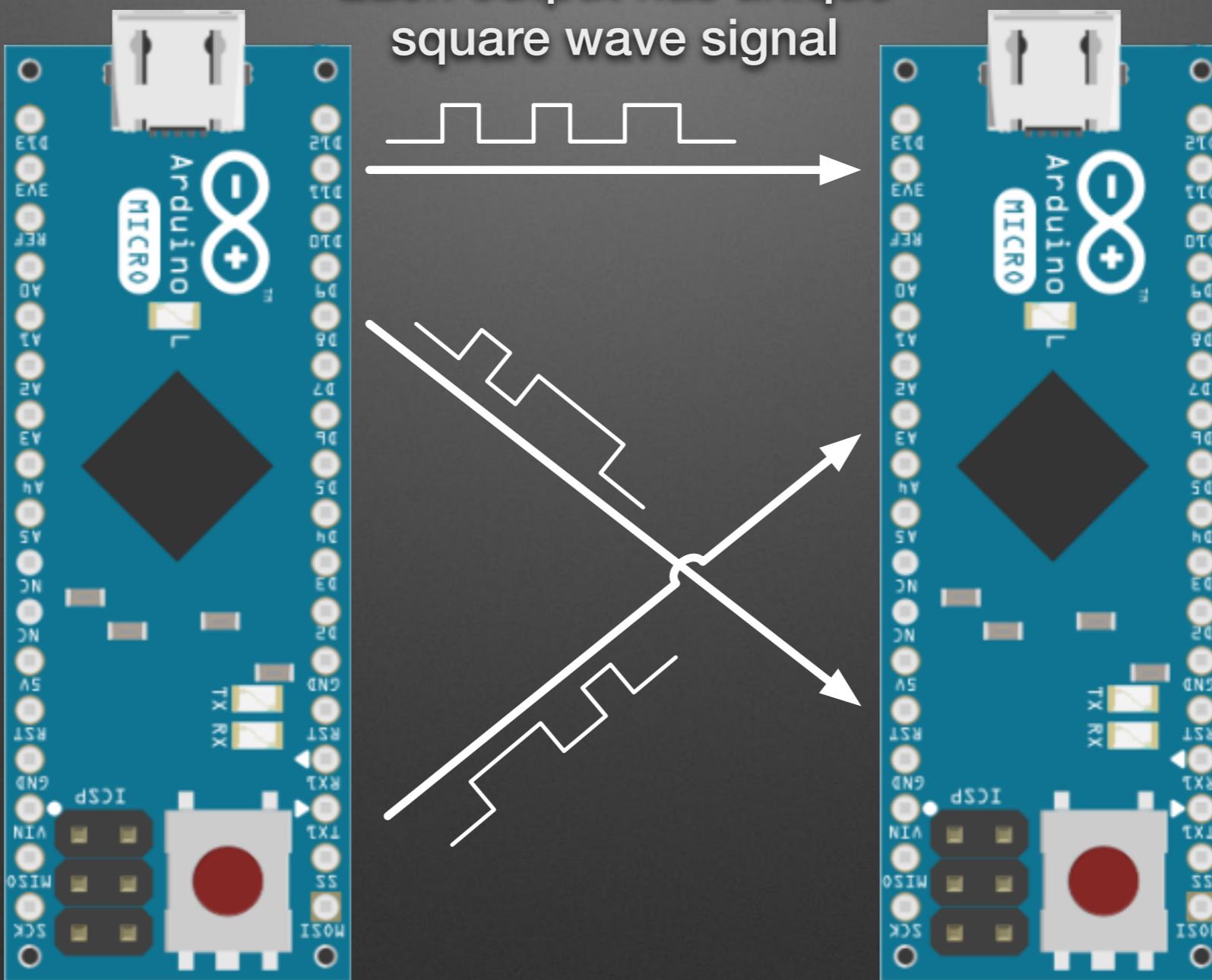


Free monad builds tree.  
Semantics provided by interpreter that runs a small step at a time in callback.

