<<> and <>> : Two simple operators for composing processes at runtime

Julian Rohrhuber

1 Abstract

Dynamic rearrangement of synth graphs can be an interesting method to move from one sound to another and from one idea to the next. It also implements the common graphical paradigm of "patching". In analogy to function composition operator <>, two new operators, <<> and <>> help to compose running processes of node proxies. This brief paper gives a few examples how they can be used for sound synthesis.

2 Composing abstract functions

2.1 Function composition

In the SuperCollider language, closures are implemented by the class Function, which is, just like unit generators, streams, routines and patterns, a subclass of AbstractFunction. Most operators may thus be used not only for calculating data, but also for composing calculations lazily – resulting in potentially infinitely long datasets whose components may even be partially unknown at the time of construction. Such operators may implement mathematical operations, like $+, -, \sin, \log$, or domain specific ones, like midicps or ampdb. A special operator is the binary *function composition* operator <>, which constructs compound functions,[1] connecting the second operand's output with the first operand's first argument:

```
f = { |x| sin(x) + (x * x) };
g = { |x| x * 2pi };
```

[1]The sign "<>" within SuperCollider approximates the mathematical sign "$\circ$": $f \circ g \circ h$, which is equal to $f(g(h(x)))$.

```
h = { |x| x + 1 };

z = f <> g <> h;
```

The result of the last expression is a function z that calculates the results from the functions f, g, h, passing its argument first to h, whose output it passes to the input of g, whose output it passes to f. This can also be written as

```
z = { |x| f.value(g.value(h.value(x))) };
z = { |x| x =  (x + 1 * 2pi); sin(x) + (x * x) }
```

The advantage of the shortened syntax is that the chain can easily be reordered, and that it is independent of the functions themselves (topologic semantics). Note that in the typical implementation, only the first argument is used: for multiple "channels" one may pass arrays or environments as an argument.

2.2 Pattern composition

In analogy to functions, also patterns and streams may be composed.

```
a = Pfunc { |x| x * 2 };
b = Pfunc { |x| if(x > 5) { 0 } { 1 } };
c = Pseq((0..6));
z = b <> a <> c;
z.asStream.next(1);
```

Pattern composition is commonly used with event patterns like Pbind, which pass an *event* up and down the chain:

```
f = Pbind(\freq, Pn(Pshuf([200, 551, 603, 881.3]/2, 16)));
g = Pbind(\dur, Pn(Pshuf([0.3, 0.1, 0.4], 8)));
z = f <> g;
z.play;
```

3 Composing signals at runtime

The class *NodeProxy* and its subclass *Ndef* return placeholders for signal processes on the server. They connect changes in code with changes in those processes, so that we can rewrite our synths while listening to their output (2). To combine the output of one proxy with the unit generator graph of another, one can embed the placeholder directly by its name:

```
Ndef(\a, { Dust.ar(8 ! 2) });
Ndef(\b, { Ringz.ar(Ndef.ar(\a), Dust.kr.lag(1) * 800 + 600, 0.3) });
Ndef(\c, { HPF.ar(CombL.ar(Ndef.ar(\a)), 8000) + Ndef(\b) });
Ndef(\c).play;
```

However convenient, this can be a problem in certain cases: in order to reroute the signal flow one needs to modify the structure that receives it.[2] In many cases, it may be better to ignore the internal usage and just try out different external routings. Optimally, the shift of focus between external and internal perspectives should be fluent.

Since the more recent introduction of audio rate routing in SuperCollider, we may now use the open sound control message */n_mapa* to connect different sources. Usually this is done by mapping audio rate buses to audio rate control names, which involves three types of entities explicitly: SynthDef, Synth and Bus. Instead, in analogy to function composition, one can also directly rearrange instances of NodeProxy or Ndef into different signal chains.[3]

The new binary operator introduced here makes some differences to usual function composition semantics:

- Unlike function composition, this composition doesn't return a new object, but instead passes the signal through one of its operands at runtime (it is an imperative operator).
- To maintain this difference, and also to allow for both directions of linking, the new operator symbols $<<>$ (right to left) and $<>>$ (left to right) are introduced.
- As there is no default input for synths, the control name *in* is assumed.
- To link other inputs, an adverb may be used: $<<>.x$ links to the x control, $<<>.freq$ to the $freq$ control.

