Q: A Functional Programming Language for Multimedia Applications

Albert GRÄF
Department of Music-Informatics
Johannes Gutenberg University
55099 Mainz
Germany
ag@muwiinfa.geschichte.uni-mainz.de

Abstract

Q is a functional programming language based on term rewriting. Programs are collections of equations which are used to evaluate expressions in a symbolic fashion. Q comes with a set of extension modules which make it a viable tool for scientific programming, computer music, multimedia, and other advanced applications. In particular, Q provides special support for multimedia applications using PortAudio, libsndfile, libsamplerate, FFTW, MidiShare and OSC (including a SuperCollider interface). The paper gives a brief introduction to the Q language and its multimedia library, with a focus on the facilities for MIDI programming and the SuperCollider interface.

Keywords

Computer music, functional programming, multimedia programming, Q programming language, SuperCollider

1 Introduction

The pseudo acronym "Q" stands for "equational programming language". Q has its roots in term rewriting, a formal calculus for the symbolic evaluation of expressions coming from universal algebra and symbolic algebra systems (Dershowitz and Jouannaud, 1990). It builds on Michael O'Donnell's ground-breaking work on equational programming in the 1980s (O'Donnell, 1985) and the author's own research on efficient term pattern matching and rewriting techniques (Gräf, 1991).

In a sense, Q is for modern functional programming languages what BASIC is for imperative ones: It is a fairly simple language, thus easy to learn and use, yet powerful enough to tackle most common programming tasks; it is an interpreted (rather than compiled) language, offering adequate (though not C-like) execution speed; and it comes with a convenient interactive environment including a symbolic debugger, which lets you play with the parts of your program to explore different solution approaches and to test things out.

Despite its simplicity, Q should not be mistaken for a "toy language"; in fact, it comes with a fairly comprehensive collection of libraries which in many areas surpasses what is currently available for its bigger cousins like ML and Haskell. Moreover, Q's SWIG interface makes it easy to interface to additional C and C++ libraries if needed.

The Q programming environment is GPL'ed software which has been ported to a large variety of different platforms, including Linux (which has been the main development platform since 1993), FreeBSD, Mac OS X, BeOS, Solaris and Windows. Q also has a cross-platform multimedia library which currently comprises MIDI (via Grame's MidiShare), audio (providing interfaces to PortAudio v19, libsndfile, libsamplerate and FFTW) and software synthesis (via OSC, the "Open Sound Control" protocol developed by CNMAT, with special support for James McCartney's SuperCollider software). Additional modules for 3D graphics (OpenGL) and video (libxine) are currently under development.

In the following we give a brief overview of the language and the standard library, after which we focus on Q's multimedia facilities. More information about Q can be found on the Q homepage at `http://q-lang.sourceforge.net`.

2 The language

At its core, Q is a fairly simple language which is based entirely on the notions of *reductions* and *normal forms* pertaining to the term rewriting calculus. A Q program or *script* is simply a collection of equations which establish algebraic identities. The equations are interpreted as rewriting rules in order to reduce expressions to normal forms. The syntax of the language was inspired by the first edition of Bird and Wadler's influential book on functional programming (Bird and Wadler, 1988) and thus is similar to other modern functional languages such as Miranda and Haskell. For instance, here is how you define a function `sqr` which squares its argument by multiplying it with itself:

```
sqr X = X*X;
```

When this equation is applied to evaluate an expression like `sqr 2`, the interpreter performs the reduction `sqr 2 => 2*2`. It then goes on to apply other equations (as well as a number of built-in rules implementing the primitive operations such as arithmetic) until a normal form is reached (an expression is said to be in normal form if no more equations or built-in rules can be applied to it). In our example, the interpreter will invoke the rule which handles integer multiplication: `2*2 => 4`. The resulting expression `4` is in normal form and denotes the "value" of the original expression `sqr 2`.

