The Haskell School of Expression

– Learning Functional Programming through Multimedia –

by

Paul Hudak

Yale University
Department of Computer Science

The Haskell School of Expression
– Learning Functional Programming through Multimedia –

by Paul Hudak

This book is dedicated to
Cathy, Cristina, Jennifer, and Rusty.

Chapter 20

Functional Music Composition

In this chapter I will describe a module for expressing *musical structures* in the same high-level, declarative style of functional programming that we have been using for graphics, animation, and other applications. These musical structures consist of primitive entities (such as notes and rests), operations to transform musical structures (such as transpose and tempo-scaling), and operations to combine musical structures to form more complex ones (such as concurrent and sequential composition). From these simple roots, much richer musical ideas can be easily developed.

For convenience, and in the style of Chapters 15 and 19 (where I defined the languages FAL and IRL, respectively), I will refer to the ideas described in this chapter as MDL, for *music description language*. MDL is a simplified version of a more complete computer music library called *Haskore*. In Chapter 21 a module will be developed for interpreting an MDL program as an abstract *performance*, and in Chapter 22 these performances will be converted into *MIDI files*, which are a standard way of interchanging electronic music and can be played on any PC with a standard sound card.[1]

Details: If you load the Haskell code for Chapter 22 into Hugs you will be able to play any of the examples presented in this chapter

[1]Haskore is described in [**?**, **?**]; see also the Haskell Home Page for information on the latest release. Other approaches to computer music from a functional programming perspective include [**?**, **?**].

by typing either:

```
testWin95 m
testNT m
testLinux m
```

depending on what kind of computer you are using. This command will convert *m* (of type *Music*, to be defined shortly) into a MIDI file, and then automatically invoke the default MIDI file player on your PC so that you can hear the result.

```
module Music where
```

20.1 The Music Data Type

I will assume that you are familiar with very basic musical concepts such as notes, rests, scales, and chords. Nothing more will be needed to understand what is going on, but of course the richer your musical background, the more applications of the ideas will be apparent to you.

Perhaps the most basic musical concept in MDL is that of a *pitch*, which consists of a *pitch class* (i.e. one of 12 semitones) and an *octave*:

```
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, Show)
type Octave      =  Int
```

Cf is read as "C-flat" and normally written as C♭, *Cs* is read "C-sharp" and normally written as C♯; and so on. A *Pitch* is a pair consisting of a pitch class and an octave. Octaves are just integers, but I have defined a separate data type for pitch classes, because distinguishing enharmonics (that is, pitches that sound the same, such as G♯ and A♭) may be important in certain contexts. When tuning instruments or entire orchestras there is a notion of "A440", which is the note A at 440 Hz; by convention, I will designate that pitch as $(A, 4)$ in the above design.

Musical structures are captured in MDL by the *Music* data type:

```
data Music  =  Note Pitch Dur
            |  Rest Dur
            |  Music :+: Music
            |  Music :=: Music
            |  Tempo (Ratio Int) Music
            |  Trans Int Music
            |  Instr IName Music
  deriving (Show, Eq)

type Dur    =  Ratio Int
```

A *Note* is its pitch paired with its duration (in number of whole notes). A *Rest* also has a duration, but of course no pitch. From these two atomic constructors we can build more complex musical structures as follows:

- *m*1 :+: *m*2 is the "sequential composition" of *m*1 and *m*2; i.e. *m*1 and *m*2 are played in sequence.
- *m*1 :=: *m*2 is the "parallel composition" of *m*1 and *m*2; i.e. *m*1 and *m*2 are played simultaneously.
- *Tempo a m* scales the rate at which *m* is played (i.e. its tempo) by a factor of *a*.
- *Trans i m* transposes *m* by interval *i* (in semitones).
- *Instr iname m* declares that *m* is to be performed using instrument *iname*, which is one of 129 names shown in Figure 20.1 (these odd names are from the *General MIDI Standard*, which is explained in more detail in Chapter 21).

It is convenient to represent these ideas in Haskell as a recursive data type because we may wish to not only build musical structures, but also take them apart, analyze their structure, print them, transform them, interpret them for performance purposes, etc. This is the same kind of argument used to justify *Shape*, *Region* and other data types in this text.

Note that durations and tempo scalings are represented using *rational numbers*; specifically, as ratios of two Haskell *Int* values. This is more

```
data IName
    = AcousticGrandPiano | BrightAcousticPiano | ElectricGrandPiano
    | HonkyTonkPiano | RhodesPiano | ChorusedPiano
    | Harpsichord | Clavinet | Celesta | Glockenspiel | MusicBox
    | Vibraphone | Marimba | Xylophone | TubularBells
    | Dulcimer | HammondOrgan | PercussiveOrgan
    | RockOrgan | ChurchOrgan | ReedOrgan
    | Accordion | Harmonica | TangoAccordion
    | AcousticGuitarNylon | AcousticGuitarSteel | ElectricGuitarJazz
    | ElectricGuitarClean | ElectricGuitarMuted | OverdrivenGuitar
    | DistortionGuitar | GuitarHarmonics | AcousticBass
    | ElectricBassFingered | ElectricBassPicked | FretlessBass
    | SlapBass1 | SlapBass2 | SynthBass1 | SynthBass2
    | Violin | Viola | Cello | Contrabass | TremoloStrings
    | PizzicatoStrings | OrchestralHarp | Timpani
    | StringEnsemble1 | StringEnsemble2 | SynthStrings1
    | SynthStrings2 | ChoirAahs | VoiceOohs | SynthVoice
    | OrchestraHit | Trumpet | Trombone | Tuba
    | MutedTrumpet | FrenchHorn | BrassSection | SynthBrass1
    | SynthBrass2 | SopranoSax | AltoSax | TenorSax
    | BaritoneSax | Oboe | Bassoon | EnglishHorn | Clarinet
    | Piccolo | Flute | Recorder | PanFlute | BlownBottle
    | Shakuhachi | Whistle | Ocarina | Lead1Square
    | Lead2Sawtooth | Lead3Calliope | Lead4Chiff
    | Lead5Charang | Lead6Voice | Lead7Fifths
    | Lead8BassLead | Pad1NewAge | Pad2Warm
    | Pad3Polysynth | Pad4Choir | Pad5Bowed
    | Pad6Metallic | Pad7Halo | Pad8Sweep
    | FX1Train | FX2Soundtrack | FX3Crystal
    | FX4Atmosphere | FX5Brightness | FX6Goblins
    | FX7Echoes | FX8SciFi | Sitar | Banjo | Shamisen
    | Koto | Kalimba | Bagpipe | Fiddle | Shanai
    | TinkleBell | Agogo | SteelDrums | Woodblock | TaikoDrum
    | MelodicDrum | SynthDrum | ReverseCymbal
    | GuitarFretNoise | BreathNoise | Seashore
    | BirdTweet | TelephoneRing | Helicopter
    | Applause | Gunshot | Percussion
  deriving (Show, Eq, Ord, Enum)
```

Figure 20.1: General MIDI Instrument Names

accurate than using floating-point numbers, as long as overflow does not occur, and for musical structures it is often just the right representation, where many concepts are often expressed as ratios ("quarter-notes," "triplets," "dotted-notes," etc.). An alternative to this design is to make *Music* polymorphic in the numeric type, but the extra complexity does not seem to be worth the trouble.

