Real time functional languages
Stéphane Letz, Yann Ortuzy, Dominique Fober
Grame 6, Quai Jean Moulin 69001 Lyon-France
letz@rd.grame.fr - ortuzy@rd.grame.fr - fober@rd.grame.fr

Abstract: This paper introduces two real-time functional programming languages. The first one aims to
capture temporal trajectories. The second allows the manipulation of both real-time and deferred time streams.
We shall describe the architecture of the real-time reduction machine used for evaluation. We shall also review
the expected consequences of this approach.

1 Introduction
Functional programming is of growing interest for the definition of musical languages. Conventional
approaches are based on functional languages like Lisp [3] or Haskell [4] specialized by adding specific data structures and
functions to describe musical objects. These languages are particularly well adapted to deferred time
composition, where values of the language are temporal objects (MIDI File or a Csound file for example) time-
rendered later by specialized applications (a MIDI File sequence for example). Using a functional language in a
real-time context is more problematic. There are indeed theoretical difficulties to describe reactive or real-time
systems in a pure functional manner when an imperative approach with states and side-effects is more suitable.
Different methods have been investigated in this domain like stream model (a stream is a lazily evaluated list
which holds all events and thus become a timeless entity[12]), data-flow models [10] and in the musical domain
Artic [1], which uses a concept of time-varying function).
Current functional approaches in musical languages are either in the deferred time programming or in the real-
time programming domain and few work has been done to unify them. Indeed, in the classical approach, the
evaluation process transforms an expression in a normal form called the value. This value will be later
rendered, but is used by an output device and for example printed on a screen or saved as a file. There is thus a
gap between the time to produce the value and the time to consume it. When values are finite or timeless objects,
this gap does not have any consequences and is even necessary when the evaluation duration exceed the
rendering duration, or when calculating a real-time incompatible value like a time inversion for example. In
return, this separation is unsuitable when the language produces infinite values whose the output process can
represent only an approximation. In a functional language denoting possibly infinite, temporal or "reactive"
values, we suggest to remove this gap between the evaluation time and the rendering time by merging the
expression evaluation with its rendering. We call this method rendering-driven evaluation.

Our approach is on two concepts:
• the stream notion which allows to describe real-time and deferred time object in a unique paradigm.
• the use of a lazy functional language implemented by a real-time reduction interpreter.
The paper is organized as follows: we shall first present a real-time language which aims is the description and
manipulation of temporal trajectories. We shall then explain more precisely how the real-time interpreter works
and describe its implementation. Finally, we shall present the stream language which handle the real-time input.

2 The Turtle language
Referring to the Logo language, the "turtle language" allows to describe and manipulate temporal trajectories,
that is geometric shapes built of temporal rectilinear moves and instantaneous rotations. These trajectories will
be generated in real-time both as a graphic output and as a MIDI events stream holding the different position
coordinates. Following an approach based on Lambda Calculus used to design programming languages [7],
we shall build a complex language by adding a limited number of constructs to a specialized descriptive language.
This descriptive language uses two elementary move "instructions" and a seq operator to put instructions in
sequence.

(move x d) instruction takes time to execute:
(1) \delta = (move x d) \rightarrow x + d
(right a) instruction is instantaneous:
(2) \delta = (right a) \rightarrow \delta

ICMC PROCEEDINGS 1995
Finally the sequent of two trajectories takes their cumulated duration:

\[ t\cdot x = \frac{t_2 - t_1}{t_2 - t_1} \cdot \left( \begin{array}{c} x_2 \\ x_1 \end{array} \right) \]

We define a rendering function \( R \) which associates a graphic output and a stream of events to any curve:

\[ R = \text{seq} (\text{move} 50 1) (\text{right} 120) \]

\[ \text{move} 50 1 \]

\[ \text{right} 120 \]

Now we add Lambda Calculus abstraction and application constructs and a limited number of primitives: (numbers, if and list constructs) to build the complete language with a Scheme-like syntax:

```scheme
<expression> ::= <number> | <expr> (copm <expr> <expr>)
| (<expr> <expr>) | (let <expr> <expr>)
| (if <expr> <expr> <expr>) | (<expr> <expr>)
| (seq <expr> <expr>) | (move <expr> <expr>) | (right <expr>)
| (seq <expr>) .

<expr> ::= 

<evaluation> ::= (define (expr expr) expr)
```

Square definition

```scheme
(define side (equiv move v1 (right 50))
(define repeat (equiv move 0)
(define lx 6 0 12 0 (seq sqare 20)
(define ly 6 0 12 0 (seq sqare 20)
```

The evaluation of an expression produces a "value", a trajectory, as a sequence of movements. Because of the language lazy semantic, this sequence will be evaluated at the rendering function request. In order to implement the previously described temporal semantic, a recursive instruction value is executed instantaneously as do a move instruction value but then the current evaluation is suspended during its duration and later resumes. We have a time-rendering-driven evaluation.

This language construction method leads to the interesting property: elements of the descriptive language become first-class objects for the complex language, so they can be used as function parameters, returned by functions. Moreover, due to lazy evaluation semantic, one can build infinite objects.

```
(definition mode x y (* x (+ x y y)))
```

Turbulence:

```
(turbulence x (a x))
```

This object can be used by the rendering function without being completely evaluated. For a practical use, evaluation is stopped by the user.

