The Role of Lambda-Abstraction in Elody

Stéphane Letz, Dominique Fober, Yann Orlarey

The Role of Lambda-Abstraction in Elody

Stéphane Letz, Dominique Fober, Yann Orlarey
{letz, fober, orlarey}@rd.grame.fr
Grame, 9 rue du Garet, BP 1185,
69202, Lyon Cedex 01, France

Abstract *: The Elody music composition environment proposes lambda-abstraction on musical structures as a fundamental mechanism to represent user-defined musical concepts and compositional processes. The user can define new musical concepts either on top of concrete musical objects by generalizing them via an abstraction operation, or by composing and transforming previously defined abstractions. As the paper will show through several examples, this approach leads to a quite natural formalization as well as a convenient* active notation *for many musical notions and compositional techniques.*

1. Introduction

Elody is a music composition environment based on a visual functional programming language, a direct manipulation user interface and Internet facilities [OFL 97]. One of its most singular aspect is that Elody doesn't have a programming language as such, separate from the music language it allows to manipulate. Conventional music composition environments are generally organized around two languages : a programming language (for instance Lisp or Smalltalk, or even a specific one) enhanced with appropriate libraries to algorithmically generate and process musical objects, and a music-data language used to describe and implement the musical objects. There usually is a strong conceptual separation between these two languages. For example musical objects will have a time dimension and structuring concepts like sequence and mix that are not applicable to user programs, while user programs may benefit from powerful abstraction mechanisms not applicable to musical objects.

This "schizophrenic" design has several disadvantages. Not only the user will have to learn a whole set of new syntactic and semantic aspects unrelated to its musical purpose, but it also implies a severe lack of modularity, reusability and expressiveness. For example, as we will see later, it is very convenient and powerful to be able to « reuse » the concept of score to time-organize programs in order to create new programs. This is only possible if programs or functions can be seen as musical objects (in order to be placed in the score) and if musical objects (here the resulting score) can be seen as a program and applied to some arguments.

Modern programming languages, especially functional languages, consider functions as first-class citizens. Functions, like ordinary data, can be passed as argument and returned as result of other functions. This allows the expression of very powerful programming concepts, like high-order functions. But the reverse is generally not true. High-order data structures, i.e. data structures containing functions (like our example of a score of functions), can't be used as ordinary functions and applied to arguments. To our best knowledge this is only possible in some theoretical extensions of the Lambda-Calculus like György Révész "applicative lists" [Révész 88][Durfee 97].

The key idea of Elody is to avoid the separation between the music language and the programming language. Elody programming language is *grounded* into the music domain. It can be seen as an *active music notation*, a slightly extend music language with programming capabilities. This approach as several advantages for the user. The number of new concepts to learn, external to the music domain, is very limited. Because user-defined programs are musical objects, the same means of visual representation and edition can be reused, leading to a visual programming language almost for free. High-order musical objects, for example scores of functions, can be easily defined. In particular user-defined compositional processes can be applied to other functions to algorithmically generate high-order scores. Moreover since the same language is used for programs and musical objects, it is easy to provide a (limited) form of *programming by example*.

The rest of the paper is organized as follows. In the next section we will introduce the music language used by Elody, in particular abstraction on musical structures and high-order scores. Then, to illustrate the role of these concepts, we will present two models, one for a Jazz score and another for Steve Reich's *Music for Pieces of Wood*.

2. The Programming Language

2.1 Basic elements and constructors

The basic elements of the language are *numbers*, *notes* and *rests* with several attributes : *duration*, *pitch*, *color*, *intensity*, etc. The *color* attribute is used to distinguish related notes in the same object, for example the *red* notes of a sequence. These elements can be assembled to form more complex objects (that in turn can be assembled to form even more complex objects, etc.) using *constructors*.

The two main constructors are *Seq* and *Mix*. They allow to time-organize musical objects in sequence or in parallel (figure 1). Other time constructors are *Begin* to take the beginning of an object according to the duration of another

object, *Rem* which allows to remove the beginning of an object, and *Xpd* to time-scale an object. The remaining constructors, like *Tr* (transpose), operate on the other musical attributes.

