Implementing a Polyphonic MIDI Software Synthesizer using Coroutines, Realtime Garbage Collection, Closures, Auto-Allocated Variables, Dynamic Scoping, and Continuation Passing Style Programming

Kjetil Matheussen
Norwegian Center for Technology in Music and the Arts. (NOTAM)
k.s.matheussen@notam02.no

Abstract

This paper demonstrates a few programming techniques for low-latency sample-by-sample audio programming. Some of them may not have been used for this purpose before. The demonstrated techniques are: Realtime memory allocation, realtime garbage collector, storing instrument data implicitly in closures, auto-allocated variables, handling signal buses using dynamic scoping, and continuation passing style programming.

Keywords

Audio programming, realtime garbage collection, coroutines, dynamic scoping, Continuation Passing Style.

1 Introduction

This paper demonstrates how to implement a MIDI software synthesizer (MIDI soft synth) using some unusual audio programming techniques. The examples are written for *Snd-RT* [Matheussen, 2008], an experimental audio programming system supporting these techniques. The techniques firstly emphasize convenience (i.e. few lines of code, and easy to read and modify), and not performance. Snd-RT[1] runs on top of Snd[2] which again runs on top of the Scheme interpreter Guile.[3] Guile helps gluing all parts together.

It is common in music programming only to compute the sounds themselves in a realtime priority thread. Scheduling new notes, allocation of data, data initialization, etc. are usually performed in a thread which has a lower priority than the audio thread. Doing it this way helps to ensure constant and predictable CPU usage for the audio thread. But writing code that way is also more complicated. At least, when all samples are calculated one by one. If however the programming only concerns handling blocks of samples where we only control a signal graph, there are several high level alternatives which makes it relatively easy to do a straightforward implementation of a MIDI soft synth. Examples of such high level music programming systems are SuperCollider [McCartney, 2002], Pd [Puckette, 2002], CSound[4] and many others.

But this paper does not describe use of block processing. In this paper, all samples are individually calculated. The paper also explores possible advantages of doing everything, allocation, initialization, scheduling, etc., from inside the realtime audio thread.

At least it looks like everything is performed inside the realtime audio thread. The underlying implementation is free to reorganize the code any way it wants, although such reorganizing is not performed in Snd-RT yet.

Future work is making code using these techniques perform equally, or perhaps even better, than code where allocation and initialization of data is explicitly written not to run in the realtime audio thread.

2 MIDI software synthesizer

The reason for demonstrating a MIDI soft synth instead of other types of music programs such as a granular synthesis generator or a reverb, is that the behavior of a MIDI soft synth is well known, plus that a MIDI soft synth contains many common challenges in audio and music programming:

1. Generating samples. To hear sound, we need to generate samples at the *Audio Rate*.
2. Handling Events. MIDI data are read at a rate lower than the audio rate. This rate is commonly called the *Control Rate*.

[1] http://archive.notam02.no/arkiv/doc/snd-rt/
[2] http://ccrma.stanford.edu/software/snd/
[3] http://www.gnu.org/software/guile/guile.html
[4] http://www.csound.com

3. Variable polyphony. Sometimes no notes are playing, sometimes maybe 30 notes are playing.

4. Data allocation. Each playing note requires some data to keep track of frequency, phase, envelope position, volume, etc. The challenges are; How do we allocate memory for the data? When do we allocate memory for the data? How do we store the memory holding the data? When do we initialize the data?

5. Bus routing. The sound coming from the tone generators is commonly routed both through an envelope and a reverb. In addition, the tones may be autopanned, i.e. panned differently between two loudspeakers depending on the note height (similar to the direction of the sound coming from a piano or a pipe organ).

3 Common Syntax for the Examples

The examples are written for a variant of the programming language Scheme [Steele and Sussman, 1978]. Scheme is a functional language with imperative operators and static scoping.

A number of additional macros and special operators have been added to the language, and some of them are documented here because of the examples later in the paper.

(<rt-stalin>...) is a macro which first transforms the code inside the block into clean R4RS code [Clinger and Rees, 1991] understood by the Stalin Scheme compiler.[5] (Stalin Scheme is an R4RS compiler). After Stalin is finished compiling the code, the produced object file is dynamically linked into Snd-RT and scheduled to immediately run inside the realtime audio thread.

