Haskore Music Tutorial

Paul Hudak

Yale University
Department of Computer Science
New Haven, CT 06520
hudak@cs.yale.edu

1 Introduction

Haskore is a collection of Haskell modules designed for expressing musical structures in the high-level, declarative style of *functional programming.* In Haskore, musical objects consist of primitive notions such as notes and rests, operations to transform musical objects such as transpose and tempo-scaling, and operations to combine musical objects to form more complex ones, such as concurrent and sequential composition. From these simple roots, much richer musical ideas can easily be developed.

Haskore is a means for describing *music*—in particular Western Music—rather than *sound.* It is not a vehicle for synthesizing sound produced by musical instruments, for example, although it does capture the way certain (real or imagined) instruments permit control of dynamics and articulation.

Haskore also defines a notion of *literal performance* through which *observationally equivalent* musical objects can be determined. From this basis many useful properties can be proved, such as commutative, associative, and distributive properties of various operators. An *algebra of music* thus surfaces.

In fact a key aspect of Haskore is that objects represent both *abstract musical ideas* and their *concrete implementations.* This means that when we prove some property about an object, that property is true about the music in the abstract *and* about its implementation. Similarly, transformations that preserve musical meaning also preserve the behavior of their implementations. For this reason Haskell is often called an *executable specification language*; i.e. programs serve the role of mathematical specifications that are directly executable.

Building on the results of the functional programming community's Haskell effort has several important advantages: First, and most obvious, we can avoid the difficulties involved in new programming language design, and at the same time take advantage of the many years of effort that went into the design of Haskell. Second, the resulting system is both *extensible* (the user is free to add new features in substantive, creative ways) and *modifiable* (if the user doesn't like our approach to a particular musical idea, she is free to change it).

In the remainder of this paper I assume that the reader is familar with the basics of functional programming and Haskell in particular. If not, I encourage reading at least *A Gentle Introduction to Haskell* [HF92] before proceeding. I also assume some familiarity with *equational reasoning*; an excellent introductory text on this is [BW88].

2 The Architecture of Haskore

Figure 1 shows the overall structure of Haskore. Note the degree of independence of high level structures from the "music platform"—it is desirable for Haskore compositions to run equally well as conventional midi-files [IMA90], NeXT MusicKit score files [JB91], and csound score files [Ver86], and to print Haskore compositions in traditional notation using the CMN (Common Music Notation) subsystem. This independence is accomplished by having abstract notions of musical object, instrument, and performance that are eventually mapped down to a particular music platform. In this paper I will provide only the details of the mapping to Midi, since it is likely to be the most popular platform for users. In any case, of most interest is the box labeled "Haskore" in the diagram.

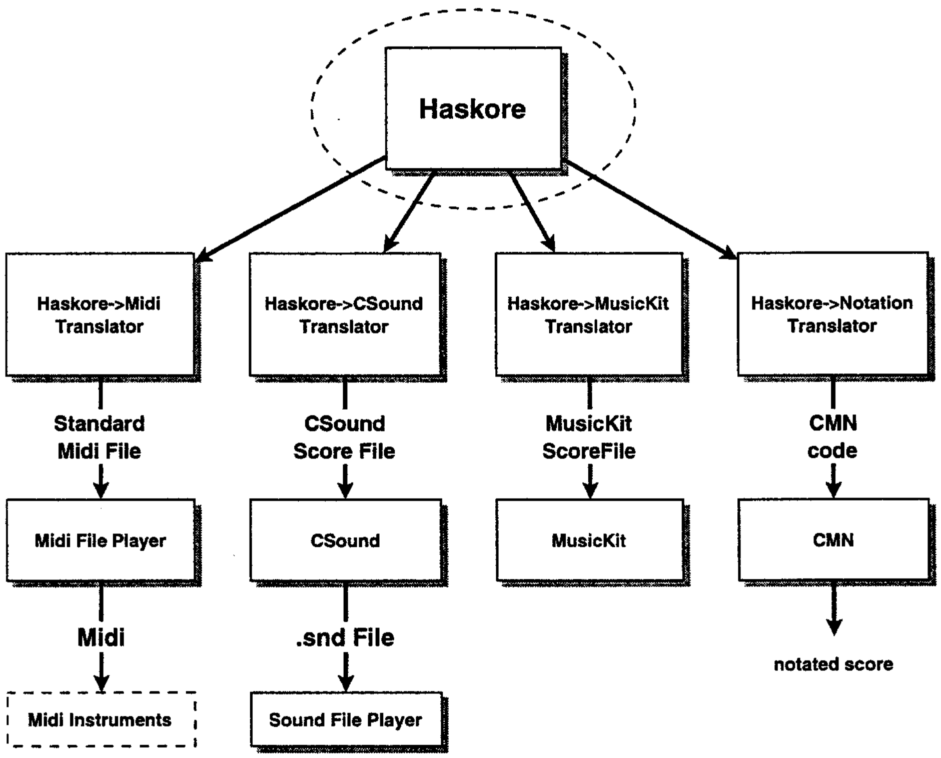


Fig. 1. Overall System Diagram

At the module level, Haskore is organized as follows:

```
> module Haskore (module Haskore, module Music, module Performance,
>                 module HaskToMidi) where    -- module Instruments,
>
> import Music           -- described in Section 3
> import Performance     -- described in Section 4
> -- import Instruments -- described in Section 5
> import HaskToMidi      -- described in Section 6
```

As I present various musical ideas in Haskell, I urge the reader to question, at every step, the decisions that I make. There is no supreme theory of music that dictates my decisions, and what I present is actually one of several versions that we have developed (this version is much richer than the one described in [HMGW94]; it is the "Haskore in practice" version alluded to in Section 6 of that paper). I believe this version is suitable for many practical purposes, but the reader may wish to modify it to better satisfy her intuitions and/or application.

This document was written in the *literate programming style*, and thus the LaTeX manuscript file from which it was generated is an *executable Haskell program*. It can be compiled under LaTeX in two ways: a basic mode provides all of the functionality that most users will need, and an extended mode in which various pieces of lower-level code are provided and documented as well (see file header for details). This version was compiled in basic mode. The document can be retrieved via WWW from:
`ftp://nebula.systemsz.cs.yale.edu/pub/yale-fp/papers/haskore`
(consult the README file for details).

The code conforms to Haskell 1.2, except in the module syntax, which conforms to Haskell 1.3. It has been tested under Hugs V1.01, Yale Release 0, which unfortunately does not yet support mutually recursive modules. For this reason all references to the module `Instruments` in this document are commented out, which in effect makes it part of module `Performance` (with which it is mutually recursive). Parts of the code should clearly be rewritten to take advantage of some Haskell 1.3 features, in particular "named fields" in datatype declarations.

3 The Basics

```
> module Music where
> infixr 5 :+:, :=:, :==
```

Perhaps the most basic musical idea is that of a *note*, which consists of a *pitch* and a *duration*:

```
> 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
>      deriving (Eq,Ord,Ix,Text)
> type Octave     = Int
> type Dur        = Float                -- in whole notes
```

So a `Pitch` is a pair consisting of a pitch class and an octave. Octaves are just integers, but we define a datatype for pitch classes, since distinguishing enharmonics (such as G# and Ab) may be important (especially for notation!). By convention, A440 = `(A,4)`.

Musical objects are captured by the `Music` datatype:[1]

```
> data Music = Note Pitch Dur [NoteAttribute]
>                                        -- a note \ atomic
>            | Rest Dur                  -- a rest /    objects
>            | Music :+: Music           -- sequential composition
>            | Music :=: Music           -- parallel composition
>            | Music :== Music           -- parallel comp (truncating)
>            | Tempo Int Int Music       -- scale the tempo
>            | Trans Int Music           -- transposition
>            | Instr IName Music         -- instrument label
>            | Phrase [PhraseAttribute] Music
>                                        -- phrase attributes
>     deriving Text
>
> type IName = String
```

Here a `Note` is its pitch paired with its duration (in number of whole notes), along with a list of `NoteAttributes` (defined later). A `Rest` also has a duration, but of course no pitch or other attributes.

