Fugue: Composition and Sound Synthesis With Lazy Evaluation and Behavioral Abstraction

Roger B. Dannenberg
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Christopher Lee Fraley
Microsoft Corporation
166011 NE 36 Way
Box 97017
Redmond, WA 98073-9717

Abstract

Fugue is an interactive language for music composition and synthesis. The goal of Fugue is to simplify the task of generating and manipulating sound samples while offering greater power and flexibility than other software synthesis languages. In contrast to other computer music systems, sounds in Fugue are abstract, immutable objects, and a set of functions are provided to create and manipulate these sound objects. Fugue directly supports behavioral abstraction whereby scores can be transformed using high-level abstract operations. Fugue is embedded in a Lisp environment, which provides great flexibility in manipulating scores and in performing other related symbolic processing. The semantics of Fugue are derived from Arctic and Canon which have been used for composition, research, and education at Carnegie Mellon University for several years.

Introduction

Fugue is a language for composition and sound synthesis. Features of Fugue include: (1) a full interactive environment based on Lisp, (2) a language which does not force a high-level distinction between the "score" and the "orchestra", (3) support for behavioral abstraction, (4) the ability to work both in terms of actual and perceptual start and stop times, and (5) a time- and memory-efficient implementation.

The Lisp environment provides an interactive interface, flexibility in manipulating sounds, and a base for performing other related symbolic processing. Sounds are first-class types in Fugue, hence they can be assigned to variables, passed as parameters, and stored in data structures. Storage for sounds is dynamically allocated as needed and reclaimed by automatic garbage collection. This allows "instruments" to be implemented as ordinary Lisp functions and eliminates the orchestra/score dichotomy.

Fugue semantics include behavioral abstraction as introduced by Arctic (Dannenberg, McAvinney, and Rubine 1986) and Canon (Dannenberg 1989). The motivation for behavioral abstraction is the idea that one should be able to describe behaviors that respond appropriately to their environment. For example, stretching a sound may mean one thing in the context of granular synthesis and another in the context of sampling. It almost never means to compute a short sound and then resample it to make it longer. Fugue allows the programmer to describe abstract behaviors that "know" how to stretch, transpose, change loudness, and shift in time. Transformation operators are provided to operate on these abstractions.

Composition requires that sounds be placed simultaneously together, in sequence, and at arbitrary offsets. Because musical sounds often have attack and release portions, we make a distinction between the absolute first and last samples of a sound and the perceptual start and end to which other sounds should be aligned.

Fugue is designed with powerful workstations in mind. The current implementation

In Proceedings of the 1989 International Computer Music Conference, pages 76-79, International Computer Music Association, San Francisco, 1989

relies upon virtual memory and a large disk memory to eliminate the need for explicit file access and buffer management. The low-level operations in Fugue are amenable to implementation on array processors or DSP chips when these are available. Fugue uses lazy evaluation to achieve reasonable performance without sacrificing its clean semantics.

Related Work

Many software synthesis and compositional systems have existed for years, each encountering and addressing a slightly different set of problems. To understand Fugue, it is beneficial to first review some other systems.

Music V takes a semi-functional approach to sound *generation* in that unit generators can be combined as functions applied to sample streams. The resulting instruments can be applied to parameter lists. Instruments cannot be applied to other instruments, nor can scores be constructed hierarchically. This division between sound manipulators (or generators) and parameter lists results in a corresponding separation between the orchestra and the score. Other consequences include a non-interactive environment. An interesting aspect of Music V is the idea that an *instance* of an instrument is created for each note specified in the score.

Kyma (Scaletti 1989) and the Sun/Mercury Workstation (Rodet and Eckel 1988) take a different approach, treating sound manipulators and sound generators as objects that can be "patched" together. This results in an intuitive system for synthesis, but there are problems. A level of indirection is required to manipulate graphs of unit generators which in turn manipulate sounds rather than to manipulate sounds directly. Various extensions have become necessary in order to handle graphs that change over time, but this also adds complexity to programs. Symbol processing and working with data structures are also difficult with these graph-oriented program representations. Both systems run all objects in synchrony, thereby assuming a global sample rate.

SRL (Kopec 1985) is a signal processing language that represents signals as parameterized computations. SRL signals are immutable objects that can be reused. SRL supports lazy evaluation and function caching by retaining a symbolic representation of all signals. SRL lacks in many musically useful concepts such as the starting time and duration of signals and behavioral abstraction. Also the user must explicitly free buffers when they are no longer needed.

Formes (Cointe and Rodet 1984) takes an object-oriented approach to the computation of functions of time, but Formes was not designed to compute audio directly. Formes was originally designed to compute control information for the Chant synthesis system.

Behavioral Abstraction

Fugue provides an elegant and hierarchical way to express scores that combines qualities of both note lists and executable programs. Note lists of classical score languages are attractive because they can be generated, stored, and manipulated as data. For example, making all the notes in a section louder is easy to do if the notes are represented as data. On the other hand, note lists suffer from the fact that they are not programs. In particular, there comes a time when "loudness" (and every other note-list parameter) must be interpreted to produce or control sound. The point at which interpretation starts defines the boundary between the "score" and the "orchestra".

Fugue avoids the boundary through the use of declarative-style programs that "feel" like note lists and by using the same language to define both scores and synthesis procedures. It is possible to alter Fugue scores by applying various transformations along the dimensions of time, loudness, pitch, articulation, and even sample rate.

A potential liability of these transformations is that they may transform the wrong thing. For example, in stretching a section of music that contains a trill, we do not necessarily

want the trill to slow down, and we almost certainly do not want the pitch to drop! Fugue provides defaults for transformations, but allows the programmer/composer to override the defaults with more appropriate behaviors.

