Real-time Composition in Elody

Stéphane Letz, Dominique Fober, Yann Orlarey

Real-time composition in Elody

Stephane Letz, Dominique Fober, Yann Orlarey
Grame 9, rue du Garet 69001 LYON
[letz, fober, orlarey]@grame.fr

Abstract

Elody was initially an environment for musical composition allowing the description and algorithmic manipulation of non real-time *musical structures. To allow the definition of real-time transformation processes, we have added a new primitive in the language : the* real-time input stream. *This object can be manipulated and transformed like non real-time objects even before being known. Evaluating a real-time expression gives as result a command sequence which drives a transformation engine. This one transforms a real-time input stream in an output stream.*

1 Introduction

Elody is a music composition environment based on a visual functional language, a direct manipulation user interface and Internet facilities [Orlarey & al.1997]. The programming language is based on a music description language extended with lambda-calculus. Programming in Elody basically consist in defining abstractions from concrete structures and applying them on new arguments to produce new results.
The functional approach is particularly well suited to non real-time composition, where values of the language are temporal objects played later in real-time by a rendering engine. Musical structures and processes are first class objects that can be freely composed in more complex ones.
Using the functional paradigm is of growing interest also for the definition of real-time multimedia languages. The idea is to describe interactions with an external environment at a high level of abstraction, and have the system deal with the problem of low level interactions or side effect primitives ordering.
Following this idea, a very interesting system called *Fran* (for *Functional Reactive Animation*) has been recently developed. Fran is a Haskell library (or "embedded language") for interactive animations with 2D and 3D graphics and sound [Elliot, Hudak]. This system allows the declarative specification of multimedia presentation in a pure functional language by combining *temporal reactive behaviors*.

This paper will show how a simple model of interaction with an input stream can be introduced in Elody. Interaction will be limited to transformation processes composed and applied on the real-time input stream.
Section 2 to 5 describe the method used on a Midi stream, and show how very complex transformation processes can be defined. Section 6 will present the same method used on a real-time audio stream.

2 Input/output in functional languages

There have been many different approaches to the problem of doing input/output within pure functional languages. The recurring problem of the integration of input/output scheme in functional languages is the integration of context-sensitive transformations rules that describe interactions of programs with an external environment, with the context-free rules of the evaluation of functional expressions. Thus they are theoretical difficulties to describe reactive or real-time systems in a pure functional manner when an imperative approach with states and side-effects seem more suitable.

The standard way is to consider input/output operations as functions which transform the current system state in a new modified state and possibly return a result. This kind of function is called an *action*. Basic actions can be composed in more complex ones using a*ction combinators* in a way that guarantee their sequential execution and the transmission of intermediate results between operations. This idea behind actions comes from the concept of *monad* where one distinguish between *simple values* and *computations* that return values, but may have additional computational effects.

A monad is represented by a triple (*M*, *unitM*, *bindM*) consisting of a type constructor M, and a pair of functions unitM and bindM [Wadler 1992]. By expressing all input/output operations using the monad concept, the good properties of functional languages, like *referential transparency* are kept.

Another interesting idea is the *unique type* concept used in the Clean functional language [Eekelen, Plasmeijer 1998]. The system global state is modeled using a *unique type object* and the language typing system guarantee it correct use : no duplication and sequential modification of the state.

To allow the description of real-time processes in Elody, the general action model has been simplified : an action is a transformation which has a side effect on the input stream, without returning any result.

3 Extension for real-time in Elody

3.1 The real-time stream

A new primitive element is added in the language : the *real-time input stream*. It is an infinite sequence that contains all events received in real-time by the system. Although this stream is only known when the expression is evaluated and reduced, it is possible to manipulate it like a *known object*. The system is in charge of translating actions described by the user into effective operations that will *operate in real-time*, during expression rendering. The user makes a specification in terms of temporal organization and manipulation of the real-time stream, the rendering engine will execute low level side effect operations according to the specification.

3.2 Integration in the language

The technique used is separated in 3 steps :

- the *evaluation* of an expression gives as result a data structure (actually a tree), which describes the temporal organization and transformation of parts of the real-time input stream. The rendering step is decomposed in two parts :
- the *compilation* produces a time ordered sequence of commands which describes the transformation to be applied on the input stream.
- the *transformation* is done by a rendering engine that receives a stream of commands and transforms a real-time input stream into an output stream.

