A Sound and Complete Axiomatization of Polymorphic Temporal Media

Paul Hudak
Department of Computer Science
Yale University
paul.hudak@yale.edu

July 1, 2008

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. We present a polymorphic data type that captures a broad range of temporal media, and study its syntactic, temporal, and semantic properties. This leads 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 that is both sound and complete.

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 [12] and a language for humanoid robot motion [7, 6]. In this paper we use two running examples throughout: an abstract representation of music (analogous to our previous work on Haskore and MDL, two DSLs for computer music [8, 9, 10, 11]), and an abstract representation of continuous animations (analogous to our previous work on Fran and FAL [4, 3, 9]).

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.

More generally, 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 our results using a combination of mathematical notation and Haskell [14] syntax. There is little to lose in using Haskell syntax, since it is close to mathematical notation, but in addition it offers several advantages, including:

- Correctness. The Haskell type-checker ensures that our declarations are type-safe and consistent.
- Executable specifications. Although we did not focus on efficiency, all of our results are executable, thus providing further assurance of correctness and laying the groundwork for a practical implementation.
- Expressiveness. Haskell’s higher-order functions and type classes allow us to present certain concepts more clearly and concisely compared to conventional mathematics.

The downside of using Haskell is that superficial coding details are occasionally necessary to ensure executability. These details, and unusual Haskell language features, are explained as they are introduced. Proofs of key theorems are presented in the running text. Simple proofs are omitted, and some proofs are relegated to the appendix.
2 Polymorphic Media

We represent temporal media by a polymorphic data type:

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

deriving (Show, Eq, Ord)
```

We refer to $T$ in $\text{Media } T$ as the base media type. This data declaration tells us that, for values $x :: T$ and $m_1, m_2 :: \text{Media } T$, a value of type $\text{Media } T$ is either a primitive value $\text{Prim } x$, a sequential composition $m_1 :+: m_2$, or a parallel composition $m_1 :+: m_2$.\(^1\) Although simple in structure, this data type is rich enough to capture a number of useful media types.

**Example 2.1 (Music)** Consider this definition of an abstract notion of a musical note:\(^2\)

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

deriving (Show, Eq, Ord)

type Dur = NNReal

type Pitch = (NoteName, Octave)

type Octave = Int

data NoteName = Cf | C | Cs | Df | D | Fs | F

| Ef | E | Es | E |
| Gf | G | Fs | F |
| A | As | Bf | B |
| Bs

deriving (Show, Eq, Ord)
```

In other words, a $\text{Note}$ is either a pitch paired with a duration, or a $\text{Rest}$ that has a duration but no pitch. $\text{Dur}$ is a measure of time (duration), which we assume to be a non-negative real number ($\text{NNReal}$); in a practical implementation a suitable approximation such as $\text{Float}$, $\text{Double}$, or $\text{Ratio Integer}$ would be used. A $\text{Pitch}$ is a pair consisting of a note name and an octave, where an octave is just an integer. The note name $\text{Cf}$ is read as “C-flat” (normally written as $C^\flat$), $\text{Cs}$ as “C-sharp” (normally written as $C^\#$), and so on.\(^3\)

Then the type:

```haskell
type Music = Media Note
```

is a temporal media for music. In particular, a value $\text{Prim } (\text{Rest } d)$ is a rest of duration $d$, $\text{Prim } (\text{Note } p d)$ is a note with pitch $p$ played for duration $d$, $m_1 :+: m_2$ is the music value $m_1$ followed sequentially in time by $m_2$, and

---

1. The **deriving** phrase automatically defines printing ($\text{Show}$), equality ($\text{Eq}$), and ordering ($\text{Ord}$) operators on elements of the $\text{Media}$ data type.
2. In Haskell, a **data** declaration introduces a new datatype, with its instances delimited by “$|$”, whereas a **type** declaration simply introduces a new name, or synonym, for an existing type. The type ($T_1, T_2$) is the Cartesian product of $T_1$ and $T_2$.
3. This representation corresponds well to that used in music theory, except that in music theory note names are called *pitch classes*.  

3
$m_1 := m_2$ is $m_1$ played simultaneously with $m_2$. This representation of music is a simplified version of that used in the Haskore computer music library [11, 8]. If we define:

\[
\text{mkNote } n d = \text{Prim } (\text{Note } n d)
\]

then the music expression:

\[
\text{let } \text{dMinor} = \text{mkNote } (D, 3) 1 :=: \text{mkNote } (F, 3) 1 :=: \text{mkNote } (A, 3) 1 \\
\text{gMajor} = \text{mkNote } (G, 3) 1 :=: \text{mkNote } (B, 3) 1 :=: \text{mkNote } (D, 4) 1 \\
\text{cMajor} = \text{mkNote } (C, 3) 2 :=: \text{mkNote } (E, 3) 2 :=: \text{mkNote } (G, 3) 2 \\
\text{in } \text{dMinor} :+: \text{gMajor} :+: \text{cMajor}
\]

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

**Example 2.2 (Animation)** Consider this definition of a base media type for continuous animations:

\[
\begin{align*}
\text{data } \text{Anim} &= \text{NullAnim } \text{Dur} \\
&\mid \text{Anim } \text{Dur} \rightarrow \text{Picture} \\
\text{type } \text{Time} &= \text{NNReal} \\
\text{data } \text{Picture} &= \text{Circle } \text{Radius } \text{Point} \\
&\mid \text{Square } \text{Length } \text{Point} \\
\text{deriving Show} \\
\text{type } \text{Point} &= (\text{Double}, \text{Double})
\end{align*}
\]

An $\text{Anim}$ is either null for a given duration, or is a value $\text{Anim } d f$, a continuous animation whose image at time $0 \leq t \leq d$ is the $\text{Picture}$ value $f t$. A $\text{Picture}$, in turn, is either a circle or square of a given size and located at a particular point. Then the type:

\[
\text{type } \text{Animation} = \text{Media } \text{Anim}
\]

is a temporal media for continuous animations. This representation is a simplified version of that used in Fran [4, 3] and FAL [9]. As a simple example:

\[
\begin{align*}
\text{let } \text{ball}_1 &= \text{Anim } 10 (\lambda t \rightarrow \text{Circle } t \text{ origin}) \\
\text{ball}_2 &= \text{Anim } 10 (\lambda t \rightarrow \text{Circle } (10 - t) \text{ origin}) \\
\text{box} &= \text{Anim } 20 (\lambda t \rightarrow \text{Square } 1 (t, t)) \\
\text{in } (\text{ball}_1 :+: \text{ball}_2) :=: \text{box}
\end{align*}
\]

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, 9].
### 3.1 Map (functor)

For starters, it is easy to define a polymorphic map function on temporal media, as follows:

- \( \text{mapM} :: (a \to b) \to \text{Media } a \to \text{Media } b \)
- \( \text{mapM } f \ (\text{Prim } n) = \text{Prim } (f \ n) \)
- \( \text{mapM } f \ (m_1 :+: m_2) = \text{mapM } f \ m_1 :+: \text{mapM } f \ m_2 \)
- \( \text{mapM } f \ (m_1 :=: m_2) = \text{mapM } f \ m_1 :=: \text{mapM } f \ m_2 \)

\( \text{mapM} \) shares many properties with the standard map function defined on lists, and in particular:

**Theorem 3.1** For any \( m :: \text{Media } T_1 \) and functions \( f, g :: T_1 \to T_2 \):

- \( \text{mapM } (f \circ g) = \text{mapM } f \circ \text{mapM } g \)
- \( \text{mapM } id = id \)

(The proof is by a straightforward induction, and is omitted.)

A common design technique used in the Haskell community is to associate a set of properties such as the above with a type class that collects together a common set of types and functions over those types. Indeed, in Haskell the following type class is pre-defined:

```haskell
class Functor f where
  fmap :: (a -> b) -> fa -> fb
```

This declaration can be read: “a type \( f \) is a member of the class \( \text{Functor} \) if it supports an operation \( \text{fmap} \) whose type is \((a \to b) \to f \ a \to f \ b\).” Recall now that the type of \( \text{mapM} \) is: \((a \to b) \to \text{Media } a \to \text{Media } b\). Thus it is easy to see that \( \text{Media} \) is a functor, which we can declare by writing:

```haskell
instance Functor Media where
  fmap = mapM
```

More importantly, the two properties stated in Theorem 3.1 are expected to hold for any type that is an instance of the \( \text{Functor} \) class. (Such properties associated with a type class are called laws.) Since these two laws hold for \( \text{mapM} \) as established in Theorem 3.1, \( \text{Media} \) is thus a valid instance of the class \( \text{Functor} \).

