An Algebraic Theory of Polymorphic Temporal Media

Research Report RR-1259

Paul Hudak
Department of Computer Science
Yale University
New Haven, CT 06520-8285
paul.hudak@yale.edu

ABSTRACT

Temporal media is information that is directly consumed by a user, and that varies with time. Examples include music, digital sound files, computer animations, and video clips. In this paper we present a polymorphic data type that captures a broad range of temporal media. We study its syntactic, temporal, and semantic properties, leading to an algebraic theory of polymorphic temporal media that is valid for underlying media types that satisfy specific constraints. The key technical result is an axiomatic semantics for polymorphic temporal media that is shown to be both sound and complete.

Categories and Subject Descriptors

D.G.1 [**Programming Languages**]: Formal Definitions and Theory

General Terms

Media, multimedia, polymorphism, temporal, algebra, formal semantics, axiomatic semantics, soundness and completeness, functional programming.

1. INTRODUCTION

The advent of the personal computer has focussed attention on the *consumer*, the person who buys and makes use of the computer. Our interest is in the consumer as a person who *consumes information*. This information takes on many forms, but it is usually dynamic and time-varying, and ultimately is consumed mostly through our visual and aural senses. We use the term *temporal media* to refer to this time-varying information. We are interested in how to represent this information at an abstract level; how to manipulate these representations; how to assign a meaning, or interpretation, to them; and how to reason about such meanings.

To achieve these goals, we define a polymorphic representation of temporal media that allows combining media values in generic ways, independent of the underlying media type. We describe three types of operations on and properties of temporal media: (a) syntactic operations and properties, that depend only on the structural representation of the media, (b) temporal operations and properties, that additionally depend on time, and (c) semantic operations and properties, that depend on the meaning, or interpretation, of the media. The latter development leads to an axiomatic semantics for polymorphic temporal media that is both sound and complete.

Examples of temporal media include music, digital sound files, computer animations, and video clips. It also includes representations of some other concepts, such as dance [10] and a language for humanoid robot motion [5]. In this paper we use two running examples throughout: an abstract representation of *music* (analogous to our previous work on *Haskore* and *MDL*, DSLs for computer music [9, 6, 7, 8]), and an abstract representation of *continuous animations* (analogous to our previous work on *Fran* and *FAL* [4, 3, 7]). We briefly address other media types in Section 8.

The key new ideas in the current work are the polymorphic nature of the media type, the exploration of syntactic and temporal properties of this media type that parallel those for lists, the casting of the semantics in a formal algebraic framework, the definition of a normal form for polymorphic temporal media, and a completeness result for the axiomatic semantics. The completeness result relies on a new axiom for swapping terms in a serial/parallel construction.

The results in this paper shed interesting light on the nature of temporal media, and cast into a rigorous framework commonalities that have been intuitively noted between languages designed for seemingly different domains.

We present all of our results using Haskell [12] syntax that, in most cases, is executable. Haskell's type classes are particularly useful in specifying constraints, via implicit laws, that constituent types must obey. Proof details of the simpler theorems are omitted in this extended abstract.

2. POLYMORPHIC MEDIA

We represent temporal media by a polymorphic data type:

```
data Media a = Prim a
             | Media a :+: Media a
             | Media a :=: Media a
```

We refer to `T` in `Media T` as the *base media type*. Intuitively, for values `x :: T` and `m1, m2 :: Media T`, a value of type `Media T` is either a primitive value `Prim x`, a sequential composition `m1 :+: m2`, or a parallel composition `m1 :=: m2`. Although simple in structure, this data type is rich enough to capture quite a number of useful media types.

Example 1 (Music): Consider this definition of an abstract notion of a *musical note*:

```
data Note = Rest Dur
          | Note Pitch Dur

type Dur    = Real
type Pitch  = (NoteName, Octave)
type Octave = Int

data NoteName = Cf | C  | Cs | Df | D  | Ds | Ef
              | E  | Es | Ff | F  | Fs | Gf | G
              | Gs | Af | A  | As | Bf | B  | Bs
```

In other words, a `Note` is either a pitch paired with a duration, or a `Rest` that has a duration but no pitch. `Dur` is a measure of time (duration), which ideally would be a real number; in a practical implementation a suitable approximation such as `Float`, `Double`, or `Ratio Int` would be used. A `Pitch` is a pair consisting of a note name and an octave, where an octave is just an integer. The note name `Cf` is read as "C-flat" (normally written as C♭), `Cs` as "C-sharp" (normally written as C♯), and so on.[1]

Then the type:

```
type Music = Media Note
```

is a temporal media for music. In particular, a value `Prim (Rest d)` is a rest of duration `d`, `Prim (Note p d)` is a note with pitch `p` played for duration `d`, `m1 :+: m2` is the music value `m1` followed sequentially in time by `m2`, and `m1 :=: m2` is `m1` played simultaneously with `m2`. This representation of music is a simplified version of that used in the Haskore computer music library [9, 6], which has been used successfully in several computer music applications. As a simple example:

```
let dMinor  =  Note (D,3) 1 :=: Note (F,3) 1
                            :=: Note (A,3) 1
    gMajor  =  Note (G,3) 1 :=: Note (B,3) 1
                            :=: Note (D,4) 1
    cMajor  =  Note (C,3) 2 :=: Note (E,3) 2
                            :=: Note (G,3) 2
in dMinor :+: gMajor :+: cMajor
```

is a ii-V-I chord progression in C major.

Example 2 (Animation): Consider this definition of a base media type for *continuous animations*:

[1]This representation corresponds well to that used in music theory, except that in music theory note names are called *pitch classes*.

```
type Anim = (Dur, Time -> Picture)
type Dur  = Real
type Time = Real

data Picture = EmptyPic
             | Circle Radius Point
             | Square Length Point
             | Polygon [Point]

type Point = (Real, Real)
```

A `Picture` is either empty, a circle or square of a given size and located at a particular point, or a polygon having a specific set of vertices. An `Anim` value `(d, f)` is a continuous animation whose image at time $0 \leq$ `t` $\leq$ `d` is the `Picture` value `f t`.

Then the type:

```
type Animation = Media Anim
```

is a temporal media for continuous animations. This representation is a simplified version of that used in Fran [4, 3] and FAL [7]. As a simple example:

```
let ball1 = (10, \t -> Circle t origin)
    ball2 = (10, \t -> Circle (10-t) origin
    box   = (20, \t -> Square 1 (t,t))
in (ball1 :+: ball2) :=: box
```

is a box sliding diagonally across the screen, together with a ball located at the origin that first grows for 10 seconds and then shrinks.

3. SYNTACTIC PROPERTIES

Before studying semantic properties, we first define various operations on the *structure* (i.e. syntax) of polymorphic temporal media values, many of which are analogous to operations on lists (and thus we borrow similar names when the analogy is strong). We also explore various *laws* that these operators obey, laws that are also analogous to those for lists [2, 7].