Note that, as in Prolog, capitalized identifiers are used to indicate the variables in an equation, which are bound to the actual values when an equation is applied. We also remark that function application is denoted simply by juxtaposition. Parentheses are used to group expressions and to indicate "tuple" values, but are *not* part of the function application syntax. This "curried" form of writing function applications is ubiquitous in modern functional languages. In addition, the Q language also supports the usual infix notation for operators such as `+` and `*`. As in other modern functional languages, these are just "syntactic sugar" for function applications; i.e., `X*X` is just a convenient shorthand for the function application `(*) X X`. Operator "sections" are also supported; e.g., `(+1)` denotes the function which adds 1 to its argument, `(1/)` the reciprocal function.

Equations may also include a condition part, as in the following (recursive) definition of the factorial function:

```
fact N = N*fact (N-1) if N>0;
       = 1 otherwise;
```

Another useful extension to standard term rewriting are the "where clauses" which allow you to bind local variables in an equation. For instance, the following equation defines a function for solving quadratic equations $x^2 + px + q = 0$. It first checks whether the discriminant $D = p^2/4 - q$ is nonnegative before it uses this value to compute the two real solutions of the equation.

```
solve P Q = (-P/2+sqrt D,-P/2-sqrt D)
  if D >= 0 where D = P^2/4-Q;
```

You can also define global variables using a `def` statement. This is useful if a value is used repeatedly in different equations and you don't want to recalculate it each time it is needed.

```
def PI = 4*atan 1;
```

Functions on structured arguments are defined by "pattern matching". E.g., the quicksort function can be implemented in Q with the following two equations. (Note that lists are written in Prolog-like syntax, thus `[]` denotes the empty list and `[X|Xs]` a list starting with the head element `X` and continuing with the list of remaining elements `Xs`. Furthermore, the `++` operator denotes list concatenation.)

```
qsort []     = [];
qsort [X|Xs] = qsort (filter (<X) Xs) ++
  [X] ++ qsort (filter (>=X) Xs);
```

Higher-order functions which take other functions as arguments can also be programmed in a straightforward way. For instance, the `filter` function used above is defined in the standard library as follows. In this case, the function argument `P` is a predicate expected to return the value `true` if an element should be included in the result list, `false` otherwise.

```
filter P []     = [];
filter P [X|Xs] = [X|filter P Xs] if P X;
                = filter P Xs otherwise;
```

In contrast to "pure" functional languages such as Haskell, Q takes the pragmatic route in that it also provides imperative programming features such as I/O operations and mutable data cells ("references"), similar to the corresponding facilities in the ML programming language. While one may argue about the use of such "impure" operations with side-effects in a functional programming language, they certainly make life easier when dealing, e.g., with complex I/O situations and thread synchronization. The `||` operator can be employed to execute such actions in sequence. For instance, using the built-in `reads` ("read string") and `writes` ("write string") functions, a simple prompt/input interaction would be written as follows:

```
prompt = writes "Input: " || reads;
```

References work like pointers to expressions. Three operations are provided: `ref` which creates a reference from its initial value, `put` which changes the referenced value, and `get` which returns the current value. With these facilities you can realize mutable data structures and maintain hidden state in a function. For instance, the following function `counter` returns the next integer at each invokation, starting at zero:

```
def COUNTER = ref 0;
counter = put COUNTER (N+1) || N
            where N = get COUNTER;
```

Despite its conceptual simplicity, Q is a full-featured functional programming language which allows you to write your programs in a concise and abstract mathematical style. Since it is an interpreted language, programs written in Q are definitely not as fast as their counterparts in C, but they are much easier to write, and the execution speed is certainly good enough for practical purposes (more or less comparable to interpreted Lisp and Haskell).

Just like other languages of its kind, Q has automatic memory management, facilities for raising and handling exceptions, constructs for defining new, application-specific data types, and means for partitioning larger scripts into separate modules. Functions and data structures using "lazy" evaluation can be dealt with in a direct manner. Q also uses dynamic typing, featuring a Smalltalk-like object-oriented type system with single inheritance. This has become a rare feature in contemporary functional languages which usually employ a static Hindley/Milner type system to provide more safety at the expense of restricting polymorphism. Q gives you back the flexibility of good old Lisp-style ad-hoc polymorphism and even allows you to extend the definition of existing operations (including built-in functions and operators) to your own data types.