Although fairly simple, this example serves as a nice demonstration of how we intend to use type classes in this paper: to collect together one or more operations on a particular type, and to state a set of laws that must hold for those operations.

\( \text{mapM} \) 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 3.1 (Music)** We define a function in terms of \( \text{mapM} \) that alters the tempo of a Music value:
tempo :: Dur → Music → Music
\[
\text{tempo } r = \text{mapM } (\text{temp } r)
\]
\[
\text{where temp } r (\text{Rest } d) = \text{Rest } (r \ast d)
\]
\[
\text{temp } r (\text{Note } p \ d) = \text{Note } p \ (r \ast d)
\]

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

\[
\text{trans} :: \text{Int} → \text{Music} → \text{Music}
\]
\[
\text{trans } i = \text{mapM } (\text{tran } i)
\]
\[
\text{where tran } i (\text{Rest } d) = \text{Rest } d
\]
\[
\text{tran } i (\text{Note } p \ d) = \text{Note } \ (\text{transPitch } i \ p) \ d
\]

where \text{transPitch } i \ p \text{ translates pitch } p \text{ by interval } i \text{ (straightforward definition omitted). These two functions obviate the need for the Tempo and Trans data constructors used in Haskore and MDL [11, 8, 9].}

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

\textbf{Corollary 3.1} For any \( r_1, r_2 :: \text{Dur} \) and \( i_1, i_2 :: \text{Int} \):

\[
\text{tempo } r_1 \circ \text{tempo } r_2 = \text{tempo } (r_1 \ast r_2)
\]
\[
\text{trans } i_1 \circ \text{trans } i_2 = \text{trans } (i_1 + i_2)
\]
\[
\text{tempo } r_1 \circ \text{tempo } r_2 = \text{tempo } r_2 \circ \text{tempo } r_1
\]
\[
\text{trans } i_1 \circ \text{trans } i_2 = \text{trans } i_2 \circ \text{trans } i_1
\]
\[
\text{tempo } r_1 \circ \text{trans } i_1 = \text{trans } i_1 \circ \text{tempo } r_1
\]

\textbf{Example 3.2 (Animation)} We define a function to scale in size and translate in space an Animation value:

\[
\text{scale} :: \text{Double} → \text{Point} → \text{Animation} → \text{Animation}
\]
\[
\text{scale } s (dx, dy) = \text{mapM } g
\]
\[
\text{where } g \ (\text{NullAnim } d) = \text{NullAnim } d
\]
\[
\ g \ (\text{Anim } d \ f) = \text{Anim } d \ (\text{scal } \circ f)
\]
\[
\text{scal } (\text{Square } l \ p) = \text{Square } (s \ast l) \ (\text{scalePt } p)
\]
\[
\text{scal } (\text{Circle } r \ p) = \text{Circle } (s \ast r) \ (\text{scalePt } p)
\]
\[
\text{scalePt } (x, y) = (s \ast x + dx, s \ast y + dy)
\]

This function obviates the need for the Scale and Translate constructors used in Fran and Fal [4, 3, 9]. In a similar way, a function \text{rate} can be defined such that \text{rate } r \ a \text{ scales the frame rate of animation } a \text{ by a factor } r.

Using Theorem 3.1 it is then straightforward to show:

\textbf{Corollary 3.2} For any \( s_1, s_2, x_1, x_2, y_1, y_2 :: \text{Double} \):

\[
\text{scale } s_1 (x_1, y_1) \circ \text{scale } s_2 (x_2, y_2) = \text{scale } (s_1 \ast s_2) \ (x_1 + x_2, y_1 + y_2)
\]
\[
\text{scale } s_1 p_1 \circ \text{scale } s_2 p_2 = \text{scale } s_2 p_2 \circ \text{scale } s_1 p_1
\]
3.2 Fold (catamorphism)

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

\[
\text{foldM} :: (a \rightarrow b) \rightarrow (b \rightarrow b) \rightarrow (b \rightarrow b) \rightarrow \text{Media} \rightarrow a \rightarrow b
\]

\[
\text{foldM} f g h (\text{Prim} \ x) = f x
\]

\[
\text{foldM} f g h (m_1 :+ : m_2) = \text{foldM} f g h \ m_1 \ g \ \text{foldM} f g h \ m_2
\]

\[
\text{foldM} f g h (m_1 := : m_2) = \text{foldM} f g h \ m_1 \ h \ \text{foldM} f g h \ m_2
\]

**Theorem 3.2** For any \( f :: T_1 \rightarrow T_2 \):

\[
\text{foldM} (\text{Prim} \circ f) (:+:) (:=:) = \text{mapM} f
\]

\[
\text{foldM} \text{Prim} (:+:) (:=:) = \text{id}
\]

More interestingly, we can state a fusion law for \( \text{foldM} \):

**Theorem 3.3 (Fusion Law)** For \( f :: T_1 \rightarrow T_2 \), \( g, h :: T_2 \rightarrow T_2 \rightarrow T_2 \), and \( g', h' :: T_1 \rightarrow T_3 \), 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: \( k \circ \text{foldM} f g h = \text{foldM} f' g' h' \).

**Proof** By induction. Base case:

\[
k (\text{foldM} f g h (\text{Prim} \ x))
\]

\[
= k (f x) \quad \text{unfold foldM}
\]

\[
= f' x \quad \text{assumption}
\]

\[
= \text{foldM} f' g' h' (\text{Prim} \ x) \quad \text{fold foldM}
\]

Induction step:

\[
k (\text{foldM} f g h (m_1 :+ : m_2))
\]

\[
= k (g (\text{foldM} f g h \ m_1) (\text{foldM} f g h \ m_2)) \quad \text{unfold foldM}
\]

\[
= g' (k (\text{foldM} f g h \ m_1)) (k (\text{foldM} f g h \ m_2)) \quad \text{assumption}
\]

\[
= g' (\text{foldM} f' g' h' m_1) (\text{foldM} f' g' h' m_2) \quad \text{induction hypothesis}
\]

\[
= \text{foldM} f' g' h' (m_1 :+ : m_2) \quad \text{fold foldM}
\]

Similarly for \( (:=:) \).

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

**Corollary 3.3 (Fold-Map Fusion Law)**

For all \( f :: T_1 \rightarrow T_2 \), \( g, h :: T_2 \rightarrow T_2 \rightarrow T_2 \), and \( j :: T0 \rightarrow T1 \):

\[
\text{foldM} f g h \circ \text{mapM} j = \text{foldM} (f \circ j) g h
\]

**Example 3.3** 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.
3.3 Reverse

We would like to define a function \( \text{reverseM} \) that reverses, \textit{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:

\[
\begin{align*}
\text{class } & \text{Reverse } a \text{ where} \hfill \\
& \text{reverseM } : : a \rightarrow a
\end{align*}
\]

\[
\begin{align*}
\text{instance } & \text{Reverse } a \Rightarrow \text{Reverse } (\text{Media } a) \text{ where} \hfill \\
& \text{reverseM } (\text{Prim } a) = \text{Prim } (\text{reverseM } a) \hfill \\
& \text{reverseM } (m_1 :+ : m_2) = \text{reverseM } m_2 :+ : \text{reverseM } m_1 \hfill \\
& \text{reverseM } (m_1 :+: m_2) = \text{reverseM } m_1 :+: \text{reverseM } m_2
\end{align*}
\]

The phrase “\( Reverse \ a \Rightarrow \ldots \)” in the instance declaration is called a constraint. The instance declaration can be read: “If the type \( a \) is reversible, then so is \( \text{Media } a \), as witnessed by the following definition of \( \text{reverseM} \).” Note that \( \text{reverseM} \) can be defined more succinctly as a catamorphism:

\[
\begin{align*}
\text{instance } & \text{Reverse } a \Rightarrow \text{Reverse } (\text{Media } a) \text{ where} \hfill \\
& \text{reverseM } = \text{foldM } (\text{Prim } \circ \text{reverseM}) (\text{flip } (:+:)) (:=:)
\end{align*}
\]

Analogous to a similar property on lists, we have:

**Theorem 3.4** For all finite \( m \), if the following law holds for \( \text{reverseM } : : T \rightarrow T \), then it also holds for \( \text{reverseM } : : \text{Media } T \rightarrow \text{Media } T \):

\[
\text{reverseM } (\text{reverseM } m) = m
\]

We take the constraint in this theorem to be a law for all valid instances of a base media type \( T \) in the class \( \text{Reverse} \).