Map. For starters, it is easy to define a polymorphic *map* on temporal media, which we do by declaring `Media` to be an instance of the `Functor` class:

```
instance Functor Media where
  fmap f (Prim n)    = Prim (f n)
  fmap f (m1 :+: m2) = fmap f m1 :+: fmap f m2
  fmap f (m1 :=: m2) = fmap f m1 :=: fmap f m2
```

`fmap` shares many properties with `map` defined on lists, most notably the standard laws for the `Functor` class:

THEOREM 3.1. For any finite `m :: Media T1` and functions `f, g :: T1 -> T2`:

```
fmap (f . g) = fmap f . fmap g
fmap id = id
```

PROOF. By a straightforward structural induction on the second argument to `fmap`. □

`fmap` allows us to define many useful operations on specific media types, thus obviating the need for a richer data type as used, for example, in our previous work on Haskore, MDL, Fran, and Fal. The following examples demonstrate this.

Example 1 (Music): We define a function in terms of `fmap` that alters the *tempo* of a `Music` value:

```
tempo :: Dur -> Music -> Music
tempo r = fmap (temp r)
          where temp r (Rest d)   = Rest (r*d)
                temp r (Note p d) = Note p (r*d)
```

and one that *transposes* a `Music` value by a given interval:

```
trans :: Int -> Music -> Music
trans i = fmap (tran i)
          where tran i (Rest d)   = Rest d
                tran i (Note p d) = Note p' d
                  where p' = transPitch i p
```

where `transPitch i p` translates pitch `p` by interval `i` (straightforward definition omitted). These two functions obviate the need for the `Tempo` and `Trans` constructors used in Haskore and MDL [9, 6, 7].

Using Theorem 3.1 we can show that `tempo` is multiplicative and `trans` is additive; in addition, they commute with respect to themselves and to each other:

COROLLARY 3.1. For any `r1,r2 :: Dur` and `i1,i2 :: Int`:

```
tempo r1 . tempo r2 = tempo (r1*r2)
trans i1 . trans i2 = trans (i1+i2)
tempo r1 . tempo r2 = tempo r2 . tempo r1
trans i1 . trans i2 = trans i2 . trans i1
tempo r1 . trans i1 = trans i1 . tempo r1
```

Example 2 (Animation): We define a function to scale in size and translate in space an `Animation` value:

```
scale :: Real -> Real -> Animation -> Animation
scale s (dx,dy) = fmap (\(d,f) -> (d, scal . f))
  where
   scal EmptyPic     = EmptyPic
   scal (Square l p) = Square (s*l) (scalePt p)
   scal (Circle r p) = Circle (s*r) (scalePt p)
   scal (Polygon vs) = Polygon (map scalePt vs)
   scalePt (x,y)     = (s*x+dx,s*y+dy)
```

This function obviates the need for the `Scale` and `Translate` constructors used in Fran and Fal [4, 3, 7]. In a similar way, a function `rate` can be defined such that `rate r a` shifts the framerate of animation `a` by `r`.

Using Theorem 3.1 it is then straightforward to show:

COROLLARY 3.2. For any `s1,s2,dx1,dx2,dy1,dy2 :: Real`:

```
scale s1 (dx1,dy1) . scale s2 (dx2,dy2)
  = scale (s1*s2) (dx1+dx2,dy1+dy2)
scale s1 p1 . scale s2 p2
  = scale s2 p2 . scale s1 p1
```

Fold (i.e. catamorphism).

A fold-like function can be defined for media values, and will play a critical role in our subsequent development of the semantics of temporal media:

```
foldM :: (a->b) -> (b->b->b) -> (b->b->b)
         -> Media a -> b
foldM f g h (Prim x) = f x
foldM f g h (m1 :+: m2) =
      foldM f g h m1 `g` foldM f g h m2
foldM f g h (m1 :=: m2) =
      foldM f g h m1 `h` foldM f g h m2
```

LEMMA 3.1. For any `f :: T1 -> T2`:

```
foldM (Prim . f) (:+:) (:=:) = fmap f
foldM Prim (:+:) (:=:) = id
```

PROOF. For the first equation, by a straightforward structural induction. The second equation then follows from the first and Theorem 3.1. □

More interestingly, we can also state a *fusion law* for `foldM`:

THEOREM 3.2. (Fusion Law)
For `f :: T1 -> T2`, `g, h :: T2 -> T2 -> T2`, `k :: T2 -> T3`, and `g', h' :: T1 -> T3`, if:

```
f' x = k (f x)
g' (k x) (k y) = k (g x y)
h' (k x) (k y) = k (h x y)
```

then:

```
k . foldM f g h = foldM f' g' h'
```

PROOF. By induction. Base case:

```
k (foldM f g h (Prim x))       unfold foldM
= k (f x)                      assumption
= f' x                         fold foldM
= foldM f' g' h' (Prim x)
```

Induction step:

```
k (foldM f g h (m1 :+: m2))    unfold foldM
= k (g (foldM f g h m1) (foldM f g h m2))
  assumption
= g' (k (foldM f g h m1)) (k (foldM f g h m2))
  induction hypothesis
= g' (foldM f' g' h' m1) (foldM f' g' h' m2)
  fold foldM
= foldM f' g' h' (m1 :+: m2)
```

Similarly for `(:=:)`. □

The following *fold-map fusion law* is a special case of the above.

COROLLARY 3.3. (Fold-Map Fusion Law)
For all `f :: T1 -> T2`, `g, h :: T2 -> T2 -> T2`, and `j :: T0 -> T1`:

```
foldM f g h . fmap j = foldM (f . j) g h
```

PROOF. Directly from the fusion law by validating the three constraints on `f`, `g`, and `h`. □

Example: In the discussion below a reverse function, and in Section 4 a duration function, are defined as catamorphisms. In addition, in Section 5 we define the standard interpretation, or semantics, of temporal media as a catamorphism.

Reverse.

We would like to define a function `reverseM` that *reverses*, in time, any temporal media value. However, this will only be possible if the base media type is itself reversible, a constraint that we enforce using type classes:

```
class Reverse a where
  reverseM :: a -> a

instance Reverse a => Reverse (Media a) where
  reverseM (Prim a)    = Prim (reverseM a)
  reverseM (m1 :+: m2) =
    reverseM m2 :+: reverseM m1
  reverseM (m1 :=: m2) =
    reverseM m1 :=: reverseM m2
```

Note that `reverseM` can be defined more succinctly as a catamorphism:

```
instance Reverse a => Reverse (Media a) where
  reverseM =
    foldM (Prim . reverseM) (flip (:+:)) (:=:)
```

Analogous to a similar property on lists, we have:

THEOREM 3.3. For all finite `m`, if the following law holds for `reverseM :: T -> T`, then it also holds for `reverseM :: Media T -> Media T`:

```
reverseM (reverseM m) = m
```

We take the constraint in this theorem to be a law for all valid instances of a base media type `T` in the class `Reverse`.

