

# “Once again text & parenthesis – sound synthesis with Foo”

Gerhard Eckel  
Ramón González-Arroyo  
Martin Rumori

April 30, 2004

**Foo** is a sound synthesis tool based on the Scheme language, a clean and powerful Lisp dialect. **Foo** is used for high-quality non-realtime sound synthesis and -processing. By scripting **Foo** like a shell it is also a neat tool for implementing common tasks like soundfile conversion, resampling, multichannel extraction etc.

## Note:

According to the talk at the Linux Audio Conference, this text will mainly cover the **Foo** kernel layer. This is because the main author of this text, Martin Rumori, is mostly involved with porting and developing the **Foo** kernel. Quotation from [5]:

Whereas the **Foo** kernel layer implements the generic sound synthesis and processing modules as well as a patch description and execution language, the **Foo** control layer offers a symbolic interface to the kernel and implements musically salient control abstractions.

Find out more about the **Foo** control layer in [4] and [5] and the **Foo** control layer’s source code at [1].

## 1 Introduction

When the **Foo**-project evolved at ZKM Karlsruhe in 1993, nobody knew how to call it. Just to be able to talk about, it got the working title “foo” according to RFC 3092. When it came to the first publicly available version, its “nickname” had got deep into the slang of the authors, and considering that “foo” may stand for the two main programming paradigms used in it (functional and object oriented) it was decided, not without some irony, to leave it as its name. Due to that, the installation of **Foo** at IRCAM’s computers was refused by their administrator first...

Since the *SourceForge* team has been granted the takeover of the sample project “foo” when the program was ten years old in late 2003, the existence of `/usr/bin/foo` is legalized now. To avoid confusion, we discourage from using the term “foo” as a general sample name in the future :-)

## 2 History of Foo

**Foo** was developed by Gerhard Eckel and Ramón González-Arroyo at ZKM Karlsruhe in 1993. Its development was continued by the authors in the context of an institutional collaboration between IRCAM and ZKM

until 1996. At that time, machines by *NeXT* running NeXTStep were the computers of choice for those tasks. Since NeXTStep is based on Objective-C, the kernel part of **Foo** was written in that language as well.

The original motivation was the lack of a high quality tool providing techniques known from the analogue audio tape, such as varispeed playback. In the digital domain, the most crucial point with those techniques is the *resampling algorithm*. Unlike other sound synthesis programs (e. g. *Csound* with the *oscil\**- or *phasor/table\**-opcodes), **Foo** allowed for scalable, high quality resampling using the Sinc-Interpolator [6] from the very beginning <sup>1</sup>.

After an infrastructure for accessing these key features *musically meaningful* had been designed and implemented, additional functionality was integrated with **Foo**, such as oscillators and filters. Thanks to its open and extensible design, **Foo** got a standalone, general purpose sound synthesis system.

## 3 Key concepts of Foo

### 3.1 Patch generation

**Foo** was also inspired by patch based sound synthesis systems such as *Max/MSP*. Similar to an analogue synthesizer, different basic modules are connected in order to form more complex signal processing entities.

Unlike *Max*, **Foo** does not copy the wiring process of an analogue synthesizer one-to-one to the screen. In fact, **Foo** is meant as a *patch generation language*. Currently, this language is *Scheme* [7], a clean and powerful *LISP* dialect. *Scheme* allows for patch generation in several abstract ways, such as recursion and high order functions. It is very easy to build patch templates which are instantiated several times with different parameters, which is quite hard with graphical languages like *Max*.

In **Foo**, everything is a signal. There is no distinction between audio rate and control rate, since that inherently holds the risk of aliasing artefacts. There are *Scheme* bindings for the constructors of each available *module* (unit generator), which evaluate to the signal produced by that modules. This value in turn may be used as an input for another module constructor.

### 3.2 Context

Dealing with patches in **Foo** is done via so called *contexts*. A *context* is kind of a container for a patch, which allows for treating a patch as an entity. From outside a *context*, the resulting signal of a complete patch is accessible via the *output* modules of that patch only.

A *context* is also a means for “executing” the associated patch. Using a *task*, one can render a *context* into a soundfile. Therefore a *context* represents exactly the sound it can produce, it is somehow a compressed *description* of the sound.

With **Foo**, it is possible to save such *contexts* in a binarily serialized form and load them again into the runtime system. This is useful especially when working with *incremental mixing* (see 3.5), since you don’t have to keep all the interim versions as probably large, space-wasting soundfiles.

### 3.3 Time

Each **Foo** patch is associated with a *context*. This is visible for the *output* modules as well as for *temporal* relations.

A **Foo** *context* has a local *time origin*, which is zero. Any patch structures inside the *context* are temporally related to this time origin by specifying a time shift. This shift can be positive or negative; the *context*’s time axis reaches from negative to positive infinity.

Time shifts can be nested, so that every shift refers to the outer time frame (with the *context*’s time origin as outmost frame).

---

<sup>1</sup>As of 2002, *Csound* allows for sinc interpolation by means of the *tablext* opcode

### 3.4 Task

A **Foo** *context* containing a complex patch is just a description for creating e. g. a sound file. A *context* itself knows nothing about the concept of sampling rate, sample format, soundfile headers etc. Thus a *context* is an abstract description which could be used in several different environments after it has been constructed.

A **Foo** *task* is an execution controller for a *context*. All the above mentioned parameters are set by the *task* object when binding the *context* to an output medium (currently a sound file, which provides those settings like sample rate and -format).

