HasBass

Javier M. Corti

15 de diciembre de 2011

1. Introducción

El propósito de este programa es generar un base para una sucesión de acordes, a un tempo dado. De esta manera se obtine una pista de acompañiamiento ("backing track") que puede ser utilizada para la práctica de ejercicios, melodías y soleo sobre alguna progresión dada.

Para generar un acompañiamiento, ya hay una infinidad de programas que son capaces de componer música. El fuerte de HasBass es que recibe como entrada una lista de acordes, que es una vista más abstracta de una cancion que cada una de las notas que la componen. De esta manera, se puede producir un acompañiamiento rapidamente, sin preocuparse por los detalles de cada nota y su duración correspondiente. La entrada del programa es casi una traducción directa del formato "leadsheet"de jazz. Es oportuno mencionar que se usarán los nombres de las notas y acordes del çifrado americano", donde C = Do, D = Re..., G = Sol, A = La y B = Si, siguiendo el formato de los leadsheets.

Una base completa está compuesta por ritmo y armonía (los acordes). En este caso, los acordes son suministrados por un sonido de piano. Se genera, además, una línea de bajo que depende del contexto armónico, pero además dota de ritmo a la base. El agregado de una batería daría mayor versatilidad rítmica, pero en este programa, la idea central radica en la sucesión de acordes, y el "swing"que proporciona la línea de bajo es un ritmo suficiente.

2. Instrucciones de Uso

Se compila ejectuando el comando

```
ghc Main.lhs
```

El uso del programa es extemadamente simple. Solo recibe por línea de comandos el nombre del archivo de entrada y el de salida, y no más argumentos. La sintaxis del archivo de entrada sigue la forma de los leadsheet tradicionales, y está dada por :

```
input = tempo sucesion

tempo = int

sucesion = compas '|' sucesion | ''

compas = acorde compas | ''
```

Los Acordes están dados por :

```
acorde = nota tipo | nota tipo '/' nota

nota = notanat | notanat b | notanat #
```

```
notanat = A | B | C | D | E | F | G
```

Los tipos de acordes :

```
"", "maj", "+", "aug", "-", "m", "m+", "m-", "0", "dim",
"sus2, "sus4", "4", "0+", "2", "6",
"7", M7", "Ma7", "9", "M9", "11", "13",
"3+", "-5", "5-", "+5", "5+", "-6", "6-", "7+", "-9", "+9", "+11"
```

3. Librerías

3.1. Haskore

La librería más importante del programa es Haskore, que proporciona tanto los tipos de datos básicos como las funciones que permiten manipular música fácilmente. Haskore propone una especificación algebraica de la música (en general representa "multimedia temporal") cuyas primitivas son la composición secuencial (:+:) y la composición paralela (:=:), dur la duración de un "temporal mediaτ none la ausencia de "temporal media"por una duración. Fue utilizado el paquete haskore-0.2.0.2 .

3.2. System.Random

Esta librería es muy útil para simular improvisación mediante pseudo-aleatoridad. En este programa se genera una secuencia pseudo aleatoria (gen) al inicio, y se la pasa a toda función que necesite un valor aleatorio. Cada una de estas funciones debe ademas, retornar una nueva secuencia de donde extraeran valores las funciones que se ejecuten despues. En este sentido la secuncia aleatoria se trata como un estado en el programa. Fue usado el paquete random-1.0.0.3 .

3.3. System.Environment

Esta libreria proporciona las funciones getArgs y getProgramName que le permiten al programa recibir argumentos de linea de comandos y comunicarse con el SO. Incluida en el paquete base-4.3.1.0 .

3.4. Parsec

Se usa parsec para construir el parser del programa, ya que proporciona buenos mensajes de error. Su uso en este programa no difiere demasiado al de la libreria de combinadores de parsers implementada en clase. Para correr el parser se usa la función parse, que da como resultado un tipo Either, donde Left lleva el mensaje de error, en caso de un error en el parseo (Right contiene el resultado de un parseo exitoso). Se usa el paquete parsec-3.1.1 .

3.5. ReadP

Para usar la implementacion del parser que reconoce los tipos de acordes de Haskore, se usa el parser de tipo ReadP. Cuando este parser falla, se invoca manualmente al fail de parsec. El ReadP esta incluida en el paquete base-4.3.1.0 .

3.6. Control.Monad

Cuando se quiere usar liftM o alguna otra función monadica, se importa esta librería, también incluida en el paquete base.