An Elody object is internally represented as a tree (actually a DAG because subtrees can be shared). The tree's nodes are the constructors used to build the object, and the leaves the basic elements. The operation a constructor represents is only computed at the rendering stage (when the object is played or displayed) and never modifies the tree. This means that an object always keeps its history. If you take, for example, the second note of a sequence, the note you obtain keeps the fact that it is the second note of that sequence. This is particularly important for the abstraction process. It allows, later on, to *make variable* in that note the original sequence in order to express concepts such as *take the second note of a sequence*, or better, to express the concept of *degree* if the sequence was from a diatonic scale.

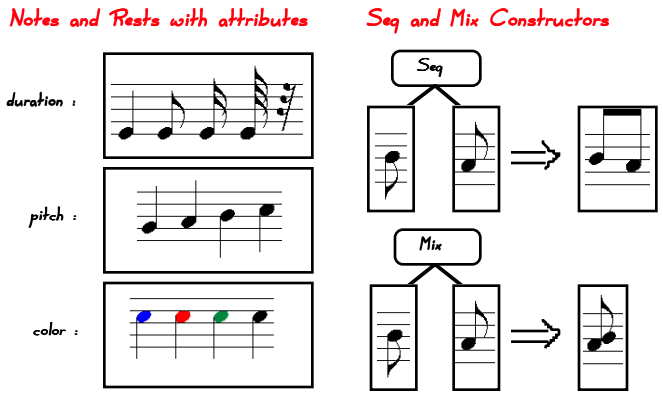


Figure 1 *: The basic elements and constructors of the language*

To textually represent Elody objects we will use a syntax loosely based on GUIDO music notation [HHFK 98]. An informal description of the syntax is given by the following rules (`M` and `N` are meta-variables representing any Elody expression) :

1. Whole notes are represented by capital letters : `C`, `D`, `E`, … with optional accidentals and octave specifications : `C#3`, `D&`,…. Rests are represented by the underscore sign (`_`).
2. Sequence constructions are enclosed within square brackets : `[C,D]`, while mix or chords are enclosed within curly brackets : `{C,E}`. In order to simplify the notation we will write `[C,D,E,F]` as a shortcut for `[C,[D,[E,F]]]` and `{C,E,G,B}` as a shortcut for `{C,{E,{G,B}}}` .
3. Chromatic transpositions are expressed with the (`+`) and (`-`) signs, time stretching operations are represented by the (`*`) and (`/`) signs, and accentuation with the (`>`) and (`<`) signs. For instance a C quarter note transposed by 7 semi tones and accentuated by 10 will be written this way : `C/4+7>10`.
4. The expression : `λx:M.N` represents an abstraction where `x` is the formal parameter, `M` is an Elody expression `x` stands for, and `N` is the body of the abstraction, an Elody expression where `x` may occur free.
5. The application of `M` to `N` is notated : `M N`

Thus the following expressions are equivalent :

```
C/4+7  = C+7/4  = G*1/4
[C,_,E,F]/8  = [C/8,_/8,E/8,F/8]
{C,E,G}+2  = {D,F#,A}
λx:M.{x, x+4, x+7, x+11}* 2 = λx:M *2.{x, x+4, x+7, x+11}
λx:M.{x, x+4, x+7, x+11} N = {N, N+4, N+7, N+11}
```

2.2 Abstractions

The word *abstraction* has two meanings depending of how we consider it, either as a process or as an entity. As a process, abstraction is the operation through which we can form a concept by extracting (*abs-trahere*) the essential parts (and neglecting the inessential details) of something. As an entity, an abstraction is the result of the abstraction process. It can represent a concept, a class of objects, a predicate, a function, etc. . Programming languages provide several means for the user to describe various kinds of abstractions, for instance functions, classes, types, etc. Moreover, abstractions play an essential part in the pure, type free Lambda-Calculus [Church 41] [Barendregt 84]. Numbers for example are not primitive objects in the lambda-calculus, but they can be simulated by abstractions that *convinc-*

ingly behave (via application and reduction rules) like numbers. The same is true for recursion, booleans and all the usual concepts of programming languages. They are not part of the lambda-calculus but can be simulated by abstractions. In fact, despite its very minimalist approach, the lambda-calculus is strong enough to represent any mechanically computable function.

