

Lambda Calculus and Music Calculi

Yann Orlarey, Dominique Fober, Stéphane Letz and Mark Bilton

GRAMÉ
6 quai Jean Moulin
69001 Lyon - France
grame@applelink.apple.com

Abstract

This article presents an approach in the design of music programming languages based on Lambda Calculus. It shows, through several examples, that a purely descriptive language, that is to say a language without any programming capability, can be equipped with programming capabilities by the addition of a limited number of simple constructs.

1. Introduction

In this paper we aim to show how a purely descriptive music language can be transformed into a music programming language by introducing the *abstraction* and *application* concepts from Lambda Calculus [Church 1941].

Instead of building suitable music data structures and functions on an actual programming language, we suggest *to build suitable programming languages on music data structures*. This method gives rise to specialized functional programming languages where the same structuring means (the same “glues” using Hughes [Hughes 1989] term) apply to both data and programs.

The method is quite general and can be used for descriptive languages of other domains. In fact our first example will be a graphic calculus whose visual aspect enables better understanding of the method. In the second example we will develop a music calculus based a textual music language and we will end with a visual version of this music calculus.

2. The transformation process

The transformation process is based on two steps:

- a) *Extension of the descriptive language syntax by introducing the abstraction and application of Lambda Calculus*

Understanding the notion of abstraction is essential. Abstraction is an operation which makes some part of an object *become variable*. The resulting object is a *generalization* of the previous one. This generalization can be given different interpretations: a predicate, a class, a concept, a set, or a function. Here we are mainly concerned by the last interpretation.

Application is in some ways the inverse of the abstraction operation. It allows us to *specialize* an abstraction by fixing some of its parts.

- b) *Extension of the Lambda Calculus reduction rules in order to deal with the new specific descriptive languages constructions*

In Lambda Calculus, the β -reduction rule gives a *functional significance* to abstractions. In the

same manner, we add new reduction rules to give a *functional significance* to the other language constructs. In this way, every construction in the language can be applied as a function.

For our examples we shall use the following structure:

- descriptive language presentation
 - syntax
 - examples
- programming language presentation
 - extended syntax
 - program examples which use β -reduction
 - reduction rules extension
 - program examples which use the new reduction rules.

Due to space restrictions we will not describe Lambda Calculus. The interested reader can refer to [Barendregt 1984]. Also we will not discuss the formal properties of the presented calculi.

3. A Graphic Calculus

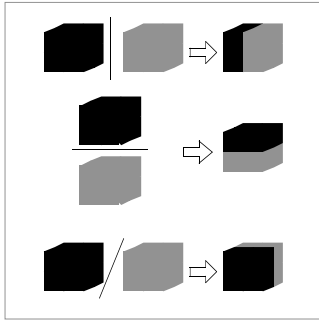
The aim of this section is to present a very simple 3-D graphic calculus based on colored cubes.

3.1. The descriptive language

Our descriptive language uses basic coloured cubes and operators to construct more complex cubes. The syntax appears as follows :

$$\begin{aligned} \text{cube} & ::= \text{color} \\ & \quad | \left[\text{cube}_1 | \text{cube}_2 \right] \\ & \quad | \left[\frac{\text{cube}_1}{\text{cube}_2} \right] \\ & \quad | \left[\text{cube}_1 / \text{cube}_2 \right] \\ \text{color} & ::= \text{white} \mid \text{red} \mid \text{blue} \mid \text{green} \\ & \quad | \text{invisible} \mid \dots \end{aligned}$$

This states that a cube is either a basic coloured cube or a construction of cubes following the three directions of space. So the construction operators respect the rules:



Note : A construction of two cubes always remains a cube, the operands are contracted following the construction axis.

Some Examples

Here are several examples of simple expressions followed by the corresponding graphic result:

a) $\left[\frac{\text{green} \mid \text{white}}{\text{white} \mid \text{green}} \right]$



b) $\left[\frac{\text{white} \mid \text{invisible}}{\text{white}} \mid \frac{\text{invisible}}{\text{white}} \right]$



c) $\left[\frac{\text{white} \mid \text{invisible}}{\text{invisible} \mid \text{white}} \mid \frac{\text{invisible} \mid \text{green}}{\text{green} \mid \text{invisible}} \right]$



3.2. The programming language

To transform our descriptive language into a graphic calculus we must:

- Extend the language syntax with abstraction and application.
- Extend the reduction rules to deal with the application of colored cubes and constructions.

