J. Functional Programming **1** (1): 1–000, January 1993

Haskore Music Notation – An Algebra of Music –

Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong

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

Abstract

We have developed a simple algebraic approach to music description and composition called *Haskore*. In this framework, 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. When these simple notions are embedded into a functional language such as Haskell, rather complex musical relationships can be expressed clearly and succinctly.

Exploiting the algebraic properties of Haskore, we have furthermore defined a notion of *literal performance* (devoid of articulation) through which *observationally equivalent* musical objects can be determined. With this basis many useful properties can be proved, such as commutative, associative, and distributive properties of various operators. An algebra of music thus surfaces.

1 Introduction

Traditional music notation (often called *common practice notation*) has many well-known limitations. From our perspective, the following are particularly acute:

1. Traditional notation is unable to adequately capture a composer's *intentions*, in particular *structural* aspects of a composition.
2. Traditional notation is biased towards music that is *humanly performable*. This is not surprising, of course, but is an obstacle when trying to notate music intended for computer performance, where the notation is often found to be deficient, inconsistent, and redundant.
3. Many well-known (not just contemporary) ideas in music *theory* are difficult if not impossible to express in traditional notation. The basic concepts of atonal music theory (For73), for example, are impossible to express without the use of a meta-logic; more preferable would be a common notation that could be used to express musical objects *and* the interrelationships between them.
4. Modern notions of *algorithmic composition* are also impossible to express in traditional notation; there is simply no notion of "algorithm" at all.

These shortcomings, along with our experience in computer music and algorithmic composition, have led us to seek alternatives to traditional music notation. Our background in the theory, design, and implementation of high-level programming languages, in particular *functional* languages, has led us to a rather satisfying solution based on the functional language *Haskell* (HPJWe92). In fact, we did not design a new language at all: our system,

which we call *Haskore*, is essentially a set of program modules written in Haskell that allow the user to express musical ideas in a high-level, higher-order, and extensible manner.

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; we don't *force* our ideas on the user).

The above advantages are perhaps obvious, but do little good if we can't design a system that is actually useful for composing music and that avoids the problems with traditional notation that we alluded to earlier. Fortunately, as we shall soon see, Haskell's high-level, declarative nature is well suited to music composition. Furthermore, relying on Haskell's underlying equational theory, the resulting system has clearly defined algebraic properties which allow one to, for example, prove interesting properties about musical objects and transform them in such a way that "meaning" is preserved.

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.

Limitations of Traditional Notation As a simple example of the limitations of traditional notation, consider the triplet (3-tuple), 5-tuple, and 7-tuple shown in Figure 1a. The rules governing the "default interpretation" of such phrases are somewhat *ad hoc*: for example, the 3 and 5 notes in the 3-tuple and 5-tuple are to be played in the space of 2 and 4 notes, respectively; but the 7-tuple is intended to be played in the space of 4 (Hin49).

To avoid this problem, traditional notation is sometimes generalized to make the implicit "denominator" more explicit, as shown in Figure 1(b). With this simple generalization we now have much more freedom in expressing more interesting phrases, as shown, for example, in Figure 1c, where we see several traditional rules being "broken":

1. Non-conventional ratios such as 7:5 are allowed.
2. The "numerator" is not required to match the number of notes in the figure. Indeed, the meaning of a figure annotated with m:n is no longer "play these m notes in the space of n," but rather, "scale the tempo by a factor of m/n."†
3. The numbers are not constrained to be integers. Indeed any ratio is allowed, even irrational ones!
4. The ratio is not always greater than one; i.e. "tempo-compression" is allowed as well as "tempo-expansion."

These generalizations are adopted in Haskore, and clearly move us rapidly toward the realm of "not humanly performable."

As an example of the redundancy in traditional notation, consider the convention of *dotting* a note to extend its duration by 50%, as shown in Figure 1d. This convention does not permit dotting more than one note, as shown in Figure 1e. The meaning of such a phrase, if it were allowed, could be expressed in generalized tuple notation as shown

† Note that our use of "tempo" here is a relative one that indicates, for a particular phrase, the rate at which the notes are played. The tempo for an overall composition is just the starting point; it may vary considerably depending on context.