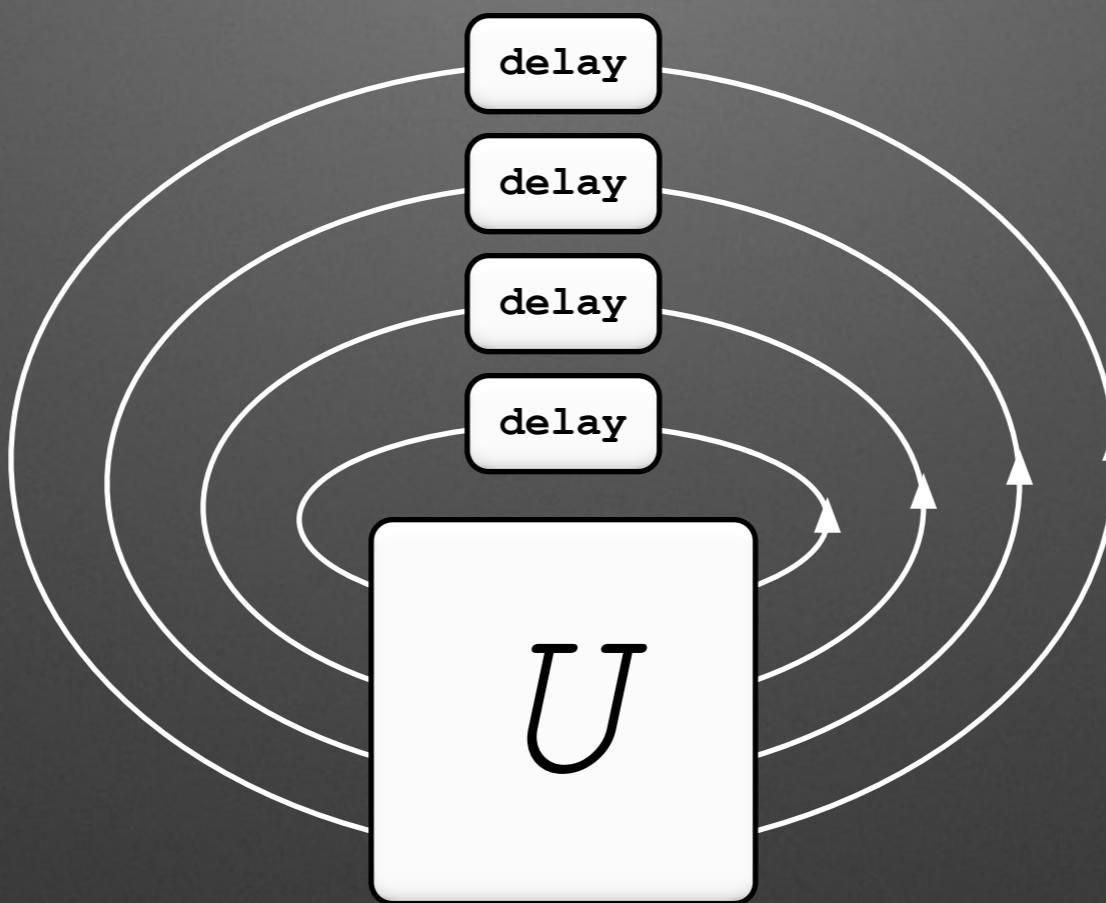
Another free monad is also used for .hs plugin to get complete separation of plugin code from Moodler internals.

