HARMONY: A System for Musical Composition

Jose Emilio Labra Gayo
Luis Ángel Oliveira Rodríguez
Juan Manuel Cueva Lovelle

Department of Computer Science
University of Oviedo
C/ Calvo Sotelo S/N, 33007
Oviedo, Spain
E-mail: labra@uniovi.es
Telephone: +34-85103394

Abstract

This paper describes HARMONY: a system that integrates the phases that take part in the creation of a musical composition. The kernel deals with automatic melody generation through fractal algorithms and harmonisation of generated melodies based on traditional theory. The system has been developed in *Haskell* and works on top of the musical description system *Haskore*.

HARMONY: A System for Musical Composition

Abstract

This paper describes HARMONY: a system that integrates the phases that take part in the creation of a musical composition. The kernel deals with automatic melody generation through fractal algorithms and harmonisation of generated melodies based on traditional theory. The system has been developed in Haskell and works on top of the musical description system *Haskore*.

1 Introduction

Using computers as practical tools for creative development is taking substantial importance. Specialised techniques for writing high level compositions can be offered to non advanced users and several algorithms for musical composition and synthesis have appeared [1]. The goal of HARMONY is to integrate these algorithms in a system that facilitates the task of musical composition.

Usually, Harmony subject ([7], [8], [21], [29]) is taught in the final years of musical studies owing to its difficulty. Automating the process of harmonising a melody eases the compositional task, allowing the user to concentrate in tasks that are more creative. The system does not try to force the composer to use some fixed set of rules. On the contrary, it tries to provide the user some essential techniques she needs to develop music with some fixed harmonic guidelines. In this way, a basic requirement is the parameterisation of the rules used in all the processes to give more freedom to the composer.

The kernel of HARMONY has been developed in *Haskell* [22], a non-strict purely functional language and uses *Haskore* [12] as the basis for musical description. The use of functional languages to describe musical structures provides a lot of interesting possibilities [6], [18]. Lazy evaluation allows the definition of infinite compositions and higher order functions increase the composer's abstraction. This system shows that the expressive power of purely functional languages can be used in the description of high level musical ideas.

This paper begins with an overview of the Harmony system, then describes the melody generation and harmonisation phases, which are the kernel of the system, and ends with some conclusions and future work.

2 System Description

Figure 1 shows the different modules that integrate Harmony. The **score recogniser** [9] and **musical editor** [10] have been developed in C++ and will not be described in this paper. The **melody generator** takes some parameters given by the user and generates melodies using fractal algorithms. The **harmonisation** phase takes a melody and harmonises it following some set of rules that depend on the kind of music desired. The instrumentation phase assigns an instrument to each part of the composition and, finally, the **performance** phase takes charge of the actions needed to transform a composition into audible music. Actually, the instrumentation phase assigns the default instruments used in *Haskore* and the performance phase transforms musical objects to MIDI files using the modules defined in *Haskore*.