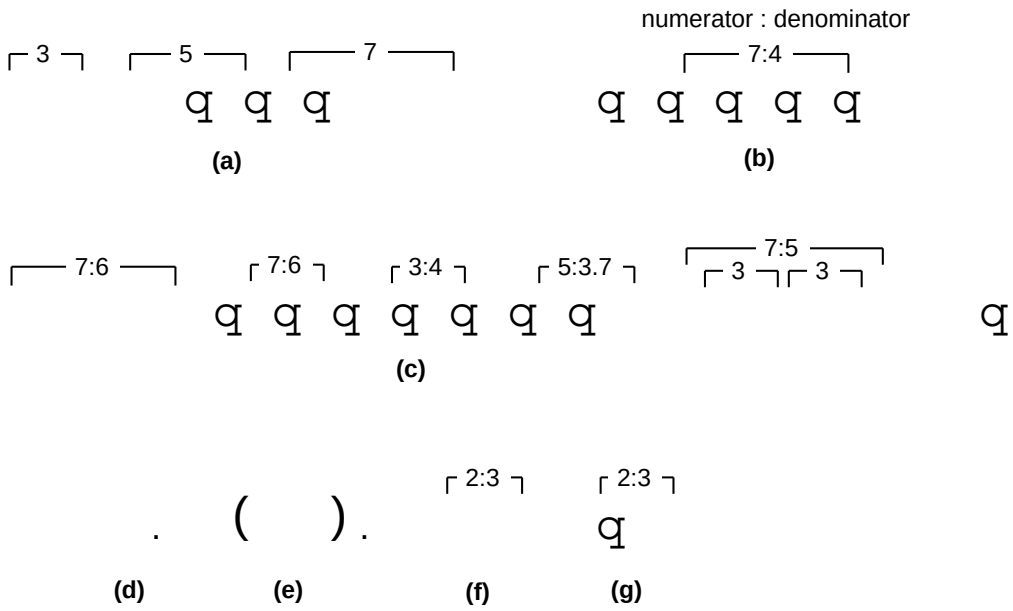

Fig. 1. Deficiencies and Redundancies of Tupling Notation

in Figure 1f. But in fact, a single dotted note could be expressed in a similar way, and therefore Figures 1d and 1g are equivalent. Thus in a sense dotted notation is a redundant form of "1-tupling."

As a final example, we note that traditional notation has very few mechanisms for allowing a composer to express how she perceives a composition's *structure*. All that is available is a few *ad hoc* labeling techniques and a mechanism for repeating phrases. For example, there is no way to describe a canon-like phrase as a structure in which "phrase A is played simultaneously with itself, but with one part delayed by 2 measures." Far more sophisticated structures and relationships exist in many compositions, limited only by the composer's creativity.

Haskore Figure 2 shows the overall structure of our system. We will not provide details of the various *translators* in this paper; they are reasonably straightforward functional programs for converting from internal abstract datatypes to the file syntax required to play Haskore compositions as conventional midi-files (IMA90), NeXT MusicKit score files (JB91), or Csound score files (Ver86), and to print Haskore compositions in traditional notation using the CMN (Common Music Notation) subsystem. Of most interest is the box labeled "Haskore."

We will use Haskell notation throughout; readers familiar with any of a number of strongly typed functional languages, such as Miranda or ML, should have little trouble following the presentation. Those completely unfamiliar with functional languages are urged to at least read (HF92) before continuing. This paper is written assuming good familiarity with functional programming in general and Haskell in particular; a different version is being written that is targeted mainly for musicians.