Analogously to the lazy initialisation in NodeProxy, if no rate and number of channels is given in the operands, they are initialised (we assume that audio rate in stereo is the default here[4]).

```
Ndef(\y) <<> Ndef(\b) <<> Ndef(\x);
Ndef(\y).play;
```

This creates an empty (silent) stereo audio rate chain, which can be filled with ugen functions later:

```
Ndef(\y, { \in.ar(0 ! 2) });
Ndef(\x, { Dust.ar(5 ! 2) });
Ndef(\b, { Ringz.ar(\in.ar(0 ! 2), LFNoise1.kr(0.1).range(550, 700), 0.2).distort });
```

Sending the message *ar* directly to a symbol (passing the number of channels as argument) is a convenient way to write an audio rate input control name within a UGen graph in a SynthDef or a NodeProxy/Ndef:

```
\in.ar([0, 0]) or: \in.ar(0 ! 2)
```

[2] In the above case, for example, in order to rout Ndef(\b) into the CombL of Ndef(\c), we have to replace Ndef.ar(\a) by Ndef.ar(\b) in the code.

[3] This method is somewhat similar to sequential signal composition in the block-diagram algebra of the Faust programming language, which uses the notation $a : b$ for the similar constellation (1).

[4] The default channel sizes may be set in a class variable of NodeProxy.

We can thereby build up line by line (in arbitrary order) a generic series of chain links which may be recombined at runtime:

```
// define a few chain links
Ndef(\comb, { CombL.ar(\in.ar(0 ! 2), 0.1, LFNoise1.kr(0.1).range(0, 0.1), 2) });
Ndef(\ring, { Ringz.ar(\in.ar(0 ! 2), LFNoise1.kr(0.1).range(550, 700), 0.2).distort });
Ndef(\filt, { RLPF.ar(\in.ar(0 ! 2), LFNoise1.kr(0.1).range(300, 7000), 0.1) });
Ndef(\dust, { Dust.ar(5 ! 2) });
Ndef(\y, { \in.ar(0 ! 2) });

// play back one of them
Ndef(\y).play;

// different combinations
Ndef(\y) <<> Ndef(\dust);
Ndef(\y) <<> Ndef(\ring) <<> Ndef(\dust);
Ndef(\y) <<> Ndef(\ring) <<> Ndef(\comb) <<> Ndef(\dust);
Ndef(\y) <<> Ndef(\ring) <<> Ndef(\comb) <<> Ndef(\filt) <<> Ndef(\dust);

// the above written in the opposite direction
Ndef(\dust) <>> Ndef(\filt) <>> Ndef(\ring) <>> Ndef(\comb) <>> Ndef(\y);
```

The *unmap* message or linking it to *nil* removes an Ndef from the chain. The method *orderNodes* allows the synth order to be enforced (for instance when an external audio signal requires minimal delays).

All these reconfigurations are done at runtime, at the same time, the content of the nodes may also be reprogrammed. Using different crossfade times (using the message *fadeTime*) for each node can be an interesting aspect of including this process in the unfolding of a musical performance.

Also, both operands may be the same – within the timing constraints of the audio processing block size (ca. 1.3 ms in the default setup), signals can be fed back to a part of their source: here, the sound output re-enters as a phase of the sine oscillator that produces it.[5]

```
Ndef(\z, {
      SinOsc.ar(
            LFNoise0.kr(12 ! 2, 100, 400),
               \in.ar(0 ! 2) * 1.5
            ) *  LFNoise0.kr(4 ! 2).max(0)
});
Ndef(\z).play;
Ndef(\z) <<> Ndef(\z);
```

Finally, a brief example of how to use *adverbs*. Adverbs are additional specifications of how a binary operator is to be applied – in other words, the adverb describes how the verb (operator) affects the nouns (operands). In SuperCollider, they are written as a string that comes after the operator, separated by a dot.

The following creates two single channel control rate inputs x and y which are mapped to two instances of Ndef reading the current cursor position.

```
Ndef(\comb, {
      CombL.ar(
            \in.ar([0, 0]),
            0.1,
            LFDNoise1.kr(\x.kr(0.1)).range(0, 0.1),
            \y.kr(2)
      )
});

Ndef(\mod1, { MouseX.kr(0.01, 100, 1) });
Ndef(\mod2, { MouseY.kr(0.01, 100, 1) });
Ndef(\comb) <<>.x Ndef(\mod1);
Ndef(\comb) <<>.y Ndef(\mod2);
```

References

[1] Y. Orlarey, D. Fober, and S. Letz. Syntactical and semantical aspects of Faust. *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, 8(9):623–632, 2004.

[5] Thanks to Alberto de Campo for the example and a number of other useful suggestions.

[2] Julian Rohrhuber, Alberto de Campo, and Renate Wieser. Algorithms today - Notes on Language Design for Just In Time Programming. In *Proceedings of International Computer Music Conference*, pages 455–458, Barcelona, 2005. ICMC.