From these two atomic constructors we can build more complex musical objects using the other constructors, as follows:

[1] I prefer to call these "musical objects" rather than "musical values" because the latter may be confused with musical aesthetics.

- `m1 :+: m2` is the sequential composition of `m1` and `m2`; i.e. `m1` and `m2` are played in sequence.
  `m1 :=: m2` is the parallel composition of `m1` and `m2`; i.e. `m1` and `m2` are played simultaneously.
- `Tempo a b m` scales the rate at which `m` is played (i.e. its tempo) by a factor of `a/b`.
- `Trans i m` transposes `m` by interval `i` (in semitones).
- `Instr iname m` declares that `m` is to be played using instrument `iname`.
- `Phrase pas m` declares that `m` is to be played using the phrase attributes (described later) in the list `pas`.

It is convenient to represent these ideas in Haskell as a recursive datatype because we wish to not only construct musical objects, but also take them apart, analyze their structure, print them in a structure-preserving way, interpret them for performance purposes, etc.

3.1 Convenient Auxiliary Functions

For convenience we first create a few names for familiar notes, durations, and rests, as shown in figure 2. Treating pitches as integers is also useful in many settings, so we define some functions for converting between `Pitch` values and `AbsPitch` values (integers). These also are shown in figure 2, along with a definition of `trans`, which transposes pitches (analogous to `Trans`, which transposes values of type `Music`).

Exercise 1. Show that `abspitch . pitch = id`, and, up to enharmonic equivalences,
`pitch . abspitch = id`.

Exercise 2. Show that `trans i (trans j p) = trans (i+j) p`.

3.2 Some Simple Examples

With this modest beginning, we can already express quite a few musical relationships simply and effectively. For example, two common ideas in music are the construction of notes in a horizontal fashion (a *line* or *melody*), and in a vertical fashion (a *chord*):

```
> line, chord :: [Music] -> Music
> line  = foldr (:+:) (Rest 0)
> chord = foldr (:=:) (Rest 0)
> cf,c,cs,df,d,ds,ef,e,es,ff,f,fs,gf,g,gs,af,a,as,bf,b,bs ::
>    Octave -> Dur -> [NoteAttribute] -> Music
>
> cf o = Note (Cf,o);  c o = Note (C,o);  cs o = Note (Cs,o)
> df o = Note (Df,o);  d o = Note (D,o);  ds o = Note (Ds,o)
> ef o = Note (Ef,o);  e o = Note (E,o);  es o = Note (Es,o)
> ff o = Note (Ff,o);  f o = Note (F,o);  fs o = Note (Fs,o)
> gf o = Note (Gf,o);  g o = Note (G,o);  gs o = Note (Gs,o)
> af o = Note (Af,o);  a o = Note (A,o);  as o = Note (As,o)
> bf o = Note (Bf,o);  b o = Note (B,o);  bs o = Note (Bs,o)
>
> wn,  hn,  qn,  en,  sn,  tn  :: Dur
> wnr, hnr, qnr, enr, snr, tnr :: Music
>
> wn = 1          ; wnr = Rest wn      -- whole note rest
> hn = 1/2        ; hnr = Rest hn      -- half note rest
> qn = 1/4        ; qnr = Rest qn      -- quarter note rest
> en = 1/8        ; enr = Rest en      -- eight note rest
> sn = 1/16       ; snr = Rest sn      -- sixteenth note rest
> tn = 1/32       ; tnr = Rest tn      -- thirty-second note rest
>
> pitchClass :: PitchClass -> Int
>
> pitchClass pc = case pc of
>      Cf -> -1;  C -> 0;  Cs -> 1    -- or should Cf be 11?
>      Df -> 1;   D -> 2;  Ds -> 3
>      Ef -> 3;   E -> 4;  Es -> 5
>      Ff -> 4;   F -> 5;  Fs -> 6
>      Gf -> 6;   G -> 7;  Gs -> 8
>      Af -> 8;   A -> 9;  As -> 10
>      Bf -> 10;  B -> 11; Bs -> 12   -- or should Bs be 0?
>
> type AbsPitch = Int
>
> absPitch :: Pitch -> AbsPitch
> absPitch (pc,oct) = 12*oct + pitchClass pc
>
> pitch    :: AbsPitch -> Pitch
> pitch    ap       = ( [C,Cs,D,Ds,E,F,Fs,G,Gs,A,As,B] !! mod ap 12,
>                       quot ap 12)
>
> trans    :: Int -> Pitch -> Pitch
> trans i p = pitch (absPitch p + i)
```

Fig. 2. Convenient note names and pitch conversion functions.

From the notes in the C major triad in register 4, I can now construct a C major arpeggio and chord as well:

```
> cMaj = map (\f->f 4 qn []) [c, e, g]  -- octave 4, quarter notes
>
> cMajArp = line  cMaj
> cMajChd = chord cMaj
```

Suppose now we wish to describe a melody `m` accompanied by an identical voice a perfect 5th higher. In Haskore we simply write "`m :=: Trans 7 m`." Similarly, a canon-like structure involving `m` can be expressed as "`m :=: delay d m`," where:

```
> delay :: Dur -> Music -> Music
> delay d m = Rest d :+: m
```

Of course, Haskell's non-strict semantics also allows us to define infinite musical objects. For example, a musical object may be repeated *ad nauseum* using this simple function:

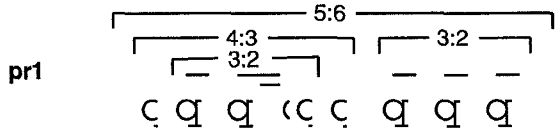
```
> repeatM :: Music -> Music
> repeatM m = m :+: repeatM m
```

Thus an infinite ostinato can be expressed in this way, and then used in different contexts that extract only the portion that's actually needed.

The notions of inversion, retrograde, retrograde inversion, etc. used in 12-tone theory are also easily captured in Haskore. First let's define a transformation from a line created by `line` to a list:

```
> lineToList :: Music -> [Music]
> lineToList n@(Rest 0) = []
> lineToList (n :+: ns) = n : lineToList ns
>
> retro, invert, retroInvert, invertRetro :: Music -> Music
> retro    = line . reverse . lineToList
> invert m = line (map inv l)
>   where l@(Note r _ _: _)  = lineToList m
>         inv (Note p d nas) = Note (pitch (2*(absPitch r)
>                                          - absPitch p)) d nas
>         inv (Rest d)       = Rest d
> retroInvert = retro  . invert
> invertRetro = invert . retro
```

Exercise 3. Show that "`retro . retro`," "`invert . invert`," and "`retroInvert . invertRetro`" are the identity on values created by `line`.