PROOF. It is straightforward to prove this result using structural induction. However, we can carry out an inductionless proof by using the fusion law of Theorem 3.2, as follows. Let `k = reverseM`. Thus:

```
(reverseM . reverseM) m
= (k . foldM (Prim . reverseM)
             (flip (:+:)) (:=:)) m  fusion law
= foldM Prim (:+:) (:=:) m          lemma 3.1
= m
```

Use of the fusion law is valid because its three conditions are met as shown below:

```
Prim x                              assumption
= Prim (reverseM (reverseM x))      fold (.)
= (Prim . reverseM) (reverseM x)    fold k
= k (Prim (reverseM x))             fold (.)
= k ((Prim . reverseM) x)

(:+:) (k x) (k y)
= flip (:+:) (k y) (k x)            fold flip
= k (y :+: x)                       fold k
= k (flip (:+:) x y)                fold flip

(:=:) (k x) (k y)
= k (x :=: y)                       unfold k
```

□

`reverseM` also interacts nicely with `fmap`, just as `reverse` interacts with `map` on lists:

THEOREM 3.4. For any `f :: T -> T`, if `f . reverseM = reverseM . f`, then:

```
fmap f . reverseM = reverseM . fmap f
```

PROOF. Using both fusion laws (Theorem 3.2 and Corollary 3.3) (details omitted). □

Finally, we can prove the following theorem, which is analogous to this well-known law about lists:

```
foldr op e xs = foldl (flip op) e (reverse xs)
```

THEOREM 3.5. For all finite `m :: Media T`, functions `g,h :: T -> T -> T`, and `f,f' :: T -> T` such that `f = f' . reverseM`:

```
foldM f g h m = foldM f' (flip g) h (reverse m)
```

PROOF. By structural induction (details omitted). □

Example 1 (Music): We declare `Note` to be an instance of class `Reverse`:

```
instance Reverse Note where
  reverseM = id
```

In other words, a note is the same whether played backwards or forwards. The constraint in Theorem 3.3 is therefore trivially satisfied, and it thus holds for music media.[2]

Furthermore:

COROLLARY 3.4. (to Theorem 3.4)

```
reverseM . tempo r = tempo r . reverseM
reverseM . trans i = trans i . reverseM
```

COROLLARY 3.5. (to Theorem 3.5)

```
foldM f g h m = foldM f (flip g) h (reverse m)
```

Example 2 (Animation): We declare `Anim` to be an instance of class `Reverse`:

```
instance Reverse Animation where
  reverseM (d, f) = (d, \t -> f (d-t))
```

Note that:

```
reverseM (reverseM (d, f))
= reverseM (d, \t -> f (d-t))
= (d, \t' -> (\t -> f (d-t)) (d-t'))
= (d, \t' -> f (d-(d-t')))
= (d, \t' -> f t')
= (d, f)
```

Therefore the constraint in Theorem 3.3 is satisfied, and the theorem thus holds for continuous animations. Furthermore:

COROLLARY 3.6. (to Theorem 3.4)

```
reverseM . scale s d = scale s d . reverseM
```

[2]The reverse of a musical passage is called its *retrograde*. Used sparingly by traditional composers (two notable examples being J.S. Bach's "Crab Canons" and Franz Joseph Haydn's Piano Sonata No. 26 in A Major (Menueto al Rovescio)), it is a standard construction in modern twelve-tone music.

4. TEMPORAL PROPERTIES

As a data structure, the `Media` type is fairly straightforward. Complications arise, however, when *interpreting* temporal media. The starting point for such an interpretation is an understanding of temporal properties, the most basic of which is *duration*. Of particular concern is the meaning of the parallel composition `m1 :=: m2` when the durations of `m1` and `m2` are different. There are at least four possibilities:

1. `m1` and `m2` begin at the same time, and when the longer one finishes, the entire construction finishes.
2. `m1` and `m2` begin at the same time, and when the shorter one finishes, the entire construction finishes (thus truncating the longer one).
3. `m1` and `m2` are "centered" in time, so that the shorter one begins after one-half of the difference between their durations.
4. This situation is disallowed: i.e. `m1` and `m2` must have the same duration in a well-formed `Media` value.

The first option is what we used in the design of Haskore and MDL [9, 6, 7]. The second option is similar to what Haskell's `zip` function does with lists. The third option is what we used in a recent paper emphasizing algebraic properties of music [8].

In the present treatment, however, we shall adopt the fourth option. Doing so simplifies the presentation, and does not lack in generality, as long as the primitive type is able to express the absence of media for a specified duration (for example a value `Rest d` in `Music`), which we enforce using type classes. Anything expressed using one of the other three options can be expressed using option four by inserting suitable "rests" in appropriate places.

Duration. To compute the *duration* of a temporal media value we first need a way to compute the duration of the underlying media type, which we enforce as before using type classes:

```
class Temporal a where
  dur  :: a -> Dur
  none :: Dur -> a

instance Temporal a => Temporal (Media a) where
  dur  = foldM dur (+) max
  none = Prim . none
```

The `none` method allows one to express the absence of media for a specified duration, as discussed earlier.

We take the constraint in the following lemma to be a law for any valid instance of a base media type `T` in the class `Temporal`:

LEMMA 4.1. If the property `dur (none d) = d` holds for `dur :: T -> Dur`, then it also holds for `dur :: Media T -> Dur`.

PROOF. (Straightforward and omitted.) □

Note that, for generality, the duration of a parallel composition is defined as the maximum of the durations of its arguments. However, as discussed earlier, we wish to restrict parallel coompositions to those whose two argument durations are the same. Thus we define:

DEFINITION 4.1. A *well-formed* temporal media value `m :: Media T` is one that is finite, and for which each parallel composition `m1 :=: m2` has the property that `dur m1 = dur m2`.

Note that `dur` is analogous to the `length` operator on lists, and obeys a law analogous to `length l1 + length l2 = length (l1 ++ l2)`.

Example 1 (Music): We declare `Note` to be `Temporal`:

```
instance Temporal Note where
  dur (Rest d)   = d
  dur (Note p d) = d
  none d         = Rest d
```

Thus `dur :: Music -> Dur` determines the duration of a `Music` value.

Example 2 (Animation): We declare `Anim` to be `Temporal`:

```
instance Temporal Anim where
  dur (d, f) = d
  none d     = (d, const EmptyPic)
```

Thus `dur :: Animation -> Dur` determines the duration of an `Animation` value.

Take and Drop. We now define two functions `takeM` and `dropM` that are analogous to Haskell's `take` and `drop` functions for lists. The difference is that instead of being parameterized by a number of elements, `takeM` and `dropM` are parameterized by *time*. As with other operators we have considered, this requires the ability to take and drop portions of the base media type, so once again we use type classes to structure the design. The expression `takeM d m` is a media value corresponding to the first `d` seconds of `m`. Similarly, `dropM d m` is all but the first `d` seconds. Both of these are very useful in practice.

```
class Take a where
  takeM :: Dur -> a -> a
  dropM :: Dur -> a -> a

instance (Take a, Temporal a) =>
         Take (Media a) where
  takeM d m | d <= 0  = none 0
  takeM d (Prim x)    = Prim (takeM d x)
  takeM d (m1 :+: m2) =
    let d1 = dur m1
    in if d <= d1 then takeM d m1
                  else m1 :+: takeM (d-d1) m2
  takeM d (m1 :=: m2) = takeM d m1 :=: takeM d m2

  dropM d m | d <= 0  = m
  dropM d (Prim x)    = Prim (dropM d x)
  dropM d (m1 :+: m2) =
    let d1 = dur m1
    in if d <= d1 then dropM d m1 :+: m2
                  else dropM (d-d1) m2
  dropM d (m1 :=: m2) = dropM d m1 :=: dropM d m2
```

Since we are only interested in the take and drop of well-formed media values, the case for parallel composition is quite simple.

