Aleatora I: Composing with Streams

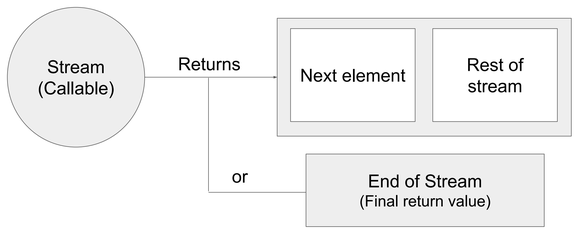
Ian Clester
MUSI 7100

INTRODUCTION

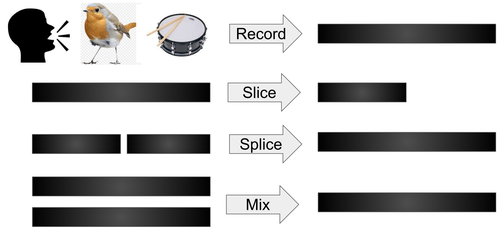
Today, there are several obstacles to creating and distributing computational compositions. **Aleatora** is a three-part project that aims to tackle the issues of language design, distribution, and tools for non-programmers. This poster deals with first issue: language for design for musical composition. Most audio programming languages today have a firm division between synthesis and control, between the score and orchestra. In several cases, this division is enshrined by separating the system into two languages (as in Max/MSP, Pure Data, or SuperCollider), while in others it exists in the syntax and semantics of a single language. Aleatora, like Nyquist before it, eschews this division and gives the composer the opportunity to define their own layers of abstraction however they deem appropriate, while adding real-time playback and inspection capabilities.

STREAMS

Aleatora is a Python library built around the abstraction of *lazy*, *effectful* streams. Since streams are *lazy*, no elements are computed until they are needed (meaning streams can be infinite), and since they are *effectful*, stream computations may have important side effects, or rely on external state, such that they have take a different path and produce different results upon replay. Streams represent suspendable chains of computation, like Python's built-in generators, with the key difference that they are replayable from any point. Aleatora's streams are related to Scheme's streams (as described in *SICP* and SRFI-41) and Haskell's lists; the main difference is the lack of (default) memoization and emphasis on side-effects, which makes their closest relative Haskell's Pipes library.



Streams are lazy linked-lists.



Common tape operations analogously supported by Aleatora streams.
Streams can be thought of as "computational tape".

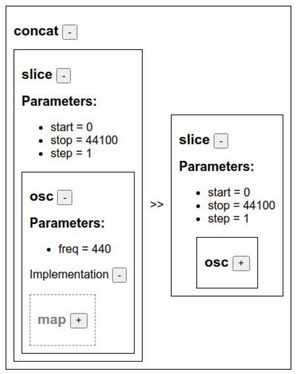
COMPOSITION

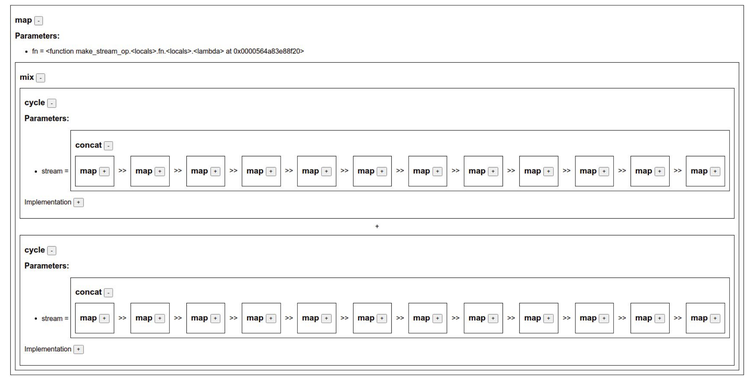
By representing a computational composition directly as a first-class value (a stream of samples), Aleatora enables working with audio like digital musique concrete—slicing, splicing, and mixing streams of (potentially endless) audio, but without the need to render it in advance. Unlike signals in many audio languages, streams may optionally come to an end, and this property allows for the definition of the concatenation (or splicing) operator, "`>>`". "`a >> b >> c`" means "play a, then play b, then play c": sequential composition. "`+`" is complimentary, as "`a + b + c`" means "play a *and* b *and* c at the same time". Streams overload many useful operators: arithmetic operators work element-wise, while slice notation allows making streams finite. For example, "`silence`" refers to the infinite stream of zeros, but "`silence[:1.0]`" creates a finite stream containing one second of silence.