3.2.1. Syntax extensions

The extended syntax is described as follows :

$$\begin{aligned} \text{cube} & ::= \text{color} \\ & \mid \left[\text{cube}_1 \mid \text{cube}_2 \right] \\ & \mid \left[\frac{\text{cube}_1}{\text{cube}_2} \right] \\ & \mid \left[\frac{\text{cube}_1}{\text{cube}_2} \right] \\ & \mid \lambda \text{color} . \text{cube} \\ & \mid (\text{cube}_1 \text{ cube}_2) \end{aligned}$$

$$\begin{aligned} \text{color} & ::= \text{white} \mid \text{red} \mid \text{blue} \mid \text{green} \\ & \mid \text{invisible} \mid \dots \end{aligned}$$

This adds two new terms to the previous one, $\lambda \text{color} . \text{cube}$ which represent an abstraction and $(\text{cube}_1 \text{ cube}_2)$ which represent the application of cube_1 to cube_2 .

In order to simplify the notation we will write $(C_1 C_2 C_3 C_4)$ instead of $((C_1 C_2) C_3) C_4$ by using left to right associativity for application.

As stated earlier, an abstraction is an object generalization obtained by making some part of the object variables. So the abstraction $\lambda \text{white} . [\text{white} \mid \text{blue}]$ is a generalization of the cube $[\text{white} \mid \text{blue}]$ obtained by making the white part variable. The $\lambda \text{white} .$ part declares that white is a bounded variable in the abstraction body $[\text{white} \mid \text{blue}]$.

If we apply $\lambda \text{white} . [\text{white} \mid \text{blue}]$ to a cube C , after β -reduction (that is to say the substitution of each occurrence of white by C in the abstraction body $[\text{white} \mid \text{blue}]$), we obtain $[C \mid \text{blue}]$. Therefore, considering the β -reduction rule, $\lambda \text{white} . [\text{white} \mid \text{blue}]$ defines the function "put a blue cube on the right of another cube".

A β -reduction example

The following abstraction is the same as example (c) where the colours white and green become variables.

$$\lambda \text{white} . \lambda \text{green} . \left[\frac{\text{white} \mid \text{invisible}}{\text{invisible} \mid \text{white}} \mid \frac{\text{invisible} \mid \text{green}}{\text{green} \mid \text{invisible}} \right]$$

If we apply this abstraction to colour blue and then colour red, we have the following β -reductions sequence : (we shall write arguments in **bold** in order to better show the substitution process)

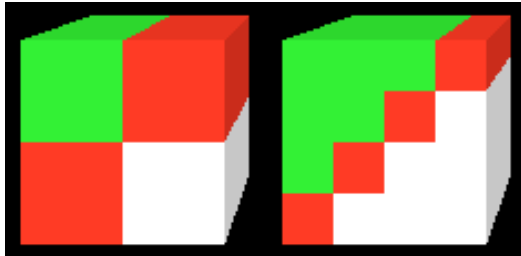
$$\begin{aligned} & \lambda_{white}. \lambda_{green}. \left[\frac{\begin{array}{c|c} white & invisible \\ \hline invisible & white \end{array}}{\begin{array}{c|c} invisible & green \\ \hline green & invisible \end{array}} \right] \mathbf{blue\ red} \\ & \Rightarrow_{\beta} \lambda_{green}. \left[\frac{\begin{array}{c|c} blue & invisible \\ \hline invisible & blue \end{array}}{\begin{array}{c|c} invisible & green \\ \hline green & invisible \end{array}} \right] \mathbf{red} \\ & \Rightarrow_{\beta} \left[\frac{\begin{array}{c|c} blue & invisible \\ \hline invisible & blue \end{array}}{\begin{array}{c|c} invisible & red \\ \hline red & invisible \end{array}} \right] \end{aligned}$$

Construction of a diagonally divided cube

We can now try to construct more complicated objects, for example this diagonally divided green and white cube.



To understand this objects construction, lets look at the following objects A and B:



$$A \equiv \left[\frac{\begin{array}{c|c} green & red \\ \hline red & white \end{array}}{\begin{array}{c|c} green & red \\ \hline red & white \end{array}} \right] \quad B \equiv \left[\frac{\begin{array}{c|c} green & \left[\frac{\begin{array}{c|c} green & red \\ \hline red & white \end{array}} \right] \\ \hline \left[\frac{\begin{array}{c|c} green & red \\ \hline red & white \end{array}} \right] & white \end{array}}{\begin{array}{c|c} green & \left[\frac{\begin{array}{c|c} green & red \\ \hline red & white \end{array}} \right] \\ \hline \left[\frac{\begin{array}{c|c} green & red \\ \hline red & white \end{array}} \right] & white \end{array}} \right]$$

The object B is obtained by replacing the red parts in A by A itself. So we can see that if we could repeat this process indefinitely, the red parts would disappear completely and the result would be a diagonally divided green and white cube.