We take the constraint in the following lemma to be a law for any valid instance of a base media type `T` in the class `Temporal`:

LEMMA 4.2. If the following laws hold for any finite `m :: T`, then they also hold for any finite well-formed `m :: Media T`:

```
takeM d m | d <= 0     = none 0
takeM d m | d >= dur m = m
dropM d m | d <= 0     = m
dropM d m | d >= dur m = none 0
```

PROOF. By structural induction on `m`. □

LEMMA 4.3. For all finite well-formed `m :: Media a` and `d :: Dur <= dur m`, if the following law holds for `takeM, dropM :: Dur -> T -> T`, then it also holds for `takeM, dropM :: Dur -> Media T -> Media T`:

```
dur (takeM d m) = d
dur (dropM d m) = dur m - d
```

PROOF. By structural induction on `m`. □

Perhaps surprisingly, `takeM` and `dropM` also share many properties analogous to their list counterparts, except that indexing is done in time, not in the number of elements. This is captured by the following key theorem:

THEOREM 4.1. For all non-negative `d1, d2 :: Dur`, if the following laws hold for `takeM, dropM :: Dur -> T -> T`, then they also hold for `takeM, dropM :: Dur -> Media T -> Media T`:

```
takeM d1 . takeM d2 = takeM (min d1 d2)
dropM d1 . dropM d2 = dropM (d1+d2)
takeM d1 . dropM d2 = dropM d2 . takeM (d1+d2)
dropM d1 . takeM d2 = takeM (d2-d1) . dropM d1
                      -- if d2>=d1
```

PROOF. See Appendix A.1. □

There is one other theorem that we would *like* to hold, whose corresponding version for lists in fact does hold:

THEOREM 4.2. For all finite well-formed `m :: Media a` and non-negative `d :: Dur <= dur m`, if the following law holds for `takeM, dropM :: Dur -> T -> T`, then it also holds for `takeM, dropM :: Dur -> Media T -> Media T`:

```
takeM d m :+: dropM d m = m
```

However, this theorem is false; in fact it does not hold for the base case:

```
takeM d (Prim x) :+: dropM d (Prim x)
=  Prim (takeM d x) :+: Prim (dropM d x)
/= Prim x
```

We cannot even state this as a constraint on the base media type, because it involves an interpretation of `(:+:)`. We will return to this issue in a later section.

Finally, we note that `takeM` and `dropM` are functionally related by the following theorem:

THEOREM 4.3. For all finite well-formed `m :: Media a` and `d :: Dur <= dur m`, if the following laws hold for `takeM, dropM :: Dur -> T -> T`, then they also hold for `takeM, dropM :: Dur -> Media T -> Media T`:

```
dropM d m =
  reverseM (takeM (dur m - d) (reverseM m))
takeM d m =
  reverseM (dropM (dur m - d) (reverseM m))
```

PROOF. Left as an exercise. □

Example 1 (Music): We declare `Note` to be an instance of `Take`:

```
instance Take Note where
  takeM d1 (Rest d2)   = Rest (min d1 d2)
  takeM d1 (Note p d2) = Note p (min d1 d2)

  dropM d1 (Rest d2)   = Rest (max 0 (d2-d1))
  dropM d1 (Note p d2) = Note p (max 0 (d2-d1))
```

The constraints on Theorems 4.1 and 4.3 hold for this instance, and thus the theorems hold for `Music` values.

Example 2 (Animation): We declare `Anim` to be an instance of `Take`:

```
instance Take Anim where
  takeM d1 (d2, f) = (max 0 (min d1 d2), f)

  dropM d1 (d2, f) = (d2 - max 0 (min d1 d2),
                      f . (d1+))
```

The constraints on Theorems 4.1 and 4.3 hold for this instance, and thus the theorems hold for `Animation` values.

5. SEMANTIC PROPERTIES

Temporal properties of polymorphic media go beyond structural properties, but do not go far enough. For example, intuitively speaking, we would expect these two media fragments:

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

to be equivalent; i.e. to deliver precisely the same information to the observer (for visual information they should *look* the same, for aural information they should *sound* the same, and so on).

In order to capture this notion of equivalence we must provide an *interpretation* of the media that properly captures its "meaning" (i.e. how it looks, how it sounds, and so on). And we would like to do this in a generic way. So once again we use type classes to constrain the design:

```
class Combine b where
  concatM :: b -> b -> b
  merge   :: b -> b -> b
  zero    :: Dur -> b

class Temporal a, Temporal b, Combine b =>
  Meaning a b where
  meaning :: a -> b

instance Meaning a b =>
         Meaning (Media a) b where
  meaning = foldM meaning concatM merge
```

Intuitively speaking, an instance `Meaning T1 T2` means that `T1` can be given meaning in terms of `T2`. More specifically, `Media T1` can be given meaning in terms of `T2`, and expressed as a catamorphism, as long as we can give meaning to the base media type `T1` in terms of `T2`.

As laws for the class `Meaning`, we require that:

```
meaning . none = zero
dur . meaning = dur
```

Also, in anticipation of the axiomatic semantics that we develop in Section 7, we requre that the following laws be valid for any instance of `Combine`:

```
b1 `concatM` (b2 `concatM` b3)
    = (b1 `concatM` b2) `concatM` b3
b1 `merge` (b2 `merge` b3)
    = (b1 `merge` b2) `merge` b3
b1 `merge` b2 = b2 `merge` b1
zero 0 `concatM` b = b
b `concatM` zero 0 = b
zero d1 `concatM` zero d2 = zero (d1+d2)
zero d `merge` b = b, if d = dur b

(b1 `concatM` b2) `merge` (b3 `concatM` b4)
  = (b1 `merge` b3) `concatM` (b2 `merge` b4),
      if  dur b1 = dur b3
      and dur b2 = dur b4
```

We then define a notion of equivalence:

DEFINITION 5.1. `m1, m2 :: Media T` are *equivalent*, written `m1 === m2`, if and only if `meaning m1 = meaning m2`.

Example 1 (Music): We take the meaning of music to be a pair: the *duration* of the music, and a *sequence of events*, where each event marks the start-time, pitch, and duration of a single note:

```
data Event = Event Time Pitch Dur

type Time = Real
type Performance = (Dur, [Event])
```

Except for the outermost duration, the interpretation of `Music` as a `Performance` corresponds well to low-level music representations such as MIDI [1] and csound [14]. The presence of the outermost duration in a `Performance` allows us to distinguish rests of unequal length; for example, `Prim (Rest d1)` and `Prim (Rest d2)`, where `d1 /= d2`. Without the durations, these phrases would both denote an empty sequence of events, and would be indistinguishable. More generally, this allows us to distinguish phrases that end with rests of unequal length, such as `m :+: Prim (Rest d1)` and `m :+: Prim (Rest d2)`.