**Proof** It is straightforward to prove this theorem using structural induction. However, one can also carry out an inductionless proof by using the fusion law of Theorem 3.3, as follows:

\[
\begin{align*}
& (\text{reverseM } \circ \text{reverseM}) \ m \\
= & (\text{reverseM } \circ \text{foldM } (\text{Prim } \circ \text{reverseM}) (\text{flip } (:+:)) (:=:)) \ m \\
= & \text{foldM } \text{Prim } (:+:) (:=:) \ m \quad \text{– unfold } \text{reverseM} \\
= & m \quad \text{– fusion law (Theorem 3.3)} \\
= & m \quad \text{– Theorem 3.2}
\end{align*}
\]

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

\[
\begin{align*}
& \text{Prim } x \\
= & \text{Prim } (\text{reverseM } (\text{reverseM } x)) \quad \text{– assumption} \\
= & \text{reverseM } (\text{Prim } (\text{reverseM } x)) \quad \text{– fold } \text{reverseM}
\end{align*}
\]

\[\text{Although the analogous theorem for lists does not hold for infinite lists, Theorem 3.4 does hold for Music values that are finite in the branching of } (+:), \text{ and thus finite in duration, but infinite in the branching of } (=:), \text{ which does not imply infinite duration.}\]
\( = \text{reverseM} \left( (\text{Prim} \circ \text{reverseM}) \ x \right) \quad \text{– fold} \ \circ \)

\( \text{(:+)} \ (\text{reverseM} \ x) \ (\text{reverseM} \ y) \)
\( = \text{flip} \ (\text{:+}) \ (\text{reverseM} \ y) \ (\text{reverseM} \ x) \quad \text{– fold} \ \text{flip} \)
\( = \text{reverseM} \ (y \text{:+} \ x) \quad \text{– fold} \ \text{reverseM} \)
\( = \text{reverseM} \ (\text{flip} \ (\text{:+}) \ x \ y) \quad \text{– fold} \ \text{flip} \)

\( \text{(:=\text{)} (\text{reverseM} \ x) (\text{reverseM} \ y)} \)
\( = \text{reverseM} \ (x \text{:=} \ y) \quad \text{– fold} \ \text{reverseM} \)

\[ \square \]

Note that \text{reverseM} also interacts nicely with \text{mapM}, just as \text{reverse} interacts with \text{map} on lists:

**Theorem 3.5** For any \( f :: T \rightarrow T \), if \( f \circ \text{reverseM} = \text{reverseM} \circ f \), then:

\[ \text{mapM} \ f \circ \text{reverseM} = \text{reverseM} \circ \text{mapM} \ f \]

**Proof** Using both fusion laws (Theorem 3.3 and Corollary 3.3) (details omitted). \( \square \)

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

\[ \text{foldr} \ \text{op} \ e \ \text{xs} = \text{foldl} \ (\text{flip} \ \text{op}) \ e \ (\text{reverse} \ \text{xs}) \]

**Theorem 3.6** For all finite \( m :: \text{Media} \ T \), functions \( g, h :: T \rightarrow T \rightarrow T \), and \( f, f' :: T \rightarrow T \) such that \( f = f' \circ \text{reverseM} \):

\[ \text{foldM} \ f \ g \ h \ m = \text{foldM} \ f' (\text{flip} \ g) \ h \ (\text{reverseM} \ m) \]

**Proof** By structural induction, starting with the base case:

\[ \text{foldM} \ f \ g \ h \ (\text{Prim} \ x) \]
\( = f \ x \quad \text{– unfold} \ \text{foldM} \)
\( = f' \ (\text{reverseM} \ x) \quad \text{– assumption} \)
\( = \text{foldM} \ f' (\text{flip} \ g) \ h \ (\text{Prim} \ (\text{reverseM} \ x)) \quad \text{– fold} \ \text{foldM} \)
\( = \text{foldM} \ f' (\text{flip} \ g) \ h \ (\text{reverseM} \ (\text{Prim} \ x)) \quad \text{– fold} \ \text{reverseM} \)

Induction steps:

\[ \text{foldM} \ f \ g \ h \ (m_1 \text{:+} m_2) \]
\( = \text{foldM} \ f \ g \ h \ m_1 \text{:'} g' \ \text{foldM} \ f \ g \ h \ m_2 \quad \text{– unfold} \ \text{foldM} \)
\( = \text{foldM} \ f' (\text{flip} \ g) \ h \ (\text{reverseM} \ m_1 \text{:'} g') \quad \text{– induction hypothesis} \)
\( = \text{foldM} \ f' (\text{flip} \ g) \ h \ (\text{reverseM} \ m_2 \text{:'} (\text{flip} \ g)') \quad \text{– fold} \ \text{foldM} \)
\( = \text{foldM} \ f' (\text{flip} \ g) \ h \ (\text{reverseM} \ m_2 \text{:+} \text{reverseM} \ m_1) \quad \text{– fold} \ \text{flip} \)
\( = \text{foldM} \ f' (\text{flip} \ g) \ h \ (\text{reverseM} \ (m_1 \text{:+} m_2)) \quad \text{– fold} \ \text{reverseM} \)
foldM \ f \ g \ h \ (m_1 :=: m_2) \\
= \ foldM \ f \ g \ h \ m_1 \ 'h' \ foldM \ f \ g \ h \ m_2 \\
= \ foldM \ f' \ (\text{flip} \ g) \ h \ (\text{reverseM} \ m_1)'h' \\
foldM \ f' \ (\text{flip} \ g) \ h \ (\text{reverseM} \ m_2) \\
= \ foldM \ f' \ (\text{flip} \ g) \ h \ (\text{reverseM} \ m_1 :=: \text{reverseM} \ m_2) \\
= \ foldM \ f' \ (\text{flip} \ g) \ h \ (\text{reverseM} \ (m_1 :=: m_2)) \\
\text{-- induction hypothesis} \\
\text{-- fold foldM} \\
\text{-- fold reverseM}

**Example 3.4 (Music)** We declare \textit{Note} to be an instance of class \textit{Reverse}:

\begin{verbatim}
instance \textit{Reverse Note} where \\
\text{reverseM} = \text{id}
\end{verbatim}

In other words, a single note is the same whether interpreted backwards or forwards. (If this seems counter-intuitive, note that this does not imply that the \textit{sound} of a note is the same if it is reversed. Rather, the \textit{representation} of the note, for example as a note on a conventional score, is the same if it is reversed.) The constraint in Theorem 3.4 is therefore trivially satisfied, and it thus holds for music media.\(^5\)

**Corollary 3.4 (to Theorem 3.5)** For any \(r :: \text{Dur}\) and \(i :: \text{Int}\):

\begin{align*}
\text{reverseM} \circ \text{tempo} \ r &= \text{tempo} \ r \circ \text{reverseM} \\
\text{reverseM} \circ \text{trans} \ i &= \text{trans} \ i \circ \text{reverseM}
\end{align*}

**Corollary 3.5 (to Theorem 3.6)** For any finite \(m :: \text{Media Note}\):

\begin{align*}
\text{foldM} \ f \ g \ h \ m &= \text{foldM} \ f \ (\text{flip} \ g) \ h \ (\text{reverse} \ m)
\end{align*}

**Example 3.5 (Animation)** We declare \textit{Anim} to be an instance of \textit{Reverse}:

\begin{verbatim}
instance \textit{Reverse Anim} where \\
\text{reverseM} \ (\text{NullAnim} \ d) = \text{NullAnim} \ d \\
\text{reverseM} \ (\text{Anim} \ d \ f) = \text{Anim} \ d \ (\lambda t \rightarrow f (d - t))
\end{verbatim}

Note that \(\text{reverseM} \ (\text{reverseM} \ (\text{Anim} \ d \ f)) = \text{Anim} \ d \ f\), therefore the constraint in Theorem 3.4 is satisfied, and the theorem thus holds for continuous animations.