In other words the problem is to find an object X solution for the following equation :

$$X = \left[\frac{\begin{array}{c|c} green & X \\ \hline X & white \end{array}}{\begin{array}{c|c} green & X \\ \hline X & white \end{array}} \right]$$

We can simply resolve this by considering the object X as the application of an object V on itself.

$$X \equiv (V V)$$

So by rewriting the previous equation in this way we have :

$$(V V) = \left[\frac{\begin{array}{c|c} green & (V V) \\ \hline (V V) & white \end{array}}{\begin{array}{c|c} green & (V V) \\ \hline (V V) & white \end{array}} \right]$$

The definition of V is now simply :

$$V \equiv \lambda_{red}. \left[\frac{\begin{array}{c|c} green & (red red) \\ \hline (red red) & white \end{array}}{\begin{array}{c|c} green & (red red) \\ \hline (red red) & white \end{array}} \right]$$

Therefore we have:

$$\left(\lambda_{red}. \left[\frac{\begin{array}{c|c} green & (red red) \\ \hline (red red) & white \end{array}}{\begin{array}{c|c} green & (red red) \\ \hline (red red) & white \end{array}} \right] V \right) \Rightarrow_{\beta} \left[\frac{\begin{array}{c|c} green & (V V) \\ \hline (V V) & white \end{array}}{\begin{array}{c|c} green & (V V) \\ \hline (V V) & white \end{array}} \right] = (V V)$$

Consequently, our diagonally divided green and white cube X is defined by the following application:

$$X \equiv \left(\lambda_{red}. \left[\frac{\begin{array}{c|c} green & (red red) \\ \hline (red red) & white \end{array}}{\begin{array}{c|c} green & (red red) \\ \hline (red red) & white \end{array}} \right] \lambda_{red}. \left[\frac{\begin{array}{c|c} green & (red red) \\ \hline (red red) & white \end{array}}{\begin{array}{c|c} green & (red red) \\ \hline (red red) & white \end{array}} \right] \right)$$

Note : In our implementation we use Normal Order Reduction. The evaluation stops automatically when an expression is known to be too small to be displayed.

3.2.2. Reduction rules extensions

The previous example used only the application of abstraction so only the β -reduction rule was required. But with our extended syntax, we can also apply basic coloured cubes and constructions to other expressions. Therefore we need to define corresponding reduction rules in order to give functional significance to these elements.

Application of basic coloured cubes

The function of the basic coloured cubes will be to colour their argument. So any object will be whitened by having a white cube applied to it. Consequently, a white cube applied on a red cube will result in a pink cube.

$$(white\ red) \Rightarrow pink$$

When a colour is applied to a construction, it "propagates" following the construction axis.

$$(white [C_1 C_2]) \Rightarrow [(white C_1)(white C_2)]$$

When a colour is applied to an abstraction, it "propagates" into the abstraction body (renaming bounded variables in case of name conflicts)

$$(white \lambda c. C) \Rightarrow \lambda c. (white C)$$

Application of constructions

The principle here is to distribute application on the construction axis. For example :

$$([C_1 C_2] [C_3 C_4]) \Rightarrow [(C_1 C_3)(C_2 C_4)]$$

These new rules are described as follows

$$(C_1 C_2 X) \Rightarrow (C_1 \mathit{left}(X))(C_2 \mathit{right}(X))$$

$$\left(\frac{C_1}{C_2} X \right) \Rightarrow \frac{C_1 \mathit{top}(X)}{C_2 \mathit{bot}(X)}$$

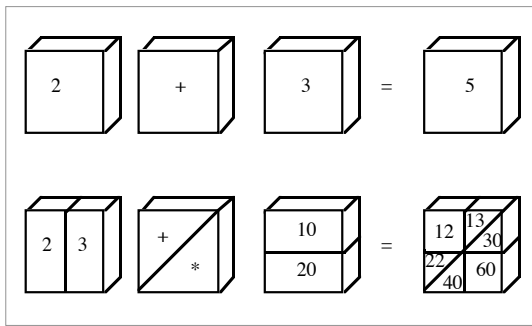
$$(C_1 / C_2 X) \Rightarrow (C_1 \mathit{front}(X)) / (C_2 \mathit{back}(X))$$

Where *left*, *right*, *top*, *bot*, *front*, *back* are cutting algorithms based on the following model (*c* is for a basic colour, *C*, *C₁* and *C₂* are for any cubes):

$$\begin{aligned} \mathit{left}\left(\frac{C_1}{C_2}\right) &\Rightarrow \frac{\mathit{left}(C_1)}{\mathit{left}(C_2)} \\ \mathit{left}\left(C_1/C_2\right) &\Rightarrow \mathit{left}(C_1)/\mathit{left}(C_2) \\ \mathit{left}(C_1C_2) &\Rightarrow C_1 \\ \mathit{left}(c) &\Rightarrow c \\ \mathit{left}(\lambda c.C) &\Rightarrow \lambda c.C \end{aligned}$$

Remark: The two last rules indicate that basic colours and abstractions are not affected by the cutting operation. The consequence is that : $\lambda x.(E \ x) \Leftrightarrow E$

These new reduction rules are very powerful. As the following picture shows, they allow to describe functions which have different behaviors in every points in space (note that the numbers and arithmetic are not part of our language, they are simply here to aid understanding).