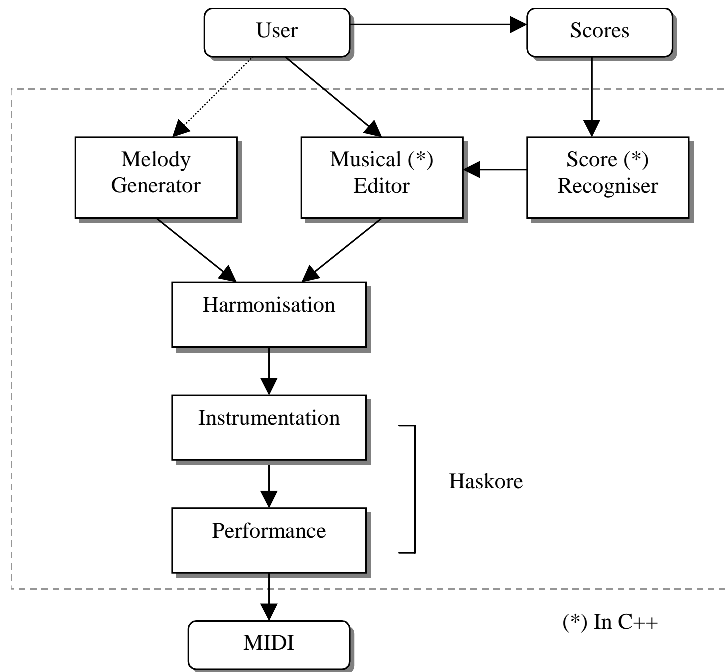

Figure 1: Harmony System

3 Melody Generation

The system implements several algorithms that generate random melodies. We represent a melody as a signal whose frequency varies over time. The aim is that the user can select the appearance of the generated melodies through some parameters such as tempo, character, etc. Of course, the melodies must have some randomness in order to select those who fit better to the composer purposes.

We have implemented a random number generator using state transformer monads and classical algorithms that generate fractal signals following [20]. We are also incorporating new techniques such as genetic algorithms or L-Systems to improve the generator quality.

Random Number Generator

The *Haskell* library provides a random number generator that generates an infinite list of random numbers. It is a little cumbersome to work with this infinite list because the functions that consume random numbers need the infinite list as an argument and must return the numbers not consumed. For this reason, we have encapsulated the random number generator in a state transformer monad as described, for example, in [15]. If the state transformer monad is defined as:

```
newtype State s a = S (s ->(a,s))
runS :: State s a -> s -> (a,s)
modS :: (s -> s) -> State s ()
setS :: s -> State s ()
getS :: State s s
instance Monad (State s)
```

Then, the random number generator could be defined as:

```
type RandM a = State [Elem] a

getRandM::RandM Seed
```

```
getRandM = do
            xs <- getS
            modS tail
            return (head xs)

runRandM::Elem->RandM a->a
runRandM seed a = fst (runS a (rands seed))
```

Where `rands s` generates an infinite list of random numbers with the seed `s`. We can define a function to get a list of n random numbers as:

```
randList::Int->Elem->[Elem]
randList n s = runRandM s (accumulate [getRandM| i <- [1..n]])
```

The above approach has a limitation; we cannot display the random numbers or execute an I/O action during the process. It can be solved if we use a state transformer monad with Input/Output actions as:

```
newtype IOS s a = IOS (s->IO(a,s))
runIOS :: IOS s a -> s -> IO (a, s)
getIOS :: IOS s s
setIOS :: s -> IOS s ()
modIOS :: (s -> s) -> IOS s s
execIO :: IO a -> IOS s a
instance Monad (IOS s)
```

Now, the random number generator can be defined as:

```
type RandMIO a = IOS [Elem] a

getRandMIO::RandM Elem
getRandMIO = do
              xs <- getIOS
              modIOS tail
              return (head xs)

runRandMIO::Elem->RandM a->IO a
runRandMIO seed act = do
                       (v,_)<-runIOS act (rands seed)
                       return v
```

And we can write a function that returns a list of n random numbers and also displays each number generated:

```
randList::Int->Elem->IO [Elem]
randList n t = do
                xs <- runRandM t (accumulate [do
                                       v <- getRandM
                                       putStrLn ((show v) ++ " ")
                                       return v
                                       |i <- [1..n]])
                return xs
```

Algorithms for Random Melody Generation

Once we have described the random number generator used, we will briefly describe some of the algorithms implemented in the following section. It is straightforward to generate white noise:

```
whiteNoise n = do
                gs <- accumulate [getGaussM | i <- [1..n]]
                return (listSignal gs)
```

Where `getGaussM` returns a Gaussian random number and `listSignal gs` converts a list of numbers in a signal.

We can also simulate one-dimensional Brownian motion as the integral of uncorrelated Gaussian noise using:

```
whiteNoiseBM n t = do
                    gs <- accumulate [getGaussM | i <- [1..n]]
                    return (listSignal (scanl1 (+) gs))
```

We can also interpret Brownian motion as the cumulative sum of a series of random cuts with n/2 gaussian numbers as:

```
randomCuts m n = foldM (createCut n) (emptySignal n) [1..m]
        where createCut n s i = do
                    pos<- getRandM
                    gs <-accumulate [getGaussM|j <-[1..n `div` 2]]
                    return (sumCut pos s n gs)
```

Where `emptySignal` generates a signal with all values set to zero, and `sumCut pos s n gs` sums the values in a signal `s` from a given position `pos` with the values in the list `gs`.