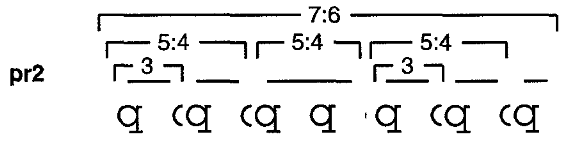


Fig. 3. Nested Polyrhythms

For some rhythmical ideas, consider first a simple *triplet* of eighth notes; it can be expressed as "`Tempo 3 2 m`," where `m` is a line of 3 eighth notes. So in fact `Tempo` can be used to create quite complex rhythmical patterns. For example, consider the "nested polyrhythms" shown in Figure 3. They can be expressed quite naturally in Haskore as follows (note the use of the *where* clause in `pr2` to capture recurring phrases):

```
> pr1, pr2 :: Pitch -> Music
> pr1 p = Tempo 5 6 (Tempo 4 3 (mkLn 1 p qn :+:
>                               Tempo 3 2 (mkLn 3 p en :+:
>                                          mkLn 2 p sn :+:
>                                          mkLn 1 p qn    ) :+:
>                               mkLn 1 p qn) :+:
>                    Tempo 3 2 (mkLn 6 p en))
>
> pr2 p = Tempo 7 6 (m1 :+:
>                    Tempo 5 4 (mkLn 5 p en) :+:
>                    m1 :+:
>                    mkLn 2 p en)
>         where m1 = Tempo 5 4 (Tempo 3 2 m2 :+: m2)
>               m2 = mkLn 3 p en
>
> mkLn n p d = line (take n (repeat (Note p d [])))
```

To play polyrhythms pr1 and pr2 in parallel using middle C and middle G, respectively, we would do the following (middle C is in the 5th octave):

```
> pr12 :: Music
> pr12 = pr1 (C,5) :=: pr2 (G,5)
```

As a final example in this section, we can can compute the duration in beats of a musical object, a notion we will need in Section 4, as follows:

```
> dur :: Music -> Dur
>
> dur (Note _ d _)    = d
> dur (Rest d)        = d
> dur (m1 :+: m2)     = dur m1   +   dur m2
> dur (m1 :=: m2)     = dur m1 `max` dur m2
> dur (Tempo a b m) = dur m * float b / float a
> dur (Trans  _  m) = dur m
> dur (Instr  i  m) = dur m
> dur (Phrase _  m) = dur m
>
> float = fromInteger . toInteger :: Int -> Float
```

Exercise 4. Find a simple piece of music written by your favorite composer, and transcribe it into Haskore. In doing so, look for repeating patterns, transposed phrases, etc. and reflect this in your code, thus revealing deeper structural aspects of the music than that found in common practice notation.

Appendix C shows the first 28 bars of Chick Corea's "Children's Song No. 6" encoded in Haskore.

3.3 Phrasing and Articulation

Recall that the `Note` constructor contained a field of `NoteAttributes`. These are values that are attached to notes for the purpose of notation or musical interpretation. Likewise, the `Phrase` constructor permits one to annotate an entire musical object with `PhraseAttributes`. These two attribute datatypes cover a wide range of attributions found in common practice notation, and are shown in figure 4. Beware that use of them requires use of an instrument that knows how to interpret them! This will be described in more detail in Section 5.

Note that some of the attributes are parameterized with a numeric value. This is used by an instrument to control the degree to which an articulation is to be applied. For example, we would expect `Legato 1.2` to create more of a legato feel than `Legato 1.1`. The following constants represent default values for some of the parameterized attributes:

```
> data NoteAttribute = Volume Float   -- by convention: 0=min, 100=max
>                    | Fingering Int
>                    | Dynamics String
>      deriving Text
>
> data PhraseAttribute = Dyn Dynamic
>                      | Art Articulation
>                      | Orn Ornament
>      deriving Text
>
> data Dynamic = Accent Float | Crescendo Float | Diminuendo Float
>      deriving Text
>
> data Articulation = Staccato Float | Legato Float | Slurred
>                   | Tenuto | Marcato | Pedal | Fermata
>                   | FermataDown | Breath | DownBow | UpBow
>                   | Harmonic | Pizzicato | LeftPizz
>                   | BartokPizz | Swell | Wedge | Thumb | Stopped
>      deriving Text
>
> data Ornament = Trill | Mordent | InvMordent | DoubleMordent
>               | Turn | TrilledTurn | ShortTrill
>               | Arpeggio | ArpeggioUp | ArpeggioDown
>               | Instruction String | Head NoteHead
>      deriving Text
>
> data NoteHead = DiamondHead | SquareHead | XHead | TriangleHead
>               | TremoloHead | SlashHead | ArtHarmonic | NoHead
>      deriving Text
```

Fig. 4. Note and Phrase Attributes.

```
> legato, staccato  :: Articulation
> accent, bigAccent :: Dynamic
>
> legato    = Legato 1.1
> staccato  = Staccato 0.5
> accent    = Accent 1.2
> bigAccent = Accent 1.5
```

To understand exactly how an instrument interprets an attribute requires knowing how instruments are defined. Haskore defines only a few standard instruments, so in fact many of the attributes in figure 4 are to allow the user to give appropriate interpretations of them for her instrument. But before looking at the structure of instruments we will need to look at the notion of a *performance* (these two ideas are tightly linked, which is why the `Instruments` and `Performance` modules are mutually recursive).

4 Interpretation and Performance

```
> module Performance (module Performance, module Music)
>                                               -- module Instruments
>        where
>
> import Music
> -- import Instruments
```

Now that we have defined the structure of musical objects, let us turn to the issue of *performance*, which we define as a temporally ordered sequence of musical *events*:

```
> type Performance = [Event]
>
> data Event = Event Time IName AbsPitch DurT Volume
>      deriving (Eq,Ord,Text)
>
> type Time      = Float
> type DurT      = Float
> type Volume    = Float
```

An event is the lowest of our music representations not yet committed to Midi, csound, or the MusicKit. An event `Event s i p d v` captures the fact that at start time `s`, instrument `i` plays pitch `p` with volume `v` for a duration `d` (where now duration is measured in seconds, rather than beats).

To generate a complete performance of, i.e. give an interpretation to, a musical object, we must know the time to begin the performance, and the proper volume, key and tempo. We must also know what *instruments* to use; that is, we need a mapping from the `INames` in an abstract musical object to the actual instruments to be used.

We can thus model a performer as a function `perform` which maps all of this information and a musical object into a performance:

```
> perform :: IMap -> Context -> Music -> Performance
>
```

```
> type IMap    = IName -> Instrument
> type Context = (Time,Instrument,DurT,Key,Volume)
> type Key     = AbsPitch
>
> perform imap c@(t,i,dt,k,v) m =
>   case m of
>     Note p d nas -> playNote i c p d nas
>     Rest d       -> []
>     m1 :+: m2    -> perform imap c m1 ++
>                     perform imap (setTime  c (t+(dur m1)*dt)) m2
>     m1 :=: m2    -> merge (perform imap c m1) (perform imap c m2)
>     Tempo a b m  -> perform imap
>                          (setTempo c (dt * float b / float a)) m
>     Trans p m    -> perform imap (setTrans c (k+p)) m
>     Instr nm m   -> perform imap (setInst  c (imap nm)) m
>     Phrase pas m -> interpPhrase i imap c pas m
```

Some things to note:

1. The `Context` is the running "state" of the performance, and gets updated in several different ways. For example, the interpretation of the `Instr` constructor uses `imap` to determine the actual instrument from the instrument name, and updates the instrument field of the context accordingly. Figure 5 defines a convenient group of selectors and mutators for contexts and events.
2. Interpretation of notes and phrases is instrument dependent. Ultimately a single note is played by the `playNote` function, which takes the instrument as an argument. Similarly, phrase interpretation is also instrument dependent, reflected in the use of `interpPhrase`. How these functions work with instruments is described in section 5.
3. The `DurT` component of the context is the duration, in seconds, of one whole note. To make it easier to compute, we can define a "metronome" function that, given a standard metronome marking (in beats per minute) and the note type associated with one beat (quarter note, eighth note, etc.) generates the duration of one whole note:

   ```
   > metro :: Float -> Dur -> DurT
   > metro setting dur = 60 / (setting*dur)
   ```

