Musical programming in Scheme

Lee Richard Boynton,
NoXT Computer, Inc.
900 Chesapeake Drive, Redwood City, CA 94063, USA
phone: (415) 366-0900, fax: (415) 780 3715, e-mail: lboynton@next.com

ABSTRACT

This paper discusses high-level programming language features desirable for musical purposes, and introduces a set of such features as extensions to the Scheme programming language. The features address parallel control flow, synchronization, and event timing, and all can be written in the Scheme language itself, so they are portable. Specific implementations may perform actions of arbitrary nature: writing to a score file, playing MIDI instruments, or playing sampled sounds, and adding a machine-dependent clock allows execution in real time, with no change in semantics. None of the abstractions described is this paper impose any particular musical representation.

1. INTRODUCTION

Programming music on a computer can be hard, for reasons that do not stem from music. Most computer languages do not provide adequate support for the concepts that music introduces, like parallel control flow and synchronization. Many efforts have been made to retrofit existing languages with facilities to make music programming easier. Typically, the solutions are complicated in one way or another, and have intrusive limitations or implementation problems. Although existing solutions make musical programming possible, they do not make it reasonable. While much progress has been made in graphical programming languages like Max (Pucette 1991), some algorithms are hard to express naturally with graphics. Many of us just want a programming language that makes expressing musical ideas reasonably painless, and we would like it to be a standard language that will not disappear in a few years.

2. SCHEME

The algorithmic language Scheme is a dialect of the Lisp programming language. It is small, as languages go, but powerful, embodying concepts that other languages do not have. Like Common Lisp, it handles all sorts of numbers (integers, rationals, reals, complex) with a distinction between exactness and inexactness (1/3 + 1/3 is exactly 2/3). In Scheme, procedures are first-class objects, delayed evaluation is directly supported, and tail recursion is properly handled. These and other desirable aspects of the language are well documented elsewhere (Clinger 1991) (Dybvig 1987) (IEEE 1991). The unique feature of Scheme takes advantage of in work described herein is the degree of continuations.

Continuations are first-class objects in Scheme that represent the state of a thread of a computation. At any time during the execution of a program, a continuation may be created that, when called, continues execution at the point it was created. The continuation captures bindings and stack information, and may be called without restrictions. This makes it possible to write coroutines in a transparent and portable manner, without ever getting into lower level operating system issues. In fact, Scheme continuations are completely independent of the underlying operating system. The following sections describe various music-oriented features implemented in Scheme.

3. TIMING

Two expressions are introduced to handle timing: sleep and now. Sleep measures time durations. For example:

(define (note pitch duration velocity)
  (midi-note-on pitch velocity)
  (sleep duration)
  (midi-note-off pitch))
defines the function note that takes a key number, a duration value, and a velocity value as arguments and plays the appropriate note on some MIDI instrument. It does not return until the note has been completed, which is generally useful for building sequences of notes:

(define (up root)
    (note root 1/3 100)
    (note (+ root 4) 1/3 100)
    (note (+ root 7) 4/3 100))

The now function returns the current time. Note that the up phrase takes 2 time units to complete:

(begin
    (display (now)) ; prints 't'
    (up c4)
    (display (now))) ; prints '2'

Begin is Scheme's name for the traditional Lisp prog construct, which executes expressions sequentially. The two expressions sleep and now determine the basic time system, in arbitrary units. The note and up functions are just examples, and could perform other operations than MIDI output, i.e., changing a parameter on a DSP or writing a line of text to a score file.

4. PARALLELISM

The note example given previously could more or less be written in any language, given a suitable library containing the MIDI output and sleep functions. The problem that arises in most other languages is the difficulty in playing several of these notes or sequential phrases at once. In Scheme, it is possible to write a parallel macro that does this. An example of such a construct is:

(parallel
    (up c4)
    (begin (sleep 1) (up c3))
    (begin (up c5) (up f5) (up c5)))

This would execute three expressions (or threads) in parallel: up at middle C, up an octave lower and delayed by a time unit, and a sequence consisting of three ups at different pitches. Parallel returns when all expressions have completed; in this case 6 time units. The underlying scheduler is non-preemptive, and calls to sleep or its equivalent need to occur in a thread to let other threads run. This tends to occur naturally, as the examples show. A preemptive scheduler is possible, but requires an interrupt facility which is not part of the Scheme standard, and has the disadvantage of not being portable or deterministic. Parallel statements may be nested arbitrarily.

5. SYNCHRONIZATION

Although the examples look a lot like the Canon score language (Dannenberg 1989) — and in fact with suitable macros could look exactly like Canon — the semantics are different: with Canon, programs like this create and return scores as data objects. Here, we are specifying a program's behavior, not data it returns, and the output results entirely from side effects of the program, which is executed (via an underlying scheduler) in time order. This allows parallel threads to communicate with each other. To take advantage of this, we need to introduce synchronization expressions to wait for and signal events, or messages between threads. The following example illustrates the use of these by playing two phrases in parallel, one of them starting when the other reaches a certain point, which may not be known in advance.