4. Main

Se extraen los nombres del archivo de entrada y el de salida de la línea de comandos y se obtiene una semilla para la sucesión pseudo-aleatoria. Luego de parsear y procesar el archivo de entrada, se almacena el resultado en el archivo de salida.

```
> module Main where
> import Tipos
> import Sonido
> import System.Random
> import System.Environment
> import Text.ParserCombinators.Parsec
>
>
>
> main :: IO()
> main = do xs     <- getArgs
>           case (length xs) of
>               2 -> do let [f,g] = xs
>                       input <- readFile f
>                       gen   <- newStdGen
>                       case (parse parseInput "" input) of
>                         Left err      -> do putStr "error de parseo en "
>                                             print err
>                         Right (s,bpm) -> saveToFile g (process gen s bpm)
>
>
>               _ -> do name <- getProgName
>                       putStrLn("Uso : " ++ name ++ " inputFile outputFile")
```

5. Tipos

```
> module Tipos(Note,AbsNote,Chord(Cons,root,bassnote,chordType),Bar,Song,parseInput) where
> import Text.ParserCombinators.Parsec
> import qualified Haskore.Basic.Pitch  as Pitch
> import qualified Haskore.Composition.ChordType as ChordType
> import qualified Text.ParserCombinators.ReadP as ReadP
> import Control.Monad
```

5.1. Notas

El tipo Note, lo definimos como sinónimo del tipo Pitch.Class suministrado por la librería Haskore. Es una representación del conjunto de equivalencia de todas las frecuencias que determinan una misma nota. Es decir, Note no distingue entre distintas teclas del piano, pero sí distingue las teclas que producen un Do de las que producen un Re.

Por otro lado, el tipo AbsNote se define como sinónimo del tipo Pitch.Absolute, que sí representa cada tecla de un piano, donde no es lo mismo un Do grave que uno agudo.

```
> type Note = Pitch.Class
> type AbsNote = Pitch.Absolute
```

En la entrada solo es necesario reconocer notas no absolutas. Para hacer el parser de ellas se definen unas funciones auxiliares :

- extr se usa para extraer el valor parseado de una lista de posibles parseos
- sToNote usa la función classParse de haskore para convertir un String en una nota. Debe usarse solo cuando se está seguro que no va a fallar.
- noteNat parsea el caracter de una nota sin accidentales
- acc parsea el caracter de un accidental

parseNote parsea una nota, primero tratando de parsear una nota con accidental, y luego una sin.

```
> extr :: [(a,b)] -> a
> extr = fst.head
>
> sToNote :: String -> Note
> sToNote = extr . Pitch.classParse
>
> noteNat :: Parser Char
> noteNat = oneOf "ABCDEFG"
>
> acc :: Parser Char
> acc = oneOf "#b"
>
> parseNote :: Parser Note
> parseNote = do n <- noteNat
>                ( (acc >>= (\c-> return $ sToNote (n:[c]))) <|> (return $ sToNote [n]))
```

5.2. Acordes

Para reprensentar los acordes se usan datos correspondientes a su notacion simbólica. La nota tónica, la nota más grave del acorde, y el tipo de acorde, representado con un tipo de Haskore. Por ejemplo : Am/G representa el acorde de tonica A, con bajo en G y tipo de acorde menor.

```
> data Chord = Cons { root      :: Note,
>                     bassnote  :: Note,
>                     chordType :: ChordType.T } deriving (Eq,Show)
```

Para parsear un acorde se debe parsear el tipo. Lo que se hace es extraer el substring que se encuentra entre el final de una nota y un separador que puede ser un espacio, el caracter '/' que indica cual es la nota más grave del acorde, y el caracter '—' que separa compases. Al extraer este String, se corre el parser de tipo de acordes implementado en Haskore de tipo ReadP y si falla, se invoca manualmente al fail de parsec.

```
> chordTfromString :: String -> [(ChordType.T,String)]
> chordTfromString = filter (null . snd) . ReadP.readP_to_S ChordType.parse
>
> sepAc :: Parser Char
> sepAc = (oneOf "/|" <|> space)
>
> parseChordT :: Parser ChordType.T
```

```
> parseChordT = do s <- manyTill anyChar (lookAhead sepAc)
>                  if null (chordTfromString s) then fail ("\""++s++"\" : Tipo de acorde inesperad
>                                               else return $ extr $ chordTfromString s
>
> parseBassNote :: Parser Note
> parseBassNote = do char '/'
>                    n <- parseNote
>                    return n
>
> parseChord :: Parser Chord
> parseChord = do r <- parseNote
>                 ct <- parseChordT
>                 b <- parseBassNote <|> (return r)
>                 return $ Cons r b ct
```

