Vuza: a functional language for creative applications

Carmine-Emanuele Cella
Casa de Velázquez - Ircam
carmine-emanuele.cella@[casadevelazquez.org, ircam.fr]

ABSTRACT

This short paper will present Vuza, a new functional language for computer music and creative coding. The keypoint of the language is to bring the expressivity and the flexiblity of functional programming to digital art and computer music and make possible to embed such power in host applications. Vuza is a general purpose language with specific extensions for sound analysis and synthesis in real-time, computer assisted composition and GUI (graphical user interface) creation.

1. INTRODUCTION

During the history of computer music and, more generally, of creative coding [1] many excellent languages and environments have been developed and have become the center of small or large communities that used their power to create and innovate. Any of these languages adopts a particular point of view on the domain it processes, thus adhering to a specific mixture among the programming paradigms. While it is beyond the scope of this paper to give an complete discussion of the topic (see [8] for more information) we would like to name, in any particular order, a few of them [2]:

- *Csound*: originally written at MIT by Barry Vercoe, from his Music 11 language, is a *must* for audio programming. It is a structured language that conceptually separates the creation of algorithms to their temporal dislocation.
- *SuperCollider*: is an environment and language created by James McCarthy for real-time audio processing and automated composition. It uses a mixture of functional and object-oriented programming and it has proved to be an excellent platform for live coding [3].
- *Nyquist*: is a sound synthesis and composition language with a double programming paradigm: a Lisp-like syntax as well as an imperative language syntax (SAL); it has been created by Roger Dannenberg at Carnegie Mellon University.
- *Common Lisp Music (CLM)*: is a huge music synthesis and signal processing environment, belonging to the Music V family of languages, created by Bill Schottstaedt at CCRMA.
- *Processing*: is a programming language and environment (IDE) for new media and visual art; it was initiated in 2001 by Casey Reas and Benjamin Fry at the MIT Media Lab.
- *Max/MSP - PureData*: are two outstanding examples of visual programming languages. Originally created by Miller Puckette for interactive computer music, they are the *de facto* standard today for real-time performances.

Each one of these environments gathered around it artists, researchers and enthusiasts thus becoming a favorite medium for innovation and discovery. Many authors, in literature, have written papers on particular aspects of the field discussing practical and aesthetic issues and elaborating on the relation between technical development and poetic inspiration (see, among others, [6]).

The project documented in this paper aims at giving some contributions to such a rich and proficient field.

The following sections will introduce and discuss *Vuza* [4], a new general purpose functional interpreted language specifically designed for computer music and creative coding. The source code, related examples and some documentation on the project can be found on the website `http://www.vuza.org`. After a short presentation of the basics of the language, some example applications will be shown.

2. LANGUAGE DESIGN

Designing a new language is not an easy task: it involves different considerations and specific needs, from implementative to theoretical. When we started designing Vuza, however, we knew that we wanted it to have a *functional* nature and other important features that will be discussed below.

[1] With this expression we mean the use of computer languages for artistic applications such as installations, interactive audio and video performances, computer vision projects, and so on.

[2] The list provided here is not meant to be exhaustive but only to give an idea on the complexity and on the size of the field.

[3] Live coding is a practice centered upon the use of improvised programming for computer music, algorithmic composition and other creative performances.

[4] The language is named after the rumanian mathematician Dan Tudor Vuza, for his important contributions in the field of mathematical theory of music. See for example `http://imar.ro/organization/people/CVs/Vuza_Dan_CV.pdf`.

Copyright: ©2014 Carmine-Emanuele Cella et al. This is an open-access article distributed under the terms of the Creative Commons Attribution 3.0 Unported License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

2.1 Functional paradigm

Functional programming is a special paradigm that treats computation as the evaluation of mathematical functions. It has its roots in the theoretical research done between the two World Wars by Alonzo Church [1] and in the work done by John McCarthy at MIT during the fifties [2]. The most famous example of such programming paradigm is the *Lisp*-family of languages and its simplified and minimalistic dialect *Scheme* [4]; both languages are standardized and the latter is described by the well known *RxRS* specifications [5] .

While Lisp has been around since a long time (its first more complete implementation dates back to the sixties [3]) it had from the very beginning incredibly expressive features that are only partially matched by more modern languages [5]:

- *function type*: functions are, in Lisp, first-class objects and can be stored in variables, passed by and have a literal representation;
- *recursion*: while well defined mathematically, recursion was not supported before Lisp and only iteration was available;
- *garbage-collection*: in Lisp, the programmer does not need to think about memory management and can focus more on high-level problems;
- *composition of expression*: there is no distinction, in the language, between statements and expressions and the programs are just trees of mathematical expressions returning values;
- *identity between code and data*: Lisp does not differentiate between programming code and user data, thus enabling exceptional reflection capabilities that exploit in the famous *macro* system, a facility to create code that generates code.