As our musical ideas are presented in Haskell, we urge the reader to question, at every step, the decisions that are being made. There is no supreme theory of music that dictates our decisions, and what we present is actually one of several versions that we have developed. We believe the simplicity and elegance of this version is suitable for many purposes,

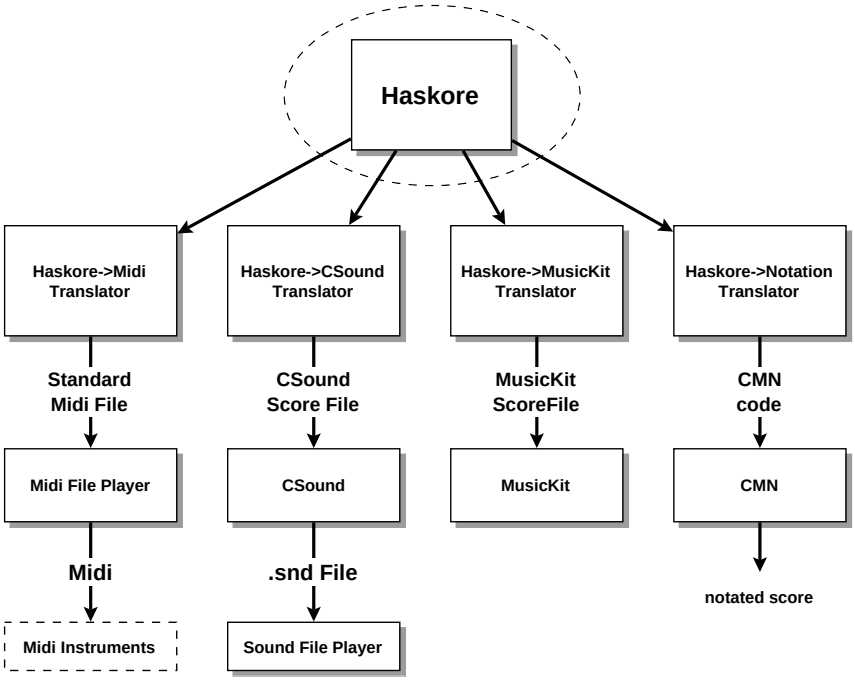

Fig. 2. Overall System Diagram

especially pedagogical ones, but the reader may likely want to modify it to better satisfy her intuitions and/or application.

This document was written in the *literate programming style*, and thus the LaTeXmanuscript file from which it was generated is an *executable Haskell program*. This file can be retrieved via the WWW from `ftp://nebula.systemsz.cs.yale.edu/pub/yale-fp/papers/haskore`, where the user will also find the latest version of the Haskore system (consult the README file for details).

2 The Basics

The most basic musical idea is a *note* (or absence thereof), which we describe using a Haskell datatype:

```
> data Note = Note Pitch Dur | Rest Dur
>      deriving Text
> type Pitch = Int
> type Dur   = Float
```

where `Pitch` is the type that represents a note's pitch (an integer) and `Dur` is the type that represents a pitch's duration (measured in number of beats). Additional note parameters (such as volume, timbre, or envelope) may easily be attached as additional `Note` constructor fields, but we omit such detail here. The `deriving Text` clause simply ensures that we can generate textual representations of `Note`'s using the overloaded `show` operator.

For convenience, we refer to a sequence of notes as a *line*:

```
> type Line = [Note]
```

From these basic primitives we will construct more complex musical ideas. For example, two basic *transformations* we may wish to perform on a musical object are:

- Scaling of the tempo (in the flavor of the generalized tupling described earlier).
- Transposition of the melody.

In addition, it is desirable to have ways to *compose* musical objects to form lager ones. In particular we may wish to:

- Play several musical objects in sequence; i.e. one after the other.
- Play several musical objects in parallel; i.e. simultaneously.

It is convenient to represent these ideas in Haskell as a recursive datatype:‡

```
> data Music = Line Line                  -- base case
>            | Scale Int Int Music        -- scale tempo
>            | Trans Int Music            -- transpose pitches
>            | Music :+: Music            -- play in sequence
>            | Music :=: Music            -- play in parallel
>            | Instr IName Music          -- use specified instrument
>     deriving Text
> type IName = String
```

So, a musical object in Haskore is either a sequence of notes (a line), a tempo scaling or melodic transposition of some other musical object, or the sequential or parallel composition of two other musical objects. The final construction, `Instr iname m`, simply declares the intent to play `m` using instrument `iname`.