Streams can also can be composed by taking other streams as arguments. While "`osc(440)`" produces an endless sine tone at a constant frequency, "`fm_osc(440 + osc(0.1)*20)`" takes a stream of frequencies as its argument (in this case causing it to slowly oscillate between 420 Hz and 460 Hz). Streams need not all run at the same rate: "`glide(freq_stream, hold_time, transition_time)`" holds frequencies and glides between them, and it only asks for a new value from frequency stream when `hold_time` + `transition_time` has elapsed. "`resample(stream, advance_stream)`" runs stream at a variable rate, based on the values in advance_stream (each of which represents a fractional number of samples to advance): so `resample(stream, const(0.5))` runs stream at half-speed, `resample(stream, const(2))` runs it at double-speed, and `resample(stream, osc(0.1) + 1)` runs it at a time-varying speed.

INSPECTION

The graph of streams that comprise a composition evolves as the composition plays, as streams are exhausted and replaced. To aid understanding and debugging, Aleatora offers facilities for inspecting a composition to see it's current state and structure. These facilities are built-in to the framework, and they include a web-based frontend for visually inspecting the stream graph. This interface allows for nesting and expansion, and it can show the current parameters applied to streams. For streams defined in terms of other streams (like "`osc`", which is defined using "`count`" and "`.map`"), the user can peek under the hood and their implementation. This functionality is automatically provided for user-defined streams (and most built-in streams) using Python's introspection facilities, as in the inspect module, which allows inpsecting function signatures at runtime. Below, the left screenshot shows the inspector view for the composition "`osc(440)[:1.0] >> osc(660)[:1.0]`" (one second of 440 Hz followed by one second of 660 Hz), while the right shows the inspector view for a four-line implementation of Steve Reich's *Piano Phase*.





DISCUSSION AND NEXT STEPS

Aleatora presents a compositional framework with some unusual properties. It is built on top of a general-purpose programming language but handles synthesis itself, in the same language. It is a textual language but has graphical tools for debugging. It eschews the typical barriers between synthesis and control, allowing streams to run at arbitrarily mixed rates. It allows real-time playback, plays well with an interactive REPL, and enables run-time inspection of compositions. Though early in the project's life, I believe this is a potent mix of features.

My intended next steps are extending the inspector into a full-fledged compositional companion, to make the system more interactive and usable. On the topic of usability, I also plan to build out integration with popular technologies (MIDI, VSTs, web APIs) and perform user-testing to ensure I'm not the project's only user. Looking ahead, I will need to find how this framework fits in with the project's other broad goals in addressing the ease of creating and distribution computational compositions.

FURTHER READING

The paper is available at ijc8.me/aleatora-i.pdf, and code with examples is available at github.com/ijc8/aleatora.

References from the paper:
[1] [n.d.]. Max/MSP. https://cycling74.com/products/max
[2] Harold Abelson and Gerald Jay Sussman. 1996. 3.5 Streams. In Structure and Interpretation of Computer Programs. MIT Press, Cambridge, Massachusetts. https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html
[3] Philip L. Bewig. 2007. SRFI-41: Streams. https://srfi.schemers.org/srfi-41/srfi-41.html
[4] Roger B. Dannenberg. 1997. The Implementation of Nyquist, A Sound Synthesis Language. Computer Music Journal 21, 3 (1997), 71–82. https://doi.org/10.2307/3681015 Publisher: The MIT Press.
[5] Roger B. Dannenberg. 1997. Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis. Computer Music Journal 21, 3 (1997), 50–60. https://doi.org/10.2307/3681013 Publisher: The MIT Press.
[6] Gabriel Gonzalez. 2020. pipes: Compositional pipelines. https://hackage.haskell.org/package/pipes.
[7] James McCartney. 2002. Rethinking the Computer Music Language: SuperCollider. Computer Music Journal 26 (Dec. 2002), 61–68. https://doi.org/10.1162/014892602320991383
[8] Miller S. Puckette. [n.d.]. Pure Data. https://puredata.info/
[9] Ge Wang and Perry R. Cook. 2003. ChucK: A Concurrent, On-the-fly, Audio Programming Language. In Proceedings of the International Computer Music Conference (ICMC). Singapore