3 The library

No modern programming or scripting language is complete without an extensive software library covering the more mundane programming tasks. In the bad old times of proprietary software, crafting such a library has always been a major undertaking, since all these components had to be created from scratch. Fortunately, nowadays there is a large variety of open source software providing more or less standardized solutions for all these areas, so that "reinventing the wheel" can mostly be avoided.

This is also the approach taken with the Q programming system, which acts as a kind of "nexus" connecting various open source technologies. To these ends, Q has an elaborate C/C++ interface including support for the SWIG wrapper generator (www.swig.org), which makes it easy to interface to existing C/C++ libraries. This enabled us to provide a fairly complete set of cross-platform extension modules which, while not as comprehensive as the facilities of other (much larger) language projects such as Perl and Python, make it possible to tackle most practical programming tasks with ease. This part of the Q library also goes well beyond what is offered with most other modern functional languages, especially in the multimedia department.

The core of the Q programming system includes a standard library, written mostly in Q itself, which implements a lot of useful Q types and functions, such as complex numbers, generic list processing functions (including list comprehensions), streams (a variant of lists featuring lazy evaluation which makes it possible to represent infinite data structures), container data structures (sets, dictionaries, hash tables, etc.), the lambda calculus, and a PostScript interface. Also included in the core is a POSIX system interface which provides, e.g., lowlevel I/O, process and thread management, sockets, filename globbing and regular expression matching.

In the GUI department, Q relies on Tcl/Tk (www.tcl.tk). While Tk is not the prettiest toolkit, its widgets are adequate for most purposes, it can be programmed quite easily, and, most importantly, it has been ported to a large variety of platforms. Using SWIG, it is also possible to embed GTK- and Qt-based interfaces, if a prettier appearance and/or more sophisticated GUI widgets are needed. (Complete bindings for these "deluxe" toolkits are on the TODO list, but have not been implemented yet.)

For basic 2D graphics, Q uses GGI, the "General Graphics Interface" (www.ggi-project.org), which has been augmented with a FreeType interface to add support for advanced font handling (www.freetype.org). Moreover, a module with bindings for the ImageMagick library (www.imagemagick.org) allows you to work with virtually all popular image file formats and provides an abundance of basic and advanced image manipulation functions.

To facilitate scientific programming, Q has interfaces to Octave, John W. Eaton's well-known MATLAB-like numerical computation software (www.octave.org), and to IBM's "Open Data Explorer", a comprehensive software for doing data visualization (www.opendx.org).

Web programming is another common occupation of the contemporary developer. In this realm, Q provides an Apache module and an XML/XSLT interface (xmlsoft.org) which allow you to create dynamic web content with ease. Moreover, an interface to the Curl library enables you to perform automated downloads and spidering tasks (curl.haxx.se). If you need database access, an ODBC module (www.iodbc.org, www.unixodbc.org) can be used to query and modify RDBMSs such as MySQL and PostgreSQL.

4 MIDI programming

Q's MIDI interface, embodied by the `midi` module, is based on Grame's MidiShare library (Fober et al., 1999). We have chosen MidiShare because it has been around since the time of the good old Atari and thus is quite mature, it has been ported to a number of different platforms (including Linux, Mac OS X and Windows), it takes a unique "client graph" approach which provides flexible dynamic routing of MIDI data between different applications, and, last but not least, it offers comprehensive support for handling standard MIDI files.

While MidiShare already abstracts from all messy hardware details, Q's `midi` module even goes one step further in that it also represents MIDI messages not as cryptic byte sequences, but as a high-level "algebraic" data type which can be manipulated easily. For instance, note on messages are denoted using data terms of the form `note_on CHANNEL NOTE VELOCITY`. The functions `midi_get` and `midi_send` are used to read and write MIDI messages, respectively. For example, Fig. 1 shows a little script for transposing MIDI messages in realtime.

The `midi` module provides all necessary data types and functions to process MIDI data in any desired way. It also gives access to MidiShare's functions to handle standard MIDI files. In order to work with entire MIDI sequences, MIDI messages can be stored in Q's built-in list data structure, where they can be manipulated using Q's extensive set of generic list operations. Q's POSIX multithreading support allows you to run multiple MIDI processing algorithms concurrently and with realtime scheduling priorities, which is useful or even essential for many types of MIDI applications.