The Elody programming language is essentially a music language extended with lambda-calculus [OFLB 94]. Elody abstractions can be defined informally as *generalized musical structures with variable parts*. These abstractions can be used to describe musical operations and applied to arguments to produce a result. But because Elody abstractions are also musical objects, they can also be used like regular ones. Abstractions can be listened to, placed in a score, mixed, transposed, stretched, etc. as simple notes. Moreover Elody provides a simple (but quite convenient) mechanism to help the user in the abstraction process. The system can compute abstractions by *making variable* any element used in the composition of an object.

Figure 2 gives an example of musical abstraction. Suppose we want to describe the concept of *Major Seventh* chord.

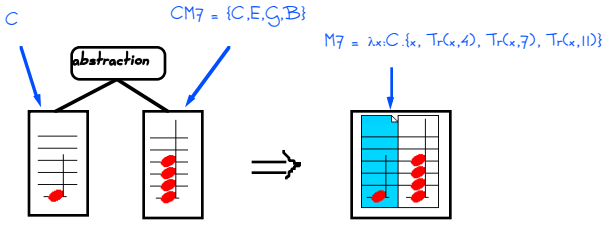


Figure 2 *: The Major Seventh abstraction*

We can start from a concrete example, a *prototype* of M7 chord, let say CM7 : `{C, E, G, B}`. Then we can ask the system to generalize this chord by *making variable* the tonic `C`. The result of this abstraction operation computed by the system is an abstraction : `λx:C.{x, x+4, x+7, x+11}` that represents the structure of the chord.

We can listen to this abstraction. When the system plays an abstraction, it replaces the variables with what they stand for. In our case we have :

PLAY(`λx:C.{x, x+4, x+7, x+11}`) = PLAY(`{C, C+4, C+7, C+11}`)
= PLAY(`{C, E, G, B}`)

We can also transform our abstraction as we do with regular musical structures. For example if we transpose our abstraction by two semi-tones, the C the variable x stands for will be thus transposed :

`λx:C.{x, x+4, x+7, x+11}+ 2` = `λx:C+2.{x, x+4, x+7, x+11}`
= `λx:D.{x, x+4, x+7, x+11}`

If we play the transposed abstraction we will hear : `{D, F#, A, C#}`.

2.3 Application

In a way the application operation is the reverse of the abstraction one. It allows to *specialize*, to *instantiate* something by fixing some of its variable parts. When an abstraction is applied to an argument, its variable parts are replaced by this argument. If we apply our Major Seventh abstraction to another tonic, for example F as in figure 4, all the free occurrences of the variable *x* in the body of the abstraction will be replaced by F, resulting in FM7 :

`λx:C.{ x, x+4, x+7, x+11} F` $\Rightarrow_{\beta}$ `{F, F+4, F+7, F+11)}`
⇒ `{F, A, C, E}`

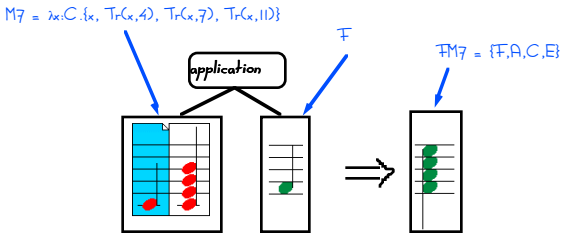


Figure 4 *: application of the Major Seventh abstraction to F*

2.4 High-order scores

As we said earlier, abstractions can be time-organized in a score like regular musical objects. A high-order score can contain sequences and chords of abstractions. If we play such a high-order score, the abstractions will be played according to the rule defined in §2.2. But we can also apply a score of abstractions to some argument. In this case each abstraction is applied to the corresponding part of the argument as illustrated figure 5.

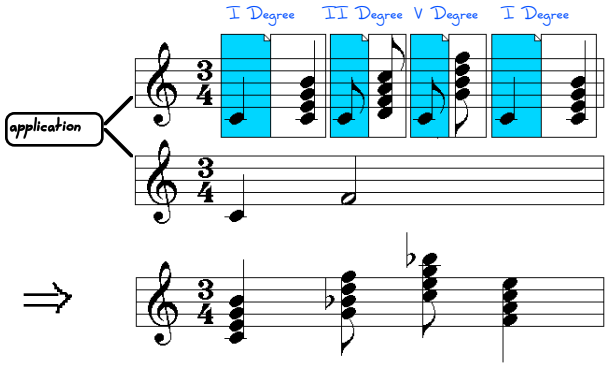


Figure 5 *: Application of a sequence of abstractions*

High-order scores are useful to describe time variant musical concepts. A typical example is a metric structure which can be define as a sequence alternating accentuation and identity functions. We can define a 4/4 metric as :

1/16	*3/16*	*1/4*	*1/16*	*3/16*	*1/4*
acc	id	id	acc'	id	id

that is

```
[[acc/16, id*3/16], id/4, [acc'/16, id*3/16], id/4]
```

where

```
acc    := λx:C.x>20
acc'   := λx:C.x>10
id     := λx:C.x
```

As we will see in the next two sections, the concept of high-order score is also a useful modeling technique allowing to describe a piece as a set of independent layers.