5.3. Compases y Canciones

Un Compás se puede pensar como una sucesión de acordes, y una canción es claramente una sucesión de compases.

```
> type Bar = [Chord]
> type Song = [Bar]
```

Para implementar los parsers se usan las funciones de parsec :

- space, consume un whitespace.
- spaces, consume reiteradamente un whitespace hasta que falla.
- sepEndBy1 p s, que corre al menos un parser p separado y opcionalmente terminado por el parser s.
- endby1, es igual a sepEndBy1, excepto que no es opcional que termine con el parser separador.

```
> spaces1 :: Parser ()
> spaces1 = skipMany1 space
>
> parseBar :: Parser Bar
> parseBar =  (parseChord `sepEndBy1` spaces1) <?> "secuencia de acordes"
>
> parseSong :: Parser Song
> parseSong = (parseBar) `endBy1` ((char '|') >> spaces)
```

5.4. Parseando la Entrada

La entrada del programa es un número indicando el tempo, y la progresion de acordes, asi que el parser final queda :

```
> parseInt :: Parser Integer
> parseInt = (liftM read $ many1 digit ) <?> "tempo (bpm)"
>
> parseInput :: Parser (Song,Integer)
> parseInput = do bpm <- parseInt
```

```
>                 spaces
>                 s <- parseSong
>                 eof
>                 return (s,bpm)
```

```
Como parseSong consume whitespaces al final, se puede revisar si ya termino la entrada, o quedo al
```

6. Funciones Básicas

```
> module Basicas(chordNotes) where
> import Tipos
> import qualified Haskore.Basic.Pitch  as Pitch
> import qualified Haskore.Composition.ChordType as ChordType
```

chordNotes es una función fundamental para el programa ya que se mueve entre dos niveles de abstracción : Toma un acorde (elemento abstracto) y devuelve una lista de notas absolutas. Para esto se debe precisar en que octava se quiere desplegar el acorde, asi que chordNotes también toma un entero. Para tener mayor cantidad de notas, el acorde se despliega sobre un rango de dos octavas, concatenando las notas de cada octava.

```
> chordNotes :: Chord -> Int -> [AbsNote]
> chordNotes (Cons r _ ct) i = map (transpose (i,r)) (ChordType.toChord ct) ++
>                              map (transpose (i+1,r)) (ChordType.toChord ct)
```

toChord es una funcion de ChordType que toma un ChordType.T y devuelve un Chord.T, que es una lista de los intervalos (en semitonos) que forman cada nota del acorde con la tónica r. Es decir un acorde mayor en tipo Chord.T sería [0,4,7] (tónica, tercera mayor y quinta). Entonces para formar las notas concretas de un C mayor, voy a tener que transponer a la tónica (un C) la cantidad de semitonos indicada en Chord.T. Es decir, si elijo como tónica a C4, el acorde resultante sería [C4+0,C4+4,C4+7].

Para esto esta la funcion transpose, que relaciona una nota con una nota traspuesta una cantidad de i de semitonos. Esta claro que para hacer la suma C4 + x, debo ver a C4 como un entero, por lo que se usa la función toInt (mapea cada tecla del piano a un entero).

```
> transpose :: Pitch.T -> Pitch.Relative -> AbsNote
> transpose p i = Pitch.toInt p + i
```

7. Armonía

```
> module Armonia(harmony,seqChords) where
> import Tipos
> import Basicas
> import qualified Haskore.Basic.Duration  as Duration
> import Ratio
```

En sí, obtener una secuncia de acordes a partir de la cancion ya parseada es muy simple: solo hay que concatenar los compases. Luego, para aproximarnos a algo que sea música, hay que asignarle a cada acorde una duración.