Three instance declarations complete our interpretation of music:

```
instance Combine Performance where
  concatM (d1, evs1) (d2, evs2) =
    (d1 + d2, evs1 ++ map shift evs2)
    where shift (Event t p d) = Event (t+d1) p d
  merge (d1, evs1) (d2, evs2) =
    (d1 `max` d2, sort (evs1 ++ evs2))
  zero d = (d, [])

instance Temporal Performance where
  dur (d, _) = d
  none = zero

instance Meaning Note Performance where
  meaning (Rest d)   = (d, [])
  meaning (Note p d) = (d, [Event 0 p d])
```

Note that, although the arguments to `(:=:)` in well-formed temporal media have equal duration, we take the `max` of the durations of the two arguments for increased generality. Also, note that the event sequences in a merge are concatenated and then sorted. A more efficient ($O(n)$ instead of $O(n \log n)$) but less concise way to express this is to define a time-ordered merge function.

We can show that the two laws for class `Meaning`, as well as the eight for class `Combine`, hold for these instances, and thus they are valid.

Example 2 (Animation): We take the meaning of animation to be a pair: the *duration* of the animation, and a *sequence of images* sampled at some frame rate `r`:

```
type Rendering = (Dur, [Image])

-- abstract Image operations
picToImage   :: Picture -> Image
combineImage :: Image -> Image -> Image
emptyImage   :: Image
```

Details of the `Image` operations are omitted.

This interpretation of animation is consistent with standard representations of videos/movies, whether digitized, on analog tape, or on film.

Three instance declarations complete our interpretation of continuous animation:

```
instance Combine Rendering where
  concatM (d1, is1) (d2, is2) =
    (d1 + d2, is1 ++ is2)
  merge (d1, is1) (d2, is2) =
    (d1 `max` d2, zipWith' combineImage is1 is2)
  zero d = (d, take (truncate (d*r))
                    [EmptyPic ..]   )

instance Temporal Rendering where
  dur (d, _) = d
  none = zero

instance Meaning Animation Rendering where
  meaning (d, f) =
    (d, map (picToImage . f)
            (take (truncate (d*r))
                  [0, 1/r ..]     ))

r :: Real
r = 30    -- frame rate, in Hertz
```

`zipWith'` is just like Haskell's `zipWith`, except that it does not truncate the result to the shorter of its two arguments.

Unfortunately, not all of the laws for classes `Meaning` and `Combine` hold for these instances. The problem stems from discretization. For example, suppose the frame rate `r = 10`. Then:

```
z1 = zero 1.06
   = (1.06, take 10 [EmptyPic ..])

z2 = zero 2.12
   = (2.12, take 21 [EmptyPic ..])
```

However, note that:

```
z1 `concatM` z1
   = (2.12, take 20 [EmptyPic ..])
```

which is not the same as `z2`. So the `Combine` law:

```
zero d1 `concatM` zero d2 = zero (d1+d2)
```

does not hold.

This problem can be remedied by requiring that all `Anim` durations be integral multiples of the frame rate `r`. We say that such animations are *integral*. With the additional assumption that the image operator `combineImage` is associative and commutative, it is then straighforward to show that all of the laws for classes `Combine` and `Meaning` hold, and thus the above are valid instances for integral animations.

Finally, returning to the motivating example in this section, we can show that:

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

In other words, `(:+:)` is associative. Indeed, there are several other such equivalences, each of which contributes to an *axiomatic semantics* of polymorphic temporal media. We discuss this in detail in Section 7, and thus delay the proof of the above axiom until then.

6. ALGEBRAIC STRUCTURE

In the previous section we defined a standard interpretation of, or a semantics for, polymorphic temporal media, using the semantic function `meaning :: Combine b => Media a -> b`. In this section we place this semantics in a formal algebraic framework, which will be useful in our development of an axiomatic semantics in Section 7.

(In what follows we take some liberty in mixing mathematical notation with Haskell syntax; the context should make the meaning clear.)

An *algebraic structure* (or just *algebra*) `<S,op1,op2,...>` consists of a non-empty carrier set (or *sort*) `S` together with one or more `n`-ary operations `op1`, `op2`, ..., on that set [13]. We define an algebra of *well-formed temporal media over type* `T` as `<Media T,:+:,:=:>`. The Haskell algebraic data type definition for `Media` can be seen as the generator of the elements of this algebra, but with the additional constraint of well-formedness discussed in Section 4. We also define an *interpretation* as an algebra `<I,concatM,merge>` for some type `I` (for example, `Performance` in the case of music, and `Rendering` in the case of animation).

THEOREM 6.1. The semantic function `meaning` is a *homomorphism* from `<Media T,:+:,:=:,none>` to `<I,concatM,merge,zero>`.

PROOF. We must show that:

```
meaning (m1:+:m2) =
  meaning m1 `concatM` meaning m2
meaning (m1:=:m2) =
  meaning m1 `merge`   meaning m2
meaning (none d) = zero d
```

This is easily done by unfolding the definition of `meaning`. □

THEOREM 6.2. `(===)` is a *congruence relation* on the algebra `<Media,:+:,:=:>`.

PROOF. We must show that, if `m1 === m2` and `m3 === m4`, then:

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

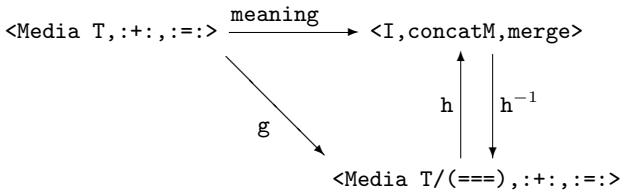

Figure 1: The Structure of Interpretation

This is easily done by unfolding the definition of `meaning` and appealing to the assumed properties of `concatM` and `merge`. □

DEFINITION 6.1. Let `[[m]]` denote the equivalence class (induced by `(===)`) that contains `m`. Let `Media T/(===)` denote the quotient set of such equivalence classes over base media type `T`, and let `<Media T/(===),:+:,:=:>` denote the quotient algebra, also called the *initial algebra*. The function:

```
g :: Media T -> Media T/(===)
g m = [[m]]
```

is called the *natural homomorphism* from `<Media T,:+:,:=:>` to `<Media T/(===),:+:,:=:>` [13]. Also define:

```
h :: Media T/(===) -> I
h [[m]] = meaning m
```

which is an isomorphism, whose inverse is:

$$h^{-1}\ p = [[m]],\ \text{if } p = \text{meaning } m$$

That `h` is an isomorphism follows from the fact that `g` is the natural homomorphism induced by `(===)`.

THEOREM 6.3. The diagram in Figure 1 commutes.