**Corollary 3.6 (to Theorem 3.5)** For all \(s, d :: \text{Double}\):

\begin{align*}
\text{reverseM} \circ \text{scale} \ s \ d &= \text{scale} \ s \ d \circ \text{reverseM}
\end{align*}

\(^5\)The reverse of a musical passage is called its \textit{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 (Menneto al Rovescio)), it is a standard construction in modern twelve-tone music [5].
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 $m_1 ::= m_2$ when the durations of $m_1$ and $m_2$ are different. There are at least four possibilities:

1. $m_1$ and $m_2$ begin at the same time, and when the longer one finishes, the entire construction finishes.
2. $m_1$ and $m_2$ begin at the same time, and when the shorter one finishes, the entire construction finishes (thus truncating the longer one).
3. $m_1$ and $m_2$ are “centered” in time, so that the shorter one begins after one-half of the difference between their durations.
4. The situation is disallowed: i.e. $m_1$ and $m_2$ 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 [11, 8, 9]. 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 [10].

In the present treatment 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 will enforce using type classes. Anything expressed using one of the other three options can be expressed using option four by padding the media value with suitable “rests” in appropriate places.

4.1 Duration

Recall from Section 2 that Dur is represented by the domain of non-negative reals. To facilitate the enforcement of this constraint when subtraction is involved, we define:

$$(\ominus) :: \text{NNReal} \rightarrow \text{NNReal} \rightarrow \text{NNReal}$$

$$x \ominus y = \max 0 (x - y)$$

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 again enforce using type classes:

```haskell
class Temporal a where
dur :: a \rightarrow Dur
none :: Dur \rightarrow a
isNone :: a \rightarrow Bool
```
\textbf{instance} Temporal \( a \Rightarrow Temporal (Media a) \) where
\[
\begin{align*}
dur &= \text{foldM } dur (+) \max \\
none &= \text{Prim } \circ \text{none} \\
isNone &= \text{foldM } isNone (\land) (\land)
\end{align*}
\]

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

We take the constraint in the following theorem to be a law for any valid instance of a base media type \( T \) in the class \textit{Temporal}:

\textbf{Theorem 4.1} If the property \( dur (\text{none } d) = d \) holds for \( dur :: T \rightarrow Dur \), then it also holds for \( dur :: Media T \rightarrow Dur \).

In addition, we require that \( isNone (\text{none } d) = \text{True} \).

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 compositions to those whose two argument durations are the same. Thus we define:

\textbf{Definition 4.1} A \textit{well-formed} temporal media value \( m :: Media T \) is one that is finite, and for which each parallel composition \( m_1 ::= m_2 \) has the property that \( dur m_1 = dur m_2 \).

Note that \( dur \) is analogous to the \textit{length} operator on lists, and obeys a law analogous to \( \text{length } l_1 + \text{length } l_2 = \text{length } (l_1 \# l_2) \).

\textbf{Example 4.1 (Music)} We declare \textit{Note} to be \textit{Temporal}:

\textbf{instance} Temporal Note where
\[
\begin{align*}
dur (\text{Rest } d) &= d \\
dur (\text{Note } p d) &= d \\
none d &= \text{Rest } d \\
\text{isNone } (\text{Rest } _) &= \text{True} \\
\text{isNone } (\text{Note } _) &= \text{False}
\end{align*}
\]

Thus \( dur :: Music \rightarrow Dur \) determines the duration of a \textit{Music} media value.

\textbf{Example 4.2 (Animation)} We declare \textit{Anim} to be \textit{Temporal}:

\textbf{instance} Temporal Anim where
\[
\begin{align*}
dur (\text{NullAnim } d) &= d \\
dur (\text{Anim } d f) &= d \\
none d &= \text{NullAnim } d \\
\text{isNone } (\text{NullAnim } d) &= \text{True} \\
\text{isNone } _ &= \text{False}
\end{align*}
\]

Thus \( dur :: Animation \rightarrow Dur \) determines the duration of an \textit{Animation} media value.
4.2 Take and Drop

We now define two functions \( \text{takeM} \) and \( \text{dropM} \) that are analogous to Haskell's \( \text{take} \) and \( \text{drop} \) functions for lists. The difference is that instead of being parameterized by a number of elements, \( \text{takeM} \) and \( \text{dropM} \) are parameterized by \textit{time} (i.e. duration). 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 \( \text{takeM} \ d \ m \) is a media value corresponding to the first \( d \) seconds of \( m \). Similarly, \( \text{dropM} \ d \ m \) is all but the first \( d \) seconds. Both of these are useful in practice.

\[
\text{class Take a where}
\begin{align*}
\text{takeM} \ :: \ & \text{Dur} \to a \to a \\
\text{dropM} \ :: \ & \text{Dur} \to a \to a
\end{align*}
\]

\[
\text{instance} \ (\text{Take a, Temporal a}) \Rightarrow \text{Take (Media a)} \text{ where}
\begin{align*}
\text{takeM} 0 \ m &= \text{none} 0 \\
\text{takeM} \ d \ (\text{Prim} \ x) &= \text{Prim} \ (\text{takeM} \ d \ x) \\
\text{takeM} \ d \ (m_1 :+; m_2) &= \text{let} \ d_1 = \text{dur} \ m_1 \\
\text{if} \ d \leq d_1 \ \text{then} \ & \text{takeM} \ d \ m_1 \\
\text{else} \ m_1 :+; \ & \text{takeM} \ (d - d_1) \ m_2 \\
\text{takeM} \ d \ (m_1 :=; m_2) &= \text{let} \ d_1 = \text{dur} \ m_1 \\
\text{if} \ d \leq d_1 \ \text{then} \ & \text{takeM} \ d \ m_1 :+; m_2 \\
\text{else} \ & \text{takeM} \ (d - d_1) \ m_2 \\
\text{dropM} 0 \ m &= m \\
\text{dropM} \ d \ (\text{Prim} \ x) &= \text{Prim} \ (\text{dropM} \ d \ x) \\
\text{dropM} \ d \ (m_1 :+; m_2) &= \text{let} \ d_1 = \text{dur} \ m_1 \\
\text{if} \ d \leq d_1 \ \text{then} \ & \text{dropM} \ d \ m_1 :+; m_2 \\
\text{else} \ & \text{dropM} \ (d - d_1) \ m_2 \\
\text{dropM} \ d \ (m_1 :=; m_2) &= \text{dropM} \ d \ m_1 :=; \text{dropM} \ d \ m_2
\end{align*}
\]

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 constraints in the following theorem to be laws for any valid instance of a base media type \( T \) in the class \textit{Temporal}:

**Theorem 4.2** If the following laws hold for any finite \( m :: T \), then they also hold for any well-formed \( m :: \text{Media} \ T \):

\[
\begin{align*}
\text{takeM} \ 0 \ m &= \text{none} 0 \\
\text{dropM} \ 0 \ m &= m \\
\text{takeM} \ d \ m \ | \ d \geq \text{dur} \ m &= m \\
\text{dropM} \ d \ m \ | \ d \geq \text{dur} \ m &= \text{none} 0
\end{align*}
\]

**Theorem 4.3** For any non-negative \( d :: \text{Dur} \), if the following law holds for \( m :: T \) and \( \text{takeM}, \text{dropM} :: \text{Dur} \to T \to T \), then it also holds for well-formed \( m :: \text{Media} \ T \) and \( \text{takeM}, \text{dropM} :: \text{Dur} \to \text{Media} \ T \to \text{Media} \ T \):

\[
\begin{align*}
\text{dur} \ (\text{takeM} \ d \ m) &= \text{min} \ d \ (\text{dur} \ m) \\
\text{dur} \ (\text{dropM} \ d \ m) &= \text{dur} \ m \ominus d
\end{align*}
\]
Perhaps surprisingly, takeM and dropM share many properties analogous to their list counterparts, except that indexing is done in time, not in the number of elements. Four of these properties are captured in the following theorem:

**Theorem 4.4** For all non-negative $d_1, d_2 :: Dur$, if the following laws hold for $takeM, dropM :: Dur \rightarrow T \rightarrow T$, then they also hold for $takeM, dropM :: Dur \rightarrow Media \ T \rightarrow Media \ T$:

\[
\begin{align*}
\text{takeM} \ d_1 \circ \text{takeM} \ d_2 & = \text{takeM} \ (\min d_1, d_2) \\
\text{dropM} \ d_1 \circ \text{dropM} \ d_2 & = \text{dropM} \ (d_1 + d_2) \\
\text{takeM} \ d_1 \circ \text{dropM} \ d_2 & = \text{dropM} \ d_2 \circ \text{takeM} \ (d_1 + d_2) \\
\text{dropM} \ d_1 \circ \text{takeM} \ d_2 & = \text{takeM} \ (d_2 \ominus d_1) \circ \text{dropM} \ % \text{if } d_2 \geq d_1
\end{align*}
\]

These properties are the same as those for lists, except that the last equation is slightly more general in that it allows for the case that $d_1 > d_2$ [9, 2].

