Demo: Functors and Music

Heinrich Apfelmus
Germany
apfelmus@quantentunnel.de

Abstract

We present work-in-progress on two projects whose combination enables live coding music in Haskell: *cnoidal*, a library for representing and transforming music, and *HyperHaskell*, a Haskell interpreter with a worksheet interface and graphical output. The library represents music as a collection of time intervals tagged with values, a data structure known as *temporal media*. Parametric polymorphism suggests various functor instances, like Applicative Functor, which we find to be highly useful for live coding. However, a lawful Monad instance can only be defined for some variants of the data type. We stress that these projects are not a specialized music environment, instead we compose a library with a general purpose interpreter.

CCS Concepts • **Applied computing → Sound and music computing**; • **Software and its engineering → Functional languages**.

Keywords music, live coding, algorave, monad, applicative functor, interpreter, Haskell, functional programming

ACM Reference Format:
Heinrich Apfelmus. 2019. Demo: Functors and Music. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design (FARM '19), August 23, 2019, Berlin, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3331543.3342582

1 Temporal Media with *cnoidal*

The *cnoidal* music library is based on the notion of *temporal media* [Hudak 2004, Hudak 2008, Hudak 2015, Archipoff 2015], a data type that essentially represents a collection of time intervals tagged with values. If the values are musical pitches, this type represents a musical score: each time interval indicates when and for how long a musical note with a given pitch is played. We generalize and allow the value type to be variable, and thus obtain a polymorphic data type

```
data Media a
toIntervals :: Media a → Set (Interval, a)
```

This is an abstract data type whose main operation, denoted `toIntervals`, returns the concrete collection of time intervals (of type `Interval`) paired with values. We assume that the type `Set` represents an unordered collection. (Practical implementations often require that the values type is a member of the type class `Ord`, but for the sake of simplicity, we ignore this here.) We require that the intervals be nonempty, but not that they are disjoint: the intervals are allowed to overlap, so that multiple notes sound at the same time and the represented snippet of music is *polyphonic*. These invariants are the main reason why we chose to discuss `Media` as an abstract data type; any concrete implementation, say, as a list of pairs, has to be careful about maintaining them. But the abstract specification also allows us to discuss variants of the type: A concrete definition would imply that two media are equal if their intervals are equal, but if we do *not* impose this condition, so that `m = n` does not follow from `toIntervals m = toIntervals n`, then the type may contain additional data. For instance, adding a total duration

```
duration :: Media a -> Time
```

would allow us to compose musical pieces in sequence, by playing the next piece after the previous one has ended. This will include a pause if the duration of the first one is different from the ending time of its last interval.

One of the key insights of typed functional programming is that higher-kinded types like `Media` are often functors obeying various laws, and that this provides very general and useful transformations on the data. Then, libraries like `Control.Monad` allow us to combine such transformations in interesting ways. We posit that this is so useful that we should not only ask whether the laws of Functor, Applicative Functor, and Monad can be satisfied for an existing data type, but to let these laws *guide the design* of new data types. We will indeed find that functors are helpful for live coding music. For example, the Applicative Functor instance is similar to zipping lists and allows us to apply a rhythm to a note or harmony, similar to how a rhythm guitar player applies a strum pattern to a chord. For some variants of the data type, there is even a Monad instance, probably novel, which allows us to embellish each note with new notes, e.g. a mordent or an arpeggio. We now discuss these instances in more detail.

1.1 Applicative Functor

Temporal media are a *Functor*, where a combinator `fmap` allows us to apply a function to each value. Its semantics are straightforward to express via `toIntervals`:

FARM '19, August 23, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6811-7/19/08.
https://doi.org/10.1145/3331543.3342582