3. Modeling of a Jazz score

Being able to manipulate abstractions like regular musical objects allows to elegantly model Jazz scores. Jazz scores are a kind of *abstract musical object* just specifying some aspects of the music to be played: the chord sequence, the melody, and usually an indication of tempo or style. The chord rhythmic structure is often not really expressed, and the music style indicates the kind of rhythmic pattern the musicians can use. An example of notation for the first 8 bars of the Misty standard is:

EbM7	Ebm7/Eb7	AbM7	Abm7/Db7	EbM7/Cm7	Fm7/Bb7	Gm7/C7	Fm7/Bb7

One part of the musician's work consists in analyzing the chord harmonic progression to detect well known harmonic patterns like II/V/I (in major of minor) or VI/II/V/I. The harmonic structure knowledge helps the musician to improvise. It's quite natural for a musician to think in terms of a chord sequence with chords seen as degrees on a diatonic scale (major or minor), separated from the tonality which is the scale itself.

Therefore the score can now be analyzed this way :

I	II/V	I	II/V	I/VI	II/V	III/VI	II/V

Eb	Ab	Gb	Eb

The separation between the harmonic layer, the tonality layer and the rhythmic layer can easily be defined using Elody. The idea is to see the harmonic layer as the sequence of *abstract chords*, the rhythmic layer as a sequence of *abstract rhythmic patterns*, and the tonality layer as a sequence of notes with one whole note per bar. The result will

be defined as the application of the rhythmic layer to the harmonic layer applied to the tonality layer. The three layers are musical objects which can individually be played and transformed in different ways.

Rhythmic Patterns							
⇓							
I	II/V	I	II/V	I/VI	II/V	III/VI	II/V
⇓							
Eb	Ab		Gb	Eb			

3.1 The chord harmonic layer

We have seen that starting from a concrete CM7 chord: `{C, E, G, B}`, we can build a I degree chord by making the `C` note become variable : `λx:C.{ x, x+4, x+7, x+11 }`. When the system plays this abstraction, it plays its body replacing the formal parameter with the musical object it stand for. Therefore we still hear the CM7 chord, a kind of prototype for all M7 chords. The same process can be used to build all diatonic chords in C major to get the different chords degrees. Abstractions can be put in sequence, and we can build the chord harmonic sequence:

I	II/V	I	II/V	I/VI	II/V	III/VI	II/V

This object can be played, and what we hear is the sequence of all prototype chords, in this case the sequence of chord degrees related to the C major scale:

CM7	Dm7/G7	CM7	Dm7/G7	CM7/Am7	Dm7/G7	Em7/Am7	Dm7/G7

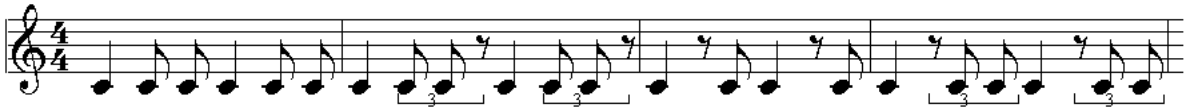
Now this chord harmonic sequence can be enriched by a process of chord substitution, replacing a chord with another chord or a group of chords which have the same harmonic function, a common operation made by jazz musicians. Therefore we can obtain the following chord sequence where the I degree chords (M7) have been replaced by a sequence of M7/M6 chords and V degree have been replaced by V9 chords:

CM7/CM6	Dm7/G9	CM7/CM6	Dm7/G9	CM7/Am7	Dm7/G9	Em7/Am7	Dm7/G9

This score is still a sequence of prototype chords and what we hear is the sequence of all prototype chords related to the C major scale. This sequence is *waiting* to be applied to the actual tonality.