These features make it possible to implement fairly sophisticated MIDI applications with moderate effort. To demonstrate this, we have employed the `midi` module to program various algorithmic composition tools and step sequencers, as well as a specialized graphical notation and sequencing software for percussion pieces. The latter program, called "clktrk", was used by the composer Benedict Mason for one of his recent projects (felt | ebb | thus | brink | here | array | telling, performed by the Ensemble Modern with the Junge Deutsche Philharmonie at the Donaueschingen Music Days 2004 and the Maerzmusik Berlin 2005).

Other generally useful tools with KDE/Qt-based GUIs can be found on the Q homepage. For instance, Fig. 2 shows the QMidiCC program, a MidiShare patchbay which can be configured to take care of your MidiShare drivers and to automatically connect new clients as soon as they show up in the MidiShare client list. QMidiCC can also be connected to other MidiShare applications to print their MIDI output and to send them MIDI start and stop messages.

5 Audio and software synthesis

The audio interface consists of three modules which together provide the necessary facilities for processing digital audio in Q. The `audio` module is based on PortAudio (v19), a cross-platform audio library which provides the necessary operations to work with the audio interfaces of the host operating system (www.portaudio.com). Under Linux this module gives access to both ALSA (www.alsa-project.org) and Jack (jackit.sf.net). The `sndfile` module uses Erik de Castro Lopo's libsndfile library which allows you to read and write sound files in a variety of formats (www.mega-nerd.com/libsndfile). The `wave` module provides basic operations to create, inspect and manipulate wave data represented as "byte strings" (a lowlevel data structure provided by Q's system interface which is used to store raw binary data). It also includes operations for sample rate conversion (via libsam-

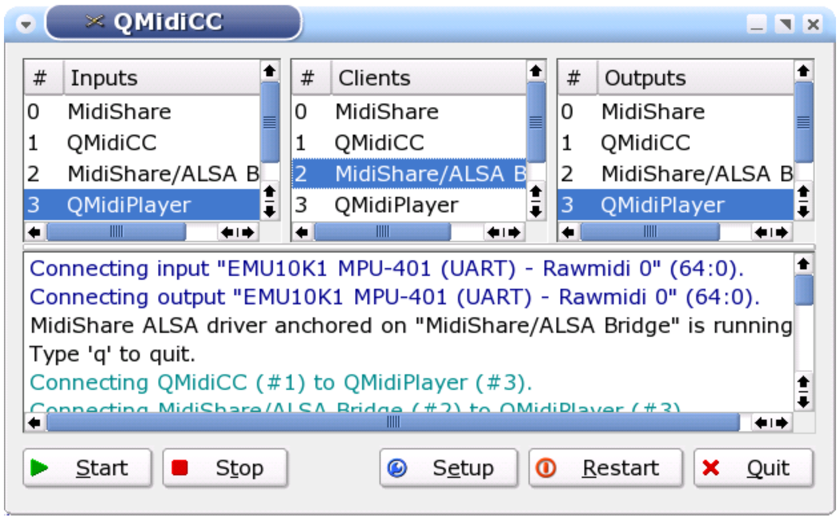
```
import midi;

/* register a MidiShare client and establish I/O connections */
def REF = midi_open "Transpose",
  IO = midi_client_ref "MidiShare/ALSA Bridge",
  _ = midi_connect IO REF || midi_connect REF IO;

/* transpose note on and off messages, leave other messages unchanged */
transp K (note_on CH N V)
                = note_on CH (N+K) V;
transp K (note_off CH N V)
                = note_off CH (N+K) V;
transp K MSG    = MSG otherwise;

/* the following loop repeatedly reads a message, transposes it and
   immediately outputs the transformed message */
transp_loop K   = midi_send REF 0 (transp K MSG) || transp_loop K
                    where (_,_,_,MSG) = midi_get REF;
```

Figure 1: Sample MIDI script.



