Music & Audio Programming Design Explorations

James McCartney

The Future of SuperCollider 2025

Contents

- A language implementation
- Code generation for synth definitions

Language

- buzzword synopsis:
  - dynamically typed, interpreted, mostly functional, and object oriented, w/ linearized multiple inheritance, multiple argument dispatch, immutable fully persistent data types, thread safe mutable data, pervasive auto-mapping (like APL), a concurrent runtime, reference counted…

Language

- buzzword synopsis:
  - dynamically typed, interpreted, mostly functional, and object oriented, w/ linearized multiple inheritance, multiple argument dispatch, immutable fully persistent data types, thread safe mutable data, pervasive auto-mapping (like APL), a concurrent runtime, reference counted…

```
let 🤎 = 520
```

```
let 🦕 = [1, 2, 3]
let 🪰 = 2
let 🖨️ = println
🖨️(🦕 * 🪰)
```

some SuperCollider issues

- Can’t dynamically add classes and methods
- No name spaces for classes
- Single threaded interpreter
- Too easy to introduce garbage collector bugs in primitives
- Double dispatch (e.g., for math) is clunky

some SuperCollider issues

- Can’t dynamically add classes and methods
- No name spaces for classes
- Single threaded interpreter
- Too easy to introduce garbage collector bugs in primitives
- Double dispatch (e.g., for math) is clunky

Dynamically add classes and functions

```
#Shape defClass(Item)

def scalarShape = Shape new { rows: 1, cols: 1 }

fn call(class Shape_class, rows Int, cols Int) {
    Shape new { rows: rows, cols: cols }
}

fn call(class Shape_class, cols Int) = Shape new { rows: 1, cols: cols }

fn len(s Shape) = s.rows * s.cols
```

Dynamically add classes and functions

```
#Shape defClass(Item)

def scalarShape = Shape new { rows: 1, cols: 1 }

fn call(class Shape_class, rows Int, cols Int) {
    Shape new { rows: rows, cols: cols }
}

fn call(class Shape_class, cols Int) = Shape new { rows: 1, cols: cols }

fn len(s Shape) = s.rows * s.cols
```

Dynamically add classes and functions

```
#Shape defClass(Item)

def scalarShape = Shape new { rows: 1, cols: 1 }

fn call(class Shape_class, rows Int, cols Int) {
    Shape new { rows: rows, cols: cols }
}

fn call(class Shape_class, cols Int) = Shape new { rows: 1, cols: cols }

fn len(s Shape) = s.rows * s.cols
```

Dynamically add classes and functions

```
#Shape defClass(Item)

def scalarShape = Shape new { rows: 1, cols: 1 }

fn call(class Shape_class, rows Int, cols Int) {
    Shape new { rows: rows, cols: cols }
}

fn call(class Shape_class, cols Int) = Shape new { rows: 1, cols: cols }

fn len(s Shape) = s.rows * s.cols
```

Dynamically add classes and functions

```
#Shape defClass(Item)

def scalarShape = Shape new { rows: 1, cols: 1 }

fn call(class Shape_class, rows Int, cols Int) {
    Shape new { rows: rows, cols: cols }
}

fn call(class Shape_class, cols Int) = Shape new { rows: 1, cols: cols }

fn len(s Shape) = s.rows * s.cols
```

```
call(Shape, rows, cols)
-- is equivalent to:
Shape(rows, cols)
```

Dynamically add classes and functions

```
#Shape defClass(Item)

def scalarShape = Shape new { rows: 1, cols: 1 }

fn call(class Shape_class, rows Int, cols Int) {
    Shape new { rows: rows, cols: cols }
}

fn call(class Shape_class, cols Int) = Shape new { rows: 1, cols: cols }

fn len(s Shape) = s.rows * s.cols
```