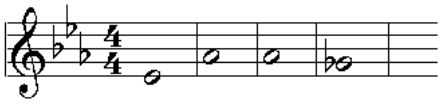
3.2 The rhythmic layer

The same idea can be used to build the rhythmic sequence. Starting from a concrete rhythm of two eight notes : `[C/8, C/8]`, we can build a first function by making the `C/4` note become variable: `λx:C/4.[x/2, x/2]`. This function applied on a quarter note gives a sequence of two eighth notes as a result. When the system plays this abstraction, it plays its body replacing the formal parameter with the musical object it stands for. Therefore we still hear `[C/8, C/8]`, a prototype for this rhythmic pattern. Using the same process, we can build a set of different rhythmic patterns and use them to build a rhythmic sequence. This object can be played, and what we hear is the sequence of all prototype rhythmic patterns played with the same pitch in this example C. Here are the first 4 bars:



3.3 The tonality layer

The tonality layer is simply defined as a sequence of one whole note per bar, where the pitch of each note defines the tonic of the tonality. Here are the first 4 bars:



3.4 The final result

In order to get the final result, we have to apply the rhythmic layer to the sequence resulting from the application of the harmonic layer to the tonality layer. An interesting point in this model is that, because each layer is actually a

musical score, it can be independently manipulated, transformed and composed as such. What is manipulated is a kind of *projection* of the harmonic, tonality and rhythmic aspects of the piece on the time dimension. This is possible because Elody abstractions have the same musical properties as concrete musical objects, especially a time dimension, and because they behave at the same time as prototypes of musical concepts and as functions.

4. Music for Pieces of Wood

Steve Reich's *Music for Pieces of Wood* is written for 5 pairs of claves, tuned using 4 different pitches. It includes 3 sections whose metrics are 6/4, 4/4 and 3/4. Their lengths are respectively 28, 18 and 12 measures. In each section an ostinato is played by voices 1 and 2, while voices 3, 4 and 5 are organized in 2 periods : an exposition period during which a rhythmic pattern is progressively unveiled note by note, followed by a rhythmic period during which the full rhythmic pattern is repeated. The exposition period of a voice starts at the end of the exposition period of the preceding voice. A section ends when all the rhythms are completed.

4.1 Analysis of the piece

All along the piece, the first voice (clave 1) alternates eighth notes and rests (figure 7). The other rhythmic patterns are built on top of a common pattern (figure 8) using 3 kind of transformations:

- rotation : the first element of the pattern is moved to the end.
- notes filtering : some notes are transformed into rests.
- time cutting : at a given position several beats of a pattern are removed.

In section 1 (figure 6) for example, the voices 2 and 5 play the common pattern while the voices 3 and 4 play a rotated version of it (figure 9).

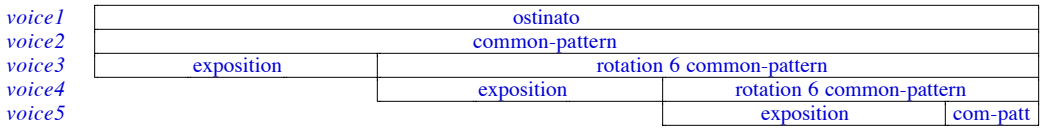


***Figure 6** : Structure of section 1*

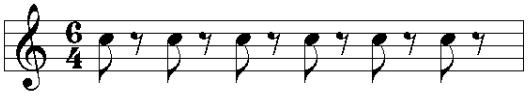


***Figure 7:** The voice 1 ostinato*

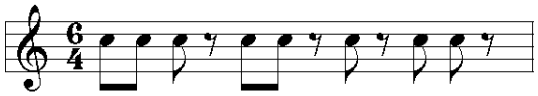


***Figure 8** :The common pattern*

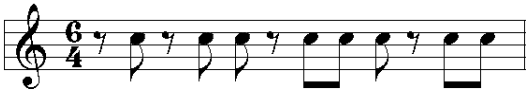


***Figure 9**: The common pattern rotated 6 times to the left*

During the exposition period of a voice, its final rhythm is progressively revealed, note by note at each new measure. The order the notes are revealed can be expressed as a list of relative positions. The relative positions for the first section are the following :

```
- voice 3 :    ( 9, 2, 3, 6, 10, 11, 8)
- voice 4 :    (11, 6, 4, 7, 10, 11, 7)
- voice 5 :    ( 5, 6, 4, 7, 10, 11, 7)
```

4.2 Implementation

The piece can be described in a quite compact way as a high-order score applied to a very simple sequence of 12 eighth notes (figure 10).

