A Functional Language for Sound Synthesis with Behavioral Abstraction and Lazy Evaluation

Roger B. Dannenberg, Christopher L. Fraley, and Peter Velikonja

Introduction

Fugue is a language for music composition and sound synthesis. Fugue extends the traditional approach to sound synthesis[1] with concepts borrowed from functional programming.[2] The resulting language can express signal processing algorithms for sound synthesis, musical scores, and higher level musical procedures. In contrast, the traditional approach requires separate languages for each of these tasks.

In addition to offering integration, Fugue is interactive, making it easy to explore new ideas or to synthesize and listen to sound examples. Because Fugue is based on Lisp, all of Lisp's abstraction capabilities are present, including function definitions and data structures. In addition, Fugue supports *behavioral abstraction*, which simplifies the expression of temporal behavior.

In this paper, we first describe the traditional approach to software for music synthesis, which originated with the Music *n* programs developed by Max Mathews at Bell Labs. In many respects, Fugue is a generalization and an extension of the concepts introduced by these early languages. Then we describe other approaches to score languages and sound synthesis, and present different aspects of Fugue. We also describe how Fugue was used to generate the composition "Spomin."

The traditional approach

Fugue is best understood by examining the features of traditional software synthesis systems, which include Music V,[3] cmusic,[1] and Csound.[4] All of these are based on similar principles: A "score language" is used to describe a list of notes or sounds to be synthesized. Each note in this list specifies a starting time, a duration, an instrument name, and other parameters particular to the instrument (such as pitch, loudness, and articulation). Figure 1 is based on an actual Music V score, but the notation has been altered to make it more readable. The score consists of statements to initialize function tables, followed by two notes in sequence.

A separate "orchestra language" defines a set of instruments, each of which specifies a particular signal processing algorithm. An instrument is defined by a set of interconnected signal processing steps known as unit generators. Typical unit generators are oscillators, filters, adders, and multipliers. Figures 2 and 3 show two representations of an instrument definition. Figure 2 is based on an actual Music V program that defines one instrument composed of five unit generators. The graphical

```
Score:
; initialize some function tables:
EnvTab = PWL 0 0 .99 20 .99 491 0 511 ; piece-wise linear function
OscTab = PWL 0 0 .99 50 .99 205 -.99 306 -.99 461 0 511
VibTab = SIN 1 ; 1 period of a sine function
         VibTone 0 2 1000 262 8 ; play a middle C
         VibTone 2 1 1000 330 8 ; play an E
         TER 3 ; stop at 3 seconds
```

Figure 1. An example of a stylized Music V score with two notes. The PWL and SIN operations initialize function tables used by the "orchestra." There are two notes to be played by the VibTone instrument, one starting at time 0 with duration 2, and one at time 2 with duration 1. The parameters of the notes specify starting time, duration, amplitude, pitch, and vibrato rate.

```
Orchestra:
VibTone: Amp, Hz, VibHz ;
   define the instrument VibTone
B1 = OSC Amp, 1/Dur, EnvTab
B2 = OSC Hz/100, VibHz, VibTab
B3 = ADD Hz, B2
B4 = OSC B1, B3, OscTab
     OUT B4
```

Figure 2. The "orchestra" consists of one instrument named VibTone, which is controlled by three explicit parameters: Amp, Hz, and VibHz (the starting time and duration are implicit parameters to every instrument). Reading from the bottom up, the output signal (B4) is generated by an oscillator (OSC) whose amplitude (B1) is generated by another oscillator, and whose frequency (B3) is produced by an adder (ADD). The adder sums an average frequency (Hz) with a vibrato signal (B2) produced by another oscillator. The amount of vibrato is chosen to be 1 percent of the frequency (Hz/100). The overall amplitude signal (B1) is generated by a very low frequency (1/Dur) oscillator that interpolates through an envelope function (Env) exactly once over the entire duration (Dur) of each note. The envelope is scaled by the note amplitude (Amp).

representation in Figure 3 illustrates the same computation in terms of a flow graph where each node corresponds to a unit generator in the instrument definition, and arcs between nodes represent streams of audio or control signal samples. This notion in which unit generators can be combined into a flow graph is an important contribution of Music V.

Each note in the score language gives rise to an instance (essentially a copy) of an instrument in the orchestra. This instrument instance computes sound for the duration of the note according to the parameters of the note statement. The resulting sound is typically added to that of the other notes in the score and written to a disk file. After computation, the synthesized music can be read from the disk in real time and converted into analog form for listening.

This approach has been used without much change for over two decades, indicating that it offers excellent and robust ideas. However, there are also some weaknesses. One of the most obvious problems is the separation of the score and orchestra languages. This creates a one-way flow of information from the score to the orchestra. The composer is also burdened with an additional language and the task of deciding at an early stage whether a function is better handled by the score or the orchestra.

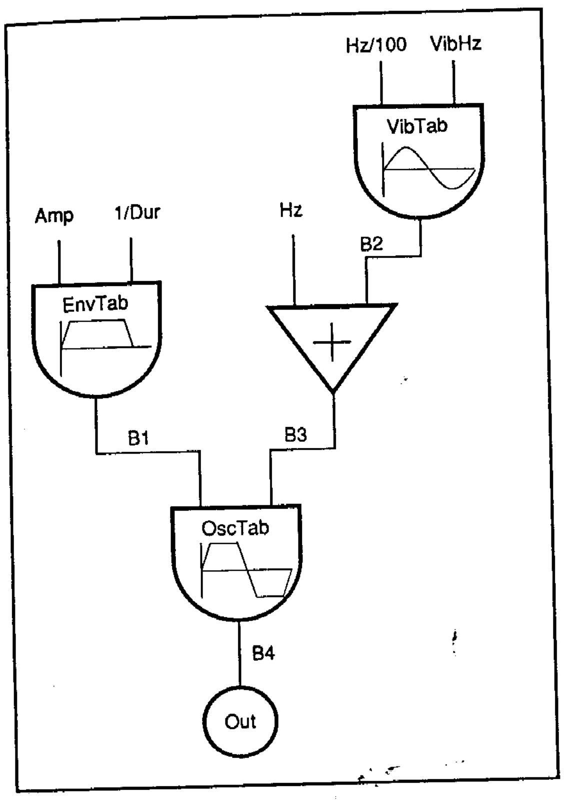


Figure 3. The VibTone instrument can be described by a dataflow graph in which arcs represent audio signals and nodes represent unit generators. There are three oscillators shown. The left input of each oscillator is the amplitude control and the right input is the frequency control. The label, for example, Env, denotes a function table that describes one period of the oscillation. Note that there is a one-to-one correspondence between this figure and the text version shown in Figure 2.