Dynamically add classes and functions

```
#Shape defClass(Item)

def scalarShape = Shape new { rows: 1, cols: 1 }

fn call(class Shape_class, rows Int, cols Int) {
    Shape new { rows: rows, cols: cols }
}

fn call(class Shape_class, cols Int) = Shape new { rows: 1, cols: cols }

fn len(s Shape) = s.rows * s.cols
```

some SuperCollider issues

- Can’t dynamically add classes and methods
- No name spaces for classes
- Single threaded interpreter
- Too easy to introduce garbage collector bugs in primitives
- Double dispatch (e.g., for math) is clunky

Modules

- Modules allow name-spacing

```
-- import the 'stuff' module
-- to use a name in stuff you need to prefix it as stuff.name
import stuff

-- import the 'stuff' module as the local name 'X'
import stuff as X

-- import every definition in the 'stuff' module into this scope
import stuff : *

-- import only the names 'funA' and 'funB' from stuff.
import stuff : funA, funB

-- import only some names, but rename them locally.
import stuff : funA as xA, funB as xB
```

Modules

- Modules allow name-spacing

```
-- import the 'stuff' module
-- to use a name in stuff you need to prefix it as stuff.name
import stuff

-- import the 'stuff' module as the local name 'X'
import stuff as X

-- import every definition in the 'stuff' module into this scope
import stuff : *

-- import only the names 'funA' and 'funB' from stuff.
import stuff : funA, funB

-- import only some names, but rename them locally.
import stuff : funA as xA, funB as xB
```

Modules

- Modules allow name-spacing

```
-- import the 'stuff' module
-- to use a name in stuff you need to prefix it as stuff.name
import stuff

-- import the 'stuff' module as the local name 'X'
import stuff as X

-- import every definition in the 'stuff' module into this scope
import stuff : *

-- import only the names 'funA' and 'funB' from stuff.
import stuff : funA, funB

-- import only some names, but rename them locally.
import stuff : funA as xA, funB as xB
```

Modules

- Modules allow name-spacing

```
-- import the 'stuff' module
-- to use a name in stuff you need to prefix it as stuff.name
import stuff

-- import the 'stuff' module as the local name 'X'
import stuff as X

-- import every definition in the 'stuff' module into this scope
import stuff : *

-- import only the names 'funA' and 'funB' from stuff.
import stuff : funA, funB

-- import only some names, but rename them locally.
import stuff : funA as xA, funB as xB
```

Modules

- Modules allow name-spacing

```
-- import the 'stuff' module
-- to use a name in stuff you need to prefix it as stuff.name
import stuff

-- import the 'stuff' module as the local name 'X'
import stuff as X

-- import every definition in the 'stuff' module into this scope
import stuff : *

-- import only the names 'funA' and 'funB' from stuff.
import stuff : funA, funB

-- import only some names, but rename them locally.
import stuff : funA as xA, funB as xB
```

Modules

- Modules allow name-spacing

```
-- import the 'stuff' module
-- to use a name in stuff you need to prefix it as stuff.name
import stuff

-- import the 'stuff' module as the local name 'X'
import stuff as X

-- import every definition in the 'stuff' module into this scope
import stuff : *

-- import only the names 'funA' and 'funB' from stuff.
import stuff : funA, funB

-- import only some names, but rename them locally.
import stuff : funA as xA, funB as xB
```

some SuperCollider issues

- Can’t dynamically add classes and methods
- No name spaces for classes
- Single threaded interpreter
- Too easy to introduce garbage collector bugs in primitives
- Double dispatch (e.g., for math) is clunky

SC interpreter single threaded

- Class library contains global mutable state
- Garbage collector is global mutable state
- All objects are mutable

Rule of safe concurrency

- **Do not share mutable state.**
  - You can have immutable state and share it
  - You can have mutable state but not share it = thread local
  - You can protect against concurrent access by using a mutex

Thread safe data

- Most objects are immutable
  - Records, Arrays, Maps, Tuples, Lists
  - You “modify” them by creating a changed copy.
- Those few objects that are mutable are protected by a mutex
  - Ref, Queue, Stack

Immutable data implementation

- Uses same kinds of data structures as found in Clojure
  - persistent Vectors, Maps, Sets
- I use the “immer” C++ library
  by Juan Pedro Bolívar Puente.
  https://github.com/arximboldi/immer

A fully persistent data structure allows

- access to all previous versions after each modification,
- preserving the complete history of changes while creating new versions
- that **share structure** with older ones to minimize memory usage.