Over time, the various programming languages have gradually evolved toward the expressive power of Lisp: some features are now widespread in the mainstream, but some (such as identity between code and data) are still unique to the language invented by McCarthy just after the second World War (for more information see `http://paulgraham.com/diff.html`).

During the last thirty years, many advocates of Lisp defended the cause providing amazing examples of its power and consequently generating vivid debates on the topic.

Deciding whether or not Lisp the key to success is out of the scope of this paper; it is sure, however, that functional programming has a great expressive power that lets the user to concentrate more on the problem to be solved than on implementation details (see for example `http://paulgraham.com/avg.html`).

[5] For more information see `http://trac.sacrideo.us/wg/`.

2.2 Other important features

Other than being functional, Vuza has by design several important features:

- **small size**: the core part of the language is roughly two thousands line of code, half of which written in C++ and the other half in Vuza itself (this also makes it very portable and maintainable);
- **embdeddable**: the whole interpreter consists of a couple of header files and it is very easy to embed it in host applications written in C++ or ObjectiveC (see section 2.5);
- **extensible**: Vuza is extensible by creating *libraries* with the language itself or by using C/C++ to create *functors*, a collection of compiled functions that can be used as built-in operators from the interpreter (see section 2.5);
- **audio-oriented libraries**: Vuza provides a powerful library called *soundmath* that supports specific functionalities for sound analysis and synthesis (such as FFT transform, digital filtering, oscillators and generators, features computation and clustering, phase vocoding, granular synthesis, etc.); moreover Vuza implements a binding to the *Csound* language for real-time audio processing (more on this in section 2.4);
- **GUI creation**: it is very easy to create GUIs thanks to the binding to the *FLTK* library, that provides common controls such as buttons, sliders, windows and so on (see section 3.3).

In a nutshell, **Vuza is an interpreted programming language of the Scheme family** that implements part of the R4RS standard [6] but sports a Lisp-like macro system. It is lexically-scoped, dinamically-typed, fully tail-recursive, with functions as first-class objects and a *mark-and-sweep* garbage collection tailored for speed. As a key-point, it implements several extensions suitable for computer music and general creative coding. The major differences with the R4RS Scheme standard are summarized below:

- no support for *continuations*, a special language feature for advanced flow control;
- simplified data types: there are no *chars* and all numbers are reals (no complex, integers, etc.);
- Lisp-like macro-system based on *quasiquotations*;
- for implementative reasons, *vectors* are just a wrapper on lists, thus behaving in a slightly different manner;
- several extensions added to improve C++ and operating system integration.

The following subsections will provide examples of the language and will discuss specific libraries and bindings.

[6] See `http://people.csail.mit.edu/jaffer/r4rs_toc.html`.

2.3 Simple examples

As all the functional languages that have roots in Lisp, also Vuza uses a prefix notation for its expressions. For example, a simple arithmetic expression can be represented as follow:

```
(+ 1 2 (- 3 5))
```

While this can be confusing at the beginning, it gives the user a great flexibility; it possible, for example, to compose the same expression in several ways:

```
(if dummy (= x 1) (= x 2))
(= x (if dummy 1 2))
```

The expressions above represent a conditional test on the variable named *dummy* and a subsequent assignment to the variable *x*.

In Vuza, it is possible to define new functions using the powerful λ-notation: λ-functions are anonymous blocks of code that can be applied on a set of parameters, stored in variables or simply passed by. The code

```
(define twice (lambda (x) (+ x x)))
```

defines a new function called *twice* that, when applied to the parameter *x*, returns its double by summing it twice.

Since any function can be also passed as a parameter to another function, it is possible to create *higher-order* application easily. The code

```
(map (lambda (x) (+ x 2)) '(1 2 3 4))
```

will add 2 to any of the numbers in the given list by *mapping* an anonymous λ-function to the subsequent arguments of the expression.

In Vuza, *recursion* is generally preferred to iteration (due to its possibility to be fully tail-recursive). The following code shows the definition of a function to compute a factorial number recursively:

```
(define fac (lambda (n)
  (if (= n 0) 1 (* n (fac (- n 1))))))
```

Thanks to the powerful built-in *macro*-system, finally, it is possible to create advanced behaviors but they will be not be examined here. For a more complete set of programming examples see `http://www.vuza.org`.