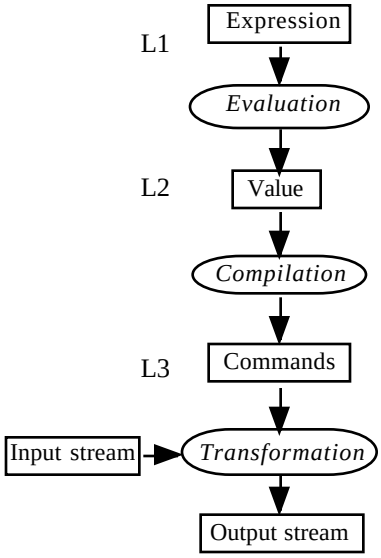

fig 1: list of treatments

4 Language semantic

Fig 1 shows the list of internal treatments used to produce the output stream starting from an user defined expression. The L1 language contains *expressions* defined by the user. After an *evaluation* step, a term of the *value* language L2 is obtained. After *compilation* of the value, a term of the *command* language L3 is obtained. Commands drive a *transformation* state machine which receives the real-time input stream and produces the output stream.

4.1 Expression evaluation

A new primitive element which denotes the infinite sequence of received events has been added : $input[d, t_{in}, c, n]$, the real-time input stream taken on a duration d, starting at the date t_{in}, with a compression/expansion ratio c, and a transposition factor n. The expression built by the user denotes the output stream in terms of temporal composition (using *Seq, Mix, Begin* and *End* constructors) and transformations of parts of the input stream. Each reduction rule of the Elody language [Orlarey & al.1997] has been extended to take account of the new element.

4.2 Value compilation

The value obtained after evaluation of an expression is still a specification of the expected output stream. The purpose of the compilation step is to *convert* this specification into a *description of real operations* to be applied on the input stream to obtain the desired output stream. A value is compiled in a sequence of commands. Each command stands for a basic transformation to be done on all events received during its duration. Each transformation works on a portion of the real-time input (characterized by a date t_{in} and a duration d_{in}) and produces a portion of the output stream using a delay value o and a expansion/compression ratio c, and possibly modify other parameters like pitch or velocity.

For a given transformation, we have the following scheme :

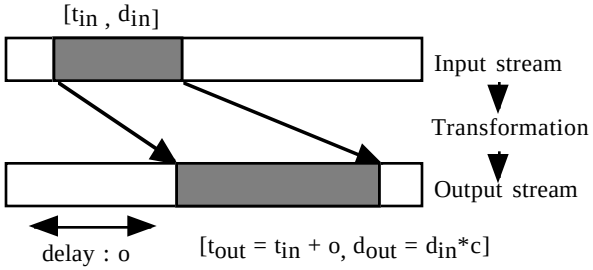

fig 2 : expansion and delay effect done on a portion of the real-time input

4.3 Real-time transformation

The result after compilation is a sequence of commands coded as a pair of associated events. The first event starts the transformation, the second event stops it. The sequence is played and drives the transformation engine, a state machine which contains, at a given date, a list of active transformations. For each received event, all active transformations are executed and the resulting event list is mixed into the output stream.

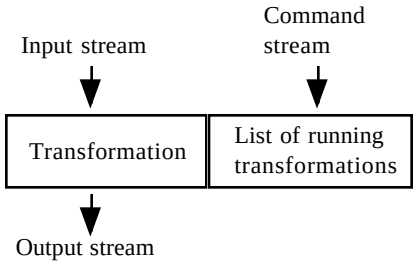

fig 3 : the transformation step

5 Examples of real-time transformations

Composing real-time transformations become as easy as composing and manipulating ordinary musical structures. *Higher order scores* (scores of programs) of real-time transformations processes can be defined using the *Seq* or *Mix* constructors. The score metaphor which gives an explicit representation of the time dimension now can be used for real-time processes. They become first class objects which can be freely composed in more complex transformations.

5.1 Temporal constructions : thru, delay, echo

The simplest program that can be described is a "*thru*" process.

It is defined by the following expression : $input[\infty,0,1,0]$

If the real-time input stream is delayed by 1 second, the output produced starting at date 0 will contain a 1 second rest followed by the input starting at date 0 : the resulting process is a delay which re-send each received event with a 1 second delay : $seq[sil[1000],input[\infty,0,1,0]]$

Several instances of the real-time stream can be used simultaneously and mixed : the following expression describes an *echo* process where each received event is sent 3 times : immediately, with a 1 second delay and with a 2 second delay.