# Recognising cables

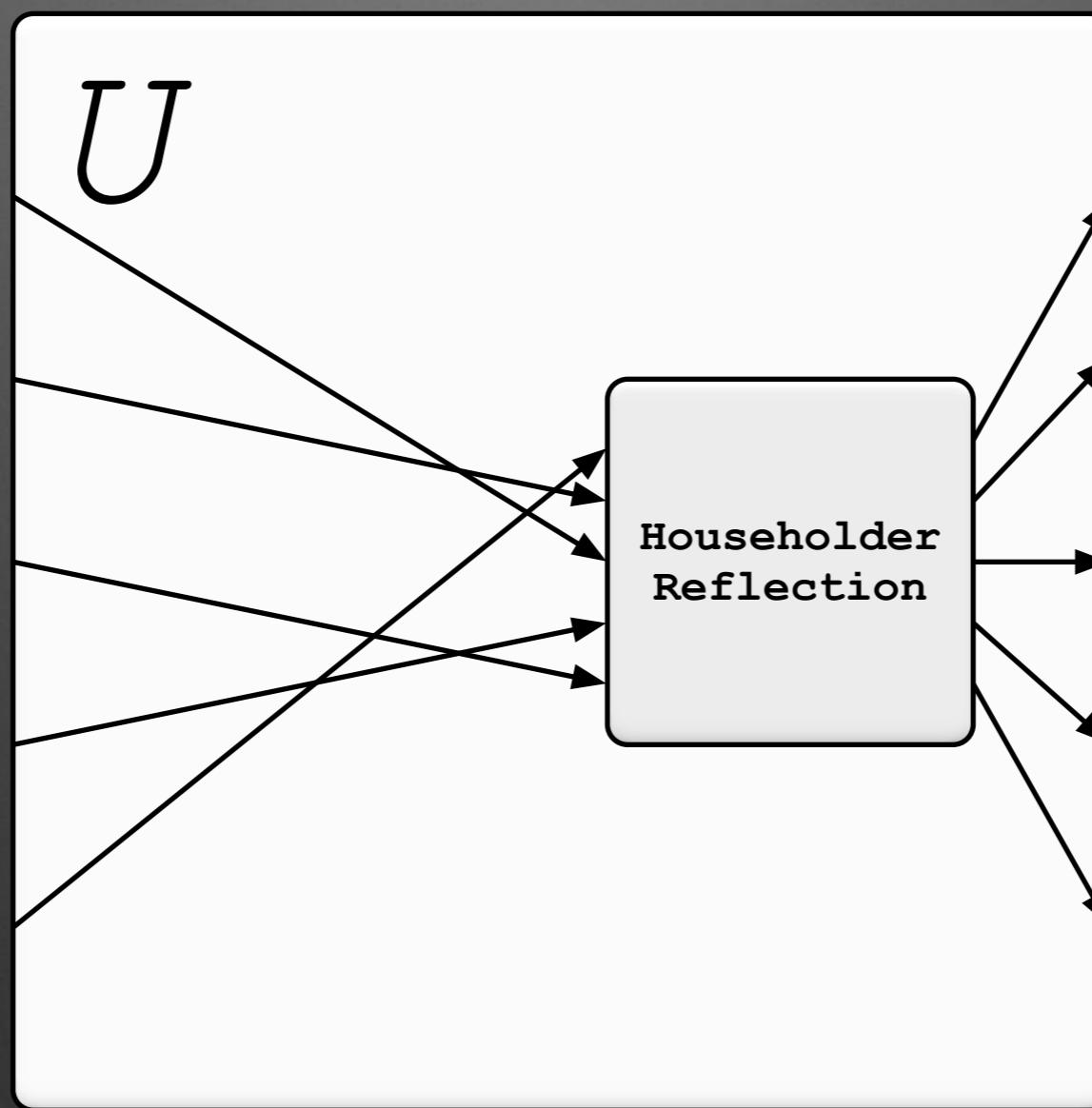
Each output has unique  
square wave signal



# Permutations with Cables



# Permutations with Cables



# Permutations with Cables



# Live code?

- Or maybe a canned example (`demo_test_bitwise`)

# Thanks

- Barnaby Robson for porting to PortAudio and Oscilloscope.