Fractional Brownian motion can be approximated using midpoint recursion as described in [20]. The algorithm starts with an initial interval (x_0, x_n), setting $x_0 = 0$ and x_n as a sample of a Gaussian random variable with mean zero and variance σ^2. Then the value in the middle point of the interval, $x_{n/2}$, is calculated as the average of x_0 and x_n plus a displacement d_1. The algorithm takes two new intervals, $(x_0, x_{n/2})$ and $(x_{n/2}, x_n)$, calculates their middle points applying a new displacement d_2 and continues recursively. The displacements satisfy:

$$d_m^2 = \frac{1}{2}\sigma^2\left(1 - \frac{1}{2^{2-2H}}\right)\frac{1}{2^{2Hm}}$$

Where $0 < H < 1$ indicates the fractal dimension $D = 2 - H$ (when $H = \frac{1}{2}$ we have brownian motion).

The implementation of the algorithm uses an array that is filled recursively:

```
midPointFM1D m sigma h = do
        gs <- accumulate [getGaussM | j <- [0..n]]
        return (midPointRecursion n sigma h (listSignal gs))
      where n = 2 ^ m

midPointRecursion n sigma h g = arraySignal a
      where
        a = array (0,n) ( [(0,0) , (n, (g!n)*sigma )] ++
                          midPointRec 0 n fill)
        fill lo hi med = 0.5 * (a!lo + a!hi) + delta lo hi
        midPointRec x y f | x == med    = []
                          | otherwise = [(med, f x y med)]  ++
                                        midPointRec x med f ++
                                        midPointRec med y f
                        where med = (x + y) `div` 2
```

Where `delta` calculates the corresponding displacements and `arraySignal` transforms the array to a signal.

4 Harmonisation

Once the melodies have been generated, the system offers the possibility to harmonise them following the traditional rules of classical western music. The main problem we had to address is computer representation of musical ideas. The first approach was to define an intermediate language based on lists of notes and define functions that work with that language. That scheme was very simple but also very restrictive. Actually, the system has adopted *Haskore* as the way to represent musical objects. *Haskore* is a set of *haskell* modules, which defines some datatypes and functions to work with musical objects. In a very short description, some of the datatypes and functions defined in Haskore are:

```
type Pitch      = (PitchClass, Octave)
data PitchClass = Cf| C | Cs| Df| D | Ds| Ef| E | Es| Ff| F|
            Fs| Gf| G | Gs| Af| A | As| Bf| B | Bs
type Octave     = Int

data Music = Note Pitch Dur   -- a note \ atomic
           | Rest Dur         -- a rest / objects
           | Music :+: Music  -- sequential composition
           | Music :=: Music  -- parallel composition
            . . .
type Dur   = Float            -- in whole notes
```

In this way, the user can work with a specialised *Haskell* for musical structures. *Haskore* also includes some modules for performance and instrument construction that have also been adopted in the actual version of *Harmony.*

In *Harmony,* we have defined some specific datatypes and functions to work with chords, intervals, modes, scales, etc

```
data Interval = Unison          |
            MinorSecond     | MajorSecond  |
                MinorThird      | MajorThird   |
                PerfectFourth   |
                DiminishedFifth | PerfectFifth |
                MinorSixth      | MajorSixth   |
                MinorSeventh    | MajorSeventh |
                Octave

-- some auxiliary functions
semitones::Interval->Int        -- Number of Semitones in an interval
transI::Pitch->Interval->Pitch  -- Transform a pitch one interval
...
-- Intervals in a major scale
intsMajor=[Unison,MajorSecond,MajorThird,PerfectFourth,
     PerfectFifth,MajorSixth,MajorSeventh]
```