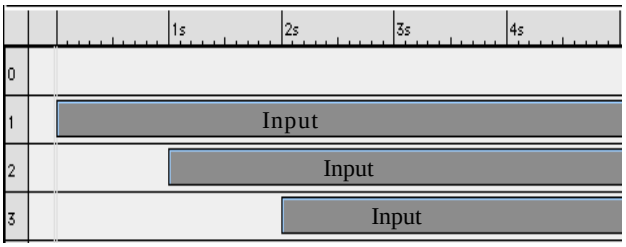

fig 4 : a triple real-time echo

5.2 Temporal manipulation

Like for a non real-time object, one can cut a part of the real-time input stream : in the following example, the first 5 second of the real-time input are cut and repeated twice. The resulting process does a Ò*thru*Ó on the first 5 second and the result is repeated twice.

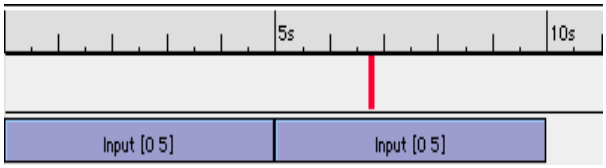

fig 5 : 2 times repetition of the real-time input 5 first seconds

5.3 Compression/expansion

The real-time stream can be compressed or expanded. The following example is similar to fig 4, but here a compression ratio of 0.5 is used. Events received between dates 0 and 10 s can not be played according to the compression ratio. They are transformed and sent as soon as they are available, that is old events which are in the past will be sent at the current date. The section between 5 and 10 s will be played correctly.

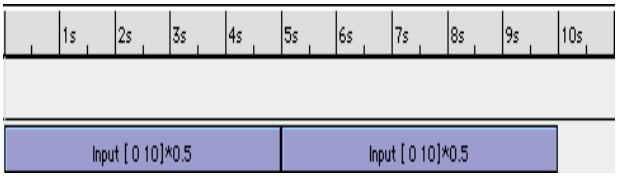

fig 6 : 2 time repetition of the 10 first seconds of the real-time input stream compressed by a ratio of 0.5

5.4 Transformations composed in time

Elody allows to build *higher order scores*, that is functions organized in time (using *Mix* or *Seq* constructors) that can be applied on arguments. [Orlarey & al.1997]. Like for usual musical objects, *score of programs* can be used to specify (for example) a sequence of transformations done on successive parts of the real-time input stream. In the following example, 2 different transformations are done : a *canon* function between dates 0 and 5 s and an *echo* function between dates 5 and 10 s.

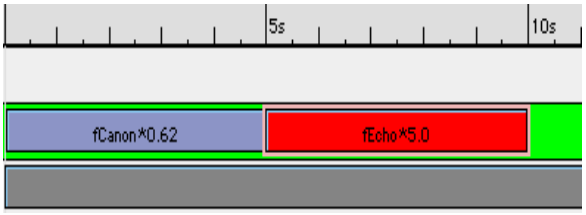

fig 7 : sequence of functions applied on the real-time input

5.5 Combination of real-time and non real-time expressions

Because composing real-time processes is similar to composing usual musical structures, it is really easy to use in real-time compositional operations previously defined and used on non real-time objects. One can also Ò*merge*Ó in a unique description, real-time and non real-time objects. In the following example, a sequence of abstractions is applied on a structure which mix the real-time stream and a sequence *seq1*.

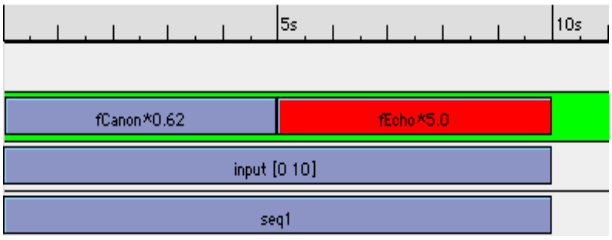

fig 8 : sequence of functions applied on the mix between a non real-time object and the real-time stream

5.6 Combination of transformation processes

A transformation expression, result of a first level of composition can be used as the base for more elaborated constructions. In the following example we have :

- first level : a sequence of transformations is applied on the real-time input stream, here a *canon* function applied on successive portions of 4 seconds.