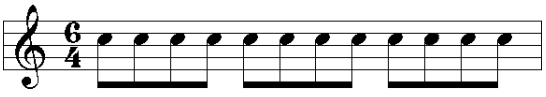


***Figure 10**: The basic sequence.*

This high-order score is composed of a mix of 5 parallel sequences of *filters* that represent the various rhythmic patterns of the piece. A filter is sequence based on two functions, an identity function that returns its argument

unchanged : `id:= λx:C/8.x`, and a rest function that transforms its argument into a rest : `rest:= λx:C/8._/8`.

For example, the voice 1 ostinato and the common pattern are expressed as filters applied to the basic sequence :

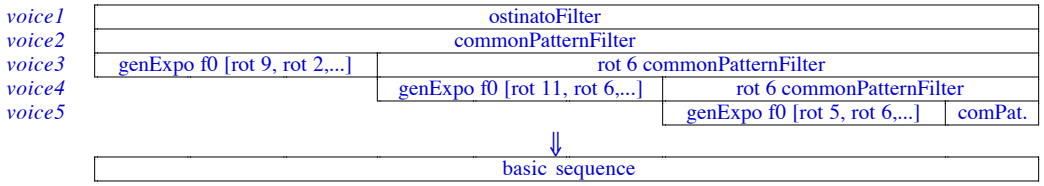
```
ostinatoFilter := [id, rest, id, rest, id, rest, id, rest, id, rest, id, rest]
commonPatternFilter := [id, id, id, rest, id, id, rest, id, rest, id, id, rest]
```

Each exposition period is also defined by a filter. These exposition filters are algorithmically generated by function `genExpo` from a list of *rotation functions* corresponding to the order the notes should be revealed. The algorithm is very simple. Starting from a first filter : `f0:=[id, rest, rest, rest, rest, rest, rest, rest, rest, rest, rest, rest]`, and a sequence of rotation functions : `[r0, r1, r2, …]`, the function `genExpo` takes the first rotation function `r0`, applies it to `f0` to produce the filter `f1`. Then it takes the second rotation `r1`, apply it to `f1`, to produce the filter `f2`, etc. Then, all these filters are assembled in a sequence to form the complete exposition filter of the voice : `[f1, {f1,f2}, {f1,f2,f3}, {f1,f2,f3,f4},…]`

Building a voice consists now in applying the exposition and rhythm filters to the basic sequence. For example for voice 3 we have :

where
```
expo3Filter := genExpo f0 [rot 9, rot 2, rot 3, rot 6, rot 10, rot 11, rot 8]
rhythm3Filter := Rot 6 basicPatternFilter
```

The full score of section 1 is :



The interesting point is that now we can easily transform the piece. We can do simple things, replace the basic sequence, change the ostinato filter or the common pattern filter. We can even modify the rotation lists to change the order the rhythmic patterns are exposed. But we can also do more complex transformations directly related to the fact that the "program" that generates the piece is actually a high-order score with a time dimension. For example we can replace the basic sequence with a rhythmic structure that evolves in time, or even recompose the abstract structure of the piece by taking portions of the program score and recombining them.

5. Conclusion and related works

The design of a music composition programming language is a particularly challenging activity :

- Music composition is a complex application domain with important issues, for instance the connection between real-time and non real-time, or between continuous and discrete time.
- A music programming language has also to face the difficult question of *end users programming* : how to give a composer that is not a programmer access to the power, the flexibility and the expressiveness of a programming language.
- A music programming language has to support an artistical activity, open to improvisations, made of trials and errors, formal and experimental at the same time, a difficult target for traditional programming languages that usually require the programmer to carefully specify a problem before writing a single line of code.

A key question is : *how adequate is a program as a notation for the musical object it describes ?* As noted by several authors, a music programming language has an important *music notation role* for the objects it describes. For example, writing about the Canon language, Roger Dannenberg says : *"Canon is something between of a cross between a music notation and a programming language and a music data language*", *"Canon scores are themselves programs"* [Dannenberg 89].