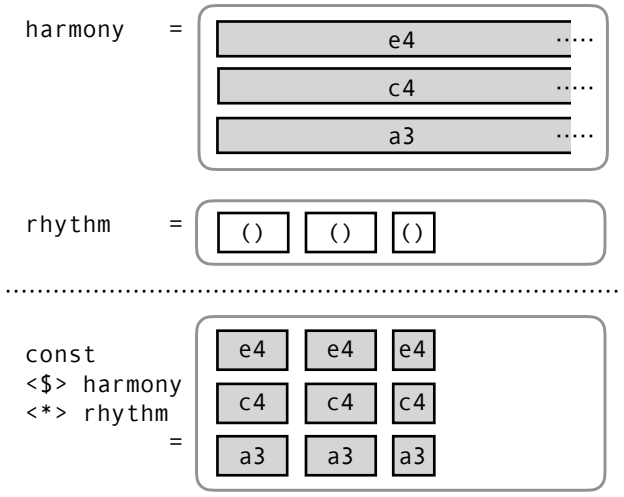

Figure 1. Illustration of the apply operation on temporal media. The horizontal direction corresponds to the passage of time, and the boxes represent time intervals with values. All intervals from the first medium are intersected with those from the second medium, while functions are applied to values. This operation is very useful for imposing a rhythm upon a chord.

```
fmap :: (a → b) → Media a → Media b
toIntervals (fmap f m) =
    fmap (\(i,a) → (i,f a)) (toIntervals m)
```

But temporal media are also an *Applicative Functor*. For this, several laws need to be satisfied, and we have to chose the `Interval` type wisely:

```
type Interval = (Time, Maybe Time)
```

The first and second component of the pair are the starting and ending time of the interval. We impose that starting times are always `>= 0`, and that an ending time of `Nothing` means that the interval is infinitely long; i.e. it never ends. This allows us to embed pure values into media:

```
pure :: a → Media a
toIntervals (pure a) = fromList [((0,Nothing), a)]
```

where we assume that `fromList` converts a list of time intervals and values to the `Media a` type. Applying media to each other is very similar to zipping lists: We arrange the intervals in two parallel tracks, and apply the functions from one track to the values in the other track. However, since intervals can overlap and do not necessarily align, we have to combine them, here by intersecting them. An example is shown in Fig. 1. Assuming combinators

```
cartesian  :: Set a → Set b → Set (a,b)
filterJust :: Set (Maybe a) → Set a
intersect  :: Interval → Interval → Maybe Interval
```

we can define the semantics of apply (`<*>`) as

```
(<*>) :: Media (a → b) → Media a → Media b
toIntervals (mf <*> ma) = filterJust $ fmap apply1 $
    cartesian (toIntervals mf) (toIntervals ma)

apply1 :: ((Interval, a → b), (Interval, a))
       → Maybe (Interval, b)
apply1 ((if,f), (ia,a)) =
    fmap (\i → (i,f a)) $ intersect if ia
```

Essentially, we define how to apply a pair of single intervals, and then lift this to sets of intervals by demanding that application distributes over unions. Put differently, if we denote the union, or parallel composition, of media by

```
(<|>) :: Media a → Media a → Media a
```

then we have the distributive laws

```
(mf <|> mg) <*> ma = (mf <*> ma) <|> (mg <*> ma)
mf <*> (ma <|> mb) = (mf <*> ma) <|> (mf <*> mb)
```

Incidentally, this gives an instance of the *Alternative* class. We skip a formal proof of the Applicative laws here, but note that they can only be established if we can reason about equality of values by using the operations of the abstract data type. For instance, this is the case if we impose that two values of type `Media a` are equal whenever applying `toIntervals` gives equal results. Then, the main idea of the proof is to observe that the type `Set (Interval, a)` is essentially the composition of two Applicative Functors: The `Set` monad, and the Writer monad [Jones 1995] where the output type is `Interval`, which forms a monoid under intersection. Then, we use that the composition of Applicative Functors is again an Applicative Functor [McBride 2007]. This argument can be extended to temporal media with an additional `duration` operation by composing with another Writer.

The apply operation is useful for imposing a rhythm upon a chord. A *rhythm* is about the timing of notes, the values are unimportant:

```
type Rhythm = Media ()
```

A popular rhythm for beginning guitarists is the "campfire strumming pattern", which we take for granted

```
campfire :: Media ()
```

To play a chord, say A minor, in this rhythm, we can intersect infinitely long intervals with this rhythm [see Fig. 1]:

```
harmony, guitar :: Media Pitch
harmony = pure a3 <|> pure c4 <|> pure e4
guitar  = const <$> harmony <*> campfire
```

Here, `a3`, `c4`, `e4` are pitches of A minor, and `<$>` is the infix synonym for `fmap`. This also works if the harmony changes over time.

1.2 Monad

The variant of temporal media where equality is determined solely by the collection of intervals also supports a *Monad* structure, which seems to be novel. Its key advantage is that it allows the addition of new intervals, while the Applicative operation only combines existing intervals.