PROOF. In the direction of `h`:

```
h (g m) = h ([[m]]) = meaning m
```

In the direction of h^{-1}:

$$h^{-1}\ (\text{meaning } m) = [[m]] = g\ m$$

□

7. AXIOMATIC SEMANTICS

In Section 5 we noted that `(:=:)` was associative. Indeed, we can treat this as one of the *axioms* in an *axiomatic semantics* for polymorphic temporal media. The full set of axioms is given in the following definition:

DEFINITION 7.1. The axiomatic semantics `A` for well-formed polymorphic temporal media consists of the eight axioms shown in Figure 2, as well as the usual reflexive, symmetric, and transitive laws that arise from `(===)` being an equivalence relation, and the substitution laws that arise from `(===)` being a congruence relation. We write $A \vdash m1 = m2$ iff `m1 === m2` can be established from the axioms of `A`.

For any finite well-formed `m`, `m1`, `m2 :: Media T`, and non-negative `d :: Dur`:

1. `(:+:)` is associative:
   `m1 :+: (m2 :+: m3) === (m1 :+: m2) :+: m3`
2. `(:=:)` is associative:
   `m1 :=: (m2 :=: m3) === (m1 :=: m2) :=: m3`
3. `(:=:)` is commutative:
   `m1 :=: m2 === m2 :=: m1`
4. `none 0` is a left (sequential) zero:
   `none 0 :+: m === m`
5. `none 0` is a right (sequential) zero:
   `m :+: none 0 === m`
6. `none d` is a left (parallel) zero:
   `none d :=: m === m`, if `d = dur m`
7. `none` is additive:
   `none d1 :+: none d2 === none (d1+d2)`
8. serial/parallel swap:
   `(m1 :+: m2) :=: (m3 :+: m4) ===`
   `(m1 :=: m3) :+: (m2 :=: m4)`,
   if `dur m1 = dur m3` and `dur m2 = dur m4`

Note that `none d` is also a right zero for `(:=:)`, but that fact is derivable from (3) and (6).

Figure 2: The Axioms of A

7.1 Soundness

THEOREM 7.1. (Soundness) The axiomatic semantics `A` is *sound*. That is, for all well-formed `m1, m2 :: Media T`:

$$\texttt{A} \vdash \texttt{m1 = m2} \;\Rightarrow\; \texttt{m1 === m2}$$

PROOF. Each of the axioms can be shown to be true by straightforward equational reasoning, using the laws of class `Combine`. The overall proof then follows by a simple induction over any derivation of equivalence between `m1` and `m2`, and the transitivity of equivalence. □

As an example of a non-trivial theorem that can be proven from these axioms, recall Theorem 4.2 from Section 4, which we pointed out was false. By changing the equality in that theorem to one of equivalence as defined in this section, we can state a valid theorem as follows:

THEOREM 7.2. For all finite `x :: T` and non-negative `d :: Dur <= dur m`, if

```
takeM d (Prim x) :+: dropM d (Prim x) === Prim x
```

then for all finite well-formed `m :: Media T`,

```
takeM d m :+: dropM d m === m
```

PROOF. See Appendix A.2. □

This theorem provides confidence that `takeM` and `dropM` take apart and put back together media values in a "meaningful" way.

Example 1 (Music): Theorem 7.2, which holds for lists, does *not* hold for `Music`, since, for example, if `m = Prim (Note p 2)`, then:

```
takeM 1 m :+: dropM 1 m
= Prim (Note p 1) :+: Prim (Note p 1)
```

which is not equivalent to `m = Prim (Note p 2)`.

Example 2 (Animation): Theorem 7.2 *does* hold for `Animation`, since, if `d2>d1`, then:

```
meaning (takeM d1 (Prim (d2,f))
         :+: dropM d1 (Prim (d2,f)))
= meaning ((d1,f) :+: (d2-d1, f . (d1+)))
= (d1, map (picToImage . f)
       (take (truncate (d1*r)) [0,1/r ..]))
  `concatM`
  (d2-d1, map (picToImage . f . (d1+))
          (take (truncate ((d2-d1)*r)) [0,1/r ..]))
= (d1, map (picToImage . f)
       (take (truncate (d1*r)) [0,1/r ..]))
  `concatM`
  (d2-d1, map (picToImage . f)
          (take (truncate ((d2-d1)*r)) [d1,1/r ..]))
= (d1+d2-d1,
   map (picToImage . f)
    (take (truncate (d1*r)) [0,1/r ..]))
   ++
   map (picToImage . f)
    (take (truncate ((d2-d1)*r)) [d1,1/r ..]))
= (d2,
   map (picToImage . f)
       ((take (truncate (d1*r)) [0,1/r ..]))
        ++
        (take (truncate ((d2-d1)*r)) [d1,1/r ..]))
= (d2, map (picToImage . f)
          (take (truncate (d2*r)) [0,1/r ..]))
= meaning (d2,f)
= meaning (Prim (d2,f))
```

A similar argument holds when `d1>d2`.

Although Theorem 7.2 holds for animation, this is not necessarily a good thing, as we will see in the next section.

7.2 Completeness

Soundness of `A` tells us that if we can prove two media values are equivalent using the axioms, then in fact they are equivalent. We are also interested in the converse: if two media values are in fact equivalent, can we prove the equivalence using only the axioms? If so, the axiomatic semantics `A` is also *complete*.

Completeness results of any kind are usually much more difficult to establish than soundness results. The key to doing so in our case is the notion of a *normal form* for polymorphic temporal media values. Recall from the previous section the isomorphism between the algebras `<P,concatM,merge>` and `<Media T/(===),:+:,:=:>`. What we need to do first is identify a canonical representation of each equivalence class in `Media T/(===)`:

```
normalize :: (Ord (Media a), Temporal a) =>
             Media a -> Media a
normalize m = sortM (norm (dur m) 0 m)

norm :: (Ord (Media a), Temporal a) =>
        Dur -> Dur -> Media a -> Media a
norm d t m | isNone m = m
norm d t (Prim x)     =
  none t :+: Prim x :+: none (d-t-dur x)
norm d t (m1 :+: m2)  =
  norm d t m1 :=: norm d (t+dur m1) m2
norm d t (m1 :=: m2)  =
  norm d t m1 :=: norm d t m2
```

Figure 3: Normalization Function

DEFINITION 7.2. A well-formed media term `m :: Media T` is in *normal form* iff it is of the form:

$$\texttt{none d},\ \ \texttt{d} \geq 0$$

or

$$\begin{aligned}
&(\texttt{none d}_{11}\ \texttt{:+: Prim x}_1\ \texttt{:+: none d}_{12})\ \texttt{:=:}\\
&(\texttt{none d}_{21}\ \texttt{:+: Prim x}_2\ \texttt{:+: none d}_{22})\ \texttt{:=:}\\
&\ldots\\
&(\texttt{none d}_{n1}\ \texttt{:+: Prim x}_n\ \texttt{:+: none d}_{n2}),\ n \geq 1,\\
&\wedge\ \forall\ (1 \leq i \leq n),\\
&\quad \texttt{d}_{i1} + \texttt{d}_{i2} + \texttt{dur x}_i = \texttt{dur m},\\
&\wedge\ \forall\ (1 \leq i < n),\\
&\quad (\texttt{d}_{i1}, \texttt{x}_i, \texttt{d}_{i2}) \leq (\texttt{d}_{(i+1)1}, \texttt{x}_{(i+1)}, \texttt{d}_{(i+1)2})
\end{aligned}$$

To be completely rigorous, we assume that the operators `(:+:)` and `(:=:)` are right associative. We denote the set of media normal-forms over type `T` as `MediaNF T`.