The major scale can be computed in the following way:

```
majScale = [transI i | i <- intsMajor]  -- Major Scale
```

The harmonisation process can be divided in three phases, see Figure 2:

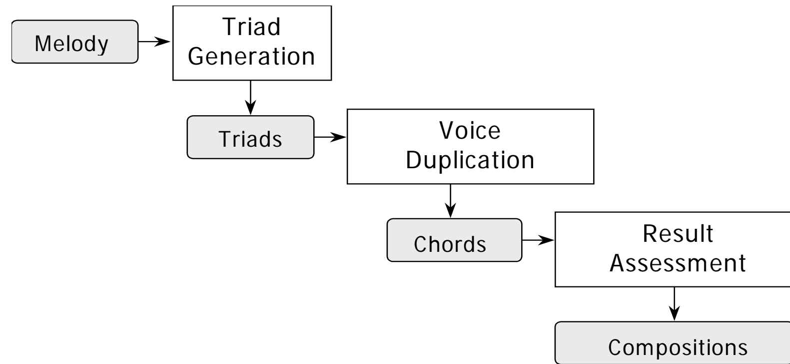

Figure 2: Harmonisation Process

- **Triad Generation** (three-part writing): It gets a melody, and generates the corresponding triad for each note. We have written a simple scheme to generate possible representations for a chord. It consists of a list of transformations that are applied to each chord. The transformations are lists of integers that indicate the number (positive or negative) of octaves to move the corresponding note of the chord. The list of transformations to be applied will be a parameter of the algorithm in order to provide greater flexibility.

- **Voice Duplication:** To obtain four-part writing we duplicate one of the voices raising or lowering it a given number of octaves.

- **Result Assessment**: We have defined functions to avoid unisons and parallel fifths and octaves, to control the compass of voices and to assess the different candidates. The assessment scheme uses fuzzy sets to represent possible preferences of transitions between chords (see Figure 3).

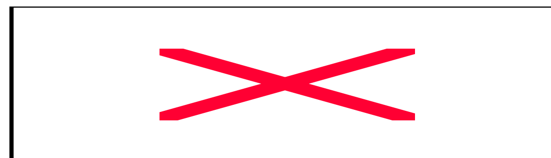
Figure 3: Some preferences

We represent each preference P as a membership function $\mu_P : Movement \to [0,1]$

```
type Preference = PRE ( Movement -> Float)

-- some preferences
indiferent = PRE (\x -> 1)
move       = PRE (\x -> min 1 (x/(2*octave)))
dontMove   = PRE (\x -> max 0 (1 - x/(2*octave)))
```

For each voice, we calculate the membership grade of the movement between notes to the desired preference of that voice; the total assessment will be computed as $\sum_{v}^{voices} \mu_{P_v}(movement_v)$

The system implements several harmonisation algorithms. All these algorithms take as parameters the possible generators, filters and assessment functions. For example, in figure 4 we show one of the algorithms. It takes a list as input (we will pass a list of chords), applies first to get the representation for the first chord. For each of the next chords, it generates the list of possible representations (generate), filters it considering their relationship with the previous chord (eliminating parallel fifths and octaves, unisons, etc.) and selects the one with best assessment.

```
getBestWithFilter::(Ord n)=>
```

```
                    [a]->               -- Input sample
                    (a->b)->            -- First transformation
                    (a->[b])->          -- Generator function
                    (b->[b]->[b])->     -- Filtering function
                    (b->b->n)->         -- Heuristic function
                    [b]                 -- Output
getBestWithFilter (c:cs) first generate filt h = scanl g (first c) cs
 where       g x y = foldl1 (getMin x) (filt x (generate y))
             getMin a x y | h a x < h a y = x
                          | otherwise     = y
```

