Lambda Calculus and Music Calculi
Yann Orlarey, Dominique Fober, Stéphane Letz and Mark Bilton

GRAME
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 1999] 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 on 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 constructs.

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 3D graphic calculus based on colored cubes.

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

```plaintext
cube 2|ω
  color
  [cube]
  [cube/]
  [cube,]
  [cube,]
color 2|ω
  white
  red
  blue
  green
  invisible
```

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:

ICMC Proceedings 1994
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:

```
cube ::= cube 1 [cube, cube] 1 cube 1 cube 1 color 1 cube 1 (cube, cube)

color ::= white | red | blue | green | invisible | ...
```

This adds two new terms to the previous one, `color:cube` which represent an abstraction and `(cube, cube)`, which represent the application of `cube` to `cube`.

In order to simplify the notation we will write `(c, c, c, c)` instead of `(((c, c, c, c)) c, c))` 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:white:blue` is a generalization of the cube `[white,blue]` obtained by making the white part variable. The `lambda:` part declares that white is a bounded variable in the abstraction body `[white,blue]`.

If we apply `lambda:white: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 `[white,blue]`), we obtain `[C,blue]`. Therefore, considering the β-reduction rule, `lambda:white: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:lambda:green:
  white | invisible
  invisible | white
  invisible | green
  green | invisible
```

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

```
lambda:lambda:green:
  white | invisible
  invisible | white
  invisible | green
  green | invisible
```