Before discussing this Monad structure in detail, we point out that it will not be compatible with the Applicative structure just discussed. The situation is similar to that for the list data type: The standard Monad structure for lists gives rise to an apply operation that corresponds to a cartesian product, but the `zipWith` function also represents an apply operation, which is different and corresponds to the intersection of lists. The Applicative structure for temporal media is analogous to the latter operation, as it corresponds to interval intersection. In other words, temporal media are a data type that supports multiple, incompatible Applicative structures, where one of them arises from a Monad structure. For lists, a newtype `ZipList` has been introduced to provide access to both instances, but the author feels that this may be premature for the `Media` type.

Unfortunately, this Monad structure seems to be at odds with the additional `duration` operation, i.e. with the desideratum that temporal media can be easily composed in sequence. As far as the `toIntervals` operation is concerned, the monad laws can be satisfied, but it is not clear to the author how the monadic bind operation can combine durations in a useful way. Thus, it seems that we have a choice: Either easy sequential composition, or a Monad instance. In practice, it appears that sequential composition is more desirable. Providing an instance of the `Monad` class that does not satisfies the laws is not an option, because that would break most library functions built on them, but we can still offer the `join` function described below as part of the regular API.

We now specify the Monad structure. Assuming combinators

```
filterJust :: Set (Maybe a) → Set a
intersect  :: Interval → Interval → Maybe Interval
start      :: Interval → Time
shift      :: Time → Interval → Interval

start    (s,e) = s
shift dt (s,e) = (dt+s, fmap (dt+) e)
```

the monadic `join` is specified by

```
join :: Media (Media a) → Media a
toIntervals (join mma) = filterJust $
  [ fmap (\k → (k,a))
    $ intersect i (shift (start i) j)
  | (i,ma) <- toIntervals mma
  , (j,a ) <- toIntervals ma  ]
```

where we have taken the liberty to use list comprehension notation for `Set`. In other words, each medium that is associated to an interval will be shifted so that it starts at the same time as this interval, and will also be cut to fit within this interval [see Fig. 2]. Now, for the variant where two values of type `Media a` are equal if applying `toIntervals` gives equal results, this implies the three monad laws

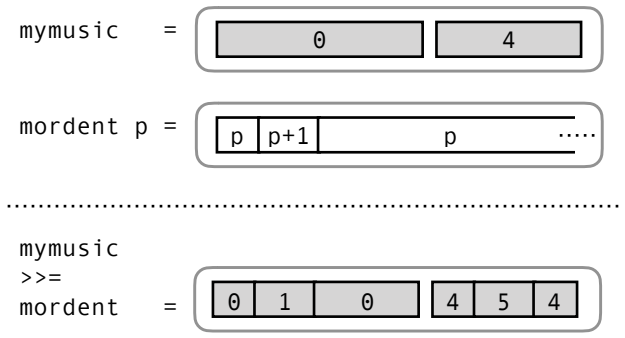

Figure 2. Illustration of the monadic bind operation on temporal media. The time intervals generated by the second argument are cut and shifted to fit into the time intervals of the first argument. This operation is useful for embellishments, e.g. a mordent.

```
      m >>= pure = m
  pure a >>= f   = f a
(m >>= f) >>= g  = m >>= (\a → f a >>= g)
```

Again, we skip the formal proof, but we note that the main insight is that the type `Set (Interval, a)` is essentially an application of the `WriterT` monad transformer [Jones 1995] to the `Set` monad. However, in this case, the binary operations that turns `Interval` into a monoid is a combination of both a shift in time and an intersection of intervals. It is remarkable that the half-infinite interval beginning at zero, `(0,Nothing)`, is both a left- and a right-identity element for this operation. Thus, this particular interval is key to designing a data type that can accommodate the Applicative Functor and Monad laws. We also see that the difficulty of adding the `duration` datum can be interpreted as a difficulty of the `Writer` type constructor not being a monad transformer.