Any alternative to the Music V class of systems must continue to offer the same advantages. One of these advantages is the ability to specify the starting time and duration of each note in the score. These temporal attributes are implicit parameters to each instrument instance, specifying when to start and how many samples to compute. An extension of this concept is supported in Fugue and described in a later section.

Another important contribution of Music V is the notion of unit generators and the idea that these can be combined into a signal flow graph. Fugue offers this same capability. Although one would expect instruments to be interchangeable with unit generators, hierarchical instruments are not supported by Music V or most other synthesis languages. In Fugue, there is no distinction between instruments and unit generators, making hierarchical descriptions possible.

Other approaches

Since Music V, there have been many other developments in sound synthesis and control. In this section, we consider some of these and their relationships to Fugue.

SRL[5] is a signal processing language that represents signals as parameterized computations. SRL signals are immutable objects that can be reused. Lazy evaluation is supported in that signals are evaluated only when they are needed. SRL also supports function caching: The system will try to find an existing signal with identical parameters to avoid computing a copy. SRL lacks many musically useful concepts such as the starting time and duration of signals and behavioral abstraction, so it would not be useful as a score language.

Formes[6] takes an object-oriented approach to the computation of control signals for music synthesis. Objects in Formes encapsulate the parameters of functions and retain the state needed to compute signals incrementally. Formes objects can be combined hierarchically to construct complex signals, but Formes is not designed to compute audio directly. Instead, Formes is used to compute control information for the Chant[7] synthesis system. Formes is interesting because it successfully integrates notions of continuous control signals with discrete events such as parameter updates or note instantiations. Formes also provides a number of features in support of the notion that programmed behaviors should take place at certain times or in certain temporal relationships to other behaviors.

Arctic,[8] like Formes, is intended to compute functions of time that in turn control audio signal processing. Unlike Formes, Arctic is a functional language in which functions of time are primitive data types. However, functions of time and real numbers are the only data types implemented in Arctic, so it has difficulty expressing many state-oriented algorithms. Arctic and Formes both assume that audio signal processing is performed by a separate system; thus both languages fail to provide an integrated composition and synthesis language. However, Arctic's functional approach, its behavioral abstraction mechanism, and its general semantics are incorporated into Fugue.

In addition to signal processing support, languages can provide support for higher levels of organization. The early, nonhierarchical lists of notes have given way to more computationally oriented score languages and notations that mirror common-practice music notation. More recently, attention has focused on visual representations of music, displaying common-practice music notation, graphical notations, or visual programming languages. A problem with visual representations is the difficulty of expressing both static (note list) information and dynamic (computation) information in a single visual notation.

Kyma[9] and the Sun/Mercury Workstation[10] treat sound manipulators and sound generators as objects that can be "patched" together into a dataflow graph, and both systems provide a graphical means for making interconnections. This results in intuitive systems for synthesis, but there are problems when these systems are extended for composing larger structures. Various extensions have become necessary to handle graphs that change over time, adding complexity to programs. Symbolic processing and working with data structures are also difficult with these graph-oriented program representations. Also, both systems run all objects in synchrony, thereby assuming a global sample rate for digital signals. This last limitation could undoubtedly be overcome, but it underscores the fact that these graphical interface systems are primarily oriented toward synthesis.

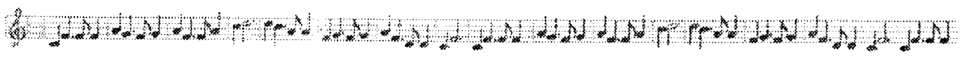
Programs as scores

Historically, musical scores have been static data structures created by composers. Traditional scores do not express computation beyond a few simple abbreviations to indicate repeats or alternate endings. There is a good reason for this: Traditional scores are data intensive and contain a wealth of detailed information. Traditionally, composers have wanted to express the results of their creativity, rather than the process of creation.

Consequently, lists of notes and their attributes have been the standard form of machine-readable scores for many years. As data structures, note lists can be transformed in time, in pitch, or along any other dimension defined by note attributes. It is generally easier to generate and manipulate data structures than programs. For example, making all the notes in a section louder is easy if the notes are represented as data. On the other hand, note lists can suffer from the fact that they are not programs. In particular, there is a schism between the "score" and the "orchestra." Controlling functions are handled by the score, while executing functions are delegated to the orchestra. There is no integration of these functions.

One way to address this dilemma is to use programs to compute note lists. This solves problems such as expanding a single "drum roll" note into a sequence of individual drum strokes. This is a very common practice, but it suffers from at least two problems. First, the composer is asked to deal with yet another language (as if two were not enough!), and second, the ultimate result does not close the gap between the score and orchestra. For example, note-generating procedures cannot use the results of signal processing functions in the orchestra, and instruments in the orchestra cannot call upon the note-generating capabilities of the score.

Fugue takes a new approach: "Scores" in Fugue are actually program expressions that, when evaluated, return audio or control signals. Two constructs are used to combine sounds or notes into



larger structures. The first combines sounds sequentially, and the second mixes sounds simultaneously:

(seq s1 s2 s3 ...) arrange each sound sequentially in time (as in a melody)
(sim s1 s2 s3 ...) arrange each sound simultaneously in time (as in a chord)

Instruments are also defined in terms of expressions that denote audio signals. For example, the following multiplies an envelope control signal by the output of an oscillator:

(s-mult (env 0.8) (osc C4))

It is possible to embed a score in an instrument as well as to call on instruments from within a score. In fact, Fugue makes no distinction whatsoever between instruments and scores.

The examples in Figures 1 and 2 can be expressed in Fugue as shown in Figure 4. This program shows how both score information and synthesis algorithms can be represented in a single expression-oriented language.