   Thus, for example, `metro 96 qn` sets a tempo of 96 quarter notes per minute.
4. In the treatment of `(:+:)`, note that the sub-sequences are appended together, with the start time of the second argument delayed by the duration of the first. The function `dur` (defined in Section 3.2) is used to compute this duration. Note that this results in a quadratic time complexity for `perform`. A

```
setTime, setInst, setTempo, setTrans, and setVolume
have type:  Context -> X -> Context, where X is obvious.

> setTime    (t,i,dt,k,v) t'  = (t',i,dt,k,v)
> setInst    (t,i,dt,k,v) i'  = (t,i',dt,k,v)
> setTempo   (t,i,dt,k,v) dt' = (t,i,dt',k,v)
> setTrans   (t,i,dt,k,v) k'  = (t,i,dt,k',v)
> setVolume  (t,i,dt,k,v) v'  = (t,i,dt,k,v')

getEventTime, getEventInst, getEventPitch, getEventDur, and getEventVol
have type:  Event -> X, where X is obvious

> getEventTime  (Event t _ _ _ _) = t
> getEventInst  (Event _ i _ _ _) = i
> getEventPitch (Event _ _ p _ _) = p
> getEventDur   (Event _ _ _ d _) = d
> getEventVol   (Event _ _ _ _ v) = v

setEventTime, setEventInst, setEventPitch, setEventDur, and setEventVol
have type:  Event -> X -> Event, where X is obvious.

> setEventTime  (Event t i p d v) t' = Event t' i  p  d  v
> setEventInst  (Event t i p d v) i' = Event t  i' p  d  v
> setEventPitch (Event t i p d v) p' = Event t  i  p' d  v
> setEventDur   (Event t i p d v) d' = Event t  i  p  d' v
> setEventVol   (Event t i p d v) v' = Event t  i  p  d  v'
```

Fig. 5. Selectors and mutators for contexts and events.

more efficient solution is to have `perform` compute the duration directly, returning it as part of its result. The current approach is pedagogically clearer, however, and thus we leave the optimization as an exercise for the reader.

5. In contrast, the sub-sequences derived from the arguments to `(:=:)` are merged into a time-ordered stream. The definition of `merge` is given below.

```
> merge :: Performance -> Performance -> Performance
>
> merge a@(e1:es1) b@(e2:es2) =
>       if e1 < e2 then e1 : merge es1 b
>                  else e2 : merge a es2
> merge [] es2 = es2
> merge es1 [] = es1
```

Note that `merge` compares entire events rather than just start times. This is to ensure that it is commutative, a desirable condition for some of the proofs used in Section 8.

5 Instruments

```
module Instruments (module Instruments, module Music,
                                                module Performance)
       where

import Music
import Performance
```

In the last section we saw how a performance involved the notion of an *instrument*. The reason for this is the same as for real performers and their instruments: many of the note and phrase attributes (see Section 3.3) are instrument dependent. For example, how many, and which staves should be used to notate an instrument? And how should "legato" be interpreted in a performance? To answer these questions, Haskore has a notion of instrument which "knows" about differences with respect to performance and notation. A pianist, for example, realizes legato in a way fundamentally different from the way a violinist does, precisely because of differences in their instruments.

A Haskore instrument is a 4-tuple consisting of a name and 3 functions: one for interpreting notes, one for phrases, and one for producing a properly notated score.

```
> data Instrument = Inst IName NoteFun PhraseFun NotateFun
>
> type NoteFun   = Context -> Pitch -> Dur -> [NoteAttribute]
>                                            -> Performance
> type PhraseFun = IMap -> Context -> [PhraseAttribute]
>                                   -> Music -> Performance
> type NotateFun = ()
```

The last line above is temporary for this executable version of Haskore, since notation only works on systems supporting CMN. The real definition should read:

```
type NotateFun = [Glyph] -> Staff
```

Note that both `NoteFun` and `PhraseFun` return a `Performance` (imported from module `Perform`), whereas `NotateFun` returns a `Staff` (imported from module `Notation`).

For convenience we define:

```
> iName         ::  Instrument -> IName
> iName         (Inst nm _ _ _) = nm
>
> playNote      ::  Instrument -> NoteFun
> playNote      (Inst _ nf _ _) = nf
>
> interpPhrase ::  Instrument -> PhraseFun
> interpPhrase (Inst _ _ pf _) = pf
>
> notateInst    ::  Instrument -> NotateFun
> notateInst    (Inst _ _ _ nf) = nf
```

5.1 Examples of Instrument Construction

A "default instrument" called `defInst` is defined for use when none other is specified in the score; it also functions as a base from which other instruments can be derived. `defInst` responds only to the `Volume` note attribute and to the `Accent`, `Staccato`, and `Legato` phrase attributes. It is defined in figure 6. Before reading this code, recall how instruments are invoked by the `perform` function defined in the last section. Then note:

1. `defPlayNote` is the only function (even in the definition of `perform`) that actually generates an event. It also modifies that event based on an interpretation of each note attribute by the function `defHasHandler`.
2. `defNasHandler` only recognizes the `Volume` attribute, which it uses to set the event volume accordingly.
3. `defInterpPhrase` calls (mutually recursively) `perform` to interpret a phrase, and then modifies the result based on an interpretation of each phrase attribute by the function `defPasHandler`.
4. `defPasHandler` only recognizes the `Accent`, `Staccato`, and `Legato` phrase attributes. For each of these it uses the numeric argument as a "scaling" factor of the volume (for `Accent`) and duration (for `Staccato` and `Lagato`). Thus `(Phrase [Legato 1.1] m)` effectively increases the duration of each note in `m` by 10%.

It should be clear that much of the code in figure 6 can be re-used in defining a new instrument. For example, to define an instrument `weird` that interprets note attributes just like `defInst` but behaves differently with respect to phrase attributes, we could write:

```
weird :: Instrument
weird = Inst "Default" (defPlayNote     defNasHandler)
                       (defInterpPhrase  myPasHandler)
                       (defNotateInst    ()           )
```

```
> defInst :: Instrument
> defInst = Inst "Default" (defPlayNote     defNasHandler)
>                          (defInterpPhrase defPasHandler)
>                          (defNotateInst   ()           )
>
> defPlayNote :: (Context->NoteAttribute->Event->Event) -> NoteFun
> defPlayNote nasHandler c@(t,i,dt,k,v) p d nas =
>       [ foldr (nasHandler c)
>               (Event t (iName i) (absPitch p + k) (d*dt) v)
>               nas ]
>
> defNasHandler :: Context-> NoteAttribute -> Event -> Event
> defNasHandler (_,_,_,_,v) (Volume v') ev = setEventVol ev (v*v'/100.0)
> defNasHandler _           _           ev = ev
>
> defInterpPhrase :: (PhraseAttribute->Performance->Performance)
>                                                        -> PhraseFun
> defInterpPhrase pasHandler imap c@(t,i,dt,k,v) pas m =
>       foldr pasHandler
>             (perform imap c m)
>             pas
>
> defPasHandler :: PhraseAttribute -> Performance -> Performance
> defPasHandler (Dyn (Accent x))      pf =
>       map (\e -> setEventVol e (x * getEventVol e)) pf
> defPasHandler (Art (Staccato x))    pf =
>       map (\e -> setEventDur e (x * getEventDur e)) pf
> defPasHandler (Art (Legato   x))    pf =
>       map (\e -> setEventDur e (x * getEventDur e)) pf
> defPasHandler _                     pf = pf
>
> defNotateInst :: () -> NotateFun
> defNotateInst    _  =  ()
```

Fig. 6. Definition of default instrument `defInst`.

and then supply a suitable definition of `myPasHandler`. That definition could also re-use code, in the following sense: suppose we wish to add an interpretation for `Crescendo`, but otherwise have `myPasHandler` behave just like `defPasHandler`.

```
myPasHandler :: PhraseAttribute -> Performance -> Performance
myPasHandler (Dyn (Crescendo x)) pf = ...
myPasHandler  pa                 pf = defPasHandler pa pf
```

Exercise 5. Fill in the ... in the definition of `myPasHandler` according to the following strategy: Assume $0 < \mathtt{x} < 1$. Gradually scale the volume of each event by a factor of 1.0 through $1.0 + \mathtt{x}$, using linear interpolation.