(define-stalin *signature* **...)** defines variables and functions which are automatically inserted into the generated Stalin scheme code if needed. The syntax is similar to *define*.

(spawn ...**)** spawns a new coroutine [Conway, 1963; Dahl and Nygaard, 1966]. Coroutines are stored in a priority queue and it is not necessary to explicitly call the spawned coroutine to make it run. The spawned coroutine will run automatically as soon[6] as the current coroutine yields (by calling ***yield*** or ***wait***), or the current coroutine ends.

Coroutines are convenient in music programming since it often turns out practical to let one dedicated coroutine handle only one voice, instead of mixing the voices manually. Furthermore, arbitrarily placed pauses and breaks are relatively easy to implement when using coroutines, and therefore, supporting dynamic control rate similar to ChucK [Wang and Cook, 2003] comes for free.

(wait n**)** waits n number of frames before continuing the execution of the current coroutine.

(sound ...**)** spawns a special kind of coroutine where the code inside ***sound*** is called one time per sample. (***sound*** coroutines are stored in a tree and not in a priority queue since the order of execution for ***sound*** coroutines depends on the bus system and not when they are scheduled to wake up.)

A simple version of the ***sound*** macro, called *my-sound*, can be implemented like this:

```
(define-stalin-macro (my-sound . body)
  `(spawn
     (while #t
       ,@body
       (wait 1))))
```

However, *my-sound* is inefficient compared to ***sound*** since *my-sound* is likely to do a coroutine context switch at every call to ***wait***.[7] ***sound*** doesn't suffer from this problem since it is run in a special mode. This mode makes it possible to run tight loops which does not cause a context switch until the next scheduled event.

(out <*channel***>** *sample***)** sends out data to the *current bus* at the *current time*. (the *current bus* and the *current time* can be thought of as global variables which are implicitly read from and written to by many

[5] Stalin - a STAtic Language ImplementatioN, http://cobweb.ecn.purdue.edu/ qobi/software.html.

[6] Unless other coroutines are placed earlier in the queue.

[7] I.e. if two or more *my-sound* blocks or ***sound*** blocks run simultaneously, and at least one of them is a *my-sound* block, there will be at least two coroutine context switches at every sound frame.

operators in the system)[8] By default, the *current bus* is connected to the sound card, but this can be overridden by using the ***in*** macro which is explained in more detail later.

If the *channel* argument is omitted, the *sample* is written both to channel 0 and 1.

It makes sense only to use ***out*** inside a ***sound*** block. The following example plays a 400Hz sine sound to the sound card:

```
(<rt-stalin>
  (let ((phase 0.0))
    (sound
      (out (sin phase))
      (inc! phase (hz->radians 400)))))
```

(range *varname start end ...***)** is a simple loop iterator macro which can be implemented like this:[9]

```
(define-macro (range varname start end . body)
  (define loop (gensym))
  `(let ,loop ((,varname ,start))
     (cond ((<,var ,end)
            ,@body
            (,loop (+ ,varname 1))))))
```

(wait-midi :*options ...***)** waits until MIDI data is received, either from an external interface, or from another program.

wait-midi has a few options to specify the kind of MIDI data it is waiting for. In the examples in this paper, the following options for ***wait-midi*** are used:

:***command note-on***
: Only wait for a *note on* MIDI message.

:***command note-off***
: Only wait for a *note off* MIDI message.

:***note*** *number*
: Only wait for a note which has MIDI note number *number*.

Inside the ***wait-midi*** block we also have access to data created from the incoming midi event. In this paper we use ***(midi-vol)*** for getting the velocity (converted to a number between 0.0 and 1.0), and ***(midi-note)*** for getting the MIDI note number.

[8]Internally, the *current bus* is a coroutine-local variable, while the *current time* is a global variable.

[9]The actual implementation used in *Snd-RT* also makes sure that "end" is always evaluated only one time.