Treating pitches simply as integers is useful in many settings, so let's also define a notion of *absolute pitch*:

```
type AbsPitch = Int
```

along with some functions for converting between *Pitch* values and *AbsPitch* values:

```
absPitch            :: Pitch → AbsPitch
absPitch (pc, oct)  =  12 * oct + pcToInt pc

pitch     :: AbsPitch → Pitch
pitch ap  =  ([C, Cs, D, Ds, E, F, Fs, G, Gs, A, As, B] !! mod ap 12,
              quot ap 12)
```

```
pcToInt    :: PitchClass → Int
pcToInt pc  = case pc of
                Cf  → −1      −− should Cf be 11?
                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     −− should Bs be 0?
```

Should *Cf* be interpreted as 11 instead of -1, and *Bs* as 0 instead of 12? I don't know. In most cases it will not matter, but it is an interesting concern.

Details: (!!) is Haskell's zero-based list-indexing function; *list* !! *n* returns the $(n + 1)$th element in *list*. It is defined in the Prelude as:

```
infixl                9  !!
(!!)                  :: [a] → Int → a
(x : _) !! 0          =  x
(_ : xs) !! n | n > 0 =  xs !! (n − 1)
(_ : _) !! _          =  error "PreludeList.!!: negative index"
[ ] !! _              =  error "PreludeList.!!: index too large"
```

mod and *quot* are methods in the *Integral* class. *mod x n* computes the value of *x* modulo n; *quot x n* computes the integer quotient of *x* divided by *n*.

We can also define a function *trans*, which transposes pitches (analogous to *Trans*, which transposes values of type *Music*):

```
trans      :: Int → Pitch → Pitch
trans i p  =  pitch (absPitch p + i)
```

Finally, for convenience, let's create simple names for familiar notes, durations, and rests, as shown in Figure 20.2. Despite the large number of them, these names are sufficiently arcane that name clashes are unlikely.

Exercise 20.1 Prove that *abspitch . pitch = id*, and, up to enharmonic equivalences, *pitch . abspitch = id*.

Exercise 20.2 Prove that *trans i (trans j p) = trans (i + j) p*.

20.2 Higher-Level Constructions

With this modest beginning, we can already express quite a few musical relationships in MDL simply and effectively.

Lines and Chords. 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)
```

For example, from the notes in the C major triad in register 4, I can construct a C major arpeggio and chord as well:

```
cMaj      =  [n 4 qn | n ← [c, e, g]]

cMajArp   =  line cMaj
cMajChd   =  chord cMaj
```

```
cf, c, cs, df, d, ds, ef, e, es, ff, f, fs, gf, g, gs, af, a, as, bf, b, bs
    :: Octave → Dur → 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
dhn, dqn, den, dsn               :: Dur

wnr, hnr, qnr, enr, snr, tnr     :: Music
dhnr, dqnr, denr, dsnr           :: Music

wn   =  1; wnr = Rest wn       -- whole
hn   =  1%2; hnr = Rest hn     -- half
qn   =  1%4; qnr = Rest qn     -- quarter
en   =  1%8; enr = Rest en     -- eight
sn   =  1%16; snr = Rest sn    -- sixteenth
tn   =  1%32; tnr = Rest tn    -- thirty-second

dhn  =  3%4; dhnr = Rest dhn     -- dotted half
dqn  =  3%8; dqnr = Rest dqn     -- dotted quarter
den  =  3%16; denr = Rest den    -- dotted eighth
dsn  =  3%32; dsnr = Rest dsn    -- dotted sixteenth
```

Figure 20.2: Convenient note names and pitch conversion functions.

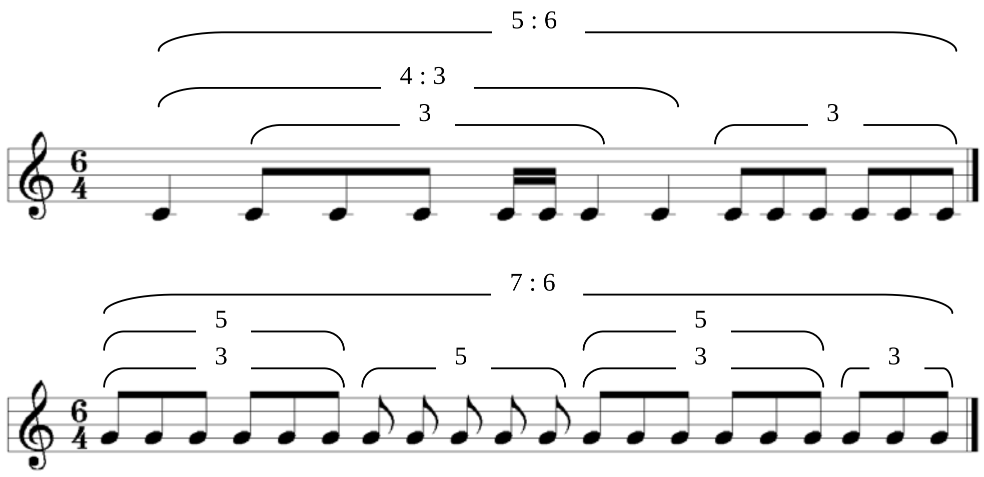

Figure 20.3: Nested Polyrhythms (top: *pr*1; bottom: *pr*2)

Delay and Repeat. Suppose that we wish to describe a melody *m* accompanied by an identical voice a perfect 5th higher. We can write this very simply as "*m* :=: *Trans* 7 *m*" (seven semitones is a perfect fifth). 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 structures. For example, a musical structure 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. This will be explained in more detail later.

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 three eighth notes. In fact *Tempo* can be used to create quite complex rhythmical patterns. For example, consider the “nested polyrhythms” shown in Figure 20.3. They can be expressed quite naturally as follows (note the use of the *where* clause in *pr*2 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 :+:
                 Tempo (3%2) m2)
  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)))
```

Note that *pr*1 and *pr*2 have the same duration: one and one-half beats (if this is not very obvious, then wait till the function *dur*, which computes the duration of *Music* values, is defined below). To play these polyrhythms in parallel using middle C and middle G, respectively, we could do the following (middle C is in the 5th octave):

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

Determining Duration It is sometimes desirable to compute the duration in beats of a musical structure; we can do so 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 m)     =  dur m/a
dur (Trans _ m)     =  dur m
dur (Instr _ m)     =  dur m
```

For example, *dur pr*12 is 3%2 (i.e. one and one-half beats).

Reversing Musical Structure. Using *dur* we can define a function *revM* that reverses any *Music* value. This is straightforward for most *Music* values:

```
revM                :: Music → Music

revM n@(Note _ _)   =  n
revM r@(Rest _)     =  r
revM (Tempo a m)    =  Tempo a (revM m)
revM (Trans i m)    =  Trans i (revM m)
revM (Instr i m)    =  Instr i (revM m)
revM (m1 :+: m2)    =  revM m2 :+: revM m1
```

but the treatment of (:=:) is tricky. The problem is, it is not symetrical with respect to time. Even if *m*1 and *m*2 are *single notes*, if they have different durations, then the reverse of *m*1 :=: *m*2 is not *m*1 :=: *m*2; rather, assuming that *dur m*1 = *d*1 >= *d*2 = *dur m*2, it is
*m*1 :=: (*Rest* (*d*1 − *d*2) :+: *m*2). With this observation, you can see that

the general case is:

```
revM (m1 :=: m2)
    = let d1 = dur m1
          d2 = dur m2
      in if d1 > d2 then revM m1 :=: (Rest (d1 − d2) :+: revM m2)
                    else (Rest (d2 − d1) :+: revM m1) :=: revM m2
```

Truncating Parallel Composition Note that the duration of *m*1 :=: *m*2 is the maximum of the durations of *m*1 and *m*2, and thus if one is infinite, so is the result. Sometimes we would rather have the result be of duration equal to the shorter of the two. This is not as easy as it sounds, since it may require interrupting the longer one in the middle of a note (or notes).

I will define a “truncating parallel composition” operator (/=), but first I will define an auxiliary function *cut* such that *cut d m* is the musical structure *m* “cut short” to have at most duration *d*:

```
cut                :: Dur → Music → Music
cut d m | d <= 0   = Rest 0
cut d (Note x d0)  = Note x (min d0 d)
cut d (Rest d0)    = Rest (min d0 d)
cut d (m1 :=: m2)  = cut d m1 :=: cut d m2
cut d (Tempo a m)  = Tempo a (cut (d ∗ a) m)
cut d (Trans a m)  = Trans a (cut d m)
cut d (Instr a m)  = Instr a (cut d m)
cut d (m1 :+: m2)  = let m1′ = cut d m1
                         m2′ = cut (d − dur m1′) m2
                     in m1′ :+: m2′
```

Note that *cut* is equipped to handle a *Music* value of infinite length.

With *cut*, the definition of (/=:) is now straightforward:

```
(/=:)      :: Music → Music → Music
m1/=: m2   = cut (min (dur m1) (dur m2)) (m1 :=: m2)
```

Unfortunately, whereas *cut* can handle infinite-duration music values, (/=:) cannot, because computing the minimum of the two durations will not terminate as written above.

Exercise 20.3 Define a version of (/=:) that shortens correctly when either one of its arguments is infinite in duration (but not both). (Hint: first define a function *minDur* that returns the minimum duration of two *Music* values, even if one is infinite.) Harder: define a version that works properly even when both arguments are infinite.

Exercise 20.4 Prove that $dur\ (cut\ d\ m) <= d$, for all $d >= 0$.

Trills A *trill* is an ornament that alternates rapidly between two (usually adjacent) pitches. The *Music* value *trill i d n* will be a trill beginning on the pitch of note *n*, with the alternate note being *i* semitones away, and with each trill note having duration *d*. The total duration of *trill i d n* should be the same as the duration of *n*.

```
trill  ::  Int → Dur → Music → Music

trill i d n@(Note p nd)
                          =  if d >= nd then n
                             else Note p d
                                  :+: trill (negate i) d
                                           (Note (trans i p) (nd − d))
trill i d (Tempo a m)     =  Tempo a (trill i (d ∗ a) m)
trill i d (Trans a m)     =  Trans a (trill i d m)
trill i d (Instr a m)     =  Instr a (trill i d m)
trill _ _ _               =  error "Trill input must be a single note"
```

It is simple to define a version of this function that starts on the alternate note rather than the start note:

```
trill'            ::  Int → Dur → Music → Music
trill' i sDur m   =   trill (negate i) sDur (Trans i m)
```

It is also convenient to define a function *roll* which generates a trill whose interval is zero. This feature is particularly useful for percussion.

```
roll          ::  Dur → Music → Music
roll dur m    =   trill 0 dur m
```

Exercise 20.5 Define a function *trilln* :: *Int* → *Int* → *Music* → *Music* which is just like *trill* except that it's second argument is the number of trill notes to be generated, rather than the duration of a single trill note. Also define: (a) *trilln′* which is to *trilln* as *trill′* is to *trill*, and (b) *rolln* which is to *roll* and *trilln* is to *trill*.

Here is a simple example of the use of *trill* and *trilln* in expressing the opening flute line in John Philip Sousa's *Stars and Stripes Forever*:

```
ssfMelody  =  m1 :+: m2 :+: m3 :+: m4

m1         =  trilln 2 5 (bf 6 en) :+:
              line [ef 7 en, ef 6 en, ef 7 en]

m2         =  line [bf 6 sn, c 7 sn, bf 6 sn, g 6 sn, ef 6 en, bf 5 en]

m3         =  line [ef 6 sn, f 6 sn, g 6 sn, af 6 sn, bf 6 en, ef 7 en]

m4         =  trill 2 tn (bf 6 qn) :+: bf 6 sn :+: denr

ssf        =  Instr Flute (Tempo 2 (ssfMelody))
```

Exercise 20.6 Prove that *dur* (*trill i d n*) = *dur n*.

Percussion Speaking of percussion, how do we express that in the MDL framework? Percussion is a difficult notion to represent in the abstract, since in a way, a percussion instrument is just another instrument, so why should it be treated differently? On the other hand, even common practice notation treats it specially, even though it has much in common with non-percussion notation. The MIDI standard is equally ambiguous about the treatment of percussion: on one hand, percussion sounds are chosen by specifying an octave and pitch, just like any other instrument, on the other hand these notes have no tonal meaning whatsoever: they are just a conveneient way to select from a large number of percussion sounds. Indeed, part of the General MIDI Standard is a set of names for commonly used percussion sounds.

Since MIDI is such a popular platform, we can at least define some handy functions for using the General MIDI Standard. We start by defining the

```
data PercussionSound
    = AcousticBassDrum    -- MIDI Key 35
    | BassDrum1    -- MIDI Key 36
    | SideStick    -- ...
    | AcousticSnare | HandClap | ElectricSnare | LowFloorTom
    | ClosedHiHat | HighFloorTom | PedalHiHat | LowTom
    | OpenHiHat | LowMidTom | HiMidTom | CrashCymbal1
    | HighTom | RideCymbal1 | ChineseCymbal | RideBell
    | Tambourine | SplashCymbal | Cowbell | CrashCymbal2
    | Vibraslap | RideCymbal2 | HiBongo | LowBongo
    | MuteHiConga | OpenHiConga | LowConga | HighTimbale
    | LowTimbale | HighAgogo | LowAgogo | Cabasa
    | Maracas | ShortWhistle | LongWhistle | ShortGuiro
    | LongGuiro | Claves | HiWoodBlock | LowWoodBlock
    | MuteCuica | OpenCuica | MuteTriangle
    | OpenTriangle    -- MIDI Key 82
      deriving (Show, Eq, Ord, Ix, Enum)
```

Figure 20.4: General MIDI Percussion Names

data type shown in Figure 20.4, which borrows its constructor names from the General MIDI standard. The comments reflecting the "MIDI Key" numbers will be explained later, but basically a MIDI Key is the equivalent of an absolute pitch in our terminology. So all we need is a way to convert these percussion sound names into a *Music* value; i.e. a *Note*:

```
perc    :: PercussionSound → Dur → Music
perc ps  = Note (pitch (fromEnum ps + 35))
```

Details: Recall that *fromEnum* is a method in the *Enum* class, and has type $(Ord\ a, Enum\ a) \Rightarrow a \rightarrow Int$ (see Chapter 24).

Since *PercussionSound* is a (derived) instance of *Enum*, *fromEnum ps* returns $(n - 1)$ if percussion sound *ps* is the *n*th constructor in the *PercussionSound* data type. Adding 35 to this yields the correct absolute pitch for this sound according to the General MIDI standard, and converting this into a pitch allows us to use the *Note* constructor.

For example, here are eight bars of a simple rock or "funk groove" that uses *perc* and *roll*:

```
funkGroove
    = let p1   =  perc LowTom qn
          p2   =  perc AcousticSnare en
      in Tempo 3 (Instr Percussion (cut 8 (repeatM
           ((p1 :+: qnr :+: p2 :+: qnr :+: p2 :+:
             p1 :+: p1 :+: qnr :+: p2 :+: enr)
            :=: roll en (perc ClosedHiHat 2))
         )))
```

Exercise 20.7 Find a simple piece of music written by your favorite composer, and transcribe it into Haskell. 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.

Exercise 20.8 If you are familiar with the terms, define Haskell functions *invert*, *retro*, *retroInvert*, and *invertRetro* to implement the concepts of inversion, retrograde, retrograde inversion, and inverted retrograde, respectively, as used in twelve-tone music theory. You may assume that the input is to these functions is created by an application of *line* above. Prove that "*retro* . *retro*," "*invert* . *invert*," and "*retroInvert* . *invertRetro*" are the identity function on values created by *line*.

Exercise 20.9 A shortcoming of our current design of musical values is that there is no representation of *dynamics*; i.e. loudness, or volume. There are several ways we could deal with this, the most straightforward being to add a constructor to the *Music* data type that expressed volume. Explore some designs for this, including direct numerical representation of the volume, as well as more traditional notations such as *pianissimo*, *piano*, *mezzo piano*, *mezzo forte*, *forte*, and *fortissimo* (often abbreviated *pp*, *p*, *mp*, *mf*, *f*, and *ff*, respectively, and for which a Haskell data type could easily be defined).

Other notions of dynamics include *legato*, *staccato*, *slurring*, *crescendo*, and *diminuendo*. In addition, with respect to tempo, there are notions of *ritardando* and *accelerando*.

If these notions are simply captured in the *Music* data type, then the most interesting aspect will be their *interpretation* in Chapter *ch* : *performance*.

Exercise 20.10 Define a data type *Mode* that enumerates the seven scale modes: ionian, dorian, phrygian, lydian, mixolydian, aeolian, and locrian. Then define a function *scale* that, given a mode and tonic (a start note), generates a scale in that mode starting on the tonic.

20.3 A Final Couple of Examples

In this section I will briefly explore two ideas for "algorithmic composition:" the idea of writing fairly concise expressions that yield interesting (hopefully...) music.

Cascades. Here is a function that recursively applies transformations *f* (to elements in a sequence) and *g* (to accumulated phrases) a total of *n* times, playing everything in unison:

$$\begin{array}{lcl} rep & :: & (Music \to Music) \to (Music \to Music) \to Int \to Music \to Music \\ rep\ f\ g\ 0\ m & = & Rest\ 0 \\ rep\ f\ g\ n\ m & = & m :=: g\ (rep\ f\ g\ (n-1)\ (f\ m)) \end{array}$$

For example, here is a rising arpeggio of perfect fourths beginning on middle C:

$$run \;=\; rep\ (Trans\ 5)\ (delay\ tn)\ 8\ (c\ 4\ tn)$$

Now suppose we use this entire phrase as an argument to *rep*, transposing it a major third each time, and delaying each phrase by an eighth note:

$$cascade \;=\; rep\ (Trans\ 4)\ (delay\ en)\ 8\ run$$

Let's do this again, this time only to repeat the phrase once, a few beats later:

$$cascades \;=\; rep\ id\ (delay\ sn)\ 2\ cascade$$

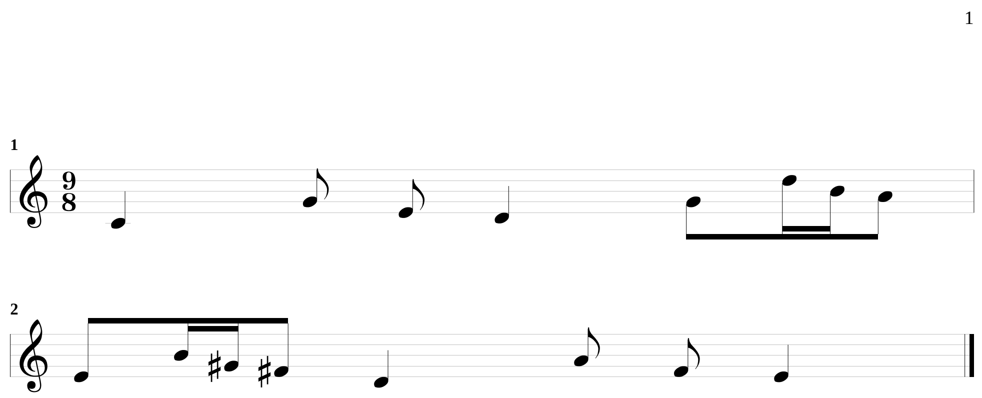

Figure 20.5: An Example of Self-Similar Music

The result is a “cascade” of sound that has some interesting properties. All of this was generated from a single starting note. We can create a final result, a cascade palindrome which I will call a “waterfall,” played using a vibraphone as follows:

$$waterfall \quad = \quad Instr\ Vibraphone\ (cascades :+: revM\ cascades)$$

Exercise 20.11 Experiment with this idea futher, using other fragments to start the process, and other transformations.

Self-Similar (Fractal) Music. Fractal images were discussed briefly in Chapter 3. But what does it mean to have fractal *music*? There are actually several notions of what this might be. My notion is as follows: start with a very simple melody of *n* notes. Now duplicate this melody *n* times, playing each in succession, but first performing the following transformation: the *i*th melody is transposed by an amount proportional to the pitch of the *i*th note in the original melody, and is shifted in tempo by a factor proportional to the duration of the *i*th note. For example, Figure 20.5 shows the result of applying this process once to a four-note melody. Now imagine that this process is repeated infinitely often, yielding an infinitely dense melody of infinitesimally shorter notes! To make the result playable, however, we will stop the process at some pre-determined level.

How can this be represented in Haskell? A *tree* seems to be the logical choice, which I will call a *Cluster*:

```
data Cluster  =  Cluster SNote [Cluster]
type SNote    =  (AbsPitch, Dur)
```

This particular kind of tree happens to be called a *rose tree*. An *SNote* is just a “simple note.”

The sequence of *SNote*s at each level of the cluster is the melodic fragment for that level. The very top cluster will contain a “dummy” note, the next level will contain the original melody, the next level will contain one iteration of the process described earlier (e.g. the melody in Figure 20.5), and so forth.

To achieve this I will define a function *selfSim* that takes the initial melody as argument and generates an infinitely deep cluster:

```
selfSim                  ::  [SNote] → Cluster
selfSim pat              =   Cluster (0, 0) (map mkCluster pat)
  where mkCluster note
              =  Cluster note (map (mkCluster . addmult note) pat)

addmult (p0, d0) (p1, d1)  =  (p0 + p1, d0 ∗ d1)
```

Note that *selfSim* itself is not recursive, but *mkCluster* is.

Next, I define a function to skim off the notes at the *n*th level, or *n*th “fringe,” of a cluster:

```
fringe                     ::  Int → Cluster → [SNote]
fringe 0 (Cluster note cls)  =  [note]
fringe n (Cluster note cls)  =  concat (map (fringe (n − 1)) cls)
```

Details: Recall that *concat* appends together a list of lists. It is defined in the Standard Prelude as:

```
concat      ::  [[a]] → [a]
concat xss  =   foldr (++) [ ] xss
```

All that is left to do is convert this into a *Music* value that we can convert to MIDI:

```
simToHask    :: [SNote] → Music
simToHask ss  = let mkNote (p, d)  =  Note (pitch p) d
                in line (map mkNote ss)
```

Putting it all together, below is a small composition whose seed is the four-note melody:

```
pat :: [SNote]
pat  = [(3, 0.5), (4, 0.25), (0, 0.25), (6, 1.0)]
```

Note that the flute line and acoustic bass line are the reverse of one another; the result is rather interesting.

```
main
   = let s  =  Trans 60
                 (Tempo 2
                    (simToHask (fringe 3 (selfSim pat))))
     in Instr Flute s
        :=: Instr AcousticBass (Trans (−24) (revM s))
```

Exercise 20.12 Experiment with this idea futher, using other melodic seeds, exploring different depths of the clusters, and so on.

Exercise 20.13 *fringe* is not very efficient, for the following reason:

concat is defined as *foldr* (++) [ ], which means that it takes a number of steps proportional to the sum of the lengths of the lists being concatenated; we cannot do any better than this. (If *foldl* were used instead, the number of steps would be proportional to the number of lists times their average length.)

The problem is, *concat* is being used over and over again, like this:

```
concat [concat [...], concat [...], concat [...]]
```

This causes a number of steps proportional to the depth of the tree times the length of the sub-lists; clearly not optimal.

Define a version of *fringe* that is linear in the total length of the final list.

Chapter 21

Interpreting Functional Music

In Chapter 20 I defined a language called MDL for describing musical *structures*. The question now is, how do we actually *interpret* the structures; that is, how do we turn them into real music? (This is analogous to the question of how to draw a *Shape* or *Region* value in a graphics window.) The approach I will take will be to convert a *Music* value into a *Standard MIDI File*, which can then be played on your computer using any standard media player. I will do this, however, in three steps:

- First I will convert a *Music* value into a value of type *Performance*, which is an abstract notion of what the music *means*.
- Then I will convert this into a value of type *MidiFile*, a data type imported from the *Haskore* library[1] that represents the structure of a Standard MIDI File.
- Finally, I will use the *outputMidiFile* function, also imported from the *Haskore* library, to write this *MidiFile* value to an actual file.

Dividing the problem into separate steps like this allows us to separate two concerns: interpreting *Music* values as music, and the details of rendering that music as a MIDI file. It also facilitates readability by having a cleaner structuring of the code, aids debugging by allowing us to look at

[1]See the Preface for instructions on how to obtain this module if it is not installed on your Haskell system.

each intermediate result, and makes it easier to later add an interface to some computer music platform other than MIDI (there are several).

In this chapter I will describe only the first step above; steps two and three are described in the next (Chapter 22). Also in this chapter I will discuss how the abstract notion of a performance can be used to uncover *algebraic properties* of musical structures.

```
module Perform where

import Music
```

21.1 Interpreting Music: A *Performance*

Our first goal is to interpret a *Music* value as an abstract *performance*, which is a temporally ordered sequence of musical *events*:

```
type Performance  =  [Event]

data Event        =  Event {eTime :: Time, eInst :: IName,
                            ePitch :: AbsPitch, eDur :: DurT}
  deriving (Eq, Ord, Show)

type Time         =  Float
type DurT         =  Float
```

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

To generate a complete performance of, i.e. give an interpretation to, a musical structure expressed in MDL, we must know the time to begin the performance, the default instrument to use, and the proper key and tempo. We can thus model a “performer” as a function *perform* which uses all of this information (which I will call the *context*) to translate a musical structure into a performance; it can also be thought of as an interpeter for an MDL program:

```
perform  ::  Context → Music → Performance
```

```
data Context   =  Context {cTime :: Time, cInst :: IName,
                           cDur :: DurT, cKey :: Key}
  deriving Show

type Key       =  AbsPitch
```

The *cDur* :: *DurT* component of the context is the duration, in seconds, of one whole note. To make it easier to compute this, 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 ∗ ratioToFloat dur)
```

For example, *metro* 96 *qn* creates a tempo of 96 quarter notes per minute.

metro uses the following coercion function, which will also be used several times later:

```
ratioToFloat      ::  Ratio Int → Float
ratioToFloat r    =   intToFloat (numerator r)/intToFloat (denominator r)


intToFloat   ::  Int → Float
intToFloat   =   fromInteger . toInteger
```

Details: *numerator* and *denominator* extract the numerator and denominator, respectively, from a *Ratio* value.

The definition of *perform* is relatively straightforward, so I will present it all at once:

```
perform c@(Context t i dt k) m =
  case m of
    Note p d     →  let d′ = ratioToFloat d ∗ dt
                    in [Event t i (transpose p k i) d′]
    Rest d       →  [ ]
    m1 :+: m2    →  perform c m1 ++
                    perform
                      (c {cTime = t + ratioToFloat (dur m1) ∗ dt}) m2
    m1 :=: m2    →  merge (perform c m1) (perform c m2)
    Tempo a m    →  perform (c {cDur = dt/ratioToFloat a}) m
    Trans p m    →  perform (c {cKey = k + p}) m
    Instr nm m   →  perform (c {cInst = nm}) m
  where transpose p k Percussion  =  absPitch p
        transpose p k _           =  absPitch p + k
```

A single note is translated into a single-event performance. Note that the pitch is transposed to correspond to the key, with one catch: no transposition is done to *Percussion*, since the note corresponds to the actual percussion instrument. A rest translates into an empty performance. Note how the *Context* is used as the running "state" of the performance, and gets updated in several different ways. For example, the interpretation of the *Tempo* constructor involves scaling *dt* appropriately and updating the *cDur* field of the context.