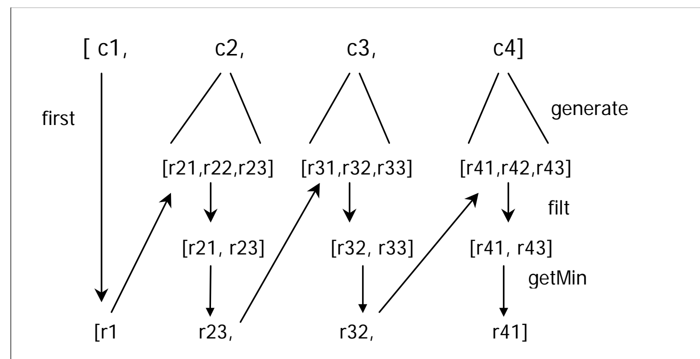

Figure 4

5 Related Work

There is a great deal of works that use fractals or genetic algorithms for musical composition [11], [17], [26], [28]. Most of them generate strange melodies and few of them allow the user to select a melody and harmonise it following the traditional rules. These systems are usually implemented with classical programming techniques and it is surprising the exiguous use of purely functional languages in the development of this kind of algorithms.

Harmony arose in parallel with Haskore [12]. At first, we developed an intermediate language for musical description but we have now adopted *Haskore* as a domain specific embedded language. In [18], a purely functional language based on lambda calculus is used to describe musical structures. R. B. Dannenberg has defined a language series based on LISP [4], [5] and [6] with an special emphasis in sound synthesis. These works, in spite of being based on LISP, emphasise the need of lazy evaluation.

It seems appropriate to mention that there are several commercial products [2][24][25] that assist in musical composition using conventional programming techniques. Most of these systems are oriented towards Jazz or similar forms that allow a greater freedom in the compositional and performance process.

6 Conclusions and Future Work

The advantages of using *Haskell* as the development language raised when we had to modify the harmonisation rules and adapt to a new intermediate language. Higher order functions enable the independence between particular algorithms and special harmonisation rules. Likewise, lazy evaluation allows the description of infinite compositions with greater abstraction.

There are a number of future lines of work. One of the priorities will be to integrate the different components of the system in a friendly user interface. The users of Harmony do not need to have much knowledge of computers and modern GUI technologies in purely functional languages have improved considerably. We are now developing the first version that integrates the components using the HUGS Graphics library [23] and we would like to offer a real-time composition system. We are

also considering the possibility to develop a special purpose language in order to allow the user to describe her harmonisation rules or melody generation algorithms. Following P. Hudak [13], it seems appropriate to embed that language in a general-purpose language as Haskell.

We are working with new fractal algorithms to melody generation and with genetic algorithms. The main problem with genetic algorithms applied to music generation is the selection of a good fitness function [3], we are considering the approximation to the traditional rules of classical music as the fitness function.

With regard to the musical skill of the system, it is necessary to research the incorporation of different musical forms and styles. It is essential to consider the influence that other parameters such as rhythm, instruments or performance attributes can exert on the music composed.

Acknowledgments

The implementation of some of the modules have been carried through Final Year Projects[1] that have been developed or are being developed Antonio Fernández Vidal [9], Mónica Freira Maira [10], Esther García Galán y Natividad Vilela Carral.

References

[1] A. Alpern, Techniques for Algorithmic Composition of Music, Hampshire College Divisional Examination,Humanities and Arts. 1995

[2] John A. Biles , GenJam: A Genetic Algorithm for Generating Jazz Solos, *International Computer Music Conference*, 1994

[3] John A. Biles, P. G. Anderson, L. W. Loggi, Neural Network Fitness Functions for a Musical IGA. *Rochestes Institute of Technology*, 1996

[4] R. B. Dannenberg, C. L. Fraley, P. Velikonja. A Functional Language for Sound Synthesis with behavioral abstraction and Lazy Evaluation. En *Computer Generated Music,* Denis Baggi (Editor). IEE Computer Society Press. 1992

[5] R. B. Dannenberg, The Implementation of Nyquist. A Sound Synthesis Language. En *Proceedings of the 1993 International Computer Music* Conference, International Computer Music Association, Sept. 1993, pp. 168-171

[6] R. B. Dannenberg, Expressing Temporal Behavior Declaratively. En *CMU Computer Science, A 25th Annyversary Commemorative.*R. F. Rashid (Editor), ACM Anthology Series (Chap. 3) pp. 47-68

[7] Dietter de la Motte. Armonía. *Ed. Labor S.A.* ISBN: 83-335-7859-6 1989.