**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 4.5** Assume that $d \leq \text{dur m}$. If the following law holds for $m :: T$ and $takeM, dropM :: Dur \rightarrow T \rightarrow T$, then it also holds for well-formed $m :: Media \ T$ and $takeM, dropM :: Dur \rightarrow Media \ T \rightarrow Media \ T$:

\[
\begin{align*}
\text{takeM} \ d \ m & + : \text{dropM} \ d \ m = m 
\end{align*}
\]

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

\[
\begin{align*}
\text{takeM} \ d \ (\text{Prim} x) & + : \text{dropM} \ d \ (\text{Prim} x) \\
& = \text{Prim} \ (\text{takeM} \ d \ x) + : \text{Prim} \ (\text{dropM} \ d \ x) \\
& \neq \text{Prim} \ x
\end{align*}
\]

It is not even reasonable to state this as a constraint on the base media type, because it necessarily 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 4.6** Assume that $d \leq \text{dur m}$. If the following law holds for $m :: T$ and $takeM, dropM :: Dur \rightarrow T \rightarrow T$, then it also holds for well-formed $m :: Media \ T$ and $takeM, dropM :: Dur \rightarrow Media \ T \rightarrow Media \ T$:

\[
\begin{align*}
\text{dropM} \ d \ m & = \text{reverseM} \ (\text{takeM} \ (\text{dur m} - d) \ (\text{reverseM} \ m)) \\
\text{takeM} \ d \ m & = \text{reverseM} \ (\text{dropM} \ (\text{dur m} - d) \ (\text{reverseM} \ m))
\end{align*}
\]

**Example 4.3 (Music)** We declare Note to be an instance of Take:

\[
\begin{align*}
\text{instance Take Note where} \\
\text{takeM} \ d_1 \ (\text{Rest} \ d_2) & = \text{Rest} \ (\min d_1, d_2) \\
\text{takeM} \ d_1 \ (\text{Note} \ p \ d_2) & = \text{Note} \ p \ (\min d_1, d_2) \\
\text{dropM} \ d_1 \ (\text{Rest} \ d_2) & = \text{Rest} \ (d_2 \ominus d_1) \\
\text{dropM} \ d_1 \ (\text{Note} \ p \ d_2) & = \text{Note} \ p \ (d_2 \ominus d_1)
\end{align*}
\]
The constraints on Theorems 4.4 and 4.6 hold for this instance, and thus the theorems hold for Music values.

Example 4.4 (Animation) We declare Anim to be an instance of Take:

```haskell
instance Take Anim where
    takeM d_1 (NullAnim d_2) = NullAnim (min d_1 d_2)
    takeM d_1 (Anim d_2 f) = Anim (min d_1 d_2) f
    dropM d_1 (NullAnim d_2) = NullAnim (max (d_2 - d_1) 0)
    dropM d_1 (Anim d_2 f) = Anim (max (d_2 - d_1) 0) (f \circ (d_1+))
```

((d_1+)) is a Haskell section, which in this case is a function that adds d_1 to its argument; i.e. (d_1+) d_2 = d_1 + d_2.) The constraints on Theorems 4.4 and 4.6 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:

\[
m_1 :+ (m_2 :+ m_3) \\
(m_1 :+ m_2) :+ m_3
\]