#	Inputs
0	MidiShare
1	QMidiCC
2	MidiShare/ALSA B
3	QMidiPlayer

#	Clients
0	MidiShare
1	QMidiCC
2	MidiShare/ALSA B
3	QMidiPlayer

#	Outputs
0	MidiShare
1	QMidiCC
2	MidiShare/ALSA B
3	QMidiPlayer

```
Connecting input "EMU10K1 MPU-401 (UART) - Rawmidi 0" (64:0).
Connecting output "EMU10K1 MPU-401 (UART) - Rawmidi 0" (64:0).
MidiShare ALSA driver anchored on "MidiShare/ALSA Bridge" is running
Type 'q' to quit.
Connecting QMidiCC (#1) to QMidiPlayer (#3).
Connecting MidiShare/ALSA Bridge (#2) to QMidiPlayer (#3).
```

Figure 2: QMidiCC program.

plerate, www.mega-nerd.com/SRC) and fast Fourier transforms (via FFTW, www.fftw.org), as well as a function for drawing waveforms in a GGI visual.

Q's audio interface provides adequate support for simple audio applications such as audio playback and recording, and provides a framework for programming more advanced audio analysis and synthesis techniques. For these you'll either have to provide your own C or C++ modules to do the necessary processing of wave data, or employ Q's `osc` module which allows you to drive OSC-aware software synthesizers (www.cnmat.berkeley.edu/OpenSoundControl). We also offer an `sc` module which provides special support for James McCartney's SuperCollider (McCartney, 2002).

The `osc` module defines an algebraic data type as a high-level representation of OSC packets which can be manipulated easily. All standard OSC features are supported, including OSC bundles. The module also implements a simple UDP transport layer for sending and receiving OSC packets. In addition, the `sc` module offers some convenience functions to control SuperCollider's `sclang` and `scsynth` applications.

Fig. 3 shows a little Q script implementing some common OSC messages which can be used to control the SuperCollider sound server. Using these facilities in combination with the `midi` module, it is a relatively straightforward matter

```
import osc, sc;

// load a synthdef into the server
d_load NAME     = sc_send (osc_message CMD_D_LOAD NAME);

// create a new synth node (add at the end of the main group)
s_new NAME ID ARGS
                = sc_send (osc_message CMD_S_NEW (NAME,ID,1,0|ARGS));

// free a synth node
n_free ID       = sc_send (osc_message CMD_N_FREE ID);

// set control parameters
n_set ID ARGS   = sc_send (osc_message CMD_N_SET (ID|ARGS));
```

Figure 3: Sample OSC script.

```
/* get MIDI input */

midiin          = (TIME,MSG) where (_,_,TIME,MSG) = midi_get REF;

/* current pitch wheel value and tuning table */

def WHEEL =  ref 0.0, TT = map (ref.(*100.0)) [0..127];

/* calculate the frequency for a given MIDI note number N */

freq N          = 440*2^((get (TT!N)-6900)/1200+get WHEEL/6);

/* The MIDI loop: Assign voices from a queue Q of preallocated SC synth units
   in a round-robin fashion. Keep track of the currently assigned voices in a
   dictionary P. The third parameter is the MIDI event to be processed next. */

/* note offs: set the gate of the synth to 0 and put it at the end of the queue */

loop P Q (_,note_on _ N 0)
                = n_set I ("gate",0) || loop P Q midiin
                    where (I,_) = P!N, P = delete P N, Q = append Q I;
                = loop P Q midiin otherwise;

loop P Q (T,note_off CH N _)
                = loop P Q (T,note_on CH N 0);

/* note ons: turn note off if already sounding, then get a new voice from the
   queue and set its gate to 1 */

loop P Q (T,note_on CH N V)
                = n_set I ("gate",0) || loop P Q (T,note_on CH N V)
                    where (I,_) = P!N, P = delete P N, Q = append Q I;
                = n_set I ("freq",FREQ,"gain",V/127,"gate",1) ||
                  loop P Q midiin
                    where [I|Q] = Q, FREQ = freq N,
                      P = insert P (N,(I,FREQ));
```

Figure 4: Excerpt from a MIDI to OSC processing loop.

to implement software synthesizers which can be played in realtime via MIDI. All actual audio processing takes place in the synthesis engine, the Q script only acts as a kind of "MIDI to OSC" translator. For instance, Fig. 4 shows an excerpt from a typical MIDI processing loop.