3.2.3. An example using extended rules

The intersection and union operations on objects are commonly used in graphics software. To reproduce these operations we must first construct Boolean functions and values. These enables us to "fill" the interior and exterior parts of objects with Boolean values and then using the Boolean functions AND and OR we can perform the intersection and union of objects.

Here is a definition of Boolean values *T*, *F*¹ and *NOT*, *AND*, *OR*. operators .

$$\begin{aligned} T &\equiv \lambda \mathit{blue}.\lambda \mathit{green}.\mathit{blue} \\ F &\equiv \lambda \mathit{blue}.\lambda \mathit{green}.\mathit{green} \\ \mathit{NOT} &\equiv \lambda \mathit{red}.\lambda \mathit{blue}.\lambda \mathit{green}.\mathit{(red \ green \ blue)} \\ \mathit{AND} &\equiv \lambda \mathit{blue}.\lambda \mathit{green}.\mathit{(blue \ green \ blue)} \\ \mathit{OR} &\equiv \lambda \mathit{blue}.\lambda \mathit{green}.\mathit{(blue \ blue \ green)} \end{aligned}$$

Now we can use these to construct a variant of Spenger sponge:

a) We begin by constructing objects A, B and C: (here displayed applied to white and invisible)



The definition of object A is :

$$A \equiv \left[\begin{array}{c} T \\ T|F|F|T \\ T|F|F|T \\ T \end{array} \right]$$

Objects B and C are based on the same model but on complementary axes.

b) Then we construct an object X which takes two arguments and applies them to the intersection of objects A, B and C.

$$X = \lambda \mathit{white}.\lambda \mathit{invisible}.\mathit{((AND A (AND BC)) \mathit{white \ invisible})}$$



c) Finally we can define our sponge by "colouring" X with itself in two levels of depth:

$$\mathit{SPONGE} = \mathit{(((X \ X \ F) \ X \ F) \ \mathit{white \ invisible})}$$



4. A Music Calculus

In this example we shall use a simplified textual music language inspired by Cadenza [Field-Richards 1993].

4.1. The descriptive language

Our descriptive language is composed of notes with pitch, velocity and duration, and two composition operators which allow us to organize notes into sequences and chords. We deliberately omit other kinds of musical information such as tempo and timbre to avoid obscuring the explanation.

The syntax is as follows :

$$\begin{aligned} \mathit{score} &::= \phi \mid \mathit{event} \mid [\mathit{score}_1; \mathit{score}_2] \mid \left[\begin{array}{c} \mathit{score}_1 \\ \mathit{score}_2 \end{array} \right] \\ \mathit{event} &::= r \mid \mathit{note} \mid \mathit{event \ modifier} \end{aligned}$$

¹ Boolean values *T* and *F* must be understood as selectors : *T* returns its first argument, *F* return its second argument.

<i>note</i>	::=	<i>pitch</i> <i>pitch octave</i> <i>note nmodifier</i>
<i>pitch</i>	::=	c d e f g a b
<i>octave</i>	::=	0 1 2 3 4 5 6 7 8 9
<i>modifier</i>	::=	. * t /
<i>nmodifier</i>	::=	+ - > <

This states that a score can be either an empty score, a musical event, a sequence of two scores $[s_1; s_2]$ following the time axis, or a chord of two scores $\left[\begin{matrix} s_1 \\ s_2 \end{matrix} \right]$ superposed on time axis.

A musical event can be either a rest (r) or a note. The event duration is by default a quarter-note. This duration can be divided by 2 (/), divided by 3 (t), multiplied by 1.5 (.) or multiplied by 2 (*).

A note is defined by a pitch value which may be followed by an octave number (octave 3 by default). The note pitch can be modified by adding (+) or subtracting (-) one semitone. In the same way, the default velocity can be accentuated (>) or diminished (<).

Here are some examples of events descriptions :

c, c3	middle C quarter-note
d4>	accentuated D4 quarter-note
f///	F3 32nd-note
a++	A3 double sharp quarter-note
r	quarter-note rest
r**	whole-note rest

In order to simplify the notation, we use associativity rules for the sequence and chord operators.