Exercise 6. Choose some of the other phrase attributes and provide interpretations of them, such as **Diminuendo**, **Slurred**, **Trill**, etc.

In a system that supports it, the default notation handler sets up a staff with a treble clef for the instrument and appends any glyphs to the end of the staff:

```
defNotateInst gs = Staff "Default" 1.0 5 (Clef Treble : gs)
```

6 Midi

Midi ("musical instrument digital interface") is a standard protocol adopted by most, if not all, manufacturers of electronic instruments. At its core is a protocol for communicating *musical events* (note on, note off, key press, etc.) as well as so-called *meta events* (select synthesizer patch, change volume, etc.). Beyond the logical protocol, the Midi standard also specifies electrical signal characteristics and cabling details. In addition, it specifies what is known as a *standard Midi file* which any Midi-compatible software package should be able to recognize.

Over the years musicians and manufacturers decided that they also wanted a standard way to refer to *common* or *general* instruments such as "acoustic grand piano," "electric piano," "violin," and "acoustic bass," as well as more exotic ones such as "chorus aahs," "voice oohs," "bird tweet," and "helicopter." A simple standard known as *General Midi* was developed to fill this role. It is nothing more than an agreed-upon list of instrument names along with a *program patch number* for each, a parameter in the Midi standard that is used to select a Midi instrument's sound.

Most "sound-blaster"-like sound cards on conventional PC's know about Midi, as well as general Midi. However, the sound generated by such modules, and the sound produced from the typically-scrawny speakers on most PC's, is often quite poor. It is best to use an outboard keyboard or tone generator, which are attached to a computer via a Midi interface and cables. It is possible to connect several Midi instruments to the same computer, with each assigned a different *channel.* Modern keyboards and tone generators are quite amazing little beasts. Not only is the sound quite good (when played on a good stereo system), but they are also usually *multi-timbral*, which means they are able to generate many different sounds simultaneously, as well as *polyphonic*, meaning that simultaneous instantiations of the same sound are possible.

Haskore provides a way to specify a Midi channel number and general Midi instrument selection for each **IName** in a Haskore composition. It also provides a means to generate a standard Midi file, which can then be played using any conventional Midi software. In this section the top-level code needed by the user to invoke this functionality will be described; the extended document contains all of the gory details.

```
> module HaskToMidi (module HaskToMidi, module GeneralMidi,
>                                                   module MidiFile)
>        where
>
> import Music
> import Performance
> import MidiFile
> import GeneralMidi
> import List(partition)
```

Intead of converting a Haskore `Performance` directly into a Midi file, Haskore first converts it into a datatype that *represents* a Midi file, which is then written to a file in a separate pass. This separation of concerns makes the structure of the Midi file clearer, makes debugging easier, and provides a natural path for extending Haskore's functionality with direct Midi capability (in fact there is a version of Haskore that does this under Windows '95, but it is not described here).

A `UserPatchMap` is a user-supplied table for mapping instrument names (`IName`'s) to Midi channels and General Midi patch names. The patch names are by default General Midi names, although the user can also provide a `PatchMap` for mapping Patch Names to unconventional Midi Program Change numbers.

```
> type UserPatchMap = [(IName,GenMidiName,MidiChannel)]
```

Given a `UserPatchMap`, a performance is converted to a datatype representing a standard Midi file using the `performToMidi` function.

```
> performToMidi :: Performance -> UserPatchMap -> MidiFile
```

A table of General Midi assignments is imported from `GeneralMidi`, which is contained in Appendix A. The Midi file datatype itself and functions for writing it to files are imported from the module `MidiFile`, briefly described below. The remaining details of this code are omitted in the basic version of this document.

```
> module MidiFile where
>
> import Monads(Output, runO, outO)
> import MonadUtils(zeroOrMore, oneOrMore)
> import Utils(unlinesS, rightS, concatS)
```

`OutputMidiFile` is the main function for writing `MidiFile` values to an actual file; its first argument is the filename:

```
> outputMidiFile :: String -> MidiFile -> IO ()
> outputMidiFile fn mf = writeFile fn (midiFileToString mf)
```

Exercise 7. Take as many examples as you like from the previous sections, create one or more `UserPatchMaps`, write the examples to a file, and play them using a conventional Midi player.

Appendix B defines some functions which should make the above exercise easier.

7 Notating Chords

I have described how to represent chords as values of type `Music`. However, sometimes it is convenient to treat chords more abstractly. Rather than think of a chord in terms of its actual notes, it is useful to think of it in terms of its chord "quality," coupled with the key it is played in and the particular voicing used. For example, we can describe a chord as being a "major triad in root position, with root middle C." Several approaches have been put forth for representing this information, and we cannot cover all of them here. Rather, I will describe two basic representations, leaving other alternatives to the skill and imagination of the reader.[2]

First, one could use a *pitch* representation, where each note is represented as its distance from some fixed pitch. 0 is the obvious fixed pitch to use, and thus, for example, `[0,4,7]` represents a major triad in root position. The first zero is in some sense redundant, of course, but it serves to remind us that the chord is in "normal form." For example, when forming and transforming chords, we may end up with a representation such as `[2,6,9]`, which is not normalized; its normal form is in fact `[0,4,7]`. Thus we define:

> A chord is in *pitch normal form* if the first pitch is zero, and the subsequent pitches are monotonically increasing.

One could also represent a chord *intervalically*; i.e. as a sequence of intervals. A major triad in root position, for example, would be represented as `[4,3,-7]`, where the last interval "returns" us to the "origin." Like the `0` in the pitch representation, the last interval is redundant, but allows us to define another sense of normal form:

> A chord is in *interval normal form* if the intervals are all greater than zero, except for the last which must be equal to the negation of the sum of the others.

In either case, we can define a chord type as:

[2] For example, Forte prescribes normal forms for chords in an atonal setting [For73].

```
> type Chord = [AbsPitch]
```

We might ask whether there is some advantage, computationally, of using one of these representations over the other. However, there is an invertible linear transformation between them, as defined by the following functions, and thus there is in fact little advantage of one over the other:

```
> pitToInt :: Chord -> Chord
> pitToInt ch = aux ch
>    where aux (n1:n2:ns) = (n2-n1) : aux (n2:ns)
>          aux [n]        = [head ch - n]
>
> intToPit :: Chord -> Chord
> intToPit ch = 0 : aux 0 ch
>    where aux p [n]    = []
>          aux p (n:ns) = n' : aux n' ns  where n' = p+n
```

Exercise 8. Show that `pitToInt` and `intToPit` are *inverses* in the following sense: for any chord `ch1` in pitch normal form, and `ch2` in interval normal form, each of length at least two:

```
intToPit (pitToInt ch1) = ch1
pitToInt (intToPit ch2) = ch2
```

Another operation we may wish to perform is a test for *equality* on chords, which can be done at many levels: based only on chord quality, taking inversion into account, absolute equality, etc. Since the above normal forms guarantee a unique representation, equality of chords with respect to chord quality and inversion is simple: it is just the standard (overloaded) equality operator on lists. On the other hand, to measure equality based on chord quality alone, we need to account for the notion of an *inversion.*

Using the pitch representation, the inversion of a chord can be defined as follows:

```
> pitInvert (p1:p2:ps) = 0 : map (subtract p2) ps ++ [12-p2]
```

Although we could also directly define a function to invert a chord given in interval representation, we will simply define it in terms of functions already defined:

```
> intInvert = pitToInt . pitInvert . intToPit
```

We can now determine whether a chord in normal form has the same quality (but possibly different inversion) as another chord in normal form, as follows: simply test whether one chord is equal either to the other chord or to one of its inversions. Since there is only a finite number of inversions, this is well defined. In Haskell:

```
> samePitChord ch1 ch2 =
>  let invs = take (length ch1) (iterate pitInvert ch1)
>  in  or (map (==ch2) invs)
>
> sameIntChord ch1 ch2 =
>  let invs = take (length ch1) (iterate intInvert ch1)
>  in  or (map (==ch2) invs)
```

For example, `samePitChord [0,4,7] [0,5,9]` returns `True` (since `[0,5,9]` is the pitch normal form for the second inversion of `[0,4,7]`).