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:

```haskell
class Combine b where
    concatM :: b \rightarrow b \rightarrow b
    merge :: b \rightarrow b \rightarrow b
    zero :: Dur \rightarrow b

class (Temporal a, Temporal b, Combine b) \Rightarrow Meaning a b where
    meaning :: a \rightarrow b

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

Intuitively speaking, an instance Meaning \(T_1\) \(T_2\) means that \(T_1\) can be given meaning in terms of \(T_2\). More specifically, the instance declaration above states that Media \(T_1\) can be given meaning in terms of \(T_2\), and expressed as a catamorphism, as long as we can give meaning to the base media type \(T_1\) in terms of \(T_2\).

As laws for the class Meaning, we require that:
Also, in anticipation of the axiomatic semantics that we develop in Section 7, we require that the following laws be valid for any instance of Combine:

\[
\begin{align*}
&b_1 \concat (b_2 \concat b_3) = (b_1 \concat b_2) \concat b_3 \\
&b_1 \merge (b_2 \merge b_3) = (b_1 \merge b_2) \merge b_3 \\
&b_1 \merge b_2 = (b_2 \merge b_1) \\
&\text{zero } 0 \concat b = b \\
&b \concat \text{zero } 0 = b \\
&\text{zero } d_1 \concat \text{zero } d_2 = \text{zero } (d_1 + d_2) \\
&\text{zero } d \merge b = b, \text{ if } d = \text{dur } b \\
&((b_1 \concat b_2) \merge (b_3 \concat b_4)) = (b_1 \merge b_3) \concat (b_2 \merge b_4), \\
&\text{ if } \text{dur } b_1 = \text{dur } b_3 \text{ and } \text{dur } b_2 = \text{dur } b_4
\end{align*}
\]

We then define a notion of equivalence:

**Definition 5.1** \( m_1, m_2 :: \text{Media } T \) are equivalent, written \( m_1 \equiv m_2 \), if and only if \( \text{meaning } m_1 = \text{meaning } m_2 \).

**Example 5.1 (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:

\[
\text{data Event} = \text{Event Time Pitch Dur} \\
\text{deriving (Show, Eq, Ord)} \\
\text{type Performance} = (\text{Dur}, [\text{Event}])
\]

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 [16]. The presence of the outermost duration in a Performance allows us to distinguish rests of unequal length; for example, \( \text{Prim (Rest } d_1 \text{)} \) and \( \text{Prim (Rest } d_2 \text{)} \), where \( d_1 \neq d_2 \). 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 \( m :+\text{ Prim (Rest } d_1 \text{)} \) and \( m :+\text{ Prim (Rest } d_2 \text{)} \).

Three instance declarations complete our interpretation of music:

\[
\begin{align*}
\text{instance Combine Performance where} \\
&\text{concat } (d_1, \text{evs}_1) (d_2, \text{evs}_2) = (d_1 + d_2, \text{evs}_1 \concat \text{evs}_2) \\
\text{where} &\text{shift } (\text{Event } t \text{ p } d) = \text{Event } (t + d_1) \text{ p } d \\
&\text{merge } (d_1, \text{evs}_1) (d_2, \text{evs}_2) = (d_1 \max d_2, \text{sort } (\text{evs}_1 \concat \text{evs}_2)) \\
&\text{zero } d = (d, [])
\end{align*}
\]

---

6After all, John Cage would not want his composition 4'33" to be confused with 4'32".
instance Temporal Performance where
dur (d, _) = d
none = zero
isNone = null \circ snd

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 5.2 (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])

picToImage :: Picture \rightarrow Image
combineImage :: Image \rightarrow Image \rightarrow Image
emptyImage :: Image

(Details of the Image operations are omitted.) This interpretation of animation is consistent with standard representations of videos and movies, whether digitized, on analog tape, or on film. Three instance declarations complete our interpretation of continuous animation:

instance Combine Rendering where
concatM (d1, i1) (d2, i2) = (d1 + d2, i1 \oplus i2)
merge (d1, i1) (d2, i2) = (d1 \max d2, zipWith' combineImage i1 i2)
zero d = (d, take (truncate (d \ast r)) (repeat emptyImage))

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

instance Meaning Anim Rendering where
meaning (NullAnim d) = (d, take (truncate (d \ast r)) (repeat emptyImage))
meaning (Anim d f) = (d, map (picToImage \circ f)
(take (truncate (d \ast r)) [0, 1 \div r ..]))

zipWith' is similar to Haskell’s zipWith, except that it does not truncate the result to the shorter of its two arguments:

zipWith' :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow [a] \rightarrow [a]
zipWith' f xs [] = xs
zipWith' f [] ys = ys
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys

17
Unfortunately, not all of the laws for classes \textit{Meaning} and \textit{Combine} hold for these instances. The problem stems from discretization. For example, suppose the frame rate $r = 10$. Then:

\[
    z_1 = \text{zero } 1.06 = (1.06, \text{take } 10 (\text{repeat emptyImage})) \\
    z_2 = \text{zero } 2.12 = (2.12, \text{take } 21 (\text{repeat emptyImage}))
\]

However, note that:

\[ z_1 \concatM z_1' = (2.12, \text{take } 20 (\text{repeat emptyImage})) \]

which is not the same as $z_2$. So the \textit{Combine} law:

\[
    \text{zero } d_1 \concatM \text{zero } d_2 = \text{zero } (d_1 + d_2)
\]

does not hold. This problem can be remedied by requiring that all \textit{Anim} durations be integral multiples of the frame period $1/r$. We say that such animations are \textit{integral}. With the additional assumption that the image operator $\text{combineImage}$ is commutative and associative, it is then straightforward to show that all of the laws for classes \textit{Combine} and \textit{Meaning} hold, and thus the above are valid instances for integral animations.

The reader might argue that it is unreasonable to assume that $\text{combineImage}$ is commutative. In particular, one might assume that $i_1 \text{combineImage} i_2$ is an image in which $i_1$ is overlaid on top of $i_2$ – indeed, parts of $i_2$ may be occluded – in which case $\text{combineImage}$ is clearly not commutative. But a commutative $\text{combineImage}$ is certainly feasible – for example, it could blend, or average, the two images pixel by pixel. A non-commutative $\text{combineImage}$ is considered in more detail in Section 7.3.

Finally, returning to the motivating example in this section, it is straightforward to show that the associativity of $(:+:)$:

\[
    m_1 :+: (m_2 :+: m_3) \equiv (m_1 :+: m_2) :+: m_3
\]

holds for both \textit{Music} and \textit{Animation} media. 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 equation until then.

\section{Algebraic Structure}

In the previous section we defined a standard interpretation of, or a semantics for, polymorphic temporal media, using the semantic function $\text{meaning} :: \text{Combine } b \Rightarrow \text{Media } a \rightarrow b$. 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.

\footnote{In Haskell, repeat $x$ is the infinite list of $x$’s.}
An algebraic structure (or just algebra) \( \langle S, op_1, op_2, \ldots \rangle \) consists of a non-empty carrier set (or sort) \( S \) together with one or more \( n \)-ary operations \( op_1, op_2, \ldots \), on that set [15]. We define an algebra of well-formed temporal media over type \( T \) as \( \langle Media \ T, ;+; ;=:; none \rangle \). 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 \( \langle I, concatM, merge, zero \rangle \) 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 \( \langle Media \ T, ;+; ;=:; none \rangle \) to \( \langle I, concatM, merge, zero \rangle \).

**Proof** We must show that:

\[
\begin{align*}
\text{meaning} (m_1 ;+; m_2) &= \text{meaning} m_1 'concatM' \text{meaning} m_2 \\
\text{meaning} (m_1 ;=:; m_2) &= \text{meaning} m_1 'merge' \text{meaning} m_2 \\
\text{meaning} (\text{none} d) &= \text{zero} d 
\end{align*}
\]

This is easily done by unfolding the definition of \( meaning \). For example, for the first equation above:

\[
\begin{align*}
\text{meaning} (m_1 ;+; m_2) &= \text{foldM} \text{meaning} concatM \text{merge} \text{foldM meaning concatM merge m_2} \\
&= \text{foldM meaning concatM merge m_2 'concatM'} \\
&= \text{meaning} m_2 'concatM' \text{meaning} m_2 \\
&= \text{meaning} (m_2 ;+; m_4) \\
\end{align*}
\]

The other two equations are proved analogously. □

**Theorem 6.2** \((\equiv)\) is a congruence relation on the algebra \( \langle Media \ T, ;+; ;=:; none \rangle \).

**Proof** We must show that, if \( m_1 \equiv m_2 \) and \( m_3 \equiv m_4 \), then:

\[
\begin{align*}
m_1 ;+; m_3 &\equiv m_2 ;+; m_4 \\
m_1 ;=:; m_3 &\equiv m_2 ;=:; m_4 
\end{align*}
\]

This is easily done by unfolding the definition of \( meaning \). For the first equation:

\[
\begin{align*}
\text{meaning} (m_1 ;+; m_3) &= \text{meaning} m_1 'concatM' \text{meaning} m_3 \\
&= \text{meaning} m_2 'concatM' \text{meaning} m_4 \\
&= \text{meaning} (m_2 ;+; m_4) \\
\end{align*}
\]

The other equation is proved analogously. □

**Definition 6.1** Let \([m]\) denote the equivalence class (induced by \((\equiv)\)) that contains \( m \). Let \( Media \ T / (\equiv) \) denote the quotient set of such equivalence classes over base media type \( T \), and let \( \langle Media \ T / (\equiv), ;+; ;=:; none \rangle \) denote the quotient algebra, also called the initial algebra. The function \( g :: Media \ T \rightarrow \)
The diagram in Figure 1 commutes.

Proof In the direction of $h$:

$$h (gm) = h [m] = \text{meaning } m$$

In the direction of $h^{-1}$:

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