:where is just another way to declare local variables. For example,

```
(+ 2 b
  :where b 50)
```

is another way of writing

```
(let ((b 50))
  (+ 2 b))
```

There are three reason for using ***:where*** instead of ***let***. The first reason is that the use of ***:where*** requires less parenthesis. The second reason is that reading the code sometimes sounds more natural this way. (I.e "add 2 and b, where b is 50" instead of "let b be 50, add 2 and b".) The third reason is that it's sometimes easier to understand the code if you know what you want to do with a variable, before it is defined.

4 Basic MIDI Soft Synth

We start by showing what is probably the simplest way to implement a MIDI soft synth:

```
(range note-num 0 128
  (<rt-stalin>
    (define phase 0.0)
    (define volume 0.0)
    (sound
      (out (* volume (sin phase))))
      (inc! phase (midi->radians note-num)))
    (while #t
      (wait-midi :command note-on :note note-num
        (set! volume (midi-vol)))
      (wait-midi :command note-off :note note-num
        (set! volume 0.0)))))
```

This program runs 128 instruments simultaneously. Each instrument is responsible for playing one tone. 128 variables holding volume are also used for communicating between the parts of the code which plays sound (running at the *audio rate*), and the parts of the code which reads MIDI information (running at the *control rate*[10]).

There are several things in this version which are not optimal. Most important is that you

[10]Note that the *control rate* in Snd-RT is *dynamic*, similar to the music programming system *ChucK*. *Dynamic control rate* means that the smallest available time-difference between events is not set to a fixed number, but can vary. In ChucK, control rate events are measured in floating numbers, while in Snd-RT the measurement is in frames. So In Chuck, the time difference can be very small, while in Snd-RT, it can not be smaller than 1 frame.

would normally not let all instruments play all the time, causing unnecessary CPU usage. You would also normally limit the polyphony to a fixed number, for instance 32 or 64 simultaneously sounds, and then immediately schedule new notes to a free instrument, if there is one.

5 Realtime Memory Allocation

As mentioned, everything inside ***<rt-stalin>*** runs in the audio realtime thread. Allocating memory inside the audio thread using the OS allocation function may cause surprising glitches in sound since it is not guaranteed to be an $O(1)$ allocator, meaning that it may not always spend the same amount of time. Therefore, *Snd-RT* allocates memory using the Rollendurchmesserzeitsammler [Matheussen, 2009] garbage collector instead. The memory allocator in Rollendurchmesserzeitsammler is not only running in $O(1)$, but it also allocates memory extremely efficiently. [Matheussen, 2009]

In the following example, it's clearer that instrument data are actually stored in closures which are allocated during runtime.[11] In addition, the 128 spawned coroutines themselves require some memory which also needs to be allocated:

```
(<rt-stalin>
  (range note-num 0 128
   (spawn
     (define phase 0.0)
     (define volume 0.0)
     (sound
       (out (* volume (sin phase))))
       (inc! phase (midi->radians note-num)))
     (while #t
       (wait-midi :command note-on :note note-num
         (set! volume (midi-vol)))
       (wait-midi :command note-off :note note-num
         (set! volume 0.0)))))
```

6 Realtime Garbage Collection. (Creating new instruments only when needed)

The previous version of the MIDI soft synth did allocate some memory. However, since all memory required for the lifetime of the program were allocated during startup, it was not necessary to free any memory during runtime.

But in the following example, we simplify the code further by creating new tones only when they are needed. And to do that, it is necessary to free memory used by sounds not playing anymore to avoid running out of memory. Luckily though, freeing memory is taken care of automatically by the Rollendurchmesserzeitsammler garbage collector, so we don't have to do anything special:

```
 1| (define-stalin (softsynth)
 2|   (while #t
 3|     (wait-midi :command note-on
 4|       (define osc  (make-oscil :freq (midi->hz (midi-note))))
 5|       (define tone (sound (out (* (midi-vol) (oscil osc)))))
 6|       (spawn
 7|          (wait-midi :command note-off :note (midi-note)
 8|            (stop tone))))))
 9|
10| (<rt-stalin>
11|   (softsynth))
```

In this program, when a *note-on* message is received at line 3, two coroutines are scheduled:

1. A ***sound*** coroutine at line 5.
2. A regular coroutine at line 6.

Afterwards, the execution immediately jumps back to line 3 again, ready to schedule new notes.