- Each operation creates a new version **without modifying** existing versions,
- enabling efficient **time travel** through the structure's entire evolution.

- Can be safely shared across threads
- Copying is as cheap as copying a pointer

Immutable data & Garbage Collection

- No mutation means no garbage collector write barriers
- Reference cycles are not possible without mutation or laziness
- So, use reference counting
- Reference counting is easy to make thread safe
- Not as error prone as remembering to insert write barriers
- Cycles CAN occur using thread-safe mutable Refs

No global variables

- The top level scope is thread local
- The top level scope is mutable (which is safe because it is not shared)
  - it is maintained in a persistent dictionary
  - Forked threads get a copy of the parent thread’s local state, which then becomes independent
  - That copy is cheap because everything is shared.
- Shared `blackboards` of data can be done using Refs

some SuperCollider issues

- Can’t dynamically add classes and methods
- No name spaces for classes
- Single threaded interpreter
- Too easy to introduce garbage collector bugs in primitives
- Double dispatch (e.g., for math) is clunky

Multimethods

- To implement math operations in SC, double dispatch is needed
  - Lots of implementations of `performBinaryOpOnSimpleNumber`
- Multiple argument dispatch -> all arguments determine method
- Classes do not contain methods
- Multimethods are defined outside of any class
- Multimethod dispatch can catch some dynamic type errors earlier

Multimethods

- Functions that belong together aren't spread out over the class library.

```
fn asSignal(x SignalExpr) = x
fn asSignal(x Real) = x scalar
fn asSignal(a Array) = a rowVec
```

- Adding multiple specialized arguments provides run-time type checking

```
-- overload operators so that these expressions can be built algebraically.
fn +(a RewriteExpr, b RewriteExpr) = UGenRewriteExpr(OpAdd, [a, b])
fn -(a RewriteExpr, b RewriteExpr) = UGenRewriteExpr(OpSub, [a, b])
fn *(a RewriteExpr, b RewriteExpr) = UGenRewriteExpr(OpMul, [a, b])
fn /(a RewriteExpr, b RewriteExpr) = UGenRewriteExpr(OpDiv, [a, b])
fn %(a RewriteExpr, b RewriteExpr) = UGenRewriteExpr(OpMod, [a, b])
```

Multimethods

- Sometimes you want multiple implementations based on the second argument.

```
fn genUGenExpr(o CCodeGen, ugen MathOpUGen, indexExprs) {...}
fn genUGenExpr(o CCodeGen, ugen SampleRateUGen, indexExprs) {...}
fn genUGenExpr(o CCodeGen, ugen SelectUGen, indexExprs) {...}
fn genUGenExpr(o CCodeGen, ugen DebugUGen, indexExprs) {...}
fn genUGenExpr(o CCodeGen, ugen Outlet, indexExprs) {...}
fn genUGenExpr(o CCodeGen, ugen Inlet, indexExprs) {...}
fn genUGenExpr(o CCodeGen, ugen URandUGen, indexExprs) {...}
fn genUGenExpr(o CCodeGen, ugen BRandUGen, indexExprs) {...}
fn genUGenExpr(o CCodeGen, ugen TSUGen, indexExprs) {...}
```

Auto-mapping

- If a function expects a single item for an argument, but instead receives an array or list, then the function is applied to every element of the array and an array or list is returned.

```
9 sqrt      -- returns 3

[1, 4, 9, 16, 25] sqrt    -- returns [1, 2, 3, 4, 5]
```