2.4 The *soundmath* library

One of the main points of Vuza is the possibility to manipulate sounds and more musical entities. This is made possible by the *soundmath* library, a special extension of the language written in C++ and Vuza that provides several advanced functionalities. The following list is only a partial description of the implemented features:

- *frequency-domain processing*: FFT transform, fast block-convolution, phase-vocoding with envelope preservation;
- *time-domain processing*: many FIR and IIR filters, variable-state systems, interpolated delays, special effects;
- *synthesis*: digital oscillators, spectral synthesis, granular synthesis, physical modeling;
- *analysis*: pitch detection, low-level features (spectral centroid, spread, skewness, curtosis, MFCC, zero-crossing ...), envelope following and onsets detection;
- *spatial processing*: convolution and traditional reverberation, ambisonic encoding and decoding;
- *feature clustering and transformations*: GMM, K-means, PCA.
- *pitch-class analysis*: set transformation, interval vector, trichordal mosaics, hexachordal combinatoriality.

The soundmath library also provides the tools to perform *soundtypes* analysis and synthesis, a special framework created by the author to represent and manipulate sounds at a quasi symbolic level (see [9] and [10]).

The following section will present the two major bindings actually implemented, for both real-time audio and GUI creation. Other bindings are currently planned; see section 4 for more information.

2.5 C/C++ interoperability

An important feature of the Vuza language is the easy interoperability with C/C++. To use the whole interpreter in a host application in C++, it will be sufficient to include a special header and to create a couple of objects:

```
#include "vuza.h"
vuza::Interpreter* interpreter =
  new vuza::Interpreter ();

vuza::Sexpr* root = interpreter->run (
  *inputdata, lineno, ctx, unparsed);

vuza::display (root, cout) << std::endl;
```

In the same way, extending the language using C++ is very easy. The following examples are taken from the FLTK binding and show that to bring a C/C++ function into the language is sufficient to wrap it into a predefined signature and expose it using the special operators *import* and *make-procedure*. The C/C++ code for importing a simple function is shown below:

```
extern "C" Sexpr* fn_fl_run (
  Sexpr* params, Environment* env) {
  Fl::lock ();
  int r = Fl::run ();
  Fl::unlock ();
  return new Number (r);
}
```

The Vuza counterpart for this is the following:

```
(begin
  (define fltklib (import "fltklib.so"))
  (define fl-run (make-procedure
    fltklib "fn_fl_run" 0))
)
```

With the described techniques, we tried to simplify as much as possible the interaction between the two languages in order to augment and facilitate their integration.

3. BINDINGS

Vuza implements two bindings to well known and largely used C++ environments: Csound and FLTK . The former provides access to real-time audio processing while the latter lets the user to create custom GUI; both libraries are portable and work on the majority of existing platforms.

While a complete discussion of this feature of the language is beyond the scope of this paper, a short description of the two bindings (called respectively *vuzcsnd* and *vuztk*) will be provided below.

3.1 Callback-lambdas

All the bindings in Vuza are possible through the concept of *callback-lambda*. Callback-lambdas are blocks of Vuza code that can be associated to any particular event; for example it is possible to create a callback that responds to an action done from the user on the graphical interface or to a time event raised by the system.

3.2 Csound and real-time

The Csound language (`http://www.csounds.com`) is a well known environment for audio generation that provideS, among other things, common facilities for real-time processing across several platforms. The

The following code shows a short example where a simple Csound script is integrated into Vuza code as string and then rendered in real-time:

```
(begin
  (load "vuzcsnd.scm")

  (csound-initialize CSOUNDINIT_NO_ATEXIT)

  (define cs (csound-create))

  (define orc "sr=44100\nksmps=32\nnchnls=2
    \n0dbfs=1\n\ninstr 1\naout vco2 0.5, 440
    \nouts aout, aout\nendin")
    (define sco "i1 0 1")

    (csound-send cs 'set-option "-odac")

  (csound-send cs 'compile-orc orc)
  (csound-send cs 'read-score sco)
  (csound-send cs 'start)
  (csound-send cs 'perform)
  (csound-destroy cs)
)
```

Other then the approach proposed above, the binding also provides a multi-threaded mechanism for audio processing with several ways for accessing internal variables of Csound.

3.3 FLTK and user interfaces

The Fast, Light Toolkit (FLTK, `http://www.fltk.org/index.php`) is a cross-platform graphical user interface library originally developed by Bill Spitzak, suitable for general UI programming. The library has a very sleek programming API and is very fast. The following code shows the implementation in Vuza of a simple example to create a window on the screen:

```
(begin
  (load "vuztk.scm")

  (define win (fl-make-widget
    'window 0 0 340 180 "FLTK"))

  (define box (fl-make-widget
    'box 20 40 300 100 "Hello, World!"))

  (fl-send box 'box FL_UP_BOX)
  (fl-send box 'labelfont (+ FL_BOLD FL_ITALIC))
  (fl-send box 'labelsize 36)
  (fl-send box 'labeltype FL_SHADOW_LABEL)

  (fl-send win 'end)
  (fl-send win 'show)

  (fl-run)
)
```

The resulting window is depicted in figure 1; the original code in C++, taken from the FLTK examples, is shown below:

```
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>

int main(int argc, char **argv) {
  Fl_Window *window = new Fl_Window(340,180);
  Fl_Box *box = new Fl_Box (20,
    40,300,100,"Hello, World!");

  box->box(FL_UP_BOX);
  box->labelfont(FL_BOLD+FL_ITALIC);
  box->labelsize(36);
  box->labeltype(FL_SHADOW_LABEL);

  window->end();
  window->show(argc, argv);

  return Fl::run();
}
```

The two examples are very similar, showing that it is very easy to port to Vuza some code written in C++ that uses the FLTK library.