Once this unification is made, it is possible to express scores and instruments in a more flexible way. For example, in traditional synthesis systems, it is difficult to alter a phrase of many notes by a single volume envelope because this requires a hierarchical nesting of notes within the

```
;; define an equivalent to the Music V OSC by
;; combining an FMOsc with a multiplier:
(defun Music5Osc (Amp Pitch Modulation Table)
  (S-Mult Amp (FMOsc Pitch Modulation 0 Table)))

;; define VibTone in terms of Music5Osc:
(defun VibTone (Pitch VibHz)
  (Music5Osc (LFO (/ 1.0 *stretch*) 1.0 1.0 0 EnvTab) ; envelope
        Pitch
        (LFO VibHz (/ (Step-To-Hz Pitch) 100)) ; vibrato
        OscTab))

;; initialize tables for oscillators:
(setf EnvTab (MakeTable (PWL .02 .99 .491 .99 .511)))
(setf OscTab (MakeTable (PWL .05 .99 .205 .99
                        .306 -.99 .461 -.99 .511)))
(play
  (sim (at 0 (stretch 2 (vibtone c4 8.0)))
       (at 2 (stretch 1 (vibtone e4 8.0)))))
```

Figure 4. This Fugue program is equivalent to the Music V example shown in Figures 1 and 2. The instrument definition is a nested expression rather than a statement list, eliminating the need for naming intermediate results (B1 through B4). The Fugue FMOsc oscillator takes both a pitch offset and a modulation input, eliminating the need for the ADD unit generator in the Music V version. However, FMOsc does not have an amplitude modulation input, so a new function, Music5Osc, is defined to realize this aspect of the Music V OSC unit generator. The score (the last two lines) is also a nested expression, using at and stretch transformations in place of the time and duration specifications of Music V. The Amp parameter in the Music V version was omitted here because the same effect can be achieved with Fugue's loud transformation.

volume envelope. In Fugue, however, scores and signals are all nested expressions, making it very natural to represent hierarchical structures. Another possibility is to use Fugue for signal analysis, using results of the analysis to determine aspects of a score. Signal analysis is a common practice in computer music, but such programs are rarely integrated into traditional sound synthesis languages.

Perhaps the most valuable feature of Fugue is that it encourages the composer to develop a personal musical vocabulary, unencumbered by a particular model of how music, or music computation, should be structured. Of course, any language, including Fugue, is bound to influence one's approach to programming or composition. However, since Fugue supports the Music V model of computation as a subset, we can at least claim an improvement in flexibility and generality. Our early experience bears this out, as demonstrated in the later section entitled "An example."

Fugue allows the composer to treat simple instruments and sounds as building blocks from which more complex sound events are constructed. This development process is supported by the abstraction capabilities of the language and an interactive language interpreter.

Behavioral abstraction

As we mentioned earlier, one of the important features of Music V is that note starting times and durations are used to determine when and how long to instantiate an instrument. In addition to our effort to unify the score and orchestra, we must also support this same idea; otherwise, temporal aspects of compositions might be very difficult to express.

We note that starting times and durations in Music V provide a special kind of abstraction: Instruments define a class of behaviors that can have any starting time or duration. Since in Fugue we wish to support nested expressions, we view a starting time or duration as a *transformation* of time rather than as a fixed value. This abstraction is important, as it allows us to denote an operation even when its implementation is not obvious.

For example, a violinist typically lengthens a note by drawing the bow across the string for a longer period of time. Yet if the note is a tremolo (a quick back-and-forth bowing), the implementation required to lengthen the duration is to add more bow strokes. Lengthening is a type of *stretching*, an abstract notion. Its implementation must be under the control of the software instrument designer, but the user need not be aware of implementation details: The instrument simply behaves properly when asked to lengthen a note. In Fugue, this concept is extended to allow transformations of articulation, loudness, and pitch. Additional qualities can be made transformable if desired, and the composer can then extend the system with new transformation operators.

The programmer/composer defines *behaviors* that describe how to generate a sound within a *transformation context*. A context in Fugue reflects the cumulative effect of nested transformations on environmental parameters such as current time, stretch, transposition, and overall loudness. Behaviors can be hierarchical compositions of other behaviors. Once defined, a behavior can have many *instances*, each of which can be evaluated in a different context and/or with different parameters. In this way, concepts such as "drum roll" or "glissando" can be defined once but applied in many different contexts.

The definition of behaviors realized according to a context is called *behavioral abstraction*.[8] A few examples should clarify how behavioral abstractions are used in Fugue. The first example is a sequence of three sounds:

```
(seq (tremolo A3) (cue wind) (osc Bf3))
```

where cue is a behavior that simply plays a sound at a given time, and tremolo and osc play pitches

(A and B-flat below middle C). Osc and cue are built-in behaviors, while tremolo is defined by the user with other built-in behaviors. Wind is a sound, perhaps loaded from a sound file.

If we want to hear the same sequence half as loud and with the wind sound delayed by 2 seconds, we can write

```
(loud 0.5
  (seq (tremolo A3) (at 2.0 (cue wind)) (osc Bf3)))
```

Instead, suppose we wish to change the pitched sounds of the sequence. We can write

```
(transpose 3 (seq (tremolo A3) (cue wind) (osc Bf3)))
```

This will transpose the sequence up by three semitones. However, the cue abstraction overrides and prevents the transposition of wind since cue is intended to be used with unpitched sounds. Hence, only the tremolo and osc behaviors are affected.

If desired, one could replace cue with a behavior that would transpose the wind sound. In general, transformations and built-in behaviors are defined to work like traditional unit generators. As a result, most instrument definitions in Fugue support the standard transformations implicitly, but the composer is always able to customize the default behavior as needed.

Behavior definition and transformations

So far, we have seen how expressions can be used to describe note lists and signal processing operations, and how transformation operators can be applied to these expressions. In this section, we examine how new behaviors can be defined and how the composer can specify how the behavior will respond to various transformations.

Behaviors are ultimately thinly disguised Lisp functions, and they are defined using defun. In the following example, motive is defined to be a sequence of three pitches:

```
(defun motive ( )
  (seq (osc G4) (osc Bf) (osc A)))
```

Transformations can now be applied to motive with no additional programming effort:

```
(transpose -12 (motive)) ;; transpose down an octave
(stretch 2 (motive))    ;; double the duration
(loud 0.1 (motive))     ;; play motive softly
```

In practice, there are times when the default effect of transformations is not what the composer wants. The tremolo example of the previous section illustrates this point: The default way to stretch notes would be to lengthen each note, but it is understood that a tremolo is lengthened by adding notes (i.e., bow strokes). It is up to the implementer of a behavior such as tremolo to ensure that instances of the behavior respond appropriately to transformations.