A *task* also provides a means for handling the *context*'s time model (even **Foo** is not really able to create sound files with an infinite duration. . .). This is done by two temporal related parameters of the *task* constructor: the *reference* and the *offset* values. The *reference* determines where in the associated output sound file the time origin of the *context* should be anchored, while the *offset* parameter specifies at which position in the *context*'s time axis the rendering process should start. Together with the *duration* parameter of the *task*'s rendering process, one can specify exactly which *part* of the *context* is being rendered.

### 3.5 Incremental mixing

The clean semantics of *task*, *context* and *time* in **Foo** allows for another neat feature: the *incremental mixing*. Consider *contexts* different layers of a composition, you might want to be able to *incrementally* construct the final composition out of these layers and perhaps do later corrections to one of the layers.

With **Foo**, a *task* is not just able to render a *context* into a new soundfile, but can also *add* the resulting sound material into an existing file at a specified time (via the *reference* parameter). When archiving each of the involved *contexts* along with the resulting file, it is later possible to render one of the layers again into a temporary file and to subtract it from the resulting file. This way, it's possible to do later corrections in the layer layout of a composition without having to keep all the intermediate versions and single layers as sound data.

### 3.6 Scripting Foo

Another one of the charming features of **Foo** is its scriptibility. Unlike with other sound synthesis systems, one does not necessarily have to enter the **Foo** environment (in other words, the **Foo** command line prompt) for doing sound synthesis tasks. Instead, it's possible to write "**Foo** scripts" like shell scripts, which could then be used as standalone signal processing applications. This is extremely useful for recurring basic tasks, like resampling, or for batch processing in general.

Similar to a shell script, a directive like `#!/usr/local/bin/foo` sets **Foo** as the interpreter for a script. The argument vector issued when calling the script is accessible via the `(command-line-args)` function from inside the script. That way it is possible to create complex scripts which are seamlessly integrated with the usual shell environment.

## 4 Future plans

The last substantial changes to **Foo** were made in 1996. Now, with having been ported to *Linux* and *Mac OS X*, **Foo** gets faced with a completely different world compared to that of the middle nineties. . .

### 4.1 Dynamically loadable modules

One major goal is to create a more flexible *module* interface for **Foo**. By now, the signal processing modules are compiled into the **Foo** kernel. A solution with dynamically loadable modules would make it easier for developers to add new modules to **Foo**.

This would also allow for interfacing with other DSP systems, such as an interface to *LADSPA* [8]. We are also thinking of using *Faust* [9] as a means for building modules for **Foo**.

## 4.2 Typed signals

To improve the flexibility of **Foo**, we think of introducing signal types other than audio signals, so that control signals, triggers etc. could be used more efficiently.

## 4.3 Further modularization of Foo

Currently, the **Foo** kernel consists of a single library, which is an extension to the *elk* [10] Scheme interpreter. To allow for more flexible use of **Foo**, it will see some restructuring.

The **Foo** kernel will be a library written in Objective-C, which could be used for other applications, too. The interface to the *elk* interpreter will be done via another lightweight library as an extension. This way, **Foo** could be easily interfaced to other Scheme interpreters as well as other languages in general.

## 4.4 Jack interface

After having ported **Foo** to *Linux*, the preliminary direct play support (via the `(play~)` module) was disabled.

To ease the process of composing with **Foo**, we think of creating an interface to the *jack* audio server [11]. This would mean just a sound file player which is better integrated with **Foo** than other players; it will *not* mean realtime rendering capabilities for **Foo**. In conjunction with an interface to the *jack* transport API, rendering and playing could be triggered by jack events, which makes using sound material created with **Foo** in other applications more comfortable.

## References

- [1] **Martin Rumori**: *Foo Website*,  
<http://foo.sourceforge.net>
- [2] **Gerhard Eckel, Ramón González-Arroyo**: *Foo Kernel Concepts*, ZKM, Karlsruhe 1996
- [3] **Gerhard Eckel, Ramón González-Arroyo**: *Foo Kernel Reference Manual*, ZKM, Karlsruhe 1994
- [4] **Gerhard Eckel, Ramón González-Arroyo**: *Foo Control Reference Manual*, ZKM, Karlsruhe 1993
- [5] **Gerhard Eckel, Ramón González-Arroyo**: *Musically Salient Control Abstractions for Sound Synthesis*, Proceedings of the 1994 International Computer Music Conference, Aarhus, 1994
- [6] **Julius O. Smith**: *The Digital Audio Resampling Home Page*,  
<http://www-ccrma.stanford.edu/~jos/resample/>
- [7] **'(schemers . org)**: *An improper list of Scheme resources*,  
<http://www.schemers.org>
- [8] **LADSPA**: *Linux Audio Developer's Simple Plugin API*,  
<http://www.ladspa.org>
- [9] **E. Gaudrain, Y. Orlarey**: *A FAUST Tutorial*,  
[ftp://ftp.grame.fr/pub/Documents/faust\\_tutorial.pdf](ftp://ftp.grame.fr/pub/Documents/faust_tutorial.pdf)
- [10] **Oliver Laumann et al.**: *Elk: The Extension Language Kit*,  
<http://sam.zoy.org/projects/elk/>
- [11] **Paul Davis et al.**: *Jack: The Jack Audio Connection Kit*,  
<http://jackit.sourceforge.net>