3 The real-time interpreter

We have developed a particular evaluation strategy to implement the previous language. An interpreter is a function taking a representation of the program, which reduces it to its value according to the language semantic. The interpreter is implemented using a defining language with its particular features like eager or lazy evaluation strategy. Because of the time-rendering-driven evaluation, we need a complete control on evaluation starts in order to concep reductions in time. We use a stack-based machine [2] which manipulates an expression, an environment and an explicit continuations. Evaluation process is a sequence of simple reductions which operates stack manipulations until the final value is reached. The stack holds the whole continuation, that is all the remaining operations which should be executed to calculate the value. Each language term has a table of methods to handle: memory allocation and destruction, evaluation, application, rendering depending of its type. A basic reduction step does the following:

- pop the expression and the method index on the top of the stack
- remove the method associated with the expression using the method index
- execute the method.
The real-time input, a sequence of MIDI events separated by time gaps is converted into the interpreter internal representation. Thus the real-time stream is handled like any other stream. The evaluation of an expression produces an output stream and is time-controlled by the rendering process that is the output stream is produced on request in real-time. The evaluation can be suspended either by the output process when rendering a temporal value or by the read process when no events are available. The implementation ensures that output events are sent as soon as they are available. The "past" of the input stream is kept in memory as long as some expression has a reference on it, because the memory management is done by reference counting.

Here are some examples of expressions written in the stream language. The manipulation of real-time and deferred time streams become completely homogenous. Users can describe the output stream as the result of a function taking equally real-time and deferred time streams as input.

```
"mix" real-time, deferred time
(define s1 (cons (event 1) (cons (middlevent 2) ())))
```

```
Streams become first-class objects. Because of the unique way to handle them, real-time transformations can be expressed in a more natural way. More important is the change of our view on the relationship between real-time interactions and deferred time objects. We describe usually an interactive process as a program which behavior is emulated to the occurrence of external events, that is where we "separate" the interaction and the program behavior. We are able now to describe objects containing both real-time and deferred time aspect of their behavior, thus interactive entities become first-class objects.

5 Conclusion

We have presented two real-time functional languages. Using a stream model and a real-time interpreter, we have showed a method to unify descriptions of real-time and deferred time musical objects. This approach is promising because it remains in a purely functional paradigm thus profits of all associated advantages: formal analysis, higher-order functions and lazy evaluation [6]. Moreover, it integrates well in the Midsquare collaborative inter-applications communication model. Indeed, we can combine applications which behavior are described in a pure functional way. For example, it will be possible to calculate the equivalent behavior of two applications inter-connected in succession just by composing their functions.

References


552

ICMC PROCEEDINGS 1995
For a real-time interpreter, memory management has to be regularly done during reduction process. The common process to reclaim unwanted memory cells is called garbage collection and is done by reference-counting in our implementation. This low-level implementation allows to:

- suspend and resume evaluation just by controlling the reduction sequence.
- segment evaluation in small temporal steps because each segment needs a small and finite number of stack manipulations. Here is an example of a reduction step for an application term: (application e1 e2)

```
        p
        x p e1 e2 : expression to reduce
        Eval, Apply : reduction method index
        p : environment continuation
```

**Time management**

We use the time management capabilities of MidShare musical operating system [6], MidShare is driven by an interrupt each ms and allows to schedule tasks in the future. Reductions are completed at interrupt level: a fixed amount of stack reductions is done at each interrupt. If the value is not reached, evaluation continues with the next interrupt call. When representing a temporal value, reductions are suspended and a task is scheduled to resume the evaluation.

```
        expression
        value representation
        evaluation
        suspension
        value representation
        evaluation
        N reductions
        Time
```

Thus, during evaluation of a temporal expression we have:
- an evaluation period with a stack reductions until a value is reached. This value is then used by the rendering function.
- a suspension period equal to the duration of the currently represented value.
- As the whole continuation is kept on the stack, evaluation can be resumed with the expression on the top of the stack.

4 The Stream language

The turtle language is a first example of a time-rendering-driven evaluation [11] fulfilling the need of evaluation temporal control while representing temporal values. We shall now present the stream language. Its aims is to provide a unique paradigm to manipulate both real-time events and deferred time objects. The method is to handle real-time events in a unique object, a stream, adding all events that becoming a timeless entity, which can be manipulated exactly the same way as deferred time streams are. Temporal rendering is in charge of the interpreter. A stream can be defined as follows:

- explicit: as the result of a language construct not involving real-time that is as a lastly evaluated list of events, either MIDI event or "duration" events.
- as constructed from the real-time input.
- as a merge of two streams.
- as the empty stream.

```
<stream> ::= input | (some <event> <stream>) | (merge <stream> <stream>) | null
<events> ::= (atinterval) | (afterinterval n) where n is a duration in seconds
```

Time between two successive MIDI events is made explicit as "duration events", thus the merge operation is perfectly deterministic. Input denotes the stream holding all events received in the real-time input. Streams are manipulated using classical functions: (first x) returns the first element of a stream, (rest x) returns the stream without its first element, null? is a predicate for empty streams.

```
(first x) == event. (rest x) == x (null? x) == True or False
```

The rendering function for a stream has the following behavior:
- it sends a MIDI event when the value is a MIDI event.

ICMC PROCEEDINGS 1995
551