A simple example will help to illustrate how a default transformation can be overridden. Consider the motive behavior defined above and imagine that the first note of the motive should always have a duration of 0.1 seconds. The remaining time should be divided equally among the other two notes. For now, we will assume that the total time is always greater than 0.1 seconds. The desired behavior is defined by the following:

```
(defun motive ( )
  (let ((str (/ (- *stretch* 0.1) 2)))
    (seq (stretch-abs 0.1 (osc G4))
        (stretch-abs str (osc Bf))
        (stretch-abs str (osc A))))))
```

In this new version of motive, the let expression computes the duration of the second and third notes and binds this value to str. This duration is the total duration minus 0.1 seconds and then divided by 2. The total duration is in turn given by *stretch*, which is part of the transformation context. The three notes are then instantiated within a seq expression as before, except each note is embedded within a stretch-abs transformation. The effect of stretch-abs (short for "stretch absolutely") is to provide a transformation in absolute rather than relative terms.

To summarize, the composer will generally override the default transformation for a behavior in two steps. First, the transformation context (e.g., *stretch*) is examined and parameters are computed to characterize the desired behavior. Second, default transformations are overridden with the computed parameters. Notice that the composer can "mix and match" default and custom transformations. In the motive example, the stretch transformation is customized, but the transpose and other transformations still operate in the standard way.

Other examples of behavioral abstractions that exhibit interesting transformations include

- instruments that get "brighter" when they are played louder,
- trills, glissandi, tremolos, and other effects that are assembled from multiple notes and are subject to various transformations,
- vibrato functions that maintain a constant frequency even when stretched,
- grace notes that should always be short,
- percussion sounds that should not be transposed or stretched,
- amplitude envelopes whose initial attack portions may remain fixed when the rest of the envelope is stretched,
- ostinato (repeating pitch and rhythm) patterns, and
- stretched tuning instruments such as the piano, where transposition by a written octave may more than double the fundamental frequency.

Developing behavioral abstractions is not always easy. In general, it is easier to program a specific instance of a sound than to describe an entire class of sounds. Behavioral abstraction is important, however, because once a behavior is implemented, it can be subjected to a wide variety of logical transformations. These are generally simpler to manage than alternate approaches, such as filling in long lists of parameters. It should also be observed that it is never a requirement to build elaborate abstractions; however, the mechanisms are available when needed.

So far, we have examined small, simple examples of Fugue expressions and transformations of them. These "toy" examples are meant to illustrate concepts without overwhelming the reader with details. In the next section, we consider a larger example from a real composition.

An example

The composition "Spomin" is derived from thousands of transformations of a single human vocal utterance. Fugue sound processing primitives were used to manipulate the source sound to varying degrees. Thus, some sounds are clearly vocal, while other more highly processed sounds bear little relation to their vocal source. Using Fugue, slices of the source were extracted, in some cases down

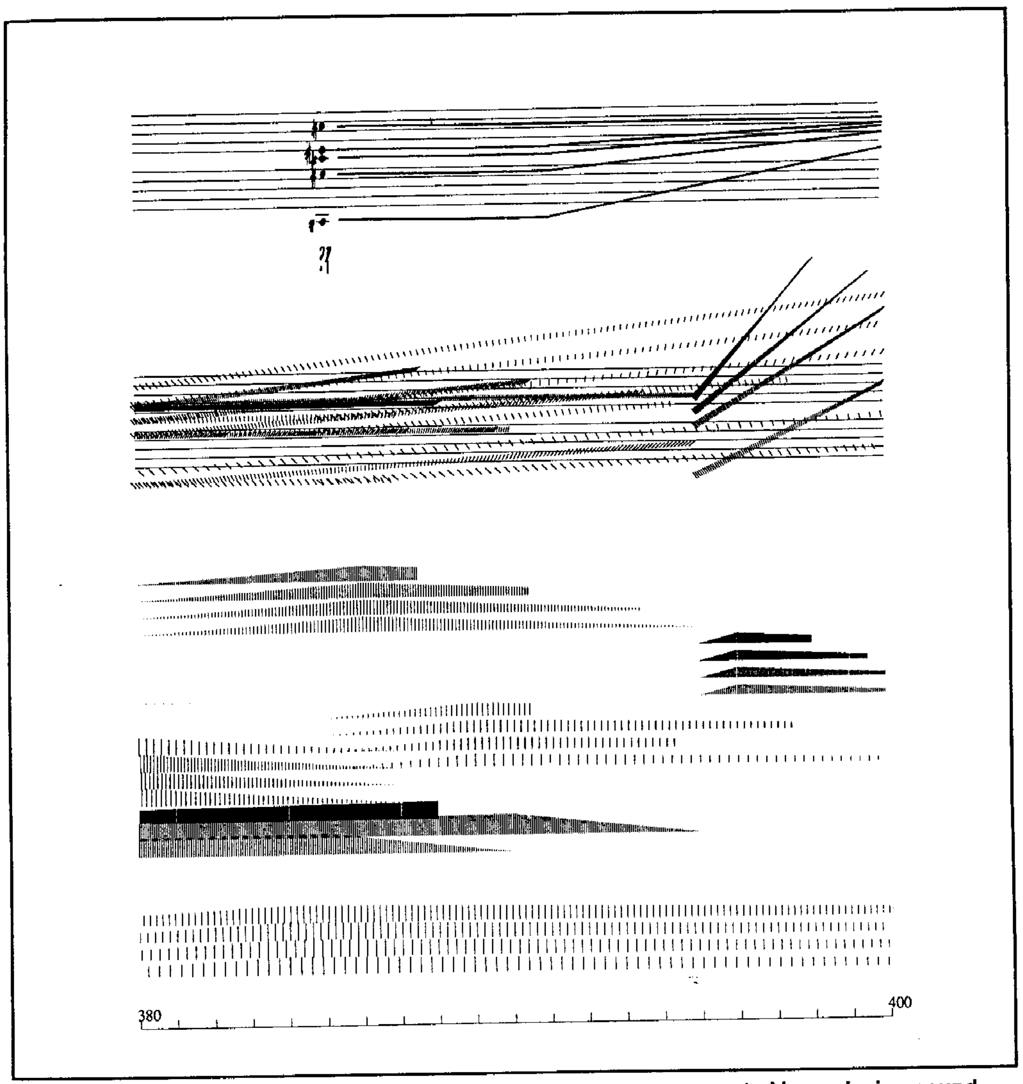


Figure 5. Page 25 from the score of "Spomin." The score was generated by replacing sound-generation modules in Fugue with modules that output graphics commands, ensuring that the score accurately reflects the sound. The final score, shown here, includes manually added music symbols.

to the level of an individual period. A period, once extracted, can be cycled to form a sustained tone (much like the technique used by sampling synthesizers). Quickly swapping periods during this cycling produces a tone with a time-varying spectrum, a technique used extensively in the latter half of the piece. In the first half, tones generated from extracted periods were used to create chords, glissandi, and chorus effects.