8 Equivalence of Literal Performances

A *literal performance* is one in which no aesthetic interpretation is given to a musical object. The function `perform` in fact yields a literal performance; aesthetic nuances must be expressed explicitly using note and phrase attributes.

There are many musical objects whose literal performances we expect to be *equivalent*. For example, the following two musical objects are certainly not equal as data structures, but we would expect their literal performances to be identical:

```
(m1 :+: m2) :+: (m3 :+: m4)
 m1 :+: m2 :+: m3 :+: m4
```

Thus we define a notion of equivalence:

Definition: Two musical objects `m1` and `m2` are *equivalent*, written `m1` $\equiv$ `m2`, if and only if:

$$(\forall \texttt{imap,c})\quad \texttt{perform imap c m1 = perform imap c m2}$$

where "=" is equality on values (which in Haskell is defined by the underlying equational logic).

One of the most useful things we can do with this notion of equivalence is establish the validity of certain *transformations* on musical objects. A transformation is *valid* if the result of the transformation is equivalent (in the sense defined above) to the original musical object; i.e. it is "meaning preserving."

The most basic of these transformation we treat as *axioms* in an *algebra of music*. For example:

Axiom 1 *For any* `r1`, `r2`, `r3`, `r4`, *and* `m`*:*

```
Tempo r1 r2 (Tempo r3 r4 m)  ≡  Tempo (r1*r3) (r2*r4) m
```

To prove this axiom, we use conventional equational reasoning (for clarity we omit `imap` and simplify the context to just `dt`):

Proof:

```
perform dt (Tempo r1 r2 (Tempo r3 r4 m))
= perform (r2*dt/r1) (Tempo r3 r4 m)          -- unfolding perform
= perform (r4*(r2*dt/r1)/r3) m                -- unfolding perform
= perform ((r2*r4)*dt/(r1*r3)) m              -- simple arithmetic
= perform dt (Tempo (r1*r3) (r2*r4) m)        -- folding perform
```

Here is another useful transformation and its validity proof (for clarity in the proof we omit `imap` and simplify the context to just `(t,dt)`):

Axiom 2 *For any* `r1`, `r2`, `m1`, *and* `m2`*:*

```
Tempo r1 r2 (m1 :+: m2)  ≡  Tempo r1 r2 m1 :+: Tempo r1 r2 m2
```

In other words, *tempo scaling distributes over sequential composition.*

Proof:

```
perform (t,dt) (Tempo r1 r2 (m1 :+: m2))
= perform (t,r2*dt/r1) (m1 :+: m2)              -- unfold perform
= perform (t,r2*dt/r1) m1 ++ perform (t',r2*dt/r1) m2
                                                -- unfold perform
= perform (t,dt) (Tempo r1 r2 m1) ++
            perform (t',dt) (Tempo r1 r2 m2)    -- fold perform
= perform (t,dt) (Tempo r1 r2 m1) ++
            perform (t'',dt) (Tempo r1 r2 m2)   -- fold dur
= perform (t,dt) (Tempo r1 r2 m1 :+: Tempo r1 r2 m2)
                                                -- fold perform
where t'  = t + (dur m1)*r2*dt/r1
      t'' = t + (dur (Tempo r1 r2 m1))*dt
```

An even simpler axiom is given by:

Axiom 3 *For any* `r` *and* `m`*:*

```
Tempo r r m  ≡  m
```

In other words, *unit tempo scaling is the identity.*

Proof:

```
perform (t,dt) (Tempo r r m)
= perform (t,r*dt/r) m                          -- unfold perform
= perform (t,dt) m                              -- simple arithmetic
```

Note that the above proofs, being used to establish axioms, all involve the definition of `perform`. In contrast, we can also establish *theorems* whose proofs involve only the axioms. For example, Axioms 1, 2, and 3 are all needed to prove the following:

Theorem 9. *For any* `r1`*,* `r2`*,* `m1`*, and* `m2`*:*

```
  Tempo r1 r2 m1 :+: m2  ≡  Tempo r1 r2 (m1 :+: Tempo r2 r1 m2)
```

Proof:

```
Tempo r1 r2 (m1 :+: Tempo r2 r1 m2)
= Tempo r1 r2 m1 :+: Tempo r1 r2 (Tempo r2 r1 m2)  -- by Axiom 1
= Tempo r1 r2 m1 :+: Tempo (r1*r2) (r2*r1) m2      -- by Axiom 2
= Tempo r1 r2 m1 :+: Tempo (r1*r2) (r1*r2) m2      -- arithmetics
= Tempo r1 r2 m1 :+: m2                            -- by Axiom 3
```

For example, this fact justifies the equivalence of the two phrases shown in Figure 7.

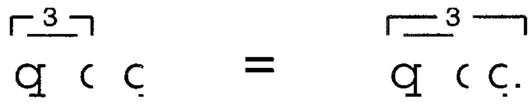


Fig. 7. Equivalent Phrases

Many other interesting transformations of Haskore musical objects can be stated and proved correct using equational reasoning. We leave as an exercise for the reader the proof of the following axioms (which include the above axioms as special cases).

Axiom 4 `Tempo` *is* multiplicative *and* `Transpose` *is* additive. *That is, for any* `r1`*,* `r2`*,* `r3`*,* `r4`*,* `p`*, and* `m`*:*

```
    Tempo r1 r2 (Tempo r3 r4 m)  ≡  Tempo (r1*r3) (r2*r4) m
           Trans p1 (Trans p2 m)  ≡  Trans (p1+p2) m
```

Axiom 5 *Function composition is* commutative *with respect to both tempo scaling and transposition. That is, for any* `r1`*,* `r2`*,* `r3`*,* `r4`*,* `p1` *and* `p2`*:*

```
Tempo r1 r2 . Tempo r3 r4  ≡  Tempo r3 r4 . Tempo r1 r2
      Trans p1 . Trans p2  ≡  Trans p2 . Trans p1
   Tempo r1 r2 . Trans p1  ≡  Trans p1 . Tempo r1 r2
```

Axiom 6 *Tempo scaling and transposition are* distributive *over both sequential and parallel composition. That is, for any* r1*,* r2*,* p*,* m1*, and* m2*:*

```
Tempo r1 r2 (m1 :+: m2)  ≡  Tempo r1 r2 m1 :+: Tempo r1 r2 m2
Tempo r1 r2 (m1 :=: m2)  ≡  Tempo r1 r2 m1 :=: Tempo r1 r2 m2
      Trans p (m1 :+: m2)  ≡  Trans p m1 :+: Trans p m2
      Trans p (m1 :=: m2)  ≡  Trans p m1 :=: Trans p m2
```

Axiom 7 *Sequential and parallel composition are* associative*. That is, for any* m0*,* m1*, and* m2*:*

```
m0 :+: (m1 :+: m2)  ≡  (m0 :+: m1) :+: m2
m0 :=: (m1 :=: m2)  ≡  (m0 :=: m1) :=: m2
```

Axiom 8 *Parallel composition is* commutative*. That is, for any* m0 *and* m1*:*

```
m0 :=: m1  ≡  m1 :=: m0
```

Axiom 9 Rest 0 *is a* unit *for* Tempo *and* Trans*, and a* zero *for sequential and parallel composition. That is, for any* r1*,* r2*,* p*, and* m*:*

```
  Tempo r1 r2 (Rest 0)  ≡  Rest 0
      Trans p (Rest 0)  ≡  Rest 0
m :+: Rest 0  ≡  m  ≡  Rest 0 :+: m
m :=: Rest 0  ≡  m  ≡  Rest 0 :=: m
```

Exercise 10. Establish the validity of each of the above axioms.