240
There can be many threads waiting for a given cue, because Signal broadcasts the cue. Signal accepts an optional argument: a value that gets passed to wait, such that when wait returns, it returns that value.

```
(parallel
  (begin (up c3) (sleep (some-calculated-quantity)) (signal 'foo) (up c2))
  (begin (wait 'foo) (up c4))
)
```

Wait also accepts an optional argument: a timeout, after which wait is aborted. In this case, wait returns false, which is printed in Scheme as "#f".

```
(parallel
  (signal 'foo 'bar)
  (display (wait 'foo)) ; prints 'bar
)
```

Signal and wait can be considered basic input and output facilities for the parallel threads of execution, and are generally useful as such (Hoare 1985). For example, event-oriented MIDI input may be implemented to (signal 'midi-input list-of-bytes), allowing several different threads to read input simultaneously by calling (wait 'midi-input duration), where duration specifies the segment of time (starting at the current time) that MIDI input is looked at. This could be useful for score following, among other things, with different threads looking simultaneously for different patterns in a single input stream, during specific time windows.

6. CONTEXTUAL DEFAULTS

Although transformations in the sense of Canon are not possible (since there is no actual score created to operate on), transformations in the xyle of the FORMEs system (Roder and Coine 1984) are possible: global values can be created, modified and shadowed by descendents in the thread hierarchy. Some Scheme implementations provide something known as fluid-let to simulate dynamic scoping (like special variables in Common Lisp), but this is not part of the Scheme standard, and is cause for some ambiguities with respect to the normal lexical scoping. It is clearer to provide a different name space for such dynamic global variables, and make use of such variables explicit in the code (Haymes and Friedman 1987). This can be done with three expressions: define-default, default, and let-default. For example, to define up with no arguments, and yet make it transposable:

```
(define-default pitch c4)
(define (up)
  (note (default pitch) 1/3 100)
  (note (+ (default pitch) 4) 1/3 100)
  (note (+ (default pitch) 7) 4/3 100))

(let-default ((pitch d3))
  (up)); play it at d3
(let-default ((pitch (+ (default pitch) 4)))
  (up)); play it at 4 semitones higher
```

Obviously, macros could be defined that make common instances of this facility much easier to use. For example, the latter case above could be the macro expansion of (transpose 4 (up)). By defining a default value to be a function instead of data, dynamically changing data (i.e. envelopes) can be handled.
7. IMPLEMENTATION

The portable version of the system presented here, which is written in terms of Scheme expressions and uses no input/output provided by a particular implementation, is a non-real-time system. A real-time version has been written for the NeXT computer. How this system runs in real-time is controlled by the latency (or scheduler advance) (Boynton et al. 1986) of the system, which may be set with the set-latency function. The latency determines how much in advance of a given time things will execute, and thus represents the expected latency, or overheard, of execution. When set to an sufficiently large number, it prevents the scheduler from ever actually waiting, and allows calculation of a state file quickly out of real time. For real-time operation, the latency can be set to some small number like a tenth of a second. In this case, up to 0.1 seconds of execution time without sleeping may be incurred before causing timing errors, but real-time input can only sensibly affect events 0.1 seconds or more in the future. Obviously, the latency involves tradeoffs involving the musical goal, the speed of the computer, and the efficiency of the Scheme interpreter.

The virtual time tags that the scheduler maintains are passed to the NeXT MIDI and sound drivers, which take advantage of them to accurately schedule the output. In effect, the data is committed to the driver early, removing it from the realm of control or interruption (Boynton 1987). For example, if a write list-of-bytes) would time-tag the bytes with the current logical time, which is some time in the future (determined by the latency), and the driver would judge them out at the latest moment, usually in an interrupt handler, for accurate timing. The list of bytes is atomic - the scheduler will ensure that they go out the MIDI port without interruption from MIDI output of other threads. To handle MIDI input, the scheduler listens on the jMIDI input port while waiting or sleeping, parses any received data, and if a complete MIDI message is assembled, performs a (signal 'midi-input list-of-bytes) to notify the program that the data is available. All pending calls of the form (wait 'midi-input) would then return with the MIDI data in a list.

Playing sound files directly on the NeXT Computer's DAC's is also supported. The NeXT 3.0 sound driver (Mimick 1992) supports real-time mixing and timed queues, so it is a simple matter to provide a function largely equivalent to one that plays a sound instead. For example, (playsoundfile filename [duration [volume [pan]]]) plays the indicated sound file and returns after the optional duration (which defaults to the duration of the sound). The optional volume and pan values, which range from 0 to 1, determine the volume and placement in the stereo field for this particular sound. MIDI and sound have synchronized onset, and playsoundfile may be used anywhere the note function was used earlier in the paper. Other output possibilities are imaginable.

8. REFERENCES


242