- Note: `x f` is equivalent to `f(x)`

Auto-mapping

- The @ operator can be used to force auto mapping, do Cartesian mapping, and for constructing data structures.

```
let a = [10, 20]
let b = [1, 2]
[a @, b] println
[a @, b @] println
[a @1, b @2] println
[a @2, b @1] println

-->
[[10, [1, 2]], [20, [1, 2]]]
[[10, 1], [20, 2]]
[[[10, 1], [10, 2]], [[20, 1], [20, 2]]]
[[[10, 1], [20, 1]], [[10, 2], [20, 2]]]
```

Code generation

- all signals are matrices
- supports 32 and 64 bit, integer and floating point signals
- supports multi rate, including intermittent, triggered or gated subgraphs
- delays are written as difference equations (like letrec in Faust)
- block diagram operators as in Faust (, : :> <: ~) are not necessary
- you have auto-mapping, algorithmic construction, a much higher level language

Graph of nodes

- Graphs are by calling functions that create nodes
- node types are
  - constants
  - math operators: `+ - * / % abs sin log FFT`
  - Delays : `init read write`
  - I/O: inputs, outputs, control
  - Control flow: `if switch for`
  - Matrix manipulation: `take skip slice stride rotate transpose reverse`

Signal types

- Every signal has:
  - a matrix shape: rows x columns
  - an element type: i32, i64, f32, f64
  - a rate: constant, init time, reset, event, audio

Graph construction

- Constant folding
- Rate inference
- Common subexpression elimination
- Algebraic simplification via graph rewriting

Expression rewriting

- Use math identities to simplify expressions
- Try to factor out lower rate operations so moved off of audio rate.

- Rules are pattern matched
- LHS is replaced by RHS
- more than 120 rules currently

```
vector rules = {
    // inverses
    rule(-(-(x)), x),
    rule(!(!(x)), x),
    rule(-(x-y), y-x),
    rule(one/(x/y), y/x),

    rule(tan(atan(x)), x),
    rule(sinh(asinh(x)), x),
    rule(asinh(sinh(x)), x),
    rule(atanh(tanh(x)), x),
```

Compiler passes

- Merge delays
- Topological sort of graph
- Calculate delay lengths
- Shape inference
- Type inference
- Cut graph into expression trees

- Put trees into matrix loops
  = loop fusion
- Separate loops by rate
  - init, reset, event, audio
- Emit C++ code
- Compile C++, link, load

Coding ugens and synthdefs

```
------------------------------------------------------------------------
--- Fundamental unit generators

#SampleRate defClass(SignalExpr)
#SampleDur defClass(SignalExpr)
#Inlet defClass(SignalExpr)
#Outlet defClass(SignalExpr)
#Control defClass(SignalExpr)

fn fs() = SampleRate new addToGraph
fn T() = SampleDur new addToGraph

fn inlet(type NumType, shape, name String = "in") =
    Inlet new { type: type, shape: shape asShape, name: name } addToGraph

fn outlet(a SignalExpr, name String = "out") =
    Outlet new { ins: [a], name: name } addToGraph

fn control(spec ControlSpec, type NumType, shape, name String) =
    Control new { spec: spec, type: type, shape: shape asShape, name: name } addToGraph
```

```
fn bubbles() =
    .4 lfsaw * 24 + [8, 7.23] lfsaw * 3 + 81 |> nnhz sinosc * 4c |> combn(.2,4) outlet
```