"Spomin" illustrates the advantage of an integrated language for expressing score information as well as signal processing. A composer can delegate signal processing routines to low-level functions, then work more abstractly using high-level functions. Fugue modules, modified to output printing information rather than digital samples, were also used to produce the graphical portion of the score. Figure 5 gives an example.

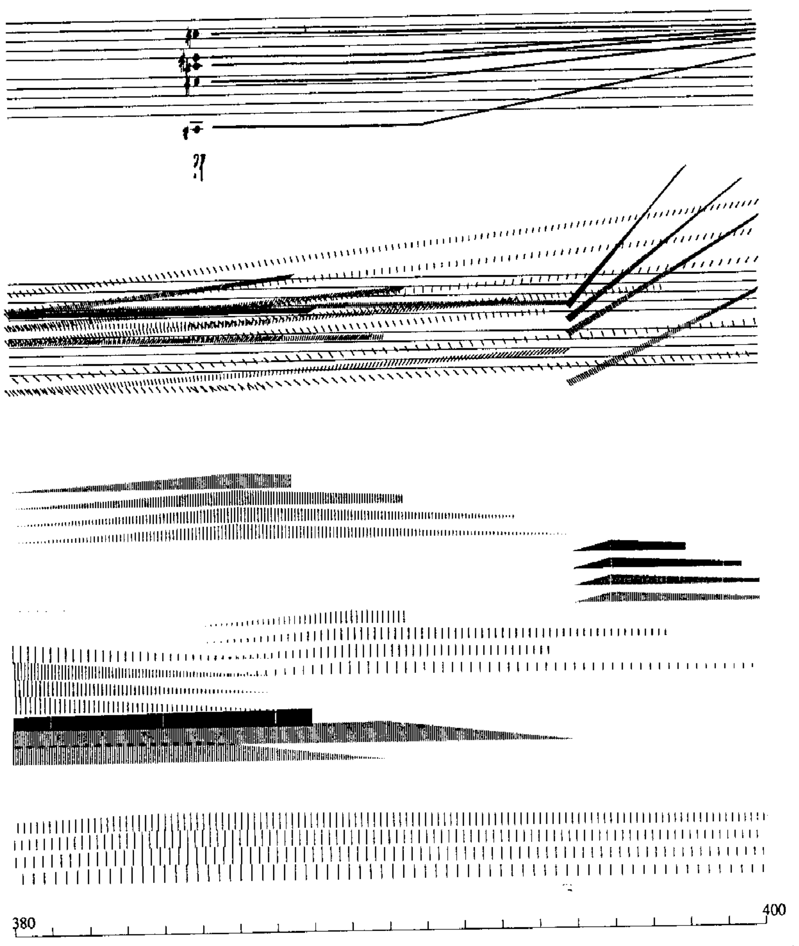


Figure 5. Page 25 from the score of "Spomin." The score was generated by replacing sound-generation modules in Fugue with modules that output graphics commands, ensuring that the score accurately reflects the sound. The final score, shown here, includes manually added music symbols.

to the level of an individual period. A period, once extracted, can be cycled to form a sustained tone (much like the technique used by sampling synthesizers). Quickly swapping periods during this cycling produces a tone with a time-varying spectrum, a technique used extensively in the latter half of the piece. In the first half, tones generated from extracted periods were used to create chords, glissandi, and chorus effects.

"Spomin" illustrates the advantage of an integrated language for expressing score information as well as signal processing. A composer can delegate signal processing routines to low-level functions, then work more abstractly using high-level functions. Fugue modules, modified to output printing information rather than digital samples, were also used to produce the graphical portion of the score. Figure 5 gives an example.

```
(defun osc-slice                              ;a slice from time start
        (sound sndpitch start end pitch dur)  ;to end is cut from
  (s-mult (pwl (* dur .5) 1.0 dur)            ;sound, oscillated at
              (osc pitch dur 0                ;pitch with an
                (extract start end sound)     ;envelope of length dur.
                sndpitch)))
                                              ;a grain consists of one such
(defun bb-grain  (period pitch dur)           ;oscillated slice. A grain
  (osc-slice bb 49.0                          ;is selected by the number
              (*period 0.008)                 ;period.  The typical
              (+ 0.008  (* period 0.008))     ;length of a slice is .008
              pitch                           ;seconds.
              dur))
                                              ;a pebble is formed from
(defun bb-pebble                              ;several (ngrains) grains.
        (ngrains period pitch dur)            ;In this example the grains
  (seqrep (g ngrains)                         ;between period and
        (bb-grain (+ period g) pitch dur)))   ;period + ngrains form
                                              ;the pebble.

(defun bb-necklace                            ;a necklace is made from a
        (npebbles ngrains pitch dur)          ;number (npebbles) of
  (seqrep (p npebbles)                        ;pebbles.
        (bb-pebble ngrains p pitch dur)))
                                              ;a necklace made from 8
(sf-od                                        ;pebbles, each containing 90
      (s-mult                                 ;grains of .05 seconds
       (pwl 25.0 1.0 28.0 .5 35.0)            ;duration pitched at e2
       (bb-necklace    8      ; npebbles      ;(tenth above middle c), has
                       90     ; ngrains       ;a piecewise-linear envelope
                       76.0   ; pitch         ;applied to it. The sound is
                       .05    ; dur           ;written (by sf-od) to an
                                              ;optical disk on a NeXT
)))                                           ;computer.
```

Figure 6. An example Fugue program taken from "Spomin," showing multiple levels of abstraction. Any level can be invoked interactively for testing or from within a higher level expression serving as a score.

The code example (see Figure 6) illustrates the layering of several levels of abstraction. In this example a short *slice* is cut from the source sound and cycled to form a *grain*. Grains of sound form *pebbles*, which are strung on a *necklace*. The second band from the top in Figure 5 (page 25 from the graphical score[11]) shows necklaces modified at the grain level to rise in pitch over time. Each grain is represented by a short line, angled to indicate where (in the source) a slice was extracted. The third, fourth, and fifth bands show timing and amplitude information. The top band shows glissandi of tones built from extracted periods.

Without the abstraction capabilities and computational support provided by Fugue, the top level of the score would consist of many thousands of notes, each corresponding to a tiny grain of sound. Such a score is technically feasible, but impractical to construct or edit by hand.