```
  invisible | green
  green | invisible
```

```
  blue red
```
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:

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:

\[ X = \text{green} \times \text{white} \]

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

\[ X = (V \times V) \]

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

\[ (V \times V) = \text{green} \times \text{red} \times \text{white} \times \text{white} \]

The definition of V is now simply:

\[ V = \text{green} \times \text{red} \times \text{white} \]

Therefore we have:

\[ \text{green} \times \text{red} \times \text{white} \]

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

\[ X = \text{green} \times \text{red} \times \text{white} \]

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:

\( \text{white red} \times \text{pink} \)

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

\( \text{white} \times [c; C] \rightarrow [\text{white} C; \text{white} C] \)

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

\( \text{white let} C \rightarrow \text{let} (\text{white} C) \)

Application of constructions

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

\[ [(c; C), (c; C)] \rightarrow [c, C; C, C] \]

These new rules are described as follows:

\( (c; C) \times (c; C) \rightarrow (c, \text{left} (x); C, \text{right} (x)) \)

\( \text{let} x \rightarrow C, \text{left} (x) \times (x; C) \)

\( \text{let} x \rightarrow C, \text{right} (x) \times (x; C) \)

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

\( \text{left} (c) \rightarrow \text{left} (c) \)

\( \text{right} (c) \rightarrow \text{right} (c) \)

ICMC Proceedings 1994

Music Languages
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. Those enable us to "fill" the interior and exterior parts of objects with Boolean values and then union the Boolean functions AND and OR we can perform the intersection and union of objects.

Here is a definition of Boolean values \( T \) and \( F \) and NOT, AND, OR, operators.

- \( T = \) blue.green.blue
- \( F = \) blue.green.green
- \( \text{NOT} = \) red.blue.green.red.green.blue
- \( \text{AND} = \) blue.green.(blue.green.blue)
- \( \text{OR} = \) blue.green.(blue.blue.green)

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

a) We begin by constructing objects A, B, and C:

(here displayed applied to white and invisible)

The definition of object A is:

\[ \text{score} \equiv \phi \text{ l event } \left[ \text{score}, \text{score2} \right] \text{ score2 } \]

\[ \text{event} \equiv \text{ l note } \text{ l event modifier} \]

\[ \text{note} \equiv \text{ pitch } \text{ l pitch octave l note modifier} \]

\[ \text{pitch} \equiv \text{ c f d e f g a b} \]

\[ \text{octave} \equiv 01235145679 \]

\[ \text{modifier} \equiv \text{ l l l l l l} \]

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

\[ X = \text{white. Invisibles} \left[ \left( \text{AND} \left( \text{AND} \left( \text{A, B, C} \right) \right) \right) \right] \text{ white instant} \]

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

\[ \text{SPONGE} = \left( \left( X \ X \ F \ X \ F \right) \right) \text{ white invisible} \]

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 timing to avoid obscuring the explanation. The syntax is as follows:

- **score**: \( \text{score} \equiv \phi \text{ l event } \left[ \text{score}, \text{score2} \right] \text{ score2 } \)
- **event**: \( \text{event} \equiv \text{ l note } \text{ l event modifier} \)
- **note**: \( \text{note} \equiv \text{ pitch } \text{ l pitch octave l note modifier} \)
- **pitch**: \( \text{pitch} \equiv \text{ c f d e f g a b} \)
- **octave**: \( \text{octave} \equiv 01235145679 \)
- **modifier**: \( \text{modifier} \equiv \text{ l l l l l l} \)

---

Music Languages

246

ICMC Proceedings 1994
This states that a score can be either as empty score, a musical event, a sequence of two scores \( \{s_1, s_2\} \) following the time axis, or a chord of two scores \( \{s_1, s_2\} \) superposed on time axis.

A musical event can be either a rest \((\cdot)\) or a note. The event duration is by default a quarter-note. This duration can be divided by 2 \((\div)\), divided by 3 \((\div)\), multiplied by 1.5 \((\times)\), or multiplied by 2 \((\times)\).

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 \(\times\) of one semitone. In the same way, the default velocity can be accentuated \(\times\) or diminished \(\times\).

Here are some examples of events descriptions:

- \(\circ\)3: middle C quarter-note
- \(\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wedge\wage
a) An infinite sequence
Let's start with an infinite sequence of 4, notes:
\[ X = [a,c,e,g,\ldots] \]
A recursive definition of \( X \) would be: \( X = \{X, A\} \). To obtain this result we shall modify the definition of \( V \) at \([b/c, f/f] \) in:
\[ V = \{[b/c, f/f] \} \]
By applying \( V \) on \( V \) we have:
\[ (V \circ V) = \{[b/c, f/f], V \} \]
\[ = \{[b/c, f/f], [b/c, f/f], \ldots \} \]
So, the required definition is:
\[ X = \{[b/c, f/f], [b/c, f/f], \ldots \} \]
A looping function
We can now generalize the preceding object \( X \) by making the note \( x \) become variable:
\[ \text{Loop} = \lambda x.\{[b/c, f/f], [b/c, f/f], \ldots \} \]