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 20.2) is used to compute this duration. Unfortunately, this strategy generates a number of steps proportional to the square of the size of the *Music* value. A more efficient solution is to have *perform* compute the duration directly, returning it as part of its result. This version of *perform* is shown in Figure 21.1.

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

```
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 21.2. Here is a more efficient version that will work just as well in practice:

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

Exercise 21.1 Prove that the two versions of *perform* are equivalent.

21.2 An Algebra of Music

A *literal performance* is a performance in which no aesthetic interpretation is given to a musical object. The function *perform* in fact yields a literal performance for an MDL program.

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

Thus I will define a formal notion of equivalence:

```
perform          ::  Context → Music → Performance
perform c m      =   fst (perf c m)

perf             ::  Context → Music → (Performance, DurT)

perf c@(Context t i dt k) m   =
  case m of
    Note p d     →  let d'  =  ratioToFloat d ∗ dt
                    in ([Event t i (transpose p k i) d'], d')
    Rest d       →  ([ ], ratioToFloat d ∗ dt)
    m1 :+: m2    →  let (pf1, d1)  =  perf c m1
                        (pf2, d2)  =  perf (c {cTime = t + d1}) m2
                    in (pf1 ++ pf2, d1 + d2)
    m1 :=: m2    →  let (pf1, d1)  =  perf c m1
                        (pf2, d2)  =  perf c m2
                    in (merge pf1 pf2, max d1 d2)
    Tempo a m    →  perf (c {cDur = dt/ratioToFloat a}) m
    Trans p m    →  perf (c {cKey = k + p}) m
    Instr nm m   →  perf (c {cInst = nm}) m
  where transpose p k Percussion  =  absPitch p
        transpose p k _           =  absPitch p + k
```

Figure 21.1: An efficient *perform* function.

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

$$(\forall\, c)\;\; perform\ c\ m1 \;\;\Rightarrow\;\; perform\ c\ m2$$

(Note the similarity of this to the notion of equivalence of regions defined in Chapter 8.)

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 $r1$, $r2$, and m:

$$Tempo\ r1\ (Tempo\ r2\ m) \;\;\equiv\;\; Tempo\ (r1 * r2)\ m$$

We can prove this axiom by calculation. For clarity I will simplify the context to just dt, the tempo duration, and will write rtf as shorthand for $ratioToFloat$:

$$
\begin{array}{l}
perform\ dt\ (Tempo\ r1\ (Tempo\ r2\ m)) \\
\;\Rightarrow\;\; \{\ \text{unfold}\ perform\ \} \\
perform\ (dt/rtf\ r1)\ (Tempo\ r2\ m) \\
\;\Rightarrow\;\; \{\ \text{unfold}\ perform\ \} \\
perform\ ((dt/rtf\ r1)/(rtf\ r2))\ m \\
\;\Rightarrow\;\; \{\ \text{arithmetic}\ \} \\
perform\ (dt/((rtf\ r1) * (rtf\ r2)))\ m \\
\;\Rightarrow\;\; \{\ \text{lemma for}\ ratioToFLoat\ \} \\
perform\ (dt/(rtf\ (r1 * r2)))\ m \\
\;\Rightarrow\;\; \{\ \text{fold}\ perform\ \} \\
perform\ dt\ (Tempo\ (r1 * r2)\ m)
\end{array}
$$

Here is another useful transformation and its validity proof (for clarity I will simplify the context to just $(t,\ dt)$, the start time and tempo):

Axiom 2 For any r, $m1$, and $m2$:

$$Tempo\ r\ (m1 :+: m2) \equiv Tempo\ r\ m1 :+: Tempo\ r\ m2$$

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

Proof:

$$\begin{array}{l}
perform\ (t, dt)\ (Tempo\ r\ (m1 :+: m2)) \\
\ \Rightarrow\ \{ \text{unfold } perform\ \} \\
perform\ (t, dt/rtf\ r)\ (m1 :+: m2) \\
\ \Rightarrow\ \{ \text{unfold } perform\ \} \\
perform\ (t, dt/rtf\ r)\ m1 +\!\!+ perform\ (t1, dt/rtf\ r)\ m2 \\
\ \Rightarrow\ \{ \text{fold } perform\ \} \\
perform\ (t, dt)\ (Tempo\ r\ m1) +\!\!+ perform\ (t1, dt)\ (Tempo\ r\ m2) \\
\ \Rightarrow\ \{ \text{arithmetic} \} \\
perform\ (t, dt)\ (Tempo\ r\ m1) +\!\!+ perform\ (t2, dt)\ (Tempo\ r\ m2) \\
\ \Rightarrow\ \{ \text{fold } dur\ \} \\
perform\ (t, dt)\ (Tempo\ r\ m1) +\!\!+ perform\ (t3, dt)\ (Tempo\ r\ m2) \\
\ \Rightarrow\ \{ \text{fold } perform\ \} \\
perform\ (t, dt)\ (Tempo\ r\ m1 :+: Tempo\ r\ m2) \\
\quad \textbf{where}\ t1 = t + rtf\ (dur\ m1) * (dt/rtf\ r) \\
\qquad\qquad t2 = t + rtf\ (dur\ m1/r) * dt \\
\qquad\qquad t3 = t + rtf\ (dur\ (Tempo\ r\ m1)) * dt
\end{array}$$

An even simpler axiom is given by:

Axiom 3 For any m:

$$Tempo\ 1\ m \equiv m$$

In other words, *unit tempo scaling is the identity function for type Music*.

Proof:

$perform\ (t, dt)\ (Tempo\ 1\ m)$
$\ \Rightarrow$ { unfold $perform$ }
$perform\ (t, dt/rtf\ 1)\ m$
$\ \Rightarrow$ { arithmetic }
$perform\ (t, dt)\ m$

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 r, $m1$, and $m2$:

$$Tempo\ r\ m1 :+: m2 \ \equiv\ Tempo\ r\ (m1 :+: Tempo\ (1/r)\ m2)$$

Proof:

$Tempo\ r\ (m1 :+: Tempo\ (1/r)\ m2)$
$\ \Rightarrow$ { by Axiom 2 }
$Tempo\ r\ m1 :+: Tempo\ r\ (Tempo\ (1/r)\ m2)$
$\ \Rightarrow$ { by Axiom 1 }
$Tempo\ r\ m1 :+: Tempo\ (r * (1/r))\ m2$
$\ \Rightarrow$ { arithmetic }
$Tempo\ r\ m1 :+: Tempo\ 1\ m2$
$\ \Rightarrow$ { by Axiom 3 }
$Tempo\ r\ m1 :+: m2$

Many other interesting transformations of MDL musical objects can be stated and proved correct via calculation. I leave as an exercise the proofs of the axioms listed below (which include the above axioms as special cases). In general, axioms such as these constitute a set of *domain-specific* properties that often capture the essence of the domain under consideration. Indeed, it is possible to start with these properties as the specification of the system that is being designed. This approach is commonly referred to as *algebraic semantics*, but I will not pursue the idea here.

Axiom 4 *Tempo* is *multiplicative* and *Transpose* is *additive*. That is, for any $r1$, $r2$, $p1$, $p2$, and m:

$$\begin{aligned} &\mathit{Tempo}\ r1\ (\mathit{Tempo}\ r2\ m) \ \equiv\ \mathit{Tempo}\ (r1 * r2)\ m \\ &\mathit{Trans}\ p1\ (\mathit{Trans}\ p2\ m) \ \equiv\ \mathit{Trans}\ (p1 + p2)\ m \end{aligned}$$

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

$$\begin{aligned} &\mathit{Tempo}\ r1 \,.\, \mathit{Tempo}\ r2 \ \equiv\ \mathit{Tempo}\ r2 \,.\, \mathit{Tempo}\ r1 \\ &\mathit{Trans}\ p1 \,.\, \mathit{Trans}\ p2 \ \equiv\ \mathit{Trans}\ p2 \,.\, \mathit{Trans}\ p1 \\ &\mathit{Tempo}\ r1 \,.\, \mathit{Trans}\ p1 \ \equiv\ \mathit{Trans}\ p1 \,.\, \mathit{Tempo}\ r1 \end{aligned}$$

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

$$\begin{aligned} &\mathit{Tempo}\ r\ (m1 :+: m2) \ \equiv\ \mathit{Tempo}\ r\ m1 :+: \mathit{Tempo}\ r\ m2 \\ &\mathit{Tempo}\ r\ (m1 :=: m2) \ \equiv\ \mathit{Tempo}\ r\ m1 :=: \mathit{Tempo}\ r\ m2 \\ &\mathit{Trans}\ p\ (m1 :+: m2) \ \equiv\ \mathit{Trans}\ p\ m1 :+: \mathit{Trans}\ p\ m2 \\ &\mathit{Trans}\ p\ (m1 :=: m2) \ \equiv\ \mathit{Trans}\ p\ m1 :=: \mathit{Trans}\ p\ m2 \end{aligned}$$

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

$$\begin{aligned} &m0 :+: (m1 :+: m2) \ \equiv\ (m0 :+: m1) :+: m2 \\ &m0 :=: (m1 :=: m2) \ \equiv\ (m0 :=: m1) :=: m2 \end{aligned}$$

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

$$m0 :=: m1 \ \equiv\ m1 :=: m0$$

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

$$\begin{aligned} &\mathit{Tempo}\ r\ (\mathit{Rest}\ 0) \ \equiv\ \mathit{Rest}\ 0 \\ &\mathit{Trans}\ p\ (\mathit{Rest}\ 0) \ \equiv\ \mathit{Rest}\ 0 \\ &m :+: \mathit{Rest}\ 0 \ \equiv\ m \ \equiv\ \mathit{Rest}\ 0 :+: m \\ &m :=: \mathit{Rest}\ 0 \ \equiv\ m \ \equiv\ \mathit{Rest}\ 0 :=: m \end{aligned}$$

Exercise 21.2 Establish the validity of each of the above axioms.

Exercise 21.3 Prove that:

$$(m0 :+: m1) :=: (m2 :+: m3) \quad \equiv \quad (m0 :=: m2) :+: (m1 :=: m3)$$

if $dur\ m0 = dur\ m2$.

Exercise 21.4 Recall the function *revM* defined in Chapter 20, and note that, in general, $revM\ (revM\ m)$ is not equal to m. However, the following is true:

$$revM\ (revM\ m) \quad \equiv \quad m$$

Prove this fact by calculation.

Chapter 22

From Performance to MIDI

MIDI is shorthand for "Musical Instrument Digital Interface," and is a standard protocol for describing electronic music. In this chapter I will describe how to convert an abstract *performance* as defined in Chapter 21 into a *MIDI file* that can be played on any modern PC with a standard sound card.