So the MIDI soft synth is still polyphonic, and contrary to the earlier versions, the CPU is now the only factor limiting the number of simultaneously playing sounds.[12]

7 Auto-Allocated Variables

In the following modification, the CLM [Schottstaedt, 1994] oscillator *oscil* will be implicitly and automatically allocated first time the function *oscil* is called. After the generator is allocated, a pointer to it is stored in a special memory slot in the current coroutine.

Since *oscil* is called from inside a ***sound*** coroutine, it is natural to store the generator in the coroutine itself to avoid all tones using the same oscillator, which would happen if the auto-allocated variable had been stored in a global variable. The new definition of *softsynth* now looks like this:

```
(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (define tone
         (sound (out (* (midi-vol)
                        (oscil :freq (midi->hz (midi-note)))))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (stop tone))))))
```

[11] Note that memory allocation performed before any *sound* block can easily be run in a non-realtime thread before scheduling the rest of the code to run in realtime. But that is just an optimization.

[12] Letting the CPU be the only factor to limit polyphony is not necessarily a good thing, but doing so in this case makes the example very simple.

The difference between this version and the previous one is subtle. But if we instead look at the reverb instrument in the next section, it would span twice as many lines of code, and the code using the reverb would require additional logic to create the instrument.

8 Adding Reverb. (Introducing signal buses)

A MIDI soft synth might sound unprofessional or unnatural without some reverb. In this example we implement John Chowning's reverb[13] and connect it to the output of the MIDI soft synth by using the built-in signal bus system:

```
(define-stalin (reverb input)
  (delay :size (* .013 (mus-srate))
    (+ (comb :scaler 0.742 :size  9601 allpass-composed)
       (comb :scaler 0.733 :size 10007 allpass-composed)
       (comb :scaler 0.715 :size 10799 allpass-composed)
       (comb :scaler 0.697 :size 11597 allpass-composed)
       :where allpass-composed
         (send input :through
           (all-pass :feedback -0.7 :feedforward 0.7)
           (all-pass :feedback -0.7 :feedforward 0.7)
           (all-pass :feedback -0.7 :feedforward 0.7)
           (all-pass :feedback -0.7 :feedforward 0.7)))))

(define-stalin bus (make-bus))

(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (define tone
        (sound
          (write-bus bus
                     (* (midi-vol)
                        (oscil :freq (midi->hz (midi-note)))))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (stop tone))))))

(define-stalin (fx-ctrl input clean wet processor)
  (+ (* clean input)
     (* wet (processor input))))

(<rt-stalin>
  (spawn
    (softsynth))
  (sound
    (out (fx-ctrl (read-bus bus)
                  0.5 0.09
                  reverb))))
```

Signal buses are far from being an "unusual technique", but in text based languages they are in disadvantage compared to graphical music languages such as Max [Puckette, 2002] or Pd. In text based languages it's inconvenient to write to buses, read from buses, and most importantly; it's hard to see the signal flow. However, signal buses (or something which provides similar functionality) are necessary, so it would be nice to have a better way to handle them.

[13] as implemented by Bill Schottstaedt in the file "jc-reverb.scm" included with Snd. The *fx-ctrl* function is a copy of the function *fxctrl* implemented in Faust's Freeverb example.

9 Routing Signals with Dynamic Scoping. (Getting rid of manually handling sound buses)

A slightly less verbose way to create, read and write signal buses is to use dynamic scoping to route signals. The bus itself is stored in a coroutine-local variable and created using the ***in*** macro.

Dynamic scoping comes from the fact that ***out*** writes to the bus which was last set up by ***in***. In other words, the scope for the *current bus* (the bus used by ***out***) follows the execution of the program. If ***out*** isn't (somehow) called from ***in***, it will instead write to the bus connected to the soundcard.