Examples With this modest beginning, we can already express quite a few complex relationships simply and effectively. For convenience we first create a few functions to generate pitch values based on familiar octave and note names, as follows:§

```
> cf,c,cs,df,d,ds,ef,e,es,ff,f,fs,gf,g,gs,af,a,as,bf,b,bs :: Pitch
> cf = -1; c = 0;  cs = 1;  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
>
> note :: Int -> Int -> Dur -> Note
> note oct pit = Note (pitch oct pit)
> pitch o p = 12*o + p
```

‡ An alternative is to represent them as *functions*, but then we lose the ability to take musical objects apart, analyze their structure, print them in a structure-preserving way, etc.

§ An alternative is to define a separate (enumerated) datatype for notes, but manipulating notes as integers is so convenient that we prefer the present approach.

Note that the expression `Note p` is a *function* that, when applied to a duration `d`, returns a note with pitch `p` and duration `d`.

Similarly, here are convenient names for common durations and rests:

```
> wn,  hn,  qn,  en,  sn  :: Dur
> wnr, hnr, qnr, enr, snr :: Note
> 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      -- eighth note rest
> sn = 1/16       ; snr = Rest sn      -- sixteenth note rest
```

Next, let's define the longest and most boring piece of music that we can imagine: an infinite sequence of a single note:

```
> infLine :: Note -> Line
> infLine note = note : infLine note   -- or, "cycle [note]"
```

From this we can define a function to create a similar line, but of a specified length:

```
> mkLine :: Int -> Note -> Line
> mkLine len = take len . infLine
```

Now consider the nested polyrhythms shown in Figure 3. They can be expressed in Haskore as follows:

```
> pr1, pr2 :: Pitch -> Music
> pr1 p = Scale 5 6 (Scale 4 3 (mkLn 1 qn :+:
>                               Scale 3 2 (mkLn 3 en :+:
>                                          mkLn 2 sn :+:
>                                          mkLn 1 qn    ) :+:
>                               mkLn 1 qn) :+:
>                     Scale 3 2 (mkLn 6 en))
>         where mkLn n d = Line (mkLine n (Note p d))
>
> pr2 p = Scale 7 6 (m1 :+:
>                     Scale 5 4 (mkLn 5 en) :+:
>                     m1 :+:
>                     mkLn 2 en)
>         where m1 = Scale 5 4 (Scale 3 2 m2 :+: m2)
>               m2 = mkLn 3 en
>               mkLn n d = Line (mkLine n (Note p d))
```

Note the use of the *where* clause in `pr2` to capture recurring phrases.

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

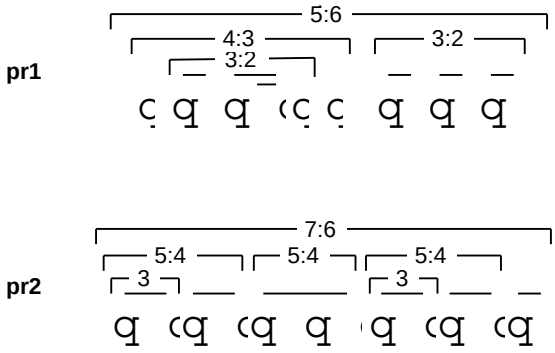

Fig. 3. Nested Polyrhythms

```
> pr12 :: Music
> pr12 = pr1 (pitch 5 c) :=: pr2 (pitch 5 g)
```

An expression of the form:

```
m :=: Trans 7 m
```

describes a melody `m` accompanied by a second voice a perfect 5th higher. Similarly, a canon-like structure involving `m` can be expressed as:

```
m :=: delay d m
```

where

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

A musical object may be repeated *ad nauseum* using this simple function:

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

For example, 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 basic notions of inversion, retrograde, retrograde inversion, etc. used in 12-tone theory are also easily captured in Haskore, as the following definitions demonstrate (we assume that these transformations are only well-defined for lines):

```
> retro, invert, retroInvert, invertRetro :: Music -> Music
> retro  (Line line) = Line (reverse line)
> invert (Line line) = Line (map inv line)
>        where Note p _        = head line
>              inv (Note p' d) = Note (2*p-p') d
>              inv  n          = n
```

```
> retroInvert = retro . invert
> invertRetro = invert . retro
```

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

```
> dur :: Music -> Dur
> dur (Scale a b m) = dur m * (float b / float a)
> dur (Trans _ m)   = dur m
> dur (m1 :+: m2)   = dur m1   +   dur m2
> dur (m1 :=: m2)   = dur m1 `max` dur m2
> dur (Line line)   = sum (map getDur line)
>                     where getDur (Rest d)   = d
>                           getDur (Note _ d) = d
> dur (Instr i m)   = dur m
>
> float = fromInteger . toInteger :: Int -> Float
```

3 Performance and Interpretation

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 Event       = (Time,IName,Pitch,DurT)
> type Performance = [Event]
> type Time        = Float
> type DurT        = Float
```

An event `(s,i,p,d)` captures the fact that at time `s`, instrument `i` plays pitch `p` for a duration `d` (where now duration is measured in seconds, rather than beats).

For a *performer* to give an interpretation to a musical object, it must know on which instrument to perform, a time to begin the performance, and the proper key and tempo. We can thus model a performer as a function `perform` which maps this information and a musical object into a performance:

```
> perform :: (Time,IName,Key,Tempo) -> Music -> Performance
> type Key   = Int
> type Tempo = Float
```

Of course, there are many kinds of performances; music can be interpreted in many different ways. Eventually we may want to develop fairly sophisticated performances, but for now the one that interests us most is what we call the *literal* performance: a flawless but emotionless (i.e. devoid of articulation) interpretation of a musical object. To begin, let's define the literal performance of a *line*:

```
> playLine :: (Time,IName,Key,Tempo) -> Line -> Performance
> playLine (s,i,k,t) (n:notes) =
>   case n of Rest d   -> playLine (s+d*60/t,i,k,t) notes
>             Note p d -> (s,i,p+k,d') : playLine (s+d',i,k,t) notes
>                         where d' = d*60/t
> playLine _ [] = []
```

With this as a foundation, we can define the literal performance of an entire musical object:

```
> perform x@(s,i,k,t) m =
>   case m of
>     Scale a b m -> perform (s,i,k,t*(float a / float b)) m
>     Trans p m   -> perform (s,i,k+p,t) m
>     m1 :+: m2   -> perform x m1 ++ perform (s+(dur m1)*60/t,i,k,t) m2
>     m1 :=: m2   -> merge (perform x m1) (perform x m2)
>     Line line   -> playLine x line
>     Instr i' m  -> perform (s,i',k,t) m
>
> merge :: Performance -> Performance -> Performance
> merge (e1:es1) (e2:es2) =
>   if e1<e2 then e1 : merge es1 (e2:es2)
>            else e2 : merge (e1:es1) es2
> merge [] es2 = es2
> merge es1 [] = es1
```

Note that `perform` invokes `playLine` for the base case (a line). The function `merge` is required to preserve the property that a performance is a *temporally ordered* sequence of events, and `dur` (defined in the last section) is needed to compute the duration of the first argument to `:+:`.¶

4 Equivalence of Literal Performances

There are many different 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:

```
Line [n1,n2] :+: Line [n3,n4]

Line [n1,n2,n3,n4]
```

Thus we define a notion of equivalence:

¶ The use of `dur` in the definition of `perform` results in quadratic time complexity. A 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. Note also that `merge` compares entire events rather than just start times. This is to ensure that it is commutative, a desirable condition for some of our proofs.

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

$$(\forall x) \quad \texttt{perform x m1 = perform x m2}$$

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 evolving *algebra of music*. For example:

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

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

To prove this axiom, we use conventional equational reasoning:

Proof:

```
perform (_,_,_,t) (Scale r1 r2 (Scale r3 r4 m))
= perform (_,_,_,r1*t/r2) (Scale r3 r4 m)       -- unfolding perform
= perform (_,_,_,r3*(r1*t/r2)/r4) m             -- unfolding perform
= perform (_,_,_,(r1*r3)*t/(r2*r4)) m           -- simple arithmetic
= perform (_,_,_,t) (Scale (r1*r3) (r2*r4) m)   -- folding perform
```

Here is another useful transformation and its validity proof:

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

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

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

Proof:

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

An even simpler axiom is given by:

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

```
Scale r r m  ≡  m
```

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

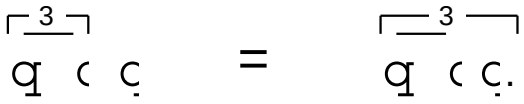

Fig. 4. Equivalent Phrases

Proof:

```
perform (s,_,_,t) (Scale r r m)
= perform (s,_,_,r*t/r) m                    -- unfold perform
= perform (s,_,_,t) 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 1
For any `r1`, `r2`, `m1`, and `m2`:

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

Proof:

```
Scale r1 r2 (m1 :+: Scale r2 r1 m2)
= Scale r1 r2 m1 :+: Scale r1 r2 (Scale r2 r1 m2)     -- by Axiom 1
= Scale r1 r2 m1 :+: Scale (r1*r2) (r2*r1) m2         -- by Axiom 2
= Scale r1 r2 m1 :+: Scale (r1*r2) (r1*r2) m2         -- simple arithmetic
= Scale r1 r2 m1 :+: m2                               -- by Axiom 3
```

For example, this fact justifies the equivalence of the two phrases shown in Figure 4.

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
`Scale` and `Transpose` are *additive*. That is, for any `r1`, `r2`, `r3`, `r4`, `p`, and `m`:

```
Scale r1 r2 (Scale r3 r4 m)  ≡  Scale (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`:

```
Scale r1 r2 . Scale r3 r4  ≡  Scale r3 r4 . Scale r1 r2
     Trans p1 . Trans p2   ≡  Trans p2 . Trans p1
  Scale r1 r2 . Trans p1   ≡  Trans p1 . Scale 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`:

```
Scale r1 r2 (m1 :+: m2)  ≡  Scale r1 r2 m1 :+: Scale r1 r2 m2
Scale r1 r2 (m1 :=: m2)  ≡  Scale r1 r2 m1 :=: Scale 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
`Line []` is a *unit* for `Scale` and `Trans`, and a *zero* for sequential and parallel composition. That is, for any `r1`, `r2`, `p`, and `m`:

```
Scale r1 r2 (Line [])  ≡  Line []
  Trans p (Line [])  ≡  Line []
m :+: Line []  ≡  m  ≡  Line [] :+: m
m :=: Line []  ≡  m  ≡  Line [] :=: m
```

Axiom 10
For any `l1` and `l2`:

```
Line l1 :+: Line l2  ≡  Line (l1 ++ l2)
```

The proofs of these axioms are no more difficult than those given earlier, except for the proof of the last (which covers the very first example given in this section), whose inductive proof we have included in the Appendix.

5 Notating Chords

So far we have built musical objects from "lines" of notes, but sometimes it is better to think in terms of *chords*. Of course, a chord consisting of the notes `n1`, `n2`, and `n3` (of equal duration) could be expressed in our current framework as:

```
Line [n1] :=: Line [n2] :=: Line [n3]
```

and for convenience we could define a function to turn an arbitrary list of notes into such a chord, as follows:

```
> mkChord :: [Note] -> Music
> mkChord = foldr (\note mus -> Line [note] :=: mus) (Line [])
```

Nevertheless, it may be preferable to add a chord constructor (having the same type as `mkChord` above) to the `Music` datatype, for the same reasons justifying the use of the datatype to begin with. In any case, the following discussion does not rely on this decision.

Rather than think of a chord in terms of its actual notes, it is also 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 do not intend to cover all of them here. Rather, we will describe two basic representations, leaving other alternatives to the skill and imagination of the reader.‖

‖ For example, Forte prescribes normal forms for chords in an atonal setting (For73).

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:

```
> type Chord = [Pitch]
```

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
```

We can in fact prove:

Theorem 2
`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]`).