```
module MDL where

import Music
import Perform
import Haskore (MidiFile (..), MidiChannel, ProgNum, MEvent,
                MFType, Velocity, MEvent (..), MidiEvent (..),
                MetaEvent (..), Division (..), MTempo,
                outputMidiFile)
import List (partition)
import System (system)
```

As mentioned in Chapter 20, *Haskore* is a library for computer music that is more extensive than MDL, and I will borrow much of the basic MIDI data types defined there, as well as the low-level function *outputMidiFile*, to be described later.

22.1 An Introduction to MIDI

MIDI is a standard 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, pedal 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. The constructor names in the *IName* data type (see Figure 20.1 in Chapter 20) come directly from this standard.

Most sound cards on conventional PC's know about MIDI and General MIDI. The sound generated by such modules, even through the typically-scrawny speakers on most PC's, is pretty good these days. For the best sound, an outboard keyboard or tone generator, attached to a computer via a MIDI cable at one end, and to a nice stereo system on the other, will provide the best sound. 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, 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.

If you decide to use the General MIDI features of your sound-card, you need to know about one other convention, namely that Channel 10 (9 in our 0-based numbering) is dedicated to *percussion*. I will use this assumption in this chapter.

22.2 The Conversion Process

Figure 22.1 is a specification, imported from the *Haskore* library, that contains as much of the *MidiFile* datatype that we will need. The details of this datatype are unimportant, except for the following points:

1. There are three types of MIDI files; the value of *MFType* makes the distinction:

   (a) A Format 0 MIDI file stores its information in a single track of events, and is best used only for monophonic music.

   (b) A Format 1 MIDI file stores its information in multiple tracks that are played simultaneously, and offers the advantage of being able to devote each track to one voice in a polyphonic piece.

   (c) A Format 2 MIDI file also has multiple tracks, but they are temporally independent.

   In this chapter we will only use Format 1, so the *MFType* field will always be 1.

2. The *Division* field refers to the "time-code division," or timing strategy, used by the MIDI file. We will always use 96 time divisions, or "ticks," per quarter-note, and thus this field will always be *Ticks* 96.

3. The main body of a MIDI file is a list of *Track*s, each of which in turn is a list of time-stamped (using the *ElapsedTime* field) *MEvent*s. There are two kinds of *MEvent*s: *MidiEvent*s and *MetaEvents*. Figure 22.1 shows just those instances of these events that we are interested in:

   (a) *NoteOn ch p v* turns on note (pitch) *p* with velocity (volume) *v* on MIDI channel *ch*. *NoteOff ch p v* performs a similar function in turning the note off. The volume is an integer in the range 0 to 127; we will always use the maximum volume 127.

   (b) *ProgChange ch pr* sets the program number for channel *ch* to *pr*. This is how an instrument is selected.

   (c) *SetTempo t* sets the tempo to *t*. For Format 1 MIDI files, *t* is the time, in microseconds, of one whole note. Using 120 beats per minute as the norm, or 2 beats per second, that works out to 500,000 microseconds per beat, which is the default value that we will use.

With this structure in mind, our goal is to define a function *performToMidi* which converts a *Performance* into the *MidiFile* data type:

```
performToMidi      :: Performance → MidiFile
performToMidi pf   =
  MidiFile mfType (Ticks division)
                  (map performToMEvs (splitByInst pf))

mfType   = 1 :: Int
division = 96 :: Int
```

There are two yet-to-be-defined functions here: *performToMEvs* and *splitByInst*.

Since we are implementing Type 1 MIDI Files, we must associate each instrument with a separate track. That is the purpose of *splitByInst*, which takes a performance *pf* and returns a list of performances, one for each unique instrument in *pf*. As part of this process it also assigns a unique channel number to each instrument, along with the appropriate program number to select the proper instrument. Thus:

```
splitByInst :: Performance → [(MidiChannel, ProgNum, Performance)]
```

Remember that channel 9 is reserved for percussion, so a special case is made for that; the other channels are selected sequentially in the range 0 to 15 (excluding 9). With this strategy there can be at most 16 instruments (15 if percussion is not used), and thus an error is signalled if this

```
data MidiFile      =  MidiFile MFType Division [Track]
  deriving (Show, Eq)

type MFType        =  Int
type Track         =  [MEvent]

data Division      =  Ticks Int | SMPTE Int Int
  deriving (Show, Eq)

data MEvent        =  MidiEvent ElapsedTime MidiEvent
                   |  MetaEvent ElapsedTime MetaEvent
                   |  NoEvent
  deriving (Show, Eq)

type ElapsedTime   =  Int

    -- Midi Events
data MidiEvent     =  NoteOff MidiChannel MPitch Velocity
                   |  NoteOn MidiChannel MPitch Velocity
                   |  ProgChange MidiChannel ProgNum
                   |  ...
  deriving (Show, Eq)
type MPitch        =  Int
type Velocity      =  Int
type ProgNum       =  Int
type MidiChannel   =  Int

    -- Meta Events
data MetaEvent   =  SetTempo MTempo
                 |  ...
  deriving (Show, Eq)
type MTempo      =  Int
```

Figure 22.1: Partial Definition of *MidiFile* Data Type

number is exceeded:

```
splitByInst p
    = aux 0 p where
        aux n [ ]  =  [ ]
        aux n pf   =  let i          =  eInst (head pf)
                          (pf1, pf2) =  partition (\e → eInst e == i) pf
                          n'         =  if n == 8 then 10 else n + 1
                      in if i == Percussion
                         then (9, 0, pf1) : aux n pf2
                         else if n > 15
                              then error "No more than 16 instruments allowed"
                              else (n, fromEnum i, pf1) : aux n' pf2
```

Details: *partition* is imported from the *List* Standard Library module. It takes a predicate and a list and returns a pair of lists: those elements of the argument list that do and do not satisfy the predicate, respectively. *partition* is defined by:

```
partition       ::  (a → Bool) → [a] → ([a], [a])
partition p xs  =
  foldr select ([ ], [ ]) xs
        where select x (ts, fs) | p x        =  (x : ts, fs)
                                | otherwise  =  (ts, x : fs)
```

Note how *partition* is used to group into *pf*1 those events that use the same instrument as the first event in the performance. The rest of the events are collected into *pf*2, which is passed recursively to the next iteration of the *aux* loop.

The crux of the conversion process is *performToMEvs*, which converts a *Performance* into a stream of *MEvents* (i.e. a *Track*).

```
performToMEvs  ::  (MidiChannel, ProgNum, Performance) → [MEvent]
performToMEvs (ch, pn, perf)
    = let setupInst     = MidiEvent 0 (ProgChange ch pn)
          setTempo      = MetaEvent 0 (SetTempo tempo)
          loop []       = []
          loop (e : es) = let (mev1, mev2)  =  mkMEvents ch e
                          in mev1 : insertMEvent mev2 (loop es)
      in setupInst : setTempo : loop perf

tempo  =  500000 :: Int      -- number of microseconds in one beat
```

An important source of incompatibilty between our abstract notion of a performance and that of MIDI is that in a performance a note is represented as one event with an onset and a duration, while in MIDI it is represented as two separate events, a note-on event and a note-off event. Thus *MkMEvents* turns an *Event* into two *MEvents*, a *NoteOn* and a *NoteOff*.

```
mkMEvents  ::  MidiChannel → Event → (MEvent, MEvent)

mkMEvents mChan (Event {eTime = t, ePitch = p, eDur = d})
            =  (MidiEvent (toDelta t) (NoteOn mChan p 127),
                MidiEvent (toDelta (t + d)) (NoteOff mChan p 127))

toDelta t  =  round (t * 4.0 * intToFloat division)
```

The time-stamp associated with an event in MIDI is called a *delta-time*, and is the time at which the event should occur expressed in time-code divisions since the beginning of the performance. Since there are 96 time-code divisions per quarter note, there are 4 times that many in a whole note; multiplying that by the time-stamp on one of our *Event*s gives us the proper delta-time.

In the code for *performToMEvs*, note that the location of the first event returned from *mkMEvents* is obvious; it belongs just where it was created. However, the second event must be inserted into the proper place in the rest of the stream of events; there is no way to know of its proper position ahead of time. The function *insertMEvent* is thus used to insert an *MEvent* into an already time-ordered sequence of *MEvent*s.

```
insertMEvent  ::  MEvent → [MEvent] → [MEvent]
insertMEvent ev1 [ ]
    = [ev1]
insertMEvent ev1@(MidiEvent t1 _) evs@(ev2@(MidiEvent t2 _) : evs′)
    = if t1 <= t2 then ev1 : evs
                  else ev2 : insertMEvent ev1 evs′
```

22.3 Putting It All Together

We are almost done. All that remains is to write the *MidiFile* value into a real file. The details of this are surprisingly ugly, however, primarily because MIDI files were invented at a time when disk space was precious, and thus a compact bit-level representation was chosen. Fortunately, there is a function in the *Haskore* library that solves this problem for us:

```
outputMidiFile :: String → MidiFile → IO ()
```

To make this easier to use, let's define a function *test* that converts a *Music* value using a default *Context* into a *MidiFile* value, and then writes that to a file "`test.mid`":

```
test     :: Music → IO ()
test m   =  outputMidiFile "test.mid"
              (performToMidi (perform defCon m))

defCon   :: Context
defCon   =  Context {cTime = 0,
                     cInst = AcousticGrandPiano,
                     cDur = metro 120 qn,
                     cKey = 0}
```

So if you type *test m* for some *Music* value *m*, it will be converted to MIDI and written to the file "`test.mid`", which you can then play using whatever MIDI-file player is supplied with your computer. If you are running the Hugs implementation of Haskell on Windows 95/NT or Linux, you can invoke the standard media player from Haskell by defining one of the following functions (for these to work you must also import *system* from the Hugs module *System*, via **import** *System* (*system*)):

```
testWin95, testNT, testLinux :: Music → IO ()

testWin95 m   = do test m
                   system "mplayer test.mid"
                   return ()

testNT m      = do test m
                   system "mplay32 test.mid"
                   return ()

testLinux m   = do test m
                   system "playmidi -rf test.mid"
                   return ()
```

For example, typing:

```
testNT funkGroove
```

using Hugs on an NT system will write the *funkGroove* example from Chapter 20 into a MIDI file and then automatically fire up the media player so that you can hear the result. Try the above for other examples from Chapter 20, such as *cMajArp*, *cMajChd*, *pr*12, *waterfall*, and *main*.