The Monad structure is useful for embellishing musical notes with more notes. For instance, a *mordent* is an embellishment where the pitch is briefly interjected with the pitch one half-tone above. We could specify it as

```
mordent :: Pitch → Media Pitch
mordent pitch = fromIntervals $
  [ ((  0, Just    dt ) , pitch  )
  , ((  dt, Just (2*dt)) , pitch+1)
  , ((2*dt, Nothing    ) , pitch  )]
  where dt = 1/32
```

In other words, a pitch is extended into a sequence of notes. A more extreme example would be an arpeggiator, where the notes of a given chord a played in sequence. The monadic bind operation (>>=) allows us to apply these embellishments to a piece of music [see Fig. 2]:

```
mymusic >>= mordent
```

This example shows that even if the monad laws are not fulfilled, the `join` combinator is still useful for transforming music.

2 Live Coding with *HyperHaskell*

HyperHaskell, "the strongly hyped Haskell interpreter", is a graphical interpreter environment with a notebook interface similar to Mathematica[1] or Jupyter[2]. It is a new, if conceptually standard, addition to the Haskell ecosystem. It is built using the Electron framework[3] and is intended to be easy to install.

Values are displayed graphically using a new `Display` class, the old `Show` class is only used when no graphical display is available yet. (Currently, graphics are specified via HTML and SVG.) For instance, we can display the intervals in temporal media as rectangles of different lengths; the author already found this useful for debugging an efficient implementation of the Applicative Functor instance.

The notebook interface facilitates live coding since expressions can be revisited and modified once entered. We also found it useful to add a function `addFinalizerSession`, which allows us to run a finalizer action whenever source code files are reloaded, like resetting the connection to a software synthesizer. This makes it easier to develop the cnoidal library with the interpreter.

3 Conclusion and Outlook

The author invites everyone to try out and use *HyperHaskell*. If it can be used for live coding music, then it can be used for live coding anything — data science, graphics, system administration…

The *cnoidal* library is still in its early iterations. To create interesting musical performances, it is desirable to not just specify musical data directly, but also to specify *patterns* that generate musical data. For instance, a pattern could specify that every other measure, a different musical notes are chosen randomly from a selected chord. This is left for the future.

A Code Example

Here is a small notebook example that produces a nice backbeat with chipper piano chords. It assumes that we are hooked up to a software synthesizer that understands standard MIDI. Each `Player` object plays a given piece of music on repeat until it is given a different one; then it will change at the next full measure.

The example requires the imports

```
import Cnoidal
import Control.Applicative
```

The code is

```
-- Initialze MIDI connection to synthesizer
s <- openMidi
addFinalizerSession $ closeMidi >>= print
show s

-- Initialize ensemble
ensemble <- newEnsemble s
drums    <- newPlayer 0
piano    <- newPlayer 1
setTempoBpm ensemble 110
together ensemble [drums, piano]
addFinalizerSession $ dissolve ensemble

-- Drums
let [kick,snare,hihat,hihat2] = [36,38,44,42]
      :: [Pitch]
let pat2 = hasten 16 $ beat "x,,, ,,x, x,,, ,,,x"
let pat3 = hasten  8 $ beat ",,x, ,,x,"
let pat4 = hasten  8 $ beat "xx,x ,x,x"
play drums $
        (kick  <$ pat2)
    <|> (snare <$ pat3)
    <|> (hihat <$ pat4)

-- Piano
let mychords = polyphony $ fromList
             $ chords "am F C G"
let rhythm   = staircase 1 $ replicate 4 $ campfire
let pattern1 = mychords <* rhythm
play piano $ pattern1
```

[1] www.wolfram.com/mathematica/
[2] jupyter.org
[3] electronjs.org

Acknowledgments

We thank the anonymous reviewers for helpful comments and suggestions. This work was self-funded and performed during the author's free time.

References

[Archipoff 2015] Simon Archipoff, An Efficient Implementation of Tiled Polymorphic Temporal Media. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*, 2015.

[Hudak 2004] Paul Hudak. An algebraic theory of polymorphic temporal media. In *Proceedings of PADL'04: 6th International Workshop on Practical Aspects of Declarative Languages*, pages 1–15, 2004.

[Hudak 2008] Paul Hudak. A sound and complete axiomatization of polymorphic temporal media, Report, 2008.

[Hudak 2015] Paul Hudak and David Janin. From out-of-time design to in-time production of temporal media, Research Report LaBRI, 2015.

[Jones 1995] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, 1995.

[McBride 2007] Conor McBride and Ross Paterson. Applicative programming with effects, J. Funct. Prog. 18, pages 1–13, 2007.