[8] R. Franko Goldman, Harmony in Western Music, *W.W. Norton & Company Inc.* ISBN: 0 393 09746 3. 1965

[9] A. Fernández Vidal, J. E. Labra Gayo, Reconocedor Óptico de partituras. Proyecto *Fin de Carrera.* Escuela Universitaria de Ingeniería Técnica en Informática de Oviedo, Spain, 1997

[10] M. Freire Maira, J. E. Labra Gayo, Editor Interactivo de Música Polifónica..*Proyecto Fin de Carrera,* Escuela Universitaria de Ingeniería Técnica en Informática de Oviedo, 1997

[11] R. Greenhouse. The Well-Tempered Fractal v3.0, A composers Tool for the Derivation of Musical Motifs, Phrases and Rhythms From the Beauty and Symmetry of Fractals, Chaotic Attractors and other Mathematical Functions.

[1] A Final Year Project is a work that the undergraduate must do to qualify as a Computer Science Engineer in the University of Oviedo

[12] P. Hudak, T. Makucevich, S. Gadde, B. Whong. Haskore Music Notation – an Algebra of Music. *Journal of Functional Programming,* 6(3), June 1996.

[13] P. Hudak, Building Domain-Specific Embedded Languages. Dept. of Computer Science, Yale University, June 1996.

[14] John Hugues. Why Functional Programming matters. *The Computer Journal*, 32(2), 98-107.

[15] G. Hutton, E. Meijer. Monadic Parser Combinators, 1996

[16] J. Jeuring, P. Janson. PolyP – a polytipic programming language extension. En POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Laguages, pages 470-482. ACM Press, 1997-07-06

[17] G. Lee Nelson, "Sonomorphs: An Application of Genetic Algorithms to the Growth and Development of Musical Organisms". *Proceedings of the Fourth Biennial Art & Technology Symposium,* Connecticut College, 4-7 Marzo 1993, pp. 155-169.

[18] O. Orlarey, D. Fober, S. Letz, M. Bilton. Lambda calculus and music calculi. En *Proceedings of International Computer Music Conference.* International Computer Music Association, 1994

[19] L.A. Oliveira Rodríguez, J. E. Labra G. Sistema de Ayuda a la Composición Musical. Proyecto Fin de Carrera. Escuela Universitaria de Ingeniería Técnica en Informática de Oviedo. 1996

[20] Heinz-Otto Peitgen, Dietmar Saupe, Editors. The Science of Fractal Images, *Springer-Verlag,* 1988

[21] G. Pratt, The Dynamics of Harmony. Principles and Practice. Open University Press. ISBN: 0 335 10595 5, 1984

[22] Peterson, J. and Hammond, K. (editors). Report on the Programming Language Haskell 1.4, A Non-strict Purely Functional Language. Technical report. Yale University. Department of Computer Science. 1997 (April)

[23] A. Reid, The HUGS Graphics Library, Distributed with HUGS 1.4

[24] The Symbolic Composer Language, Experimental Music Laboratory For The PowerMacintosh, http://www.xs4all.nl/~psto/index.html

[25] Nathan Tenny, Orfeo: Fractal Music, http://www.qualcomm.com/~ntenny/fmusic/

[26] Claus-Dietter Schulz, The Fractal Music Bibliography, Computer Center Univ. of Stuttgart (RUS) Dept. Communication Systems 70550 Stuttgart, Germany

[27] M.Vestin. Genetic Algorithms in Haskell with polytypic programming. Febrero de 1997. Master's Thesis. Göteborg University. 1997. http://www.cs.chalmers.se/~johanj/polytipism/genetic.ps

[28] Wentian Li, A Bibliography on 1/f Noise, *Rockefeller university*, http://linkage.rockefeller.edu/wli/1fnoise/

[29] J. Zamacois. Tratado de Armonía. Libros I, II y III. *Ed. Labor*, 1948