Since Fugue allows a composer to combine components to form complex structures, large sound events can be controlled with a small number of commands. The interactive environment is an important feature, as rapid feedback allows a composer greater control over sound materials. Finally, Fugue provides a supportive framework for exploring new musical forms and compositional methods. This framework allows for a composer-defined musical syntax, where music can be defined as a process rather than a simple series of notes. (It might also be mentioned that Fugue was this composer's first exposure to Lisp; Fugue is not restricted to use by programmers.)

Implementation

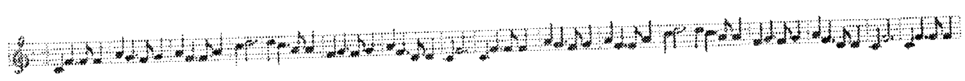
Fugue is implemented in a combination of C and XLisp to run on Unix workstations. (Written by David Betz, XLisp is an interpreter which is in turn implemented in C.) We use XLisp because it is fairly easy to extend with new data types, and it is also easy to interface with C programs for signal processing. While Lisp provides convenient and powerful interaction, C allows for efficient implementation of low-level functionality. It would be possible to use a more efficient compiled Lisp, but most of the computation time is taken by signal processing primitives, so the Lisp interpretation represents only a small overhead.

The transformation context in Fugue is implemented within Lisp. Operators such as transpose are macros that bind an element of the context (*transpose* in this case) and then evaluate the embedded expression. The binding is restored upon exit. The context is simply a set of global variables; however, it is essential that they not be set except using the transformation operators. If desired, new transformations can be added simply by defining new transformation macros, but these macros would only serve to manage additional context variables. Behaviors would also need to be created or modified to respond to the extended context as well.

New synthesis techniques can be introduced into Fugue by combining existing behaviors or by writing new sound synthesis algorithms in C and calling them from Lisp. The unit generator approach, corresponding closely to Fugue signal processing behaviors, does not seem to be sufficient to express arbitrary signal processing algorithms. Therefore, it is sometimes necessary to resort to C to add new synthesis techniques. This is one area where Fugue does not entirely live up to our goal of integrating the full spectrum of synthesis and compositional activities into one language. It is possible to do signal processing directly in Lisp, but the overhead is quite high. Nevertheless, this technique was used in the creation of "Spomin" to analyze some of the source sounds for zero-crossings and period length.

Several steps have been taken to increase time and memory efficiency, including multiple sample rates and lazy evaluation. Multiple sample rates allow for "control" signals at a low sample rate, as in the Music-11 system and Csound,[4] reducing time and memory requirements. When it becomes necessary to manipulate two sounds with different sample rates, linear interpolation is used by default to resample the lower sample rate signal to the higher sample rate. Other types of interpolation can be specified explicitly. Since there is no distinction between control and audio signals, filters can be used to modify spectra or to smooth envelopes, and multiplication can be used uniformly for gain control, amplitude envelopes, or audio-rate amplitude modulation.

The signal data type in Fugue is called a *sound*. Sounds in Fugue are implemented as an extension to Lisp. Sounds are immutable values, meaning that once a sound is created, it cannot be altered. Therefore, the implementation cannot add several sounds directly into a single buffer. Instead, each addition of two sounds produces a new sound. One might expect an implementation with immutable values to be very inefficient, but we avoid this problem through lazy evaluation. When additions (and many other operations) are performed, our implementation merely builds a small data structure describing the desired operation without actually computing any samples until absolutely necessary.



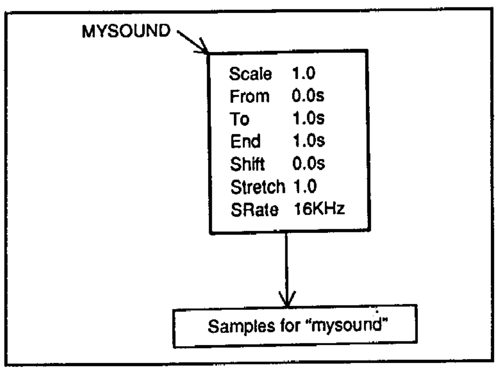


Figure 7. Mysound after (setf Mysound (SFLoad "mysound")), assuming the duration of Mysound is 1 second. The samples have been loaded from a file.

Operations can then be combined; for example, it is common that only one array is allocated to hold the result of many additions. This technique avoids many needless copy operations but is completely hidden from the user.

Example of lazy evaluation

To illustrate how the implementation works, we show what the memory structures look like at each step of a sequence of operations. The operations are

```
(setf Mysound  (sfload "mysound"))
(setf Demo  (scale 2.0  (seq  (cue Mysound)
                              (cue Mysound))))
(play Demo)
```

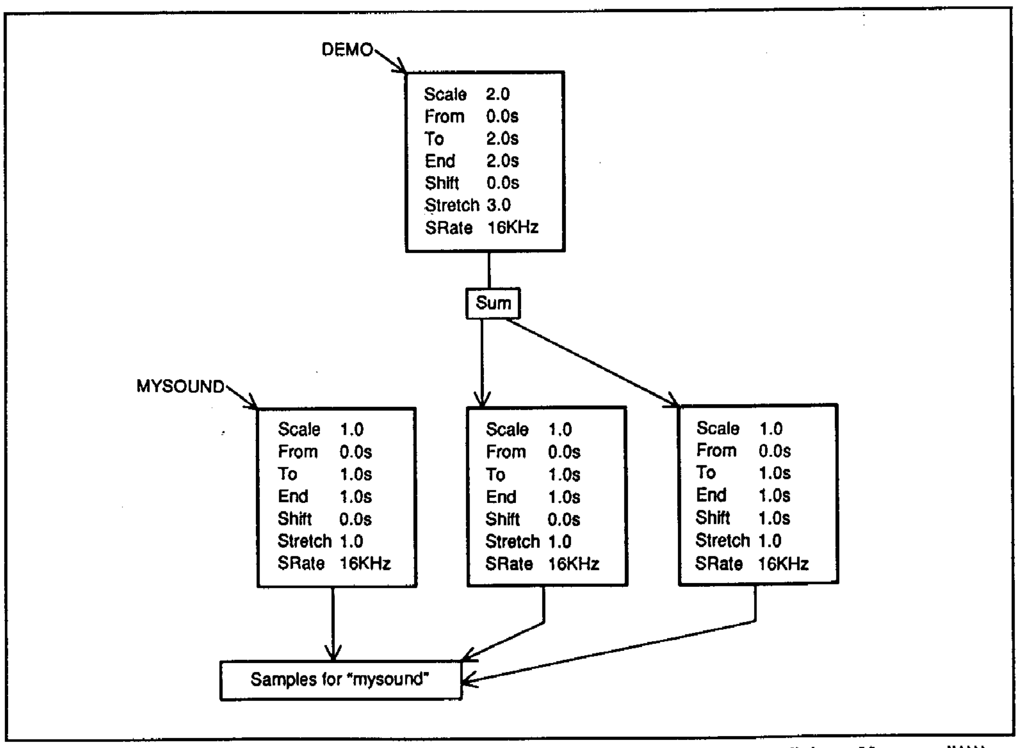
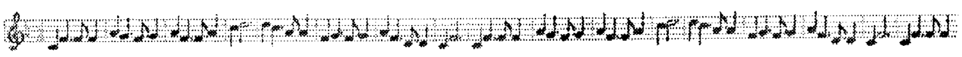