For instance, instead of writing:

```
(define-stalin bus (make-bus))

(define-stalin (instr1)
  (sound (write-bus bus 0.5)))

(define-stalin (instr2)
  (sound (write-bus bus -0.5)))

(<rt-stalin>
  (instr1)
  (instr2)
  (sound
    (out (read-bus bus))))
```

we can write:

```
(define-stalin (instr1)
  (sound (out 0.5)))

(define-stalin (instr2)
  (sound (out -0.5)))

(<rt-stalin>
  (sound
    (out (in (instr1)
             (instr2)))))
```

What happened here was that the first time ***in*** was called in the main block, it spawned a new coroutine and created a new bus. The new coroutine then ran immediately, and the first thing it did was to change the *current bus* to the newly created bus. The ***in*** macro also made sure that all ***sound*** blocks called from within the ***in*** macro (i.e. the ones created in *instr1* and *instr2*) is going to run before the main ***sound*** block. (That's how ***sound*** coroutines are stored in a tree)

When transforming the MIDI soft synth to use ***in*** instead of manually handling buses, it will look like this:

```
;; <The reverb instrument is unchanged>

;; Don't need the bus anymore:
(define-stalin bus (make-bus))

;; softsynth reverted back to the previous version:
(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (define tone
         (sound (out (* (midi-vol)
                        (oscil :freq (midi->hz (midi-note)))))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (stop tone))))))

;; A simpler main block:
(<rt-stalin>
  (sound
    (out (fx-ctrl (in (softsynth))
                  0.5 0.09
                 reverb))))
```

10 CPS Sound Generators. (Adding stereo reverb and autopanning)

Using coroutine-local variables was convenient in the previous examples. But what happens if we want to implement autopanning and (a very simple) stereo reverb, as illustrated by the graph below?

```
              +-- reverb -> out ch 0
             /
softsynth--<
             \
              +-- reverb -> out ch 1
```

First, lets try with the tools we have used so far:

```
(define-stalin (stereo-pan input c)
  (let* ((sqrt2/2 (/ (sqrt 2) 2))
         (angle (- pi/4 (* c pi/2)))
         (left  (* sqrt2/2 (+ (cos angle) (sin angle))))
         (right (* sqrt2/2 (- (cos angle) (sin angle)))))
    (out 0 (* input left))
    (out 1 (* input right))))

(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (define tone
        (sound
          (stereo-pan (* (midi-vol)
                         (oscil :freq (midi->hz (midi-note))))
                      (/ (midi-note) 127.0))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (stop tone))))))

(<rt-stalin>
  (sound
    (in (softsynth)
        (lambda (sound-left sound-right)
          (out 0 (fx-ctrl sound-left  0.5 0.09 reverb))
          (out 1 (fx-ctrl sound-right 0.5 0.09 reverb))))))
```

At first glance, it may look okay. But the reverb will not work properly. The reason is that auto-generated variables used for coroutine-local variables are identified by their position in the source. And since the code for the reverb is written only one place in the source, but used two times from the same coroutine, both channels will use the same coroutine-local variables used by the reverb; a delay, four comb filters and four all-pass filters.

There are a few ways to work around this problem. The quickest work-around is to recode 'reverb' into a macro instead of a function. However, since neither the problem nor any solution to the problem are very obvious, plus that it is slower to use coroutine-local variables than manually allocating them (it requires extra instructions to check whether the data has been allocated[14]), it's tempting not to use coroutine-local variables at all.

Instead we introduce a new concept called CPS Sound Generators, where CPS stands for Continuation Passing Style. [Sussman and Steele, 1975]

10.1 How it works

Working with CPS Sound Generators are similar to Faust's Block Diagrams composition. [Orlarey et al., 2004] A CPS Sound Generator can also be seen as similar to a Block Diagram in Faust, and connecting the CPS Sound Generators is quite similar to Faust's Block Diagram Algebra (BDA).

CPS Sound Generators are CPS functions which are able to connect to other CPS Sound Generators in order to build up a larger function for processing samples. The advantage of building up a program this way is that we know what data is needed before starting to process samples. This means that auto-allocated variables don't have to be stored in coroutines, but can be allocated before running the *sound* block.