9 Related and Future Research

Many proposals have been put forth for programming languages targeted for computer music composition [Dan89, Sch83, Col84, AK92, DFV92, HS92, CR84, OFLB94], so many in fact that it would be difficult to describe them all here. None of them (perhaps surprisingly) are based on a *pure* functional language, with one exception: the recent work done by Orlarey et al. at GRAME [OFLB94], which uses a pure lambda calculus approach to music description, and bears a strong resemblance to our effort (but unfortunately has not been implemented). There are some other related approaches based on variants of Lisp, most notably Dannenberg's *Fugue* language [DFV92], in which operators similar to ours can be found but where the emphasis is more on instrument synthesis rather than note-oriented composition. Fugue also highlights the utility of lazy evaluation in certain contexts, but extra effort is needed to make this work in Lisp, whereas in a non-strict language such as Haskell it essentially comes "for free." Other efforts based on Lisp utilize Lisp primarily as a convenient vehicle for "embedded language design," and the applicative nature of Lisp is not exploited well (for example, in Common Music the user will find a large number of macros which are difficult if not impossible to use in a functional style).

We are not aware of any computer music language that has been shown to exhibit the kinds of algebraic properties that we have demonstrated for Haskore. Indeed, none of the languages that we have investigated make a useful distinction between music and performance, a property that we find especially attractive about the Haskore design. On the other hand, Balaban describes an abstract notion (apparently not yet a programming language) of "music structure," and provides various operators that look similar to ours [Bal92]. In addition, she describes an operation called *flatten* that resembles our literal interpretation `perform`. It would be interesting to translate her ideas into Haskell; the match would likely be good.

Perhaps surprisingly, the work that we find most closely related to ours is not about music at all: it is Henderson's *functional geometry*, a functional language approach to generating computer graphics [Hen82]. There we find a structure that is in spirit very similar to ours: most importantly, a clear distinction between object *description* and *interpretation* (which in this paper we have been calling musical objects and their performance). A similar structure can be found in Arya's *functional animation* work [Ary94].

There are many interesting avenues to pursue with this research. On the theoretical side, we need a deeper investigation of the algebraic structure of music, and would like to express certain modern theories of music in Haskore. The possibility of expressing other scale types instead of the thus far unstated assumption of standard equal temperament is another area of investigation. On the practical side, the potential of a graphical interface to Haskore is appealing. We are also interested in extending the methodology to sound synthesis. Our primary goal currently, however, is to continue using Haskore as a vehicle for interesting algorithmic composition (for example, see [HB95]).

A General Midi

```
> module GeneralMidi where
>
> import MidiFile
>
> type GenMidiName = String
> type GenMidiTable = (GenMidiName,ProgNum)
>
> genMidiMap :: [GenMidiTable]
> genMidiMap =[
>  ("Acoustic Grand Piano",0),      ("Bright Acoustic Piano",1),
>  ("Electric Grand Piano",2),      ("Honky Tonk Piano",3),
> ("Rhodes Piano",4),                 ("Chorused Piano",5),
> ("Harpsichord",6),                  ("Clavinet",7),
> ("Celesta",8),                      ("Glockenspeil",9),
> ("Music Box",10),                   ("Vibraphone",11),
> ("Marimba",12),                     ("Xylophone",13),
> ("Tubular Bells",14),               ("Dulcimer",15),
> ("Hammond Organ",16),               ("Percussive Organ",17),
> ("Rock Organ",18),                  ("Church Organ",19),
> ("Reed Organ",20),                  ("Accordion",21),
> ("Harmonica",22),                   ("Tango Accordion",23),
> ("Accoustic Guitar (nylon)",24),    ("Acc. Guitar (steel)",25),
> ("Electric Guitar (jazz)",26),      ("Ele. Guitar (clean)",27),
> ("Electric Guitar (muted)",28),     ("Overdriven Guitar",29),
> ("Distortion Guitar",30),           ("Guitar Harmonics",31),
> ("Accoustic Bass",32),              ("Ele. Bass (fingered)",33),
> ("Electric Bass (picked)",34),      ("Fretless Bass",35),
> ("Slap Bass 1",36),                 ("Slap Bass 2",37),
> ("Synth Bass 1",38),                ("Synth Bass 2",39),
> ("Violin",40),                      ("Viola",41),
> ("Cello",42),                       ("Contrabass",43),
> ("Tremolo Strings",44),             ("Pizzicato Strings",45),
> ("Orchestral Harp",46),             ("Timpani",47),
> ("String Ensemble 1",48),           ("String Ensemble 2",49),
> ("Synth Strings 1",50),             ("Synth Strings 2",51),
> ("Choir Aahs",52),                  ("Voice Oohs",53),
> ("Synth Voice",54),                 ("Orchestra Hit",55),
> ("Trumpet",56),                     ("Trombone",57),
> ("Tuba",58),                        ("Muted Trumpet",59),
> ("French Horn",60),                 ("Brass Section",61),
> ("Synth Brass 1",62),               ("Synth Brass 2",63),
> ("Soprano Sax",64),                 ("Alto Sax",65),
> ("Tenor Sax",66),                   ("Barinote Sax",67),
> ("Oboe",68),                        ("Basoon",69),
> ("English Horn",70),                ("Clarinet",71),
> ("Piccolo",72),                     ("Flute",73),
> ("Recorder",74),                    ("Pan Flute",75),
> ("Blown Bottle",76),                ("Shakuhachi",77),
> ("Whistle",78),                     ("Occarina",79),
> ("Lead 1 (square)",80),             ("Lead 2 (sawtooth)",81),
> ("Lead 3 (calliope)",82),           ("Lead 4 (chiff)",83),
> ("Lead 5 (charang)",84),            ("Lead 6 (voice)",85),
> ("Lead 7 (fifths)",86),             ("Lead 8 (bass+lead)",87),
> ("Pad 1 (new age)",88),             ("Pad 2 (warm)",89),
> ("Pad 3 (polysynth)",90),           ("Pad 4 (choir)",91),
> ("Pad 5 (bowed)",92),               ("Pad 6 (metallic)",93),
>  ("Pad 7 (halo)",94),                 ("Pad 8 (sweep)",95),
>  ("FX1 (train)",96),                  ("FX2 (soundtrack)",97),
>  ("FX3 (crystal)",98),                ("FX4 (atmosphere)",99),
>  ("FX5 (brightness)",100),            ("FX6 (goblins)",101),
>  ("FX7 (echoes)",102),                ("FX8 (sci-fi)",103),
>  ("Sitar",104),                       ("Banjo",105),
>  ("Shamisen",106),                    ("Koto",107),
>  ("Kalimba",108),                     ("Bagpipe",109),
>  ("Fiddle",110),                      ("Shanai",111),
>  ("Tinkle Bell",112),                 ("Agogo",113),
>  ("Steel Drums",114),                 ("Woodblock",115),
>  ("Taiko Drum",116),                  ("Melodic Drum",117),
>  ("Synth Drum",118),                  ("Reverse Cymbal",119),
>  ("Guitar Fret Noise",120),           ("Breath Noise",121),
>  ("Seashore",122),                    ("Bird Tweet",123),
>  ("Telephone Ring",124),              ("Helicopter",125),
>  ("Applause",126),                    ("Gunshot",127)]
```