The latter inequality above is defined lexicographically left-to-right. Note that it orders the media values in time, and also establishes an ordering on simultaneous media values.

Defining a normal form is not quite enough, however. We must show that (a) each normal form is unique: i.e. it is not equivalent to any other, and (b) any media value can be transformed into an equivalent normal form using only the axioms of `A`. We will treat (a) as an assumption, and return later to study situations where this is not true. For (b), we prove the following lemma:

LEMMA 7.1. Any `m : Media T` can be transformed into a media normal-form using only the axioms of `A`.

PROOF. We define a normalization function `normalize` in Figure 3. The auxiliary function `norm` does most of the work, returning a media value that is either `none d`, or is such that all interior nodes are parallel constructions (i.e. applications of `(:=:)`) and all of the leaves are of the form $(\texttt{none d}_{i1}\ \texttt{:+: Prim x}_i\ \texttt{:+: none d}_{i2})$. So this is *almost* in media normal form: what remains to be done is simply flatten and sort this structure, which is what `sortM` does. This is a straightforward task involving only the associativity of `(:=:)`, and thus we omit the details.

We focus instead on the function `norm`. In particular, we must prove, using only the axioms of `A`, that the normal form that `norm` generates has the same meaning as the original term. That is, we must prove that $\texttt{A} \vdash \texttt{norm (dur m) 0 m = m}$. We do so by relying on a more general result, that stated in Lemma 7.2 (see below). With that result we proceed straightforwardly as follows:

```
norm (dur m) 0 m                      lemma
=== none 0 :+: m :+: none (dur m - 0 - dur m)
=== none 0 :+: m :+: none 0           Axioms 4 and 5
=== m
```

□

LEMMA 7.2. For all `d, t :: Dur` and finite well-formed `m :: Media T`:

```
norm d t m === none t :+: m :+: none (d-t-dur m)
```

PROOF. See Appendix A.3. □

With these lemmas we can now prove our main result.

THEOREM 7.3. (Completeness) The axiomatic semantics `A` is *complete*, that is: for all `m1, m2 :: Media T`:

$$\texttt{m1 === m2} \ \Rightarrow\ \texttt{A} \vdash \texttt{m1 = m2}$$

if and only if the normal forms in `MediaNF T` are *unique*, that is, for all `nf1, nf2 :: MediaNF T`:

$$\texttt{nf1} \neq \texttt{nf2} \ \Rightarrow\ \neg(\texttt{nf1 === nf2})$$

PROOF. Assume that the normal forms are unique. If `m1 === m2`, then `p = meaning m1 = meaning m2`. Let `n1 = normalize m1` and `n2 = normalize m2`. Then $\texttt{A} \vdash \texttt{n1 = m1}$ and $\texttt{A} \vdash \texttt{n2 = m2}$. Thus:

```
meaning n1
 = meaning m1
 = p
 = meaning m2
 = meaning n2
```

But we know from Section 6 that there is an isomorphism between `Media T/(===)` and `I`. Therefore `p` corresponds uniquely to some normal form, namely $\texttt{h}^{-1}\ \texttt{p}$. This implies that $\texttt{n1 = h}^{-1}\ \texttt{p = n2}$, and thus $\texttt{A} \vdash \texttt{m1 = m2}$.

Now assume that the axioms are complete. We will show that the normal forms must therefore be unique by contradiction. If they are not unique, then there must be two normal forms `nf1` and `nf2` whose meanings are the same; i.e. `nf1 === nf2`. If just one of these is of the form `none d`, then it is clear that no axiom can establish its equivalence to the other, therefore the axioms must not be complete. This contradicts our assumption, so the normal forms must be unique. On the other hand, if `nf1` and `nf2` are each of the form `none d1 :+: Prim x :+: none d2`, then a similar argument follows: If either pair of corresponding durations are different, then no axiom can establish their equivalence. If both pairs of corresponding durations are the same, then it must be that two `Prim` values are equivalent, but that also cannot be proven by any axiom. Thus if `nf1` and `nf2` are in fact equivalent, the axioms are not complete. But that contradicts our assumption, so the normal forms must be unique. □

Theorem 7.3 is important not only because it establishes completeness, but also because it points out the special nature of the normal forms. That is, there can be no other

choice of the normal forms – they are uniquely tied to completeness.

Example 1 (Music): The normal forms for `Music`, i.e. `MusicNF Note`, are unique. In fact, the domain is isomorphic to `Performance`. To see this, we can define a bijection between the two domains as follows:

1. The music normal form `none d` corresponds to the interpretation `meaning (none d)` = `zero d`.

2. The non-trivial normal form, call it `m`, written in Definition 7.2, corresponds to the performance:

```
meaning m
= ( dur m, [ (d11, p1, d12),
       (d21, p2, d22), ...
       (dn1, pn, dn2) ], dur m)
```

   This correspondence is invertible because each `di3` is computable from the other durations; i.e. `di3 = dur m - di1 - di2`.

Example 2 (Animation): The normal forms of the `Animation` media type are *not* unique. There are several reasons for this. First, there may be pairs of primitive images that are equivalent, such as a circle of radius zero and a square of length zero, or a square and a polygon that mimics a square. Second, there may be pairs of animations that are equivalent because of the effect of occlusion. For example, a large box completely occluding a small circle is equivalent to a large box completely occluding any other image. It is possible to include additional axioms to cover these special cases, in which case the resulting axiomatic semantics may be complete, but the proof will not be automatic and cannot rely exclusively on the uniqueness of the normal forms.

8. OTHER CONCRETE MEDIA

In the final paper we will outline the representation of a *sound file* as an array of floating-point signal values sampled at a standard rate, and discuss the applicability to other media as well.

9. RELATED WORK

There has been a fair amount of work in embedding semantic descriptions *in* multimedia frameworks (XML, UML, the Semantic Web, and so on), but we are not aware of any work attempting to formalize the semantics *of* concrete media, at least not from a programming languages point of view. There are also many authoring tools, scripting languages, and so on for designing multimedia applications. The one closest to a programming language is SMIL [15]. This language can be seen as treating multimedia in a polymorphic way. Our own work on Haskore and MDL [9, 6, 7, 8] is of course highly related, but specialized to music. Graham Hutton shows how fold and unfold can be used to describe denotational and operational semantics, respectively [11], and thus our use of fold to describe the semantics of temporal media is an instance of his framework.

10. ACKNOWLEDGEMENTS

The NSF provided partial support for this research under grant number CCR9900957. Also thanks to the many students who have worked with me over the years on Haskore, which inspired this work more than any other effort.