An example of such a program, called "QSCSynth", can be found on the Q homepage (cf. Fig. 5). QSCSynth is a (KDE/Qt based) GUI frontend for the `sclang` and `scsynth` programs which allows you to play and control SuperCollider synthesizers defined in an SCLang source file. It implements a monotimbral software synth which can be played via MIDI input and other MidiShare applications. Moreover, with MidiShare's ALSA driver, QSCSynth can easily be wired up with ALSA-based sequencer applications like Rosegarden, employing it as a fully programmable realtime software synthesizer. The audio stream generated by SuperCollider can be watched in an integrated waveform/FFT display, and can also be recorded in an audio file. QSCSynth can also be configured to map arbitrary MIDI controller messages to corresponding OSC messages which change the control parameters of the synthesizer and effect units defined in the SCLang source file. Moreover, QSCSynth also provides its own control surface (constructed automatically from the parameter descriptions found in the binary synth definition files) which lets you control synth and effect units from the GUI as well.

6 The future

While Q's multimedia library already provides a fairly complete framework for programming multimedia and computer music applications on Linux, there still remain a few things to be done:

- Finish the OpenGL and video support.
- Provide modules for some Linux-specific libraries such as Jack, LADSPA and DSSI.
- Provide high-level interfaces for computer music applications such as algorithmic composition. There are a few lessons to be learned from existing environments here, such as Rick Taube's Common Music (Taube, 2005), Grame's Elody (Letz et al., 2000) and Paul Hudak's Haskore (Hudak, 2000b).
- Add graphical components for displaying and editing music (piano rolls, notation, etc.). For this we should try to reuse parts from existing open source software, such as Lilypond (lilypond.org), the GUIDO library (www.salieri.org/guido) and Rosegarden (www.rosegardenmusic.com).
- Add a "patcher"-like visual programming interface, such as the one found in IRCAM's OpenMusic.

7 Conclusion

Functional programming has always played an important role in computer music, because it eases the symbolic manipulation of complex structured data. However, to our knowledge no other "modern-style" functional language currently provides the necessary interfaces to implement sophisticated, realtime-capable multimedia applications. We therefore believe that Q is an interesting tool for those who would like to explore MIDI programming, sound synthesis and other multimedia applications, in the context of a high-level, general-purpose, non-imperative programming language.