Thus, the programmer/composer defines behaviors that "do the right thing" in the context of a specified set of transformations. The definition of a class of behaviors that are realized according to a context is called *behavioral abstraction*. A few examples should clarify how Fugue works. The first example is a sequence of three sounds:

```
(seq (cue wind) (cue water) (osc Bf3))
```

where `cue` is a behavior that simply plays a sound at a given time, and `osc` is a behavior that plays a given pitch. `wind` and `water` are two sounds, perhaps loaded from sound files. If we wanted to hear the same sequence at a lower amplitude and with the `water` sound delayed by 2 seconds, we could write:

```
(loud 0.2 (seq (cue wind) (at 2.0 (cue water)) (osc Bf3)))
```

Now suppose we wish to change the pitch. We could write

```
(transpose 3 (seq (cue wind) (cue water) (osc Bf3)))
```

This would have the effect of transposing the sequence up by 3 semitones. However, since the `cue` abstraction overrides and prevents transposition, only the `osc` behavior will be affected.

Signal Processing

Fugue is intended as a versatile system for the analysis, synthesis, and processing of sound. Thus far, our efforts have focussed on building an extensible kernel for Fugue and implementing some simple synthesis primitives. In the current implementation, sounds may be obtained using a generalized oscillation function or by loading sounds from files.

Primitives are also supplied to manipulate the environment in which sounds are generated and composed. These operations are used to perform cutting and splicing, stretching, controlling the amount of *legato* (sound overlap), loudness, and pitch. These operations manipulate the environment in which sounds are computed and may be applied from the score level all the way down to sound generation primitives.

System Organization

Fugue is implemented in a combination of C (Kernighan and Richie 1978) and XLisp (Betz 1986). We use XLisp because it is fairly easy to extend with a new type. (XLisp is itself written in C.) The use of two languages reflects our goal to provide an interactive and efficient environment. New synthesis techniques can be introduced by combining existing Lisp functions on sounds or by writing new sound synthesis algorithms in C and making them callable from Lisp.

Multiple sample rates allow "control" signals to exist at a low sample rate as in Music-11 (Vercoe 1981) and Csound (Vercoe 1986), reducing time and memory requirements. Linear interpolation is used (by default) when it becomes necessary to manipulate two sounds with different sample rates. There is no distinction between control signals 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 fact that sounds in Fugue are immutable values implies that the implementation cannot add several sounds directly into a buffer. If sounds are immutable, then each addition of two sounds produces a new sound and requires storage allocation. 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. This technique avoids many redundant copy operations but is completely hidden from the user.

Also hidden from the user is the use of reference counting (Pratt 1975) to reclaim storage from sounds that are no longer referenced. This reference counting scheme is integrated with the XLisp mark-and-sweep garbage collector (Shorr and Waite 1967).

Conclusion

Fugue is a new language that provides high-level operations on sounds. Fugue is unique in that is spans a range of computational tasks from score manipulation to synthesis within a single integrated language. Fugue already has an efficient implementation running on Unix workstations. We intend to improve this further by taking advantage of virtual copy and mapped file capabilities of the Mach (Accetta *et. al.* 1986) operating system and a DSP chip for signal processing. We also plan to extend Fugue with more sound functions from other systems such as Moore's Cmusic (Moore 1982), Vercoe's Csound (Vercoe 1986), NeXT's Sound Kit (Jaffe and Boynton 1989), and Lansky's Cmix (Lansky 1987).

References

Accetta, M., Baron, R. Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. "Mach: A New Kernel Foundation for UNIX Development." *Proc. of Summer Usenix*, July 1986.

Betz, D. 1986. XLISP: An Experimental Object-oriented Language, Version 1.7. (program documentation).

Cointe, P. and Rodet, X. 1984. "Formes: an Object & Time Oriented System for Music Composition and Synthesis." In *1984 Symposium on LISP and Functional Programming*. ACM Press, pp. 85-95.

Dannenberg, R. B., McAvinney, P., Rubine, D. 1986. "Arctic: A Functional Language for Real- Time Systems." *Computer Music Journal* 10(4):67-78.

Dannenberg, R. B. 1989. "The Canon Score Language." *Computer Music Journal* 13(1):47-56.

Jaffe, D. and Boynton, L. 1989. "An Overview of the Sound and Music Kits for the NeXT Computer." *Computer Music Journal* 13(2):48-55.

Kernighan, B. M. and Richie, D. M. 1978, *The C Programming Language*. Englewood Cliffs: Prentice-Hall.

Kopec, G. E. 1985. "The Signal Representation Language SRL." *IEEE Transactions Acoustics, Speech and Signal Processing*. 33(4):921-932.

Lansky, P. 1987. "CMIX" Princeton Univ. (Software and documentation).

Moore, F. R. 1982. "The Computer Audio Research Laboratory at UCSD." *Computer Music Journal* (6)1:18-29.

Pratt, T. 1975. *Programming Languages: design and implementation*. Englewood Cliffs: Prentice Hall.

Rodet, X. and Eckel, G. 1988. "Dynamic Patches: Implementation and Control in the Sun-Mercury Workstation." In *Proceedings of the 1988 International Computer Music Conference*. Computer Music Association. pp 82-89.

Scaletti, C. 1989. "The Kyma/Platypus Computer Music Workstation." *Computer Music Journal* 13(2):23-38.

Schorr, H. and Waite, W. 1967. "An Efficient and Machine Independent Procedure for Garbage Collection in Various List Structures." *Comm. ACM* 10(8):501-506.

Vercoe, B. 1981. *Reference Manual for the MUSIC 11 Sound Synthesis Language*. MIT Experimental Music Studio.

Vercoe, B. 1986. *CSOUND: A Manual for the Audio Processing System and Supporting Programs*. MIT Media Lab.