B Auxiliary Functions for Testing Haskore

```
> module Examples where
> import Haskore
> import ChildSong
>
> -- performance stuff
>
> -- an IMap
> imap :: String -> Instrument
> imap "Default" = defInst
> imap "Ch1Inst" = ch1Inst
> imap "Ch2Inst" = ch2Inst
>
> -- some instruments
> ch1Inst,ch2Inst :: Instrument
> ch1Inst = Inst "Ch1Inst" (defPlayNote     defNasHandler)
>                          (defInterpPhrase defPasHandler)
>                          (defNotateInst   ()           )
> ch2Inst = Inst "Ch2Inst" (defPlayNote     defNasHandler)
>                          (defInterpPhrase defPasHandler)
>                          (defNotateInst   ()           )
>
```

```
> -- a UserPatchMap
> defPatch :: UserPatchMap
> defPatch = [("Default","Acoustic Grand Piano",1),
>             ("Ch1Inst","Acoustic Grand Piano",1),
>             ("Ch2Inst","Acoustic Grand Piano",2)]
>
> -- from a Music object, we can:
>
> -- a) generate a performance
> testPerf :: Music -> Performance
> testPerf m = perform imap (0, defInst, metro 120 qn, 0, 80) m
>
> -- b) generate a midifile datatype
> testMidi :: Music -> MidiFile
> testMidi m = performToMidi (testPerf m) defPatch
>
> -- c) generate a midifile
> test     :: Music -> IO ()
> test     m = outputMidiFile "foo.mid" (testMidi m)
>
> -- try calling the above with things like pr12, cMajArp, cMajChd, etc.
> -- (examples from the tutorial)
> -- also try inversions, retrogrades, legato, etc. on the same examples
> -- also try "childSong6" imported from module ChildSong
```

C Partial Encoding of Chick Corea's "Children's Song No. 6"

```
> module ChildSong where
> import Haskore
>
> -- Preliminaries: define some dotted durations
> dhn, dqn, den, dsn, dtn :: Float
> dhn = 3/4;  dqn = 3/8;  den = 3/16;  dsn = 3/32;  dtn = 3/64
>
> -- note updaters for mappings
> fd d n = n d v
> vol  n = n   v
> v      = [Volume 80]
> lmap f l = line (map f l)
>
> -- repeat something n times
> times  1    m = m
> times (n+1) m = m :+: (times n m)
>
```

```
> -- BASELINE:
> b1 = lmap (fd dqn) [b  3, fs 4, g  4, fs 4]
> b2 = lmap (fd dqn) [b  3, es 4, fs 4, es 4]
> b3 = lmap (fd dqn) [as 3, fs 4, g  4, fs 4]
>
> bassLine = times 3 b1 :+: times 2 b2 :+: times 4 b3 :+: times 5 b1
>
> -- MAIN VOICE:
> v1  = v1a :+: v1b
> v1a = lmap (fd en) [a 5, e 5, d 5, fs 5, cs 5, b 4, e 5, b 4]
> v1b = lmap vol    [cs 5 tn, d 5 (qn-tn), cs 5 en, b 4 en]
>
> v2  = v2a :+: v2b :+: v2c :+: v2d :+: v2e :+: v2f
> v2a = lmap vol [cs 5 (dhn+dhn), d 5 dhn,
>                 f 5 hn, gs 5 qn, fs 5 (hn+en), g 5 en]
> v2b = lmap (fd en) [fs 5, e 5, cs 5, as 4] :+: a 4 dqn v :+:
>       lmap (fd en) [as 4, cs 5, fs 5, e 5, fs 5, g 5, as 5]
> v2c = lmap vol [cs 6 (hn+en), d 6 en, cs 6 en, e 5 en] :+: enr :+:
>       lmap vol [as 5 en, a 5 en, g 5 en, d 5 qn, c 5 en, cs 5 en]
> v2d = lmap (fd en) [fs 5, cs 5, e 5, cs 5, a 4, as 4, d 5,
>                     e 5, fs 5] :+:
>       lmap vol [fs 5 tn, e 5 (qn-tn), d 5 en, e 5 tn,
>                 d 5 (qn-tn),
>                 cs 5 en, d 5 tn, cs 5 (qn-tn), b 4 (en+hn)]
> v2e = lmap vol [cs 5 en, b 4 en, fs 5 en, a 5 en, b 5 (hn+qn),
>                 a 5 en, fs 5 en, e 5 qn, d 5 en, fs 5 en,
>                 e 5 hn, d 5 hn, fs 5 qn]
> v2f = Tempo 3 2 (lmap vol [cs 5 en, d 5 en, cs 5 en]) :+:
>                                                   b 4 (3*dhn+hn) v
>
> mainVoice = Phrase [Art legato] (times 3 v1 :+: v2)
>
> -- PUTTING IT ALL TOGETHER
> childSong6 = Tempo 3 1 (bassLine :=: mainVoice)
```

References

[AK92] D.P. Anderson and R. Kuivila. Formula: A programming language for expressive computer music. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[Ary94] K. Arya. A functional animation starter-kit. *Journal of Functional Programming*, 4(1):1–18, 1994.

[Bal92] M. Balaban. Music structures: Interleaving the temporal and hierarchical aspects of music. In M. Balaban, K. Ebcioglu, and O. Laske, editors, *Understanding Music With AI*, pages 110–139. AAAI Press, 1992.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, New York, 1988.

[Col84] D. Collinge. Moxie: A languge for computer music performance. In *Proc. Int'l Computer Music Conference*, pages 217–220. Computer Music Association, 1984.

[CR84] P. Cointe and X. Rodet. Formes: an object and time oriented system for music composition and synthesis. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programmming*, pages 85–95. ACM, 1984.

[Dan89] R.B. Dannenberg. The Canon score language. *Computer Music Journal*, 13(1):47–56, 1989.

[DFV92] R.B. Dannenberg, C.L. Fraley, and P. Velikonja. A functional language for sound synthesis with behavioral abstraction and lazy evaluation. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[For73] A. Forte. *The Structure of Atonal Music*. Yale University Press, New Haven, CT, 1973.

[HB95] P. Hudak and J. Berger. A model of performance, interaction, and improvisation. In *Proceedings of International Computer Music Conference*. Int'l Computer Music Association, 1995.

[Hen82] P. Henderson. Functional geometry. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programmming*. ACM, 1982.

[HF92] P. Hudak and J. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

[HMGW94] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation – an algebra of music, September 1994. To appear in the Journal of Functional Programming; preliminary version available via
`ftp://nebula.systemsz.cs.yale.edu`
`/pub/yale-fp/papers/haskore/hmn-lhs.ps`.

[HS92] G. Haus and A. Sametti. Scoresynth: A system for the synthesis of music scores based on petri nets and a music algebra. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[IMA90] Midi 1.0 detailed specification: Document version 4.1.1, February 1990.

[JB91] D. Jaffe and L. Boynton. An overview of the sound and music kits for the NeXT computer. In S.T. Pope, editor, *The Well-Tempered Object*, pages 107–118. MIT Press, 1991.

[OFLB94] O. Orlarey, D. Fober, S. Letz, and M. Bilton. Lambda calculus and music calculi. In *Proceedings of International Computer Music Conference*. Int'l Computer Music Association, 1994.

[Sch83] B. Schottstaedt. Pla: A composer's idea of a language. *Computer Music Journal*, 7(1):11–20, 1983.

[Ver86] B. Vercoe. Csound: A manual for the audio processing system and supporting programs. Technical report, MIT Media Lab, 1986.