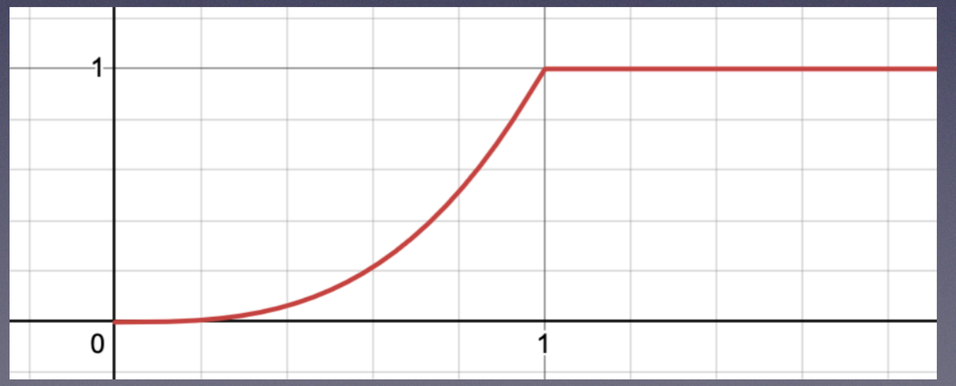
- A 0.4 Hz low frequency sawtooth times 24 semitones
- plus 2 channel (8 Hz and 7.23 Hz) sawtooth times 3 semitones
- plus 81 semitones
- |> piped into nnhz which converts MIDI note numbers to Hertz
- into a sine oscillator times 0.04 (4c)
- |> piped to a fixed delay comb echo delay
- sent to an outlet

Defining a unit generator

```
fn onepole(x, a) {
    let y = DelayVar()
    y write(x + a * (y(1) - x))
}

fn onezero(x, a) = x + a * (z1(x) - x)
```

```
fn fadein(x, t) {
    let dt = 1 / (t * fs())
    let y = DelayVar()
    min(1, dt + y(1)) write(y) cb * x
}
```



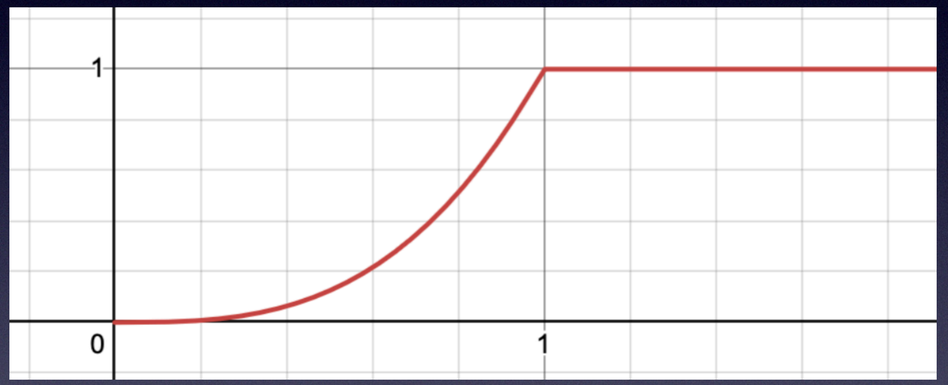
Defining a unit generator

```
fn onepole(x, a) {
    let y = DelayVar()
    y write(x + a * (y(1) - x))
}

fn onezero(x, a) = x + a * (z1(x) - x)
```

Defining a unit generator

```
fn fadein(x, t) {
    let dt = 1 / (t * fs())
    let y = DelayVar()
    min(1, dt + y(1)) write(y) cb * x
}
```



A lot remains to do

- Events
  - Code generation
  - Scheduler
- Synth Server protocol
- SIMD code gen - existed in a previous version, needs to be re-done

Related work

- Language
  - Dylan - multimethods, multiple inheritance
  - Clojure - persistent data structures, Hash array mapped tries.
- Code generation
  - Kronos - Vesa Norilo - multi-rate, reactive
  - Arrp - Jakob Leben - multi-rate, multi dimensional signals

Video SynthExperiment

- Generates random trees of functions from (x, y) to (r, g, b)
- Generates Metal shader code from the tree.
- Renders a tree + a vector of random parameters as an image.
- Renders a tree + a list of paramter vectors as a video

Demo

Other talks

- Codefest 2021
  https://www.youtube.com/watch?v=fmVdfQNPzkE
- Darwin Grosse podcast 2021
  https://www.youtube.com/watch?v=qmayIRViJms