- Like this we can write $[s_1; s_2; s_3; s_4]$ instead of $[s_1; [s_2; [s_3; s_4]]]$ by using right to left associativity for the sequence operator.

- In the same way we can write $\left[\begin{matrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{matrix} \right]$ instead of

$\left[\begin{matrix} s_1 \\ \left[\begin{matrix} s_2 \\ \left[\begin{matrix} s_3 \\ s_4 \end{matrix} \end{matrix} \right] \end{matrix} \right] \right]$ by using bottom to top associativity for the chord operator.

Here is a simple example :



$[c4; b // ; \left[\begin{matrix} d4 // \\ f4 // \end{matrix} \right]; r; r^*]$

4.2. The programming language

To transform the descriptive language into a music calculus, we first introduce abstraction and application, and then extend the reduction rules.

Here follows an example to explain the role of abstraction and application. Using our descriptive language we can explicitly describe repetition of notes, like : $[c;c]$, $[f3//;f3//]$ or $[b2+;b2+]$, but we can't describe the underlying concept of repetition : *the sequence of two identical objects*. Abstraction will allow us to describe such concepts.

We can generalize a particular repetition, for example $[c;c]$, by making the note c become *variable*. In this case the abstraction is written $\lambda c.[c;c]$, where the λc part declares that c is an abstract note in the abstraction body $[c;c]$ (In other terms c is a bounded variable, this will be written in *italics* to distinguish it from a real note).

If we now apply this abstraction to note $a4$, written $(\lambda c.[c;c] a4)$, after β -reduction (that is substitution of each c occurrence with $a4$) we obtain the sequence $[a4;a4]$. So the abstraction $\lambda c.[c;c]$ really defines an operation that repeats the same object two times.

4.2.1. Syntax extension

The syntax of our musical calculus is described as follows :

<i>score</i>	::=	ϕ <i>event</i> $[score_1; score_2]$ $\left[\begin{matrix} score_1 \\ score_2 \end{matrix} \right]$
		$\lambda event.score$ $(score_1 score_2)$

It extends the descriptive language syntax with two new forms : $\lambda event.score$ which is for abstraction, and $(score_1 score_2)$ for application .

Again in order to simplify the notation, we assume associativity rules to write the applications : like this we can write $(s_1 s_2 s_3 s_4)$ instead of $((s_1 s_2) s_3) s_4$ by using left to right associativity for the application.

β -reduction examples

Here are some examples of applications and their sequence of corresponding β -reductions. (we shall write arguments in **bold** in order to better show the substitution process).

a) 3 time repetition :

$$\left(\lambda c.[c;c;c] \left[\begin{matrix} \mathbf{c4} \\ \mathbf{e4} \\ \mathbf{g4} \\ \mathbf{b5} \end{matrix} \right] \right) \Rightarrow_{\beta} \left[\begin{matrix} \mathbf{c4} \\ \mathbf{e4} \\ \mathbf{g4} \\ \mathbf{b5} \end{matrix} \right]; \left[\begin{matrix} \mathbf{c4} \\ \mathbf{e4} \\ \mathbf{g4} \\ \mathbf{b5} \end{matrix} \right]; \left[\begin{matrix} \mathbf{c4} \\ \mathbf{e4} \\ \mathbf{g4} \\ \mathbf{b5} \end{matrix} \right]$$

b) ABBA form:

$$\begin{aligned} & (\lambda a.\lambda b.[a;b;b;a] \mathbf{b5} [\mathbf{c4};\mathbf{e4}]) \\ & \Rightarrow_{\beta} (\lambda b.[\mathbf{b5};b;b;\mathbf{b5}] [\mathbf{c4};\mathbf{e4}]) \\ & \Rightarrow_{\beta} [\mathbf{b5};[\mathbf{c4};\mathbf{e4}];[\mathbf{c4};\mathbf{e4}];\mathbf{b5}] \end{aligned}$$

c) canon form :

$$\left(\lambda c. \left[\begin{matrix} c \\ r;c \end{matrix} \right] [\mathbf{c4};\mathbf{e4};\mathbf{g4};\mathbf{c3}] \right) \Rightarrow_{\beta} \left[\begin{matrix} \mathbf{c4};\mathbf{e4};\mathbf{g4};\mathbf{c3} \\ r;\mathbf{c4};\mathbf{e4};\mathbf{g4};\mathbf{c3} \end{matrix} \right]$$

Infinite sequences

As with Lambda Calculus we can't directly define recursive functions, however they can be easily sim-

ulated. To understand this principle let's look at the following abstraction :

$$V = \lambda f.(f f)$$

When we apply V on V, after β -reduction :

$$(V V) = (\lambda f.(f f) V) \Rightarrow_{\beta} (V V)$$

This is V applied on V again, consequently the evaluation on such a form never terminates (i.e. it doesn't have a normal form). We shall use this principle to define infinite sequences.