Concerning Elody, we can reverse this last quotation, and say : "*Elody programs are themselves scores*". Instead of embedding musical data structures into a programming language, we have embedded programming capabilities into a music language. As a result, an Elody program has all the attributes of a musical object. It is an *active music notation* and the programming activity is a quite natural extension of the compositional activity.

As we have seen, Elody programs are either *abstractions* (generalized musical objects obtained by making variable parts of a concrete musical object) or *high order scores* (time organized scores of programs). The idea of abstraction on musical objects has also been considered by M. Balaban [Balaban 1994]. The concept of high-order score is close to G. Assayag and C. Agon concept of *maquette* [AAFH 97] and to CyberBand Score Sheets + Modifiers [WOJPF 97], but Elody high-order scores have the great advantage to have an *applicative semantic*. This approach leads to elegant models where a musical object can be seen as a explicit composition of different layers, each layer stands for a prototype of the described concept but keeps it's functional property and where abstract concepts like chord degrees, rhythmic patterns can be expressed and manipulated like concrete objects.

Being based on lambda-calculus, Elody benefits from the advantages of the functional paradigm [Hughes 1989]. R. Dannenberg proposed several functional music languages that demonstrate the advantages of features like lazy evaluation and high-order functions [Dannenberg 84, 89, 91]. The functional approach is also central to the solution proposed by P. Desain and H. Honing for the representation of control functions [DH 91]. Moreover, Haskore, an elegant *algebra of music* developed by P. Hudak et al. with the functional language Haskell [HMGW 96], shows clearly the advantages of the functional approach as an adequate notation to represent both abstract musical ideas and their concrete implementations.

6. References

[AAFH 97] Assayag G., C. Agon, J. Fineberg, P. Hanappe, "An Object Oriented Visual Environment For Musical Composition", Proc. ICMC 97, San Francisco: ICMA Publishing, 1997.

[Balaban 94], Balaban M., "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 84] Barendregt H. P., *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, Amsterdam, 1984.

[Church 41], Church A., *The Calculi of Lambda Conversion*, Princeton University Press, Princeton, N.J., 1941.

[Dannenberg 84], Dannenberg R. B., "Artic: A Functional Language for Real-Time Control", in *1984 ACM Symposium on LISP and Functional Programming*, ACM, New-York, 1984.

[Dannenberg 89], Dannenberg R. B., "The Canon Score Language", *Computer Music Journal*, 13 (1), pp. 47-56, 1989.

[DFV 91], Dannenberg R. B., C. L. Fraley and P. Velikonja, "Fugue : A Functional Language for Sound Synthesis", *Computer*, 24 (7), pp. 36-42, 1991.

[DH 91], P. Desain and H. Honing, "Time Functions Function Best as Functions of Multiple Times", *Computer Music Journal*, 16 (2), pp. 17-34, 1991.

[Durfee 97] Durfee G., "A Model for a List-oriented Extension of the Lambda Calculus", CMU technical report CMU-CS-97-151, May 1997.

[HHFK 98] Hoos H., K. Hamel, K. Flade, J. Kilian, "GUIDO Music Notation - Towards an Adequate Representation of Score Level Music", Proc. JIM 98, LMA-CNRS, 1998.

[HMGW 96] Hudak P., T. Makucevich, S. Gadde, B. Whong, "Haskore Music Notation - An Algebra of Music", Journal of Functional Programming, Cambridge University Press, 6(3), June 1996.

[Hughes 1989] J. Hughes, Why Functional Programming Matters, *The computer Journal* 32 (2), pp. 98-107, 1989.

[OFL 97] Orlarey, Y., D. Fober, S. Letz. "Elody : a Java+MidiShare based Music Composition Environment", Proc. ICMC 97, San Francisco: ICMA Publishing, 1997.

[OFLB 94] Orlarey, Y., D. Fober, S. Letz and M. Bilton. "Lambda-Calculus and Music Calculi", Proc. ICMC 94, San Francisco: ICMA Publishing, 1994.

[Révész 88] Révész G., "Lambda-Calculus Combinators and Functional Programming", Cambridge Tracts in Theoretical Computer Science volume 4, Cambridge U. Press, 1988.

[WOJPF 97] Wright J., D. Oppenheim, D. Jameson, D. Pazel, R. Fuhrer, "CyberBand : A "Hands-On" Music Composition Program", Proc. ICMC 97, San Francisco: ICMA Publishing, 1997.

7. Acknowledgments

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