Para eso está la función withDurations : dado un compás, cuya duracion es una redonda, divide esta duración entre los acordes que lo componen. La división es equitativa, salvo cuando un compás esta compuesto de 3 acordes, donde se reparten las duraciones cortas al final.

```
> seqChords :: Song -> [(Chord,Duration.T)]
> seqChords = concatMap withDurations
>
> withDurations :: [Chord] -> [(Chord,Duration.T)]
> withDurations xs | length xs == 1          = map (\t -> (t,Duration.wn)) xs
>                  | length xs == 2          = map (\t -> (t,Duration.hn)) xs
>                  | length xs == 3          = (head xs,Duration.hn) :
>                                              (map (\t -> (t,Duration.qn)) (tail xs))
>                  | length xs == 4          = map (\t -> (t,Duration.qn)) xs
>                  | otherwise               = withDurations (take 4 xs)
```

Ahora solo resta obtener notas absolutas a partir de los acordes para tener la informacion concreta de qué se va a tocar, similar a la de una partitura.

```
> harmony :: [(Chord,Duration.T)] -> [([AbsNote],Duration.T)]
> harmony = map (\(x,y) -> (chordNotes x 1,y))
```

8. Walking Bass

```
> module Bajo(composeBass) where
> import Tipos
> import Basicas
> import Armonia
> import System.Random
> import qualified Haskore.Basic.Pitch  as Pitch
> import qualified Haskore.Basic.Duration  as Duration
> import Ratio
```

Primero defino unas funciones que nos serán útiles más adelante :

- leadTone toma una nota absoluta, y devuelve una lista de notas que se encuentran a un semitono de ella. Esto se hace para que el Walking Bass incorpore cromatismos (notas consecutivas que difieren en un semitono).
- cerca elimina aquellas notas que estén lejos (a muchos semitonos) de una nota dada. Evita que haya saltos demasiado grandes en la melodia.
- laMasCerca toma una nota n, una nota absoluta s y calcula cual es la nota absoluta perteneciente a la misma clase que n, que esté más cerca de s. Para lograrlo solo hace falta comparar con 2 notas ya que el bajo no toca notas fuera de las dos primeras octavas.
- oneOf elige un elemento al azar de una lista.

```
> leadTone :: AbsNote -> [AbsNote]
> leadTone s = [s-1,s-1,s-1,s+1,s+1,s+1]
>
> cerca :: Int -> AbsNote -> [AbsNote] -> [AbsNote]
```

```
> cerca maxinterval n ct = let r = filter (\x -> abs(n - x) < maxinterval) ct in
>                          if (null r) then (take 5 ct) else r
>
> laMasCerca :: Note -> AbsNote -> AbsNote
> laMasCerca n s = if (abs(n0 - s) < abs(n1 - s)) then n0 else n1
>                  where n0 = Pitch.toInt (0,n)
>                        n1 = Pitch.toInt (-1,n)
>
> oneof :: (RandomGen g) => g -> [a] -> (a,g)
> oneof g xs = (xs !! ri,rg)
>              where (ri,rg) = randomR (0,length xs - 1) g
```

Ahora ya podemos hacer un walking bass :

```
> walkingBass :: (RandomGen g) => g -> Int -> Note -> AbsNote -> [AbsNote] -> ([AbsNote],g)
> walkingBass g 1 t s _ = ([laMasCerca t s],g)
> walkingBass g n t s cn | mod n 2 == 0      = let (ne,g') = oneof g ((cerca 5 s cn) ++
>                                                                     leadTone s)
>                                                (w,g'') = walkingBass g' (n-1) t ne cn
>                                            in (w ++ [ne] , g'')
>
>                       | otherwise          = let (ne,g') = oneof g (cerca 9 s cn)
>                                                (w,g'') = walkingBass g' (n-1) t ne cn
>                                            in (w ++ [ne] , g'')
```

En un walking bass es importante saber cual es la primer nota que se debe tocar y cual es la ultima (un punto de partida y un objetivo de llegada). En este caso se contruye de atrás para adelante, es decir walkingBass g n t s cn, es igual a una secuencia de n notas tal que t es la nota por la que se empieza y s es la nota objetivo, donde las notas del acorde son cn. (Es decir, s se tocaría despues de la última nota del walking bass).

Ahora solo resta definir las notas candidatas a ser tocadas antes que s, de las cuales se elegira una al azar.