a) An infinite sequence

Let's start with an infinite sequence of c4 notes:

$$X = [c4;c4;c4;...]$$

A recursive definition of X would be : $X = [c4;X]$. To obtain this result we shall modify the definition of $V = \lambda f.(f f)$ in :

$$V@ = \lambda f.[c4;(f f)]$$

By applying V' on V' we have :

$$\begin{aligned} (V@ V@) &= (\lambda f.[c4;(f f)] V@) \\ &\Rightarrow_{\beta} [c4;(V@ V@)] \\ &\Rightarrow_{\beta^*} [c4;c4;c4;...] \end{aligned}$$

So, the required definition is:

$$X = (\lambda f.[c4;(f f)] \lambda f.[c4;(f f)])$$

b) A looping function

We can now generalize the preceding object X by making the note c4 become variable :

$$Loop = \lambda c4.(\lambda f.[c4;(f f)] \lambda f.[c4;(f f)])$$

So if we apply Loop to the [a2;a2//] sequence we obtain :

$$(Loop [a2;a2//]) \Rightarrow_{\beta^*} [[a2;a2//];[a2;a2//];[a2;a2//];...]$$

c) Infinite alternation

By slightly modifying the preceding definition, we can define an infinite alternation of two objects :

$$Alt = \lambda c.\lambda d.(\lambda f.[c;d;(f f)] \lambda f.[c;d;(f f)])$$

If we apply Alt on c3 and c3+ we obtain :

$$(Alt c3 c3+) \Rightarrow_{\beta^*} [c3;c3+;c3;c3+;c3;c3+;...]$$

Notice that this result is not equivalent to :

$$(Loop [c3;c3+]) \Rightarrow_{\beta^*} [[c3;c3+];[c3;c3+];[c3;c3+];...]$$

(although the sound result would be the same)

4.2.2. Reduction rules extension

The preceding examples only used the usual β -reduction rule. By extending reduction rules we shall give functional capabilities to each language construction.

Let's take for example the application of sequence [E;F;G] on sequence [H;I;J]. Following the intuitive concepts of sequence we consider [E;F;G] as a time ordering of functions. This means that when applied to a sequence the time ordering is maintained, as we show in the following example :

$$([E;F;G] [H;I;J]) \Rightarrow [(E H);(F I);(G J)]$$

In the same way we can consider $\left[\frac{E}{F}\right]$ as time superimposed functions, therefore :

$$\left(\left[\frac{E}{F}\right] H\right) \Rightarrow \left[\frac{(E H)}{(F H)}\right]$$

With these two rules, we are able to time-organize functions in the same way as other musical objects.

We must also give functional significance to basic objects like notes and rests. The principle we choose is to consider a note as a function which transforms its argument according to the differences between the note and a reference note (the c3 quarter-note in our case). So if we represent a note as a tuple <pitch, velocity, duration> then the application of a note to another one $(\langle h_1, i_1, d_1 \rangle \langle h_2, i_2, d_2 \rangle)$ gives a new note :

$$\Rightarrow \langle h_2 + h_1 - h_{ref}, i_2 + i_1 - i_{ref}, d_2 * d_1 / d_{ref} \rangle$$

To summarize, the new reduction rules are as follows (we give only the principal rules, E, F, G, H are for any terms, N is for a note) :

a) Sequence application:

$$([E;F] [G;H]) \Rightarrow [(E G);(F H)]$$

$$\left([E;F] \left[\frac{G}{H}\right]\right) \Rightarrow \left[\frac{([E;F] G)}{([E;F] H)}\right]$$

$$([E;F] N) \Rightarrow (E N)$$

b) Chord application :

$$\left(\left[\frac{E}{F}\right] G\right) \Rightarrow \left[\frac{(E G)}{(F G)}\right]$$

c) Note application :

$$(N [G;H]) \Rightarrow [(N G);(N H)]$$

$$\left(N \left[\frac{G}{H}\right]\right) \Rightarrow \left[\frac{(N G)}{(N H)}\right]$$

$$(N_1 N_2) \Rightarrow \langle h_2 + h_1 - h_{ref}, i_2 + i_1 - i_{ref}, d_2 * d_1 / d_{ref} \rangle$$

d) Application of ϕ :

$$(\phi E) \Rightarrow \phi$$

Simples examples

The following examples use the reduction rules for note application :

a) One semi-tone transposition

$$(c3 + [e4;f3;g4]) \Rightarrow [f4;f3+;g4 +]$$

b) Duration division by 4

$$(c3 // [e4;f3;g4]) \Rightarrow [e4 // ;f3 // ;g4 //]$$

c) Transposition, expansion and accentuation

$$(e3 > * [e4;f3;g4]) \Rightarrow [g4 + > *; a3 > *; b4 > *]$$

Treatment of sequences

The following examples take advantage of the new rule for the application of sequences. Indeed, in order to treat each element of a sequence, we now just have to describe a sequence of treatments and apply it to a sequence.