□

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 $(\equiv)$ being an equivalence relation, and the substitution laws that arise from $(\equiv)$ being a congruence relation. We write $A \vdash m_1 = m_2$ iff $m_1 \equiv m_2$ can be established from the axioms of $A$.

7.1 Soundness

Theorem 7.1 (Soundness) The axiomatic semantics $A$ is sound. That is, for all well-formed $m_1, m_2 :: Media T$:

$$A \vdash m_1 = m_2 \Rightarrow m_1 \equiv m_2$$
For any well-formed \( m, m_1, m_2 :: Media\ T, \) and non-negative \( d :: Dur\):

1. \((+:)\) is associative: \( m_1 :+: (m_2 :+: m_3) \equiv (m_1 :+: m_2) :+: m_3\)
2. \((::=)\) is associative: \( m_1 ::= (m_2 ::= m_3) \equiv (m_1 ::= m_2) ::= m_3\)
3. \((::=)\) is commutative: \( m_1 ::= m_2 \equiv m_2 ::= m_1\)
4. \(\text{none } 0\) is a left (sequential) zero: \(\text{none } 0 :+: m \equiv m\)
5. \(\text{none } 0\) is a right (sequential) zero: \(m :+: \text{none } 0 \equiv m\)
6. \(\text{none } d\) is a left (parallel) unit: \(\text{none } d ::= m \equiv m\) if \(d = \text{dur } m\)
7. \(\text{none } d\) is additive: \(\text{none } d_1 :+: \text{none } d_2 \equiv \text{none } (d_1 + d_2)\)
8. serial/parallel inversion:
   \((m_1 :+: m_2) ::= (m_3 :+: m_4) \equiv (m_1 ::= m_3) :+: (m_2 ::= m_4),\)
   if \(\text{dur } m_1 = \text{dur } m_3\) and \(\text{dur } m_2 = \text{dur } m_4\)

Note: \(\text{none } d\) is also a right unit for \((::=)\), but is derivable from (3) and (6).

**Proof** Each of the axioms can be shown to be true by straightforward equational reasoning, using critically the laws of class Combine. For example, for Axiom 1:

\[
\text{meaning } (m_1 :+: (m_2 :+: m_3)) \\
= \text{meaning } m_1 '\text{concatM}' (\text{meaning } m_2 '\text{concatM}' \text{meaning } m_2) \quad \text{– unfold meaning} \\
= (\text{meaning } m_1 '\text{concatM}' \text{meaning } m_2)'\text{concatM} \text{meaning } m_2 \quad \text{– Combine law} \\
= \text{meaning } ((m_1 :+: m_2) :+: m_3) \quad \text{– fold meaning}
\]

The overall proof then follows by a simple induction over any derivation of equivalence between \(m_1\) and \(m_2\), and the transitivity of equivalence. □

As an example of a non-trivial theorem that can be proved using the axioms of polymorphic temporal media, recall Theorem 4.5 from Section 4, which we noted 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 \(d :: Dur \leq \text{dur } x\), if 
\(\text{takeM } d \ (\text{Prim } x) :+: \text{dropM } d \ (\text{Prim } x) \equiv \text{Prim } x\) then for all well-formed \(m :: Media\ T, \) \(\text{takeM } d \ m :+: \text{dropM } d \ m \equiv m.\)

**Proof** See Appendix A.2. □

**Example 7.1 (Music)** Theorem 7.2, which holds for lists, does not hold for Music, since, for example:

\[
\text{meaning } (\text{takeM } 1 \ (\text{Prim } \text{Note } p 2)) :+: \text{dropM } 1 \ (\text{Prim } \text{Note } p 2)) \\
= \text{meaning } (\text{Prim } \text{Note } p 1) :+: \text{Prim } \text{Note } p 1) \\
= (2, [\text{Event } 0 \ p 1, \text{Event } 1 \ p 1])
\]

which is not the same as:
Example 7.2 (Animation) Theorem 7.2 does hold for integral Animation, since, if \(d_2 > d_1\), then:

\[
\text{takeM } d_1 \ (\text{Prim } (\text{Anim } d_2 f)) :+= \text{dropM } d_1 \ (\text{Prim } (\text{Anim } d_2 f))
\]
\[
\equiv \text{Prim } (\text{Anim } d_1 f) :+= \text{Prim } (\text{Anim } (d_2 - d_1) \ (f \circ (d_1+)))
\]
\[
\equiv \text{Prim } (\text{Anim } d_2 f)
\]

A similar argument holds when \(d_1 > d_2\). 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 is usually more difficult to establish than soundness. 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 algebra \(\langle P, \text{concatM, merge} \rangle\) and the algebra \(\langle \text{Media } T / (\equiv), :+=, ::= \rangle\). What we need to do first is identify a canonical representation of each equivalence class in \(\text{Media } T / (\equiv)\).

Definition 7.2 A well-formed media term \(m::\text{Media } T\) is in normal form iff it is of the form:

\[
\begin{align*}
\text{(none } d_11 :+= \text{ Prim } x_1 :+= \text{ none } d_{12}) :== \\
\text{(none } d_{21} :+= \text{ Prim } x_2 :+= \text{ none } d_{22}) :== \\
\vdots \\
\text{(none } d_{n1} :+= \text{ Prim } x_n :+= \text{ none } d_{n2}) :== \\
\text{none } (\text{dur } m), \ n \geq 0, \\
\land \forall (1 \leq i \leq n), d_{i1} + d_{i2} + \text{dur } x_i = \text{dur } m, \\
\land \forall (1 \leq i < n), (d_{i1}, x_i, d_{i2}) \leq (d_{(i+1)1}, x_{(i+1)}, d_{(i+1)2})
\end{align*}
\]

To be completely rigorous, we assume that the operators (:+:) and (:=:) are right associative. Also, the last inequality above is defined lexicographically left-to-right. Note that it orders the media values in time, and by relying on an ordering of the base media type it also establishes an ordering on simultaneous media values.

We denote the set of media normal-forms over type \(T\) as \(\text{MediaNF } T\).

Example 7.3 Applying the normalize function to the ii-V-I chord progression given in Example 2.1 yields the following normal form:
normalize :: (Ord (Media a), Temporal a) ⇒ Media a → Media a

normalize m =
  let d = dur m
  norm t m | isNone m = m
  norm t (Prim x) = none t :+: (Prim x :+: none (d - t - dur x))
  norm t (m1 :+: m2) = norm (t + dur m1) m2
  norm t (m1 :+: m2) = norm t m1 :+: norm t m2
  in sortM (norm 0 m)

sortM m = sort m (none (dur m))
  where sort m acc | isNone m = acc
  sort (m1 :+: m2) acc = sort m1 (sort m2 acc)
  sort p@(m1 :+: (Prim x :+: m2)) acc = insert p acc

insert p m | isNone m = p :+: m
insert p (n :+: m) | p ≤ n = p :+: (n :+: m)
  otherwise = n :+: insert p m

Figure 3: Normalization Function

(Prim (Rest 0.0) :+: Prim (Note (D, 3) 1.0) :+: Prim (Rest 3.0)) ::=
(Prim (Rest 0.0) :+: Prim (Note (F, 3) 1.0) :+: Prim (Rest 3.0)) ::=
(Prim (Rest 0.0) :+: Prim (Note (A, 3) 1.0) :+: Prim (Rest 3.0)) ::=
(Prim (Rest 1.0) :+: Prim (Note (D, 4) 1.0) :+: Prim (Rest 2.0)) ::=
(Prim (Rest 1.0) :+: Prim (Note (G, 3) 1.0) :+: Prim (Rest 2.0)) ::=
(Prim (Rest 1.0) :+: Prim (Note (B, 3) 1.0) :+: Prim (Rest 2.0)) ::=
(Prim (Rest 2.0) :+: Prim (Note (C, 3) 2.0) :+: Prim (Rest 0.0)) ::=
(Prim (Rest 2.0) :+: Prim (Note (E, 3) 2.0) :+: Prim (Rest 0.0)) ::=
(Prim (Rest 2.0) :+: Prim (Note (G, 3) 2.0) :+: Prim (Rest 0.0)) ::=
(Prim (Rest 4.0))

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 it may not be true. For (b), we prove the following theorem:

**Theorem 7.3** The function `normalize` in Figure 3 converts any `m :: Media T`
into a media normal-form using only the axioms of A. In other words, for all
well-formed `m :: Media T, normalize m` is in normal form, and:

\[ A ⊢ normalize m = m \]

**Proof** Note in Figure 3 that 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 \((\text{none } d_1 :+ \text{Prim } x_1 :+ \text{none } d_2)\). So this is \textit{almost} in media normal form: what remains to be done is simply flatten and sort this structure, which is what \textit{sortM} does. The flattening is achieved using \textit{sortM}'s accumulator argument, and the sorting is achieved via the \textit{insert} function (this of course results in an \(O(n^2)\) algorithm, but efficiency is not our concern). Sort and insert utilize only the commutativity and associativity of \((:+)\).

We now must prove, using only the axioms of \(A\), that the normal form that \textit{norm} generates has the same meaning as the original term. That is, we must prove that \(A \vdash \text{norm } 0\ m = m\). We do so by relying on a more general result, that stated in Lemma 7.1 (see below). With that result we proceed straightforwardly as follows:

\[
\begin{align*}
\text{norm } 0\ m &\equiv \text{none } 0 :+ \text{none } (\text{dur } m - 0 - \text{dur } m) &\text{– Lemma 7.1} \\
&\equiv \text{none } 0 :+ \text{none } 0 &\text{– arithmetic} \\
&\equiv m &\text{– Axioms 4 and 5}
\end{align*}
\]

□

\textbf{Lemma 7.1} For all \(t::\text{Dur}\), well-formed \(m::\text{Media } T\), and \(d::\text{Dur} \geq t + \text{dur } m\):

\[
\text{norm } t\ m \equiv \text{none } t :+ \text{none } (d - t - \text{dur } m)
\]

\textbf{Proof} See Appendix A.3. □

We can now state our main result.

\textbf{Theorem 7.4 (Completeness)} The axiomatic semantics \(A\) is \textit{complete}, that is: for all \(m_1, m_2 :: \text{Media } T\):

\[
m_1 \equiv m_2 \Rightarrow A \vdash m_1 = m_2
\]

iff the normal forms in \textit{MediaNF }\(T\) are \textit{unique}, i.e. for all \(nf_1, nf_2 :: \text{MediaNF } T\):

\[
nf_1 \neq nf_2 \Rightarrow nf_1 \neq nf_2
\]

\textbf{Proof} Assume that the normal forms are unique. If \(m_1 \equiv m_2\), then \(p = \text{meaning } m_1 = \text{meaning } m_2\). Let \(n_1 = \text{normalize } m_1\) and \(n_2 = \text{normalize } m_2\). Then by Theorem 7.3, \(A \vdash n_1 = m_1\) and \(A \vdash n_2 = m_2\). Thus:

\[
\text{meaning } n_1 = \text{meaning } m_1 = p = \text{meaning } m_2 = \text{meaning } n_2
\]

But we know from Section 6 that there is an isomorphism between \textit{Media }\(T/(\equiv)\) and \(I\). Therefore \(p\) corresponds uniquely to some normal form, namely \(h^{-1}\ p\). This implies that \(n_1 = h^{-1}\ p = n_2\), and thus \(A \vdash m_1 = m_2\).