APPENDIX

A. PROOFS

A.1 Proof of Theorem 4.1

For the first equation, beginning with the base cases:

```
If d2 = 0:
takeM d1 (takeM d2 m)           unfold takeM
= takeM d1 (none 0)             theorem above
= none 0                        fold takeM
= takeM d2 m                    fold min
= takeM (min d1 d2) m

If d1 = 0:
takeM d1 (takeM d2 m)           unfold takeM
= none 0                        fold takeM
= takeM d1 m                    fold min
= takeM (min d1 d2) m

takeM d1 (takeM d2 (Prim x))    unfold takeM
= takeM d1 (Prim (takeM d2 x))  unfolf takeM
= Prim (takeM d1 (takeM d2 x))  assumption
= Prim (takeM (min d1 d2) x)    fold takeM
= takeM (min d1 d2) (Prim x)
```

Induction steps:

```
takeM d1 (takeM d2 (m1 :+: m2))    unfold takeM
if d2 <= dur m1:
  = takeM d1 (takeM d2 m1)         ind hyp
  = takeM (min d1 d2) m1           fold takeM
  = takeM (min d1 d2) (m1 :+: m2)
if d2 > dur m1:
  = takeM d1 (m1 :+: takeM (d2 - dur m1) m2)
                                   unfold takeM
    if d1 <= dur m1:
    = takeM d1 m1                  fold min
    = takeM (min d1 d2) m1         fold takeM
    = takeM (min d1 d2) (m1 :+: m2)
    if d1 > dur m1
    = m1 :+: takeM (d1 - dur m1)
                (takeM (d2 - dur m1) m2)
    ind hyp
    = m1 :+: takeM (min (d1 - dur m1)
                        (d2 - dur m1)) m2
    arith
    = m1 :+: takeM (min d1 d2 - dur m1) m2
    fold takeM
    = takeM (min d1 d2) (m1 :+: m2)

takeM d1 (takeM d2 (m1 :=: m2))
  unfold takeM
= takeM d1 (takeM d2 m1 :=: takeM d2 m2)
  unfold takeM
= takeM d1 (takeM d2 m1) :=: takeM d1 (takeM d2 m2)
  ind hyp
= takeM (min d1 d2) m1 :=: takeM (min d1 d2) m2
  fold takeM
= takeM (min d1 d2) (m1 :=: m2)
```

(Remainder of proof omitted because of space limitations.)

A.2 Proof of Theorem 7.2

The base case is trivially true from the assumption. For the first induction step:

```
takeM d (m1 :+: m2) :+: dropM d (m1 :+: m2)
  unfold takeM and dropM
```

Now if `d <= dur m1`:

```
  = takeM d m1 :+: (dropM d m1 :+: m2)
  assoc
  = (takeM d m1 :+: dropM d m1) :+: m2
  ind hyp
  = m1 :+: m2
```

And if `d > dur m1`:

```
  = (m1 :+: take (d - dur m1) m2) :+:
    dropM (d - dur m1) m2
  assoc
  = m1 :+: (takeM (d - dur m1) m2 :+:
            dropM (d - dur m1) m2)
  ind hyp
  = m1 :+: m2
```

Finally, for the second induction step:

```
takeM d (m1 :=: m2) :+: dropM d (m1 :=: m2)
  unfold takeM and dropM
= (takeM d m1 :=: takeM d m2) :+:
  (dropM d m1 :=: dropM d m2)      ser/par axiom
= (takeM d m1 :+: dropM d m1) :=:
  (takeM d m2 :+: dropM d m2)      ind hyp
= m1 :=: m2
```

A.3 Proof of Lemma 7.2

Base cases:

```
norm d t (none d')
= none d
= none t :+: none d' :+: none (d-t-d')

norm d t (Prim x)
= none t :+: Prim x :+: none (d-t-dur x)
= none t :+: Prim x :+: none (d-t-dur(Prim x))
```

For the first induction step, let `d1 = dur m1`, `d2 = dur m2`, and `d12 = dur (m1:=:m2)`. Then:

```
norm d t (m1 :=: m2)
=   norm d t m1 :=: norm d t m2
=== (none t :+: m1 :+: none (d-t-d1)) :=:
    (none t :+: m2 :+: none (d-t-d2))
=== (none t :=: none t) :+:
    (m1 :=: m2) :+:
    (none (d-t-d1) :=: none (d-t-d2))
=== none t :+: (m1 :=: m2) :+:
    (none (d-t-d1) :=: none (d-t-d2))
=== none t :+: (m1 :=: m2) :+:
    none (d-t-d12)
```

For the second induction step, let `d1 = dur m1`, `d2 = dur m2`, and `d12 = dur (m1:+:m2)`. Then:

```
norm d t (m1 :+: m2)
=   norm d t m1 :=: norm d (t+d1) m2
=   (none t      :+: m1 :+: none (d-t-d1)) :=:
    (none (t+d1) :+: m2 :+: none (d-t-d1-d2))
=   (none t      :+: m1 :+: none (d-t-d1)) :=:
    (none t :+: none d1 :+: m2 :+: none (d-t-d1-d2))
=== (none t :=: none t) :+:
    (m1 :=: none d1)    :+:
    (none (d-t-d1) :=: (m2 :+: none (d-t-d1-d2)))
=== none t :+: m1 :+:
    (m2 :+: none (d-t-d1-d2)))
=== none t :+: (m1 :+: m2) :+: none (d-t-d12)
```

B. REFERENCES

[1] International MIDI Association. Midi 1.0 detailed specification: Document version 4.1.1, February 1990.

[2] Richard S. Bird. *Introduction to Functional Programming using Haskell (second edition)*. Prentice Hall, London, 1998.

[3] Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the first conference on Domain-Specific Languages*, pages 285–296. USENIX, October 1997.

[4] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 263–273, June 1997.

[5] Liwen Huang and Paul Hudak. Dance: A language for humanoid robot motion. In *International Conference on Functional Programming*. ACM Press, 2003. Submitted for publication.

[6] Paul Hudak. Haskore music tutorial. In *Second International School on Advanced Functional Programming*, pages 38–68. Springer Verlag, LNCS 1129, August 1996.

[7] Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, New York, 2000.

[8] Paul Hudak. *Describing and Interpreting Music in Haskell*, chapter 4. Palgrave, 2003. *The Fun of Programming*, edited by Jeremy Gibbons and Oege de Moor.

[9] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3):465–483, May 1996.

[10] Ann Hutchinson. *Labanotation*. Routledge Theatre Arts Books, New York, 1991.

[11] Graham Hutton. Fold and unfold for program semantics. In *International Conference on Functional Programming*. ACM Press, 1998.

[12] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[13] J.P. Tremblay and R. Manohar. *Discrete Mathematical Structures with Applications to Computer Science*. McGraw-Hill, New York, 1975.

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

[15] World Wide Web Consortium (W3C). Synchronized Multimedia Integration Language (SMIL), 2003. http://www.w3.org/AudioVideo.