a) *Repetition of each sequence element*

We want here to apply the function $\lambda c.[c;c]$ on each element of a sequence. For a specific sequence $[e;f;g]$ we can write :

$$([\lambda c.[c;c];\lambda c.[c;c];\lambda c.[c;c]] [e;f;g]) \Rightarrow [[e;e];[f;f];[g;g]]$$

The problem is now how to make it work for sequences of any length. Note that our reduction rules can treat expression as follows :

$$([A;B;C;D;\dots] [U;V;W]) \Rightarrow [(A U);(B V);(C W)]$$

So the solution consist in creating an infinite sequence of $\lambda c.[c;c]$ functions (using the preceding *Loop* function) and applying it on the argument sequence :

$$(Loop \lambda c.[c;c] [e;f;g]) \Rightarrow [[e;e];[f;f];[g;g]]$$

Here the treatment sequence could be more complex. For example :

$$(Alt \lambda c.[c;c] \lambda c.[c;c;c] [e;f;g;a]) \Rightarrow [[e;e];[f;f;f];[g;g];[a;a;a]]$$

b) *Sequences interlacing*

Suppose we want a function *Inter* such that :

$$(Inter [c;d;c] [e;f;g]) \Rightarrow [[c;e];[d;f];[c;g]]$$

Following the previous example the definition we need is :

$$Inter = (Loop \lambda a.\lambda b.[a;b])$$

Indeed we have :

$$\begin{aligned} (Loop \lambda a.\lambda b.[a;b] [c;d;c] [e;f;g]) \\ \Rightarrow ([\lambda a.\lambda b.[a;b]; \lambda a.\lambda b.[a;b]; \dots] [c;d;c] [e;f;g]) \\ \Rightarrow ([\lambda b.[c;b]; \lambda b.[d;b]; \lambda b.[c;b]] [e;f;g]) \\ \Rightarrow [[c;e];[d;f];[c;g]] \end{aligned}$$

c) *Sequence multiplication*


Here we want to define a *Mult* function that takes two sequences as arguments and applies each note of the first sequence on the entire second sequence :

$$(Mult [a;b;c] S) \Rightarrow [(a S);(b S);(c S)]$$

The definition we need is simple :

$$Mult = \lambda a.\lambda b.(Loop \lambda c.(c b) a)$$

Here is a example of multiplication and its result :

$$Mult [c3;e3+;c3-] \left[e/; \begin{matrix} c// \\ e// \\ g// \end{matrix}; a// \right]$$


By multiplying the same sequence with itself several times we can define a sort of "fractal" structure.

$$Fractalize = \lambda c.(Mult c (Mult c (Mult c c)))$$

5. A Visual Music Calculus

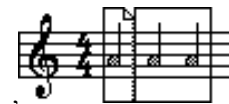
Here we introduce a visual representation for our musical calculus. Here our descriptive language can be graphically represented using common musical notation, then we only have to introduce a graphical representation for abstraction and application to define a purely graphical musical calculus. We shall also maintain the same reductions rules.

5.1. Abstraction

Abstractions are represented in a rectangle. The abstract note is declared first and separated from the abstraction body by a vertical line. Abstract notes are coloured gray to be distinguishable from real notes.

Here are some abstractions representations.

a) *Two times repetition*



$$\lambda a3.[a3; a3]$$

b) *ABBA form*



$$\lambda a.\lambda b.[a; b; b; a]$$

c) *Canon form*



$$\lambda a. \left[\begin{matrix} a \\ r; a \end{matrix} \right]$$

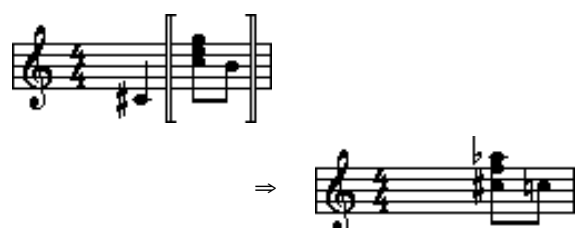
5.1.1. Application

Compared to the textual representation, here the application of a to b will be written using brackets a[b].

Below are some application examples and their corresponding evaluation.