6 Haskore in Practice

The version of Haskore presented in this paper has been simplified for pedagogical purposes. Indeed, since Haskore is not a new language, but rather is simply a collection of datatypes and functions written in Haskell, there has been a tendency for it to evolve as we have gained experience using it. This is both a blessing and a curse. It is a blessing because we (and our users) are never constrained by past design decisions, and can easily adapt the system to meet current needs. It is a curse because compositions written in previous versions of Haskore will not always run in newer ones! Nevertheless, the system seems to be stabilizing fairly rapidly, and in this section we briefly describe some specific changes and extensions that have become part of the new design. These changes are primarily at the level of the `Music` datatype and its associated interpretation via `perform`; the various translators to midi, Csound, etc. remain unaltered.

The uncomfortable treatment of a "line" as something special led us at one point to also include a constructor for chords, as suggested in Section 5, thus providing two base cases for the `Music` datatype. However, further reflection convinced us that making *either* of them special was wrong. Instead, why not make the note and rest primitive music objects, and use the constructors `(:+:)` and `(:=:)` to build lines and chords? The resulting datatype:

```
data Music = Note Pitch Dur            -- base case
           | Rest Dur                  -- base case
           | Music :+: Music           -- play in sequence
           | Music :=: Music           -- play in parallel
           ...
```

is more stream-lined, and allows us to write simple definitions such as:

```
line,chord :: [Music] -> Music
line  = foldr (:+:) (Rest 0)
chord = foldr (:=:) (Rest 0)
```

when we wish to work with the sometimes-more-convenient list representation of notes.

The `Music` datatype has also been extended with annotations to allow more expressiveness over both notes and larger phrases:

```
data Music = Note Pitch Dur [NoteAttribute]
           | Phrase [PhraseAttribute] Music
           ...
```

where the attributes are defined by:

```
data NoteAttribute = Volume Float
                   | Timbre Timbre
                   | Envelope Envelope

data PhraseAttribute = Dyn Dynamic
                     | Art Articulation
                     | Orn Ornament

data Dynamic = Accent Int | Crescendo Int | Diminuendo Int

data Articulation = Staccato Float | Legato Float | Slurred
                  | Tenuto | Marcato
                  | Fermata | FermataDown | Breath

data Ornament = Trill | Mordent | InvMordent | DoubleMordent
              | Turn | TrilledTurn | ShortTrill | Arpeggio
```

Note that many of the above annotations are borrowed from traditional common practice notation, but nevertheless may be useful to the composer. However, they immediately pose more difficult issues with respect to interpretation than those we have encountered so far, and it is no longer clear whether a notion of *literal* performance is valid anymore. The use of traditional terms conjures the image of human performance, further complicating the situation. In general, articulations are subjectively interpreted by humans, and are usually not played the same every time even by the same musician. A legato phrase can be expressed using our notion of musical events by overlapping slightly the beginning of a note with the end of the preceding note, but how *much* overlap should be used, and should it vary with context? A composer of computer music may in fact wish to specify these details, thus some of the constructors listed above are provided with numeric arguments to specify the degree to which the articulation is to be expressed.