If we apply \( \text{Loop} \) to the \([a,2,2,2,\ldots] \) sequence we obtain:
\[ \text{Loop}([a,2,2,2,\ldots]) = \{[a,2,2,2,2,2,\ldots], [a,2,2,2,2,2,\ldots], \ldots \} \]
\[ \text{c) Infinite alternation} \]
By slightly modifying the preceding definition, we can define an infinite alternation of two objects:
\[ \text{Alternation} = \lambda x.\{[b/c, f/f], [b/c, f/f], \ldots \} \]
If we apply \( \text{Alternation} \) to \([c,3,3,3,\ldots] \) and \([a,3,3,3,\ldots] \) we obtain:
\[ \text{Alternation}([c,3,3,3,\ldots], [a,3,3,3,\ldots]) = \{[c,3,3,3,3,3,\ldots], [a,3,3,3,3,3,\ldots], \ldots \} \]
Notice that this result is not equivalent to:
\[ \text{Loop}([c,3,3,3,\ldots]) = \{[c,3,3,3,3,3,3,3,3,\ldots], \ldots \} \]
(although the sound result would be the same)

4.2.2. Reduction rules extension
The preceding examples used the usual \( \beta \)-reduction rule. By extending reduction rules we shall give functional capabilities to each language construction.
Let's take for example the application of sequence (SEQ) on sequence (HELLO). Following the intuitive concepts of sequence we consider \([HELLO] \) 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:
\[ \text{SEQ}(HELLO) = [HELLO] \]
In the same way we can consider \([F] \) as time superimposed functions, therefore:
\[ \text{SEQ}(F) = [F] \]
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 \((\text{pitch}, \text{velocity}, \text{duration}) \) gives a new note:
\[ \text{note} = (\text{pitch} + \text{velocity}, \text{velocity}, \text{duration}) \]
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):

\[ a) \quad \text{Sequence application:} \]
\[ [E, F, G] \rightarrow [E, G, (F \circ G)] \]
\[ [E, F, G] \rightarrow [E, G, \text{F} \circ \text{G}] \]
\[ [E, F, G] \rightarrow [E, G, \text{F} \circ \text{G}] \]
\[ b) \quad \text{Chord application:} \]
\[ [E, F, G] \rightarrow [E, G, \text{F} \circ \text{G}] \]
\[ c) \quad \text{Note application:} \]
\[ (N) \rightarrow [N, G, (N \circ G)] \]
\[ N \rightarrow [N, G, (N \circ G)] \]
\[ (N, N) \rightarrow (N, N) \]
\[ d) \quad \text{Application of a note:} \]
\[ \text{seq} \rightarrow \text{seq} \]

Simplex examples
The following examples use the reduction rules for note application:

\[ a) \quad \text{One semi-tones transposition} \]
\[ [c3] \rightarrow [c3 + 1/243] \]
\[ b) \quad \text{Duration division by 4} \]
\[ [c3] \rightarrow [c3/4] \]
\[ c) \quad \text{Transposition, expansion and accentuation} \]

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.

\[ a) \quad \text{Repetition of each sequence element} \]
We want here to apply the function \( \text{seq} \rightarrow \text{seq} \) on each element of a sequence. For a specific sequence \([c3, c3, c3, c3, c3] \), we can write:
\[ [\text{seq} \circ [c3, c3, c3, c3, c3] \rightarrow [c3, c3, c3, c3, c3, c3, c3, c3, c3] \]

Music Languages
The problem is now how to make it work for sequences of any length. Note that our reduction rules can treat expression as follows:
\[
(A \circ B \circ C \circ D \ldots) \circ (E \circ F \circ G \circ H \circ I) = [(A \circ B \circ C \circ D \ldots) \circ (E \circ F \circ G \circ H \circ I)]
\]
So the solution consist in creating an infinite sequence of \( \text{lc}(a) \) functions (using the preceding two sequences) and applying it on the argument sequence:
\[
(\text{loop } \text{lc}(a) \text{lc}(b) \text{lc}(c) \text{lc}(d) \ldots) \text{lc}(e) \text{lc}(f) \text{lc}(g) \text{lc}(h) \text{lc}(i) \ldots) \text{lc}(a) \text{lc}(b) \text{lc}(c) \text{lc}(d) \ldots)
\]
Here the treatment sequence could be more complex. For example:
\[
(\text{loop } \text{lc}(a) \text{lc}(b) \text{lc}(c) \text{lc}(d) \ldots) \text{lc}(e) \text{lc}(f) \text{lc}(g) \text{lc}(h) \text{lc}(i) \ldots) \text{lc}(a) \text{lc}(b) \text{lc}(c) \text{lc}(d) \ldots)
\]
\[
\text{lc}(e) \text{lc}(f) \text{lc}(g) \text{lc}(h) \text{lc}(i) \ldots) \text{lc}(a) \text{lc}(b) \text{lc}(c) \text{lc}(d) \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
\]
\[
(a \circ b \circ c \circ d \ldots)
c) ABBA form
Application of the \(\lambda s.h[c; b; a; v]\) abstraction on two sequences.

\[
\begin{align*}
&\downarrow \\
&\downarrow \\
&\downarrow
\end{align*}
\]

\[
\begin{align*}
&\downarrow \\
&\downarrow \\
&\downarrow
\end{align*}
\]

\[
\begin{align*}
&\downarrow \\
&\downarrow \\
&\downarrow
\end{align*}
\]

\[
\begin{align*}
&\downarrow \\
&\downarrow \\
&\downarrow
\end{align*}
\]

\[
\begin{align*}
&\downarrow \\
&\downarrow \\
&\downarrow
\end{align*}
\]

d) Infinite sequence
This example is the visual translation of the infinite sequence of c4 defined in 1.2.1.2.;

\[
X = ([v.[c4s/f]] [v.[c4s/f]])
\]

\[
\begin{align*}
&\downarrow \\
&\downarrow \\
&\downarrow
\end{align*}
\]

\[
\begin{align*}
&\downarrow \\
&\downarrow \\
&\downarrow
\end{align*}
\]

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

\[
\text{Loop } \equiv \lambda s.t.[s[v.[c4s/t]] [s[v.[c4s/t]]]
\]

\[
\begin{align*}
&\downarrow \\
&\downarrow \\
&\downarrow
\end{align*}
\]

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 functions [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


[Church 1941], A. Church, The Calculi of Lambda Conversion, Princeton University Press, Princeton, N.J., 1941.


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