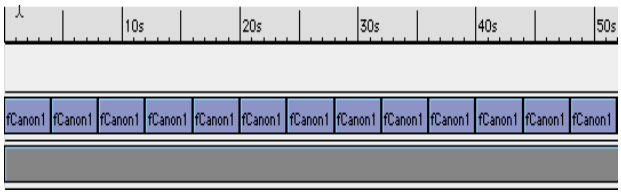

fig 9 : sequence of *canon* functions applied on the real-time input stream

- second level : the resulting stream is separated in sections of 16 seconds, named A, B and C. These elements are used to build the following temporal construction :

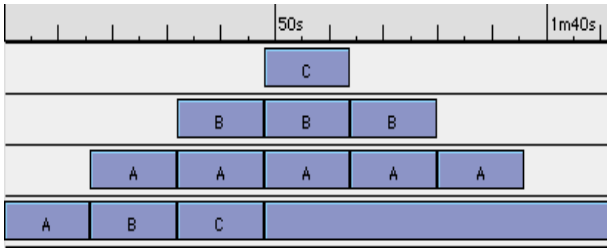

fig 10 : construction using the result stream obtained in fig 9

The resulting process is now quite complex : the first 16 seconds of the real-time input stream are transformed according to the first level of transformation. Between dates 16 and 32 s, the *whole result* of this first transformation (A section) is repeated and mixed with the result of the first process, which is still applied on the real-time input (B section).

Any already transformed object can be freely re-used in more complex expressions. Abstractions defined using non real-time expressions can be used to build real-time processes and vice-versa. Thus very complex transformations processes can be built by *combining* and *re-using* simpler ones.

6 Audio version

An audio version of the mechanism previously described using a MIDI stream has been implemented. It is based on the new audio extension developed recently in MidiShare [Fober & al. 1995].

- audio streams are defined as sequences of time-stamped audio events that contain sample buffers.

- the audio In driver is in charge of splitting the continuous audio input stream in time-stamped audio events, that are distributed to each connected application.

- the audio Out driver receives and mix audio event streams coming from all connected applications, and produces a continuous audio output stream.

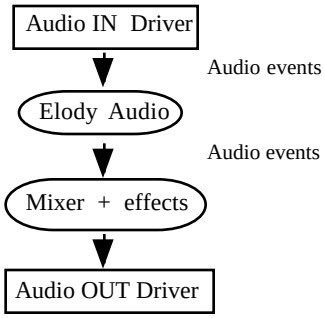

fig 11 : Elody audio architecture

In the Elody audio version, the real-time audio input can be temporally manipulated and transformed by applying audio effects (chorus, reverb...). After evaluation and compilation, the command sequence drives the transformation engine which produces a stream composed of audio events and effects activation events. This stream is finally sent to the audio output driver.

7 Conclusion

We have presented an extension implemented in Elody which allows the description of real-time transformations processes. A new primitive element is added in the language : the real-time input stream. It can be manipulated and transformed with the existing constructors and tools. Real-time transformation processes are described as expressions which manipulate and modify the real-time stream. They are evaluated and compiled into a sequence of time-ordered simple commands executed in real-time. A rendering engine driven by commands transforms an input stream into an output stream.

Working at the level of abstract descriptions of interactions allows the user to free itself about the problems of low level interactions or side effect primitives ordering. This is achieved through the definition of more abstract operations that are first class objects and thus can be freely composed into larger interactions processes.

This work is a step towards the definition of music interactive programs using a pure functional language. We plan to add new primitives to deal with *reactive processes* which are activated when external events are received. This will allow to describe a larger class of interactive programs.

References

[Eekelen, Plasmeijer 1998] Marco Van Eekelen, Rinus Plasmeijer. *Concurrent Clean language report.* High Level Software Tools B.V and University of Nijmegen 1998.

[Elliot, Hudak] Conal Elliott, Paul Hudak. *Functional Reactive Animation.* In the proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97).

[Fober & al. 1995] Fober D., Letz S., Orlarey Y. - *MidiShare, un syst me d'exploitation musical pour la communication et la collaboration* - Actes des Journ es d'Informatique Musicale JIM95, Paris, pp.91-100.

[Orlarey & al.1997] Y Orlarey, D Fober, S Letz. *LÕenvironnement de composition musicale Elody* . Actes des 4¡ Journ es dÕinformatique Musicale JIM 97 Lyon pp 122- 136.

[Wadler 1992] Philip Wadler *Comprehending monads*. Mathematical Structures in Computer Science, Special issue of selected papers from 6'th Conference on Lisp and Functional Programming,2 461-493, 1992.