We also note that the interpretation of many annotations is *instrument* dependent; for example, a legato phrase will be interpreted quite differently when using a piano versus, say, a violin. We have dealt with this problem by treating instruments as a pair of *functions*:

```
type Instrument = (Pitch -> Dur -> [NoteAttribute] -> Event,
                   [PhraseAttribute] -> Music -> Music)
```

which are used to interpret individual notes and phrases. The relevant lines in the revised definition of `perform` look something like:

```
perform x@(s,i,k,t) m =
  case m of
    Note p d nas -> fst i (p+k) d' nas : perform (s+d',i,k,t) notes
                    where d' = d*60/t
    Phrase pas m -> perform x (snd i pas m)
    ...
```

This "object oriented" approach to instrument definition should permit us to prove properties about the interpretation of note and phrase attributes for specific instruments, although we have not attempted to do so. (The `Event` datatype, by the way, has also been extended to express aspects of dynamics, timbre, etc. that arise from interpretation of the extended `Music` datatype described earlier.)

To aid in the construction of new instruments, we have also defined *default* instruments giving default interpretations to such things as legato and staccato, and from which more complex instruments may be derived. Several such defaults have been defined to capture certain *classes* of instruments such as string, woodwind, and percussion, since members of these classes tend to be constrained by the same physical features.

7 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 Proof of Axiom 10

We first state a simple lemma (proof omitted):

```
dur (Line (Note p d : ns))  ≡  d + dur (Line ns)
 dur (Line (Rest d : ns))  ≡  d + dur (Line ns)
```

With this lemma, the proof of Axiom 10 is as follows:

```
pf x@(s,_,_,t) (Line l1 :+: Line l2)
= pf x (Line l1) ++ pf (s+dur(Line l1),_,_,t) (Line l2)
```

We proceed by induction on `l1`. The basis is when `l1 = []`:

```
= pf x (Line []) ++ pf (s+dur(Line []),_,_,t) (Line l2)
= play x [] ++ pf (s+dur(Line []),_,_,t) (Line l2)
= [] ++ pf (s+dur(Line []),_,_,t) (Line l2)
= pf (s+dur(Line []),_,_,t) (Line l2)
= pf (s,_,_,t) (Line l2)
= pf x (Line ([] ++ l2))
```

Thus proving the basis case. For the induction step, we have `l1 = n:ns`:

```
= pf x (Line (n:ns)) ++ pf (s+(dur (Line (n:ns)))*60/t,_,_,t) (Line l2)
= play x (n:ns) ++ pf (s+(dur (Line (n:ns)))*60/t,_,_,t) (Line l2)
```

At this point there are two possibilities for `n`. First, it could be a note; i.e. `n = Note p d`:

```
= play x (Note p d : ns) ++ pf (s+(dur(Line(Note p d : ns)))*60/t,_,_,t) (Line l2)
= play x (Note p d : ns) ++ pf (s+d'+(dur(Line ns))*60/t,_,_,t) (Line l2)
  where d' = d*60/t
= ((s,_,_,d') : play (s+d',_,_,_) ns) ++
                pf (s+d'+(dur(Line ns))*60/t,_,_,t) (Line l2)
= ((s,_,_,d') : pf (s+d',_,_,_) (Line ns)) ++
                pf (s+d'+(dur(Line ns))*60/t,_,_,t) (Line l2)
= (s,_,_,d') : (pf (s+d',_,_,_) (Line ns) ++
                pf (s+d'+(dur(Line ns))*60/t,_,_,t) (Line l2))
= (s,_,_,d') : pf (s+d',_,_,_) (Line ns :+: Line l2)
= (s,_,_,d') : pf (s+d',_,_,_) (Line (ns++l2))      -- induction hypothesis
= (s,_,_,d') : play (s+d',_,_,_) (ns++l2)
= play (s,_,_,_) (Note p d : (ns++l2))
= play (s,_,_,_) (l1++l2)
= pf (s,_,_,_,) (Line (l1++l2)
```

Thus completing the proof for the induction step when `n` is a note. The proof for the case when `n = Rest d` is almost identical to the above, and is thus omitted.

References

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.

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

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.

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

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.

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

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.

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

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.

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

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

P. Hindemith. *Elementary Training for Musicians*. Associated Music Publishers, Inc., New York, 2 edition, 1949.

P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

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.

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

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.

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.

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

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