The following example, finally, shows how to create a callback-lambda that reacts to a user action; when the user will press a button in the GUI, the program will print a string on the screen:

```
(begin
  (load "vuztk.scm")

  (define window (fl-make-widget
    'window 0 0 395 180 "FLTK"))

  (define slider1 (fl-make-widget
    'hor-value-slider 5 5 380 30 "Sl1"))
  (define slider2 (fl-make-widget
    'hor-value-slider 5 65 380 30 "Sl2"))
  (define slider3 (fl-make-widget
    'hor-fill-slider 5 125 380 30
    "(Sl3 - connected to Sl1)"))

  (fl-send slider3 'set_output)

  (define (cback1 caller)
    (define v (fl-send caller 'value))
    (display "Sl1 value = ") (display v)
      (newline)
    (fl-send slider3 'value! v))

  (define (cback2 caller)
    (define v (fl-send caller 'value))
    (display "Sl2 value = ") (display v)
      (newline))

  (fl-send slider1 'callback cback1)
  (fl-send slider2 'callback cback2)
  (fl-send slider3 'color 127)

  (fl-send window 'end)
  (fl-send window 'show)

  (fl-run)
)
```

The callback-lambda is implemented, in the previous example, as a λ-function and is passed to the operator *fl-callback* as a parameter.

4. CONCLUSIONS

The Vuza project is still in an early stage; however, the language is already pretty powerful and usable. The continuation of the development will focus on the following areas:

- *improve R4RS compatibility*: while Vuza is pretty compatible with the standard R4RS of the Scheme language, there are still some incompatibilities that we would like to address;
- *complete current bindings*: not all functions from FLTK and Csound are exported into Vuza now; we want to complete the implementation;
- *add new bindings for creative coding*: we think that adding a binding to libraries for creative coding could increase Vuza scope; for this reason we would like to support to the OpenFrameworks library (`http://www.openframeworks.cc/`) and we would like to provide some integration with the Max/MSP environment.

Creating a new programming language means, ultimately, giving a new perspective on reality [7]. We strongly believe that this is a really important activity, especially in the present moment where all is evolving very fast.

Acknowledgments

We would like to thank Anthony Hay [7] and Leo Uino [8] for their useful suggestions and comments regarding the internal implementation of Vuza. This language will be used during a research project at Ircam in Paris, from october 2014; for this reason we would like to thank Arshia Cont for making this possibility concrete.

Figure 1. The Hello world example.

[7] See `http://howtowriteaprogram.blogspot.com.es/2010/11/lisp-interpreter-in-90-lines-of-c.html`.

[8] See `http://www.lwh.jp/lisp/`.

5. REFERENCES

[1] Church, Alonzo, *The Calculi of Lambda Conversion*, 1941, Princeton University Press.

[2] McCarthy, John, *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*, Commun. ACM, April 1960, pages 184–195, ACM, New York, NY, USA.

[3] McCarthy, John, *LISP 1.5 Programmer's Manual*, 1962, The MIT Press.

[4] Hal Abelson, Jerry Sussman and Julie Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, 1984, ISBN 0262010771.

[5] Graham, Paul, *On Lisp*, Prentice Hall, 1993, 432 pages, ISBN 0130305529.

[6] Bernardini Nicola and Rocchesso Davide, *Making sounds with numbers: a tutorial on music software dedicated to digital audio*, Proceedings of the Conference on Digital Audio Effects (DAFX-98), pages 192–201, Barcelona / Spain, November 1998.

[7] Scaletti, Carla, *Computer Music Languages, Kyma, and the Future*, Comput. Music Journal, Winter 2002, pages 69–82, MIT Press, Cambridge, MA, USA.

[8] Ge, Wang, *A History of Programming and Music*, Cambridge Companion to Electronic Music, 2008, Cambridge University Press.

[9] Cella, Carmine-Emanuele, *Sound-types: a new framework for symbolic sound analysis and synthesis*, ICMC 2011, Huddersfield, United Kingdom.

[10] Cella, Carmine-Emanuele and Burred Juan José, *Advanced Sound Hybridizations by Means of the Theory of Sound-Types*, Proc. International Computer Music Conference (ICMC), Perth, Australia, August 2013.