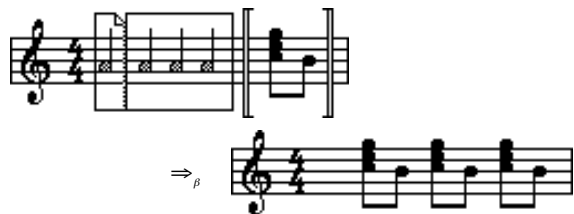
a) *One semi-tone transposition*

All notes are in effect transformation functions who's transposition value is the difference between their pitch and the reference pitch c3. Therefore by applying c3+ to an argument we transpose it by one semi-tone.



b) *Three time repetition*

Application of the "repeat three times" operation on a sequence.



c) *ABBA form*

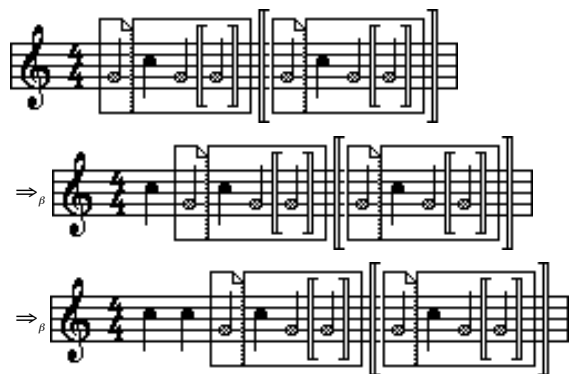
Application of the $\lambda a. \lambda b. [a; b; b; a]$ abstraction on two sequences.



d) *Infinite sequence*

This example is the visual translation of the infinite sequence of c4 defined in 1.2.1.2. :

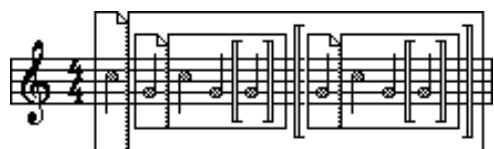
$$X = (\lambda f. [c4; (f f)] \lambda f. [c4; (f f)])$$



e) *Looping function*

The loop function is defined by making the note c4 (from the preceding example) become variable.

$$Loop = \lambda c4. (\lambda f. [c4; (f f)] \lambda f. [c4; (f f)])$$



6. Conclusion

We believe that the functional programming model is of great interest for music languages. Since 1984 R. Dannenberg proposed several functional music languages that demonstrate the advantages of features like lazy evaluation and high-order func-

tions [Dannenberg 1984, 1989, 1991]. The functional approach is also central to the solution proposed by P. Desain and H. Honing for the representation of control functions [Desain and Honing 1991].

By returning to the root of functional programming, Lambda calculus, we are able to propose a new approach in designing music programming languages. The central idea of our approach is to start from the music data structures and to introduce Lambda Calculus abstraction and application. Abstract music objects lead to a "natural" expression of music functions and operations. This has also been proposed by M. Balaban in a recent paper [Balaban 1994].

A very interesting consequence of our approach is that, being music objects, functions can be composed, processed and represented in the same way as real music objects. Therefore, the programming activity is naturally in keeping with the composition activity which is more familiar to the user.

References

- [Balaban 1994], M. Balaban, "Introducing Formal Processing into Music - The Music Structure Approach", *Technical Report FC-94-07*, Dept. of Math. and Comp. Science, Ben-Gurion University of the Negev, 1994.
- [Barendregt 1984] H. P. Barendregt, *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, Amsterdam, 1984.
- [Church 1941], A. Church, *The Calculi of Lambda Conversion*, Princeton University Press, Princeton, N.J., 1941.
- [Dannenberg 1984], R. B. Dannenberg, "Artic: A Functional Language for Real-Time Control", in *1984 ACM Symposium on LISP and Functional Programming*, ACM, New-York, 1984.
- [Dannenberg 1989], R. B. Dannenberg, "The Canon Score Language", *Computer Music Journal*, 13 (1), pp. 47-56, 1989.
- [Dannenberg et al. 1991], R. B. Dannenberg, C. L. Fraley and P. Velikonja, "Fugue : A Functional Language for Sound Synthesis", *Computer*, 24 (7), pp. 36-42, 1991.
- [Desain and Honing 1991], P. Desain and H. Honing, "Time Functions Function Best as Functions of Multiple Times", *Computer Music Journal*, 16 (2), pp. 17-34, 1991.
- [Field-Richards 1993] H. S. Field-Richards, "Cadenza : A Music Description Language", *Computer Music Journal*, 17 (4), pp. 60-72, 1993.
- [Hughes 1989] J. Hughes, Why Functional Programming Matters, *The computer Journal* 32 (2), pp. 98-107, 1989.

Acknowledgment

This research was sponsored by the music research department of French Ministry of Culture.