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 \(nf_1\) and \(nf_2\) whose meanings are the same; i.e. \(nf_1 \equiv nf_2\). If exactly one of these is of the form \(\text{none } d\), then by inspection of the axioms it is clear that none of them can establish \(\text{none } d\)'s equivalence to some other term, therefore the axioms must not be complete. This contradicts
our assumption, so the normal forms must be unique. On the other hand, if
$nf_1$ and $nf_2$ are each of the form $none d_1 :+ Prim x :+ none d_2$, 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 $nf_1$ and $nf_2$ are in fact equivalent, the
axioms are not complete. But that contradicts our assumption, so the normal
forms must be unique. $\square$

Theorem 7.4 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 – $MediaNF$ is uniquely tied to
completeness.

Example 7.4 (Music) The normal forms for well-formed $Music$, i.e. $MediaNF$
$Note$, 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 in Definition 7.2 corresponds to the perform-
   ance:

   $\langle dur m, \{Event d_{11} p_1 (dur m - d_{11} - d_{12}),$
   $Event d_{21} p_2 (dur m - d_{21} - d_{22}), ...$
   $Event d_{n1} p_n (dur m - d_{n1} - d_{n2})\} \rangle$

Conversely, any performance of the form:

   $\langle d, \{Event t_1 p_1 d_1,$
   $Event t_2 p_2 d_2, ...$
   $Event t_n p_n d_n\} \rangle$

corresponds to the normal form:

   $\langle none t_1 :+ Prim (Note p_1 d_1) :+ none (d - t_1 - d_1) \rangle$ :=:
   $\langle none t_2 :+ Prim (Note p_2 d_2) :+ none (d - t_2 - d_2) \rangle$ :=:
   $\ldots$
   $\langle none t_n :+ Prim (Note p_n d_n) :+ none (d - t_n - d_n) \rangle$ :=:
   $none d$

This correspondence is invertible because each of the three durations is
computable from the other two.

It is worth pointing out that in our definition of $Music$, notes having different
pitches are distinguishable, even if they have zero duration. It might seem
reasonable to insist that any notes with zero duration are indistinguishable,
regardless of the pitch. We could accomplish this by changing the meaning
function defined in Section 5 for $Note$ to:
instance Meaning Note Performance where
\[
\begin{align*}
\text{meaning (Rest } d\text{)} &= (d, []) \\
\text{meaning (Note } p \ 0\text{)} &= (0, []) \\
\text{meaning (Note } p \ d\text{)} &= (d, [\text{Event } 0 \ p \ d])
\end{align*}
\]
(Note the addition of the second line to the definition.) This would do no harm to our soundness result, but it would render our axiomatic semantics incomplete, since there would be more equalities than we are able to prove with the axioms.

We could regain completeness, however, in one of two ways:

- Without loss of generality, we could redefine the notion of well-formed media to exclude primitive values whose duration is zero, since the same effect can be achieved by \texttt{none} 0.
- We could add an axiom that identified zero-duration values with \texttt{none} 0.

Example 7.5 (Animation) The normal forms of the integral Animation media type are not unique. There are two problems, one fundamental and unavoidable, the other depending on the details of the primitive image operators, which so far we have left undefined.

The fundamental problem is that the meaning function for animations is “lossy” – that is, the meaning of an abstract animation involves sampling the animation function at discrete points in time, thereby throwing away all of the information at other points in time. Thus it is possible for two distinct functions to look identical when sampled, but there are no axioms to equate the original functions. In other words, there will be pairs of normal forms that are semantically equivalent. This does not affect soundness, of course, but does affect completeness, and there is little we can do to solve the problem, short of designing a non-lossy meaning function.

The second problem stems from the fact that we have said little about the primitive image operators such as \texttt{combineImage}, \texttt{picToImage}, and \texttt{emptyImage}. In particular:

- It is reasonable to assume that there exist images \( i_1, i_2, \) and \( i_3 \), such that \( i_3 = \text{combineImage } i_1 i_2 \). This means that there will be animations \( a_1, a_2, \) and \( a_3 \) such that \( a_3 \equiv a_1 :=: a_2 \). In other words, once again, there will be pairs of normal forms that are semantically equivalent. To remedy this we would have to add axioms that somehow captured all of these equivalences.

- Similarly, as is revealed in Theorem 7.2, there are pairs of primitive animations that, when combined sequentially, are indistinguishable from a single primitive animation. The only way around this dilemma is to add Theorem 7.2 (or something like it) as an axiom.

[Contrast this with Music media: Playing two notes simultaneously is not the same as playing any other single note. This is a fact not only about our representation of music, but also (usually) in practice – for example, two notes played sequentially on a piano are not the same as any one note.]
• Finally, in order to know if the normal forms of the Animation media type are unique, we need to know more about the function picToImage. In particular, a circle of radius zero and a square of length zero should probably be rendered in the same way – namely, as an emptyImage – in which case they are indistinguishable. Yet we have no axiom to allow us to conclude that fact. This problem can be fixed in a way similar to the problem with zero-duration notes (see previous example): either disallow circles and squares of size zero, or provide an axiom to equate them.

These problems reveal the subtlety and fragility of completeness, even in a “non-lossy” framework. By adding constraints or new axioms to capture the extra equivalences we can restore completeness, but at the expense of having a more complex axiom set.

7.3 A Non-Commutative Semantics

In Example 5.2 we discussed the possibility of combineImage being non-commutative, thus rendering merge, and in turn (=: :), non-commutative, and therefore invalidating Axiom 3. This raises the question of whether it is possible to devise a non-commutative semantics that is both sound and complete. It is easy to see that losing commutativity does not affect soundness, as long as the commutative axiom is dropped. In addition, we would probably want to add an axiom to capture the fact that done d is a right unit for (=: :) (previously derived from Axiom 3). The resulting semantics is sound, but is it complete?

In the case of animation, note that there would 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 conceivable to include additional axioms to cover these special cases and thus restore completeness, but the proof cannot rely exclusively on the uniqueness of the normal forms as previously defined.

More generally, to further complicate matters and to highlight the centrality of normal forms, note that to say that the commutative axiom is invalid does not imply that it is never the case that \( m_1 :=: m_2 \equiv m_2 :=: m_1 \). For a simple example, using Axiom 6, none 0 :=: m \equiv m :=: none 0. As a more subtle example, if dur x \equiv dur y \equiv d, then using Axioms 6 and 8 we have:

\[
(\text{Prim } x \oplus: \text{None } d) :=: (\text{None } d \oplus: \text{Prim } y) \\
\equiv (\text{Prim } x :=: \text{None } d) \oplus: (\text{None } d :=: \text{Prim } y) \\
\equiv \text{Prim } x \oplus: \text{Prim } y \\
\equiv (\text{None } d :=: \text{Prim } x) \oplus: (\text{Prim } y :=: \text{None } d) \\
\equiv (\text{None } d :=: \text{Prim } y) :=: (\text{Prim } x :=: \text{None } d)
\]

We conjecture that the resulting non-commutative axiomatic semantics is in fact complete, but it is not easy to see how to design a normal form. For example, what should the normal form be for the last example above?
8 Related Work

There has been considerable work on embedding semantic descriptions in multimedia (XML, UML, the Semantic Web, etc.), but not on formalizing the semantics of concrete media. There are also many authoring tools and scripting languages for designing multimedia applications. The one closest to a programming language is probably SMIL [17], which can be seen as treating multimedia in a polymorphic way. Our own work on Haskore and MDL [11, 8, 9, 10] is of course related, but specialized to music. Our animations are similar in spirit to those in Fran [4, 3], except that they are not reactive. Graham Hutton shows how fold and unfold can be used to describe denotational and operational semantics, respectively [13], and thus our use of fold to describe the semantics of temporal media is an instance of his framework.

Acknowledgements The NSF provided partial support for this research under grant numbers CCR-9900957 and CCR-0205542. Also thanks to the many students who worked with me on Haskore, which was the original inspiration for the current work.

A Proofs

A.1 Proof of Theorem 4.4

For the first equation, beginning with the base cases:

If $d_2 = 0$:
\[
\begin{align*}
takeM d_1 (takeM d_2 m) &= \text{none } 0 & \text{– unfold } takeM \\
\text{none } 0 &= \text{none } 0 & \text{– theorem 4.2} \\
\text{none } 0 &= \text{none } 0 & \text{– fold } takeM \\
\end{align*}
\]

If $d_1 = 0$:
\[
\begin{align*}
takeM d_1 (takeM d_2 m) &= \text{none } 0 & \text{– unfold } takeM \