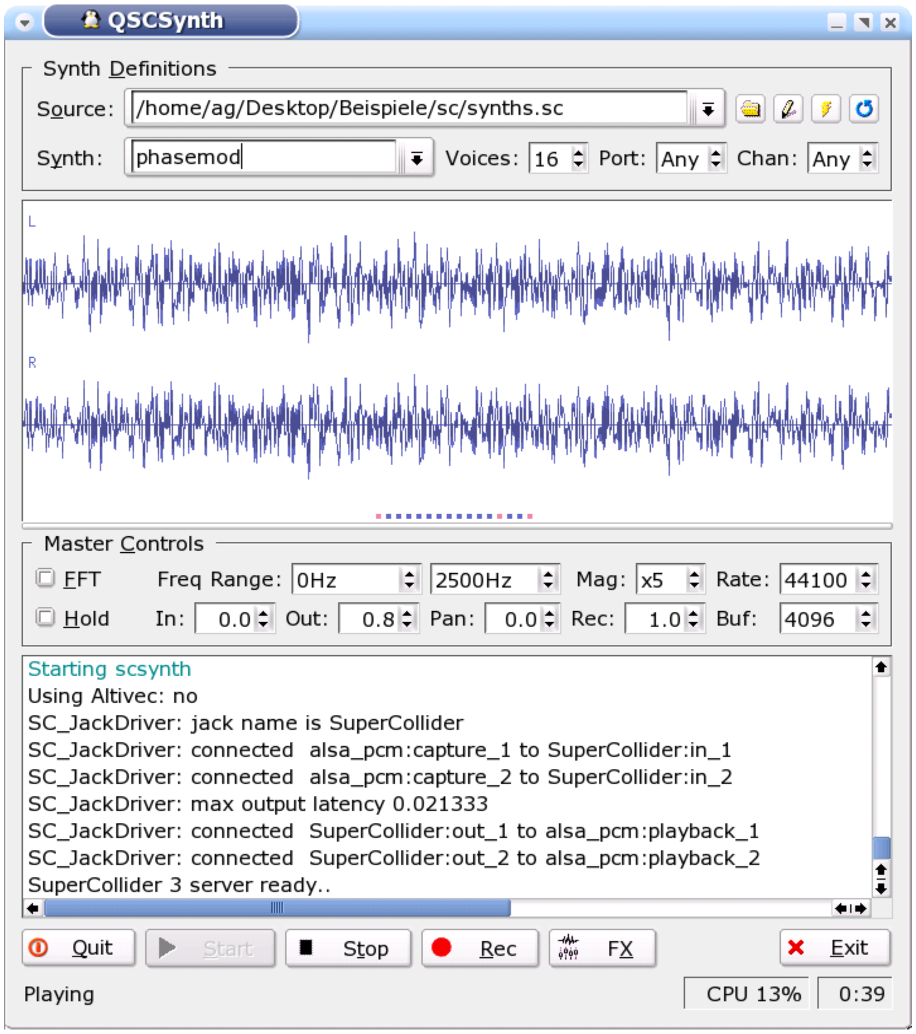
While the Q core system is considered stable, the language and its libraries continue to evolve, and it is our goal to turn Q into a viable tool for rapid application development in many different areas. We think that multimedia is an attractive playground for functional programming, because modern FP languages allow many problems in this realm to be solved in new and interesting ways; see in particular Paul Hudak's book on multimedia programming with Haskell (Hudak, 2000a) for more examples. As the multithreading and realtime capabilities of mainstream functional languages mature, it might also be an interesting option to port some of Q's libraries to other environments such as the Glasgow Haskell compiler which offer better execution speed than an interpreted language, for the benefit of both the functional programming community and multimedia application developers.

References

Richard Bird and Philip Wadler. 1988. *Introduction to Functional Programming.* Prentice Hall, New York.

Nachum Dershowitz and Jean-Pierre Jouannaud. 1990. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier.

Dominique Fober, Stephane Letz, and Yann Orlarey. 1999. MidiShare joins the open sources



```
Starting scsynth
Using Altivec: no
SC_JackDriver: jack name is SuperCollider
SC_JackDriver: connected  alsa_pcm:capture_1 to SuperCollider:in_1
SC_JackDriver: connected  alsa_pcm:capture_2 to SuperCollider:in_2
SC_JackDriver: max output latency 0.021333
SC_JackDriver: connected  SuperCollider:out_1 to alsa_pcm:playback_1
SC_JackDriver: connected  SuperCollider:out_2 to alsa_pcm:playback_2
SuperCollider 3 server ready..
```

Figure 5: QSCSynth program.

softwares. In *Proceedings of the International Computer Music Conference*, pages 311–313, International Computer Music Association. See also `http://www.grame.fr/MidiShare`.

Albert Gräf. 1991. Left-to-right tree pattern matching. In Ronald V. Book, editor, *Rewriting Techniques and Applications*, LNCS 488, pages 323–334. Springer.

Paul Hudak. 2000a. *The Haskell School of Expression: Learning Functional Programming Through Multimedia*. Cambridge University Press.

Paul Hudak. 2000b. Haskore Music Tutorial. Yale University, Department of Computer Science. See `http://www.haskell.org/haskore`.

Stephane Letz, Dominique Fober, and Yann Orlarey. 2000. Realtime composition in Elody. In *Proceedings of the International Computer Music Conference*, International Computer Music Association. See also `http://www.grame.fr/Elody`.

James McCartney. 2002. Rethinking the computer music language: SuperCollider. *Computer Music Journal*, 26(4):61–68. See also `http://supercollider.sourceforge.net`.

Michael O'Donnell. 1985. *Equational Logic as a Programming Language*. Series in the Foundations of Computing. MIT Press, Cambridge, Mass.

Heinrich K. Taube. 2005. *Notes from the Metalevel: Introduction to Algorithmic Music Composition*. Swets & Zeitlinger. To appear. `http://pinhead.music.uiuc.edu/~hkt/nm`.