For instance, the following code is written in generator-style and plays a 400Hz sine sound to the sound card:

```
(let ((Generator (let ((osc (make-oscillator :freq 400)))
                   (lambda (kont)
                     (kont (oscil osc))))))
  (sound
    (Generator (lambda (sample)
                 (out sample)))))
```

The variable *kont* in the function *Generator* is the continuation, and it is always the last argument in a CPS Sound Generator. A continuation is a function containing the rest of the program. In other words, a continuation function

[14] It is possible to optimize away these checks, but doing so either requires restricting the liberty of the programmer, some kind of JIT-compilation, or doing a whole-program analysis.

will never return. The main reason for programming this way is for generators to easily return more than one sample, i.e have more than one output.[15]

Programming directly this way, as shown above, is not convenient, so in order to make programming simpler, some additional syntax have been added. The two most common operators are Seq and Par, which behave similar to the '**:**' and ',' infix operators in Faust.[16]

Seq creates a new generator by connecting generators in sequence. In case an argument is not a generator, a generator will automatically be created from the argument.

For instance, (Seq (+ 2)) is the same as writing

```
(let ((generator0 (lambda (arg1 kont0)
                    (kont0 (+ 2 arg1)))))
  (lambda (input0 kont1)
    (generator0 input0 kont1)))
```

and (Seq (+ (random 1.0)) (+ 1)) is the same as writing

```
(let ((generator0 (let ((arg0 (random 1.0)))
                    (lambda (arg1 kont0)
                      (kont0 (+ arg0 arg1)))))
      (generator1 (lambda (arg1 kont1)
                    (kont1 (+ 1 arg1)))))
  (lambda (input kont2)
    (generator0 input (lambda (result0)
                        (generator1 result0 kont2)))))

;; Evaluating ((Seq (+ 2) (+ 1)) 3 display)
;; will print 6!
```

Par creates a new generator by connecting generators in parallel. Similar to *Seq*, if an argument is not a generator, a generator using the argument will be created automatically.

For instance, (Par (+ (random 1.0)) (+ 1)) is the same as writing:

```
(let ((generator0 (let ((arg0 (random 1.0)))
                    (lambda (arg1 kont0)
                      (kont0 (+ arg0 arg1)))))
      (generator1 (lambda (arg1 kont1)
                    (kont1 (+ 1 arg1)))))
  (lambda (input2 input3 kont1)
    (generator0 input2
      (lambda (result0)
        (generator1 input3
          (lambda (result1)
            (kont1 result0 result1)))))))

;; Evaluating ((Par (+ 2)(+ 1)) 3 4 +) will return 10!
```

(gen-sound :*options* **generator)** is the same as writing

```
(let ((gen generator))
  (sound :options
    (gen (lambda (result0)
           (out 0 result0)))))
```

...when the generator has one output. If the generator has two outputs, it will look like this:

```
(let ((gen generator))
  (sound :options
    (gen (lambda (result0 result1)
           (out 0 result0)
           (out 1 result1)))))
```

...and so forth.

The Snd-RT preprocessor knows if a variable or expression is a CPS Sound Generator by looking at whether the first character is capitol. For instance, (Seq (Process 2)) is equal to (Process 2), while (Seq (process 2)) is equal to (lambda (input kont) (kont (process 2 input))), regardless of how 'Process' and 'process' are defined.

10.2 Handling auto-allocated variables

oscil and the other CLM generators are macros, and the expanded code for *(oscil :freq 440)* looks like this:

```
(oscil_ (autovar (make_oscil_ 440 0.0)) 0 0)
```

Normally, *autovar* variables are translated into coroutine-local variables in a separate step performed after macro expansion. However, when an auto-allocated variable is an argument for a generator, the *autovar* surrounding is removed. And, similar to other arguments which are normal function calls, the initialization code is placed before the generator function. For example, (Seq (oscil :freq 440)) is expanded into:[17]

```
(let ((generator0 (let ((var0 (make_oscil_ 440 0.0)))
                    (lambda (kont)
                      (kont (oscil_ var0 0 0))))))
  (lambda (kont)
    (generator0 kont)))
```

[15] Also note that by inlining functions, the Stalin scheme compiler is able to optimize away the extra syntax necessary for the CPS style.

[16] Several other special operators are available as well, but this paper is too short to document all of them.

[17] Since the Snd-RT preprocessor doesn't know the number of arguments for normal functions such as *oscil_*, this expansion requires the preprocessor to know that this particular *Seq* block has 0 inputs. The preprocessor should usually get this information from the code calling *Seq*, but it can also be defined explicitly, for example like this: (Seq 0 Cut (oscil :freq 440)).