La función sigue dos reglas :

- Los tiempos impares son fuertes armonicamente, es decir, solo se tocan notas que pertenecen al acorde (alguna de las notas de cn)
- Los tiempos pares son fuertes rítmicamente, así que se pueden tocar tanto notas del acorde o también cromatismos (notas que difieren de un semitono)

Ahora que ya se tiene una funcion para caminar por un acorde dado, faltan definir las funciones que generen el bajo de toda una cancion :

Dado un acorde y su duracion, bassOverChord retorna las notas (negras = quarter notes = qn) que se tocaran sobre ese acorde

bassLine genera el bajo para una sucesión de acordes, y composeBass usa a bassLine para poder generarlo a partir de una canción. No es necesario que composeBass le de duración a cada nota, porque el estilo requiere que sean todas negras.

```
> bassOverChord :: (RandomGen g) => g -> Chord -> Duration.T -> AbsNote -> ([AbsNote],g)
> bassOverChord g c d dest = walkingBass g (fromInteger(Duration.divide d Duration.qn))
>                                        (bassnote c) dest (chordNotes c (-2))
>
> bassLine :: (RandomGen g) => g -> [(Chord,Duration.T)] -> [AbsNote]
> bassLine _ [] = []
```

```
> bassLine _ (x : [])     = [Pitch.toInt (-1,bassnote (fst x))]
> bassLine g (x : y : xs) = b ++ bassLine g' (y:xs)
>                           where (b,g') = bassOverChord g (fst x) (snd x)
>                                                          (Pitch.toInt (-1,bassnote (fst y)))
>
> composeBass :: (RandomGen g) => g -> Song -> [AbsNote]
> composeBass g s = bassLine g (seqChords s)
```

9. Haciendo que suene

```
> module Sonido(process,saveToFile) where
> import Tipos
> import Armonia
> import Bajo
> import System.Random
> import qualified Haskore.Basic.Pitch  as Pitch
> import qualified Haskore.Basic.Duration  as Duration
> import Ratio
> import qualified Haskore.Music.GeneralMIDI as MidiMusic
> import qualified Haskore.Melody         as Melody
> import qualified Haskore.Music          as Music
> import qualified Sound.MIDI.File.Save    as SaveMidi
> import qualified Haskore.Interface.MIDI.Render       as Render
```

Hasta ahora tenemos una idea de como obtener las notas absolutas que formaran parte de la salida. Pero para Haskore una lista de notas no es Música, asi que se debe generar algo de tipo Melody.T antes de poder pasar a un formato MIDI de salida (MidiMusic.T) . Mucho del código de esta sección fue sacado de la parte de ejemplos (Haskore.Example.Miscellaneous), para obtener música directamente de los datos ya obtenidos.

Primero se genera música a partir de una nota, luego a partir de un acorde, y despues a partir de una sucesión de acordes, fijando el tempo :

```
> makeNote :: Duration.T -> AbsNote -> Melody.T ()
> makeNote d p = Melody.note' n o d ()
>               where (o,n) = Pitch.fromInt p

> makePChord :: ([AbsNote],Duration.T) -> Melody.T ()
> makePChord (xs,d) = Music.chord (map (makeNote d) xs)

> acompiano :: [([AbsNote],Duration.T)] -> Integer -> Melody.T ()
> acompiano xs t = Music.changeTempo (t Duration.%+ 60) (Music.line (map makePChord xs))
```

Finalmente, se genera lo pasa a formato MIDI :

```
> pianoFromMusic :: Melody.T () -> MidiMusic.T
> pianoFromMusic = MidiMusic.fromMelodyNullAttr MidiMusic.AcousticGrandPiano
```

Para generar la musica del bajo se usan funciones similares, notar que Music.line secuencializa la musica que se va generando, mientras que Music.chord que se uso en el piano la paraleliza.

```
> musicBass :: [AbsNote] -> Integer -> Melody.T ()
> musicBass xs t = Music.changeTempo (t Duration.%+ 60)  (Music.line ((map (makeNote Duration.qn)
>                                                                      (map (makeNote Duration.wn)
>
> bassFromMelody :: Melody.T () -> MidiMusic.T
> bassFromMelody = MidiMusic.fromMelodyNullAttr MidiMusic.AcousticBass
```

La función final, process, junta todo, y usa el operador de composición paralela de Haskore, para que el piano y el bajo suenen simultáneamente.

```
> process :: (RandomGen g) => g -> Song -> Integer -> MidiMusic.T
> process g s t =   pianoFromMusic (acompiano (harmony (seqChords s) ) t) Music.=:=
>                   bassFromMelody (musicBass (composeBass g s) t)
>
> saveToFile :: String -> MidiMusic.T -> IO ()
> saveToFile f m = SaveMidi.toFile f (Render.generalMidiDeflt m)
```