Figure 8. Demo after (setf Demo (stretch 3.0 (scale 2.0 (seq (cue 4 Mysound) (cue Mysound))))). The transformations and summation are reflected in the data structure, so no new sound samples need to be computed.



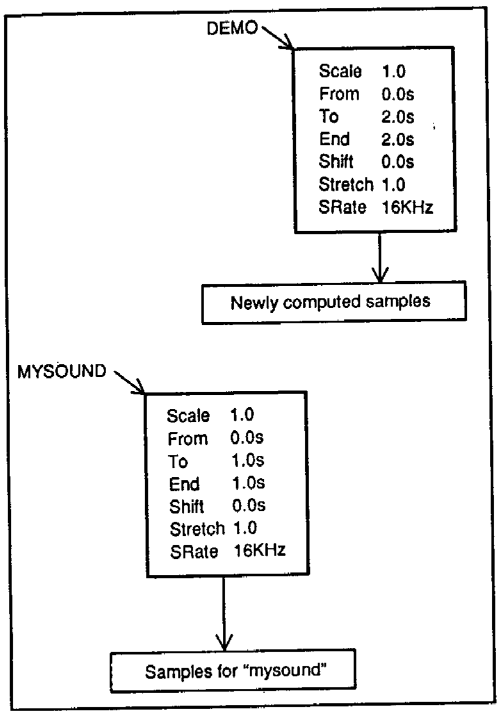


Figure 9. Demo after (play Demo). Samples have been computed according to the transformations in Figure 8. The resulting samples are cached, replacing the previous data structure with a new one.

The first line sets the variable Mysound to a sound stored in the file "mysound." Figure 7 shows the resulting configuration. The second line evaluates a score consisting of two copies of Mysound in sequence. Since only time shifting and addition are involved here, essentially no computation takes place. Figure 8 shows the resulting configuration. Notice that no addition is performed yet; instead, the sum is represented by a data structure. Finally, the third line forces the system to produce samples for Demo. The representation for Demo is replaced by one in which the actual samples have been computed and storage for samples has been allocated, as shown in Figure 9. Note that this last step is the only time that new storage for sample data is allocated and a new sound sample is actually computed. Also, note that the multiplication by 2.0 can be performed when the new sound sample is computed. Since modern processors can perform multiplications as fast as they can access memory, it is important to avoid writing intermediate results to memory and then reading them back again.

Imagine now a typical computation of the form

```
for i := 1 to 100 do
    myPiece := myPiece + MakeNote(i);
```

In Fugue, a roughly equivalent program would be

```
(seqrep (i 100) (make-note i))
```

where seqrep is a control construct that concatenates some number of instances of a behavior — in this case 100 copies of make-note.

Typically, MakeNote(i) generates a relatively short signal to be added to a much longer myPiece. Without lazy evaluation, each addition requires

(1) the allocation of memory at least the size of myPiece to hold the sum of the two signals,
(2) copying myPiece into the new memory area, and
(3) adding the result of MakeNote to form the new signal.

The cost of this computation is dominated by the cost of allocating memory and copying signals.

With lazy evaluation, each "lazy" addition simply adds another level to a tree of sum nodes like the one in Figure 8. When the final result is needed, the tree is traversed to determine the size of the result, memory is allocated, and the leaves of the summation tree are added together. Note that this technique eliminates memory allocation and signal copying to form intermediate results. Another approach (not supported by Fugue) to efficient execution would be to require the composer to explicitly allocate a buffer to hold the final sum of the MakeNote signals, and to allow the buffer to be modified by an *add-signal* operation. This approach would violate the principle that signals are immutable and place more storage management burden on the composer. In short, lazy evaluation allows Fugue to exhibit clean and simple semantics without loss of efficiency.

Storage management

Sounds are managed at two levels in our implementation. At the Lisp level, the core of the Lisp interpreter treats sounds as pointers (memory addresses). When operations are performed on sounds, the interpreter passes a sound pointer to a C function that implements the operation. Similarly, the garbage collector treats sounds like any other Lisp data.

The garbage collector locates and marks all data that can be accessed directly or indirectly, starting from variables and the runtime stack. Anything that cannot be accessed is placed on a free list for reuse at a later time. In the case of sounds, the garbage collector passes the sound pointer to a function — the destructor for sounds — before reclaiming the pointer storage.

The pointers managed by Lisp point to small Fugue structures that keep track of transformations such as amplitude scaling and time shifting. Since there may be several different structures that represent various transformations of a single sample, we store the actual sound samples separately. Each sound structure has a pointer to the samples. We use reference counts to keep track of how many structures are referencing a given set of samples so that we can deallocate the samples when they are no longer referenced.

An alternative to the current memory management system would be to represent sound structures and even sound samples within Lisp. This would allow the garbage collector to collect sounds directly. We rejected this approach to make our sound management system more efficient and less dependent on a particular Lisp implementation. Another alternative would be to store large sounds in files rather than in virtual memory. This would facilitate computing long compositions that take 100 Mbytes of storage or more. (Ten minutes of stereo audio sampled at 50 kHz and stored as 32-bit floating-point values take 240 Mbytes of storage.)

Future directions

We need to extend Fugue with more sound functions as in Moore's cmusic (distributed by the University of California at San Diego), Vercoe's Csound (distributed by the MIT Media Lab), NeXT's Sound Kit, and Lansky's Cmix (distributed by the Princeton University Music Department). These systems are popular in part because of the library of synthesis techniques they provide.

A possibility for investigation is the use of Fugue in parallel computation. Because of its functional style, Fugue programs contain explicit parallelism in the form of the sim (for "simultaneous") construct. Even when there are data dependencies such as in the seq construct, lazy evaluation often defers signal computations so that the data dependencies can be resolved immediately. Then the signal processing can proceed in parallel. If sounds in Fugue were implemented as streams, even more parallelism could be obtained by lazily evaluating streams. Furthermore, this could dramatically reduce the memory requirements for intermediate results in Fugue expressions. (Even with large virtual memories and automatic garbage collection, storage is a serious problem in the current implementation.)

An exciting potential of the lazy evaluation of streams is that of real-time execution. This would require real-time garbage collection as well. There are many opportunities for compilation and optimization of Fugue behaviors that we have not yet explored.

Many of the ideas of Fugue seem appropriate for computer graphics and computer animation. The idea of behavioral abstraction seems to fit nicely with graphical transformations (e.g., "make this truck longer" or "make this tree bigger") and also with action in animations ("run faster"). In computer animation, Fugue's notions of explicit timing and constructs for parallel and sequential behavior might be useful. For images, new constructs might be added to represent spatial as well as temporal relationships.

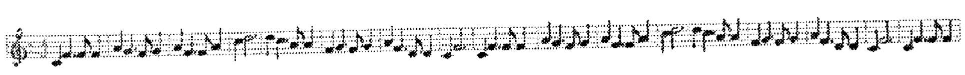
The semantics of Fugue can be extended in several ways. Currently, the context in Fugue can be extended only by someone with a fair understanding of how contexts are implemented. This should be simpler. Fugue should also support the use of MIDI files as scores so that existing music editors can be used as a source of data. In its current form, Fugue has very little support for the conventions of common-practice Western music such as beats, measures, and key signatures. Another area of improvement would be to allow time-varying transformations,[12] using signals in place of real numbers to achieve musical effects such as *accelerando* (gradual increase in overall tempo) and *crescendo* (gradual increase in overall loudness). Finally, multidimensional signals need to be supported. We plan to make these changes in a future version.

Conclusions

The word "composition" has a musical meaning and a mathematical one. In Fugue, musical composition is supported by the mathematical composition of functions. We have shown how this elegant model can unify the score and orchestra languages of traditional music synthesis systems, and how behavioral abstraction can be used to extend the temporal semantics of earlier systems. Fugue is an interpreted language and uses lazy evaluation to achieve an efficient implementation.

Acknowledgments

We thank Carnegie Mellon University and Yamaha Music Technologies for their support of this work. Dean Rubine provided Lisp consultation for the production of "Spomin," and his Strips library was used in the production of the graphical score. George Polly contributed some of the signal processing functions to Fugue.



References

1. F.R. Moore, *Elements of Computer Music*, Prentice Hall, Englewood Cliffs, N.J., 1990.
2. A.J. Field and P.G. Harrison, *Functional Programming*, Addison-Wesley, Reading, Mass., 1988.
3. M.V. Mathews, *The Technology of Computer Music*, MIT Press, Boston, 1969.
4. B. Vercoe, "Csound: A Manual for the Audio Processing System and Supporting Programs," MIT Media Lab, MIT, Cambridge, Mass., 1986.
5. G.E. Kopec, "The Signal Representation Language SRL," *IEEE Trans. Acoustics, Speech and Signal Processing*, Vol. ASSP-33, No. 4, Aug. 1985, pp. 921–932.
6. P. Cointe and X. Rodet, "Formes: An Object and Time Oriented System for Music Composition and Synthesis," *1984 Symp. LISP and Functional Programming*, ACM, New York, 1984, pp. 85–95.
7. X. Rodet, Y. Potard, and J.-B. Barriere, "The CHANT Project: From Synthesis of the Singing Voice to Synthesis in General," *Computer Music J.*, Vol. 8, No. 3, Fall 1984, pp. 15–31.
8. R.B. Dannenberg, "Expressing Temporal Behavior Declaratively," *Carnegie Mellon Computer Science 25th Anniversary Proc.*, Addison-Wesley, Reading, Mass., 1991, pp. 47–68.
9. C. Scaletti and E. Johnson, "An Interactive Graphic Environment for Object-Oriented Music Composition and Sound Synthesis," *Proc. 1988 Conf. Object-Oriented Languages and Systems*, ACM, New York, 1988, pp. 18–26.
10. X. Rodet and G. Eckel, "Dynamic Patches: Implementation and Control in the Sun-Mercury Workstation," *Proc. Int'l Computer Music Conf.*, Computer Music Assoc., San Francisco, 1988, pp. 82–89.
11. P. Velikonja, "Spomin," unpublished score, CD or cassette (companion to July 1991 *Computer*) available from IEEE Computer Society, Los Alamitos, Calif.
12. R.B. Dannenberg, "The Canon Score Language," *Computer Music J.*, Vol. 13, No. 1, Spring 1989, pp. 47–56.

Note: The musical and audio results produced by the system described in this article are an integral part of the project. For a reference of the publication in audio form and availability, and for a discussion of these musical results, see the Discography.

Roger B. Dannenberg is a senior research computer scientist at Carnegie Mellon University. His research interests include programming-language design and implementation, and the application of computer science techniques to the generation, control, and composition of computer music. He is codirector of the Piano Tutor project, whose goal is applying music understanding and expert system technology to music education. He frequently performs jazz and experimental music on trumpet or electronically.

Dannenberg received a BSEE from Rice University in 1977, an MS in computer engineering from Case Western Reserve University in 1979, and a PhD in computer science from Carnegie Mellon in 1982. He is a member of Phi Beta Kappa, Sigma Xi, Tau Beta Pi, Phi Mu Alpha, ACM, and SIGCHI, and research coordinator for the Computer Music Association.

Christopher Lee Fraley is a computer design engineer at Microsoft Corporation, where he is working on graphical user interface builders. His interests include computer applications in music and poetry. He received his BA in computer engineering from Carnegie Mellon University in 1989.

Peter Velikonja is an oboe player and composer. He has performed with several of this country's major orchestras, and for the past few years has been writing music using digital synthesis at the School of Computer Science at Carnegie Mellon University.

Velikonja received his training at Northwestern University; the Folkwang Musikhochschule in Essen, Germany, on a Fulbright grant; and Mannes College in New York City. He is presently a graduate student at Princeton University.