10.3 The Soft Synth using CPS Sound Generators

```
(define-stalin (Reverb)
  (Seq (all-pass :feedback -0.7 :feedforward 0.7)
       (all-pass :feedback -0.7 :feedforward 0.7)
       (all-pass :feedback -0.7 :feedforward 0.7)
       (all-pass :feedback -0.7 :feedforward 0.7)
       (Sum (comb :scaler 0.742 :size 9601)
            (comb :scaler 0.733 :size 10007)
            (comb :scaler 0.715 :size 10799)
            (comb :scaler 0.697 :size 11597))
       (delay :size (* .013 (mus-srate)))))

(define-stalin (Stereo-pan c)
  (Split Identity
         (* left)
         (* right)
    :where left  (* sqrt2/2 (+ (cos angle) (sin angle)))
    :where right (* sqrt2/2 (- (cos angle) (sin angle)))
    :where angle (- pi/4 (* c pi/2))
    :where sqrt2/2 (/ (sqrt 2) 2)))

(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (define tone
        (gen-sound
          (Seq (oscil :freq (midi->hz (midi-note)))
               (* (midi-vol))
               (Stereo-pan (/ (midi-note) 127)))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (stop tone))))))

(define-stalin (Fx-ctrl clean wet Fx)
  (Sum (* clean)
       (Seq Fx
            (* wet))))

(<rt-stalin>
  (gen-sound
    (Seq (In (softsynth))
         (Par (Fx-ctrl 0.5 0.09 (Reverb))
              (Fx-ctrl 0.5 0.09 (Reverb))))))
```

11 Adding an ADSR Envelope

And finally, to make the MIDI soft synth sound decent, we need to avoid clicks caused by suddenly starting and stopping sounds. To do this, we use a built-in ADSR envelope generator (entirely written in Scheme) for ramping up and down the volume. Only the function *softsynth* needs to be changed:

```
(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (gen-sound :while (-> adsr is-running)
        (Seq (Prod (oscil :freq (midi->hz (midi-note)))
                   (midi-vol)
                   (-> adsr next))
             (Stereo-pan (/ (midi-note) 127))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (-> adsr stop)))
      :where adsr (make-adsr :a 20:-ms
                             :d 30:-ms
                             :s 0.2
                             :r 70:-ms))))
```

12 Conclusion

This paper has shown a few techniques for doing low-latency sample-by-sample audio programming.

13 Acknowledgments

Thanks to the anonymous reviewers and Anders Vinjar for comments and suggestions.

References

William Clinger and Jonathan Rees. 1991. Revised Report (4) On The Algorithmic Language Scheme.

Melvin E. Conway. 1963. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408.

O.-J. Dahl and K. Nygaard. 1966. SIMULA: an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678.

Kjetil Matheussen. 2008. Realtime Music Programming Using Snd-RT. In *Proceedings of the International Computer Music Conference.*

Kjetil Matheussen. 2009. Conservative Gargage Collectors for Realtime Audio Processing. In *Proceedings of the International Computer Music Conference*, pages 359–366 (online erratum).

James McCartney. 2002. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(2):61–68.

Y. Orlarey, D. Fober, and S. Letz. 2004. Syntactical and semantical aspects of faust, *soft computing.*

Miller Puckette. 2002. Max at Seventeen. *Computer Music Journal*, 26(4):31–43.

W. Schottstaedt. 1994. Machine Tongues XVII: CLM: Music V Meets Common Lisp. *Computer Music Journal*, 18(2):30–37.

Jr. Steele, Guy Lewis and Gerald Jay Sussman. 1978. The Revised Report on SCHEME: A Dialect of LISP. *Technical Report 452, MIT.*

Gerald Jay Sussman and Jr. Steele, Guy Lewis. 1975. Scheme: An interpreter for extended lambda calculus. In *Memo 349, MIT AI Lab.*

Ge Wang and Perry Cook. 2003. ChucK: a Concurrent and On-the-fly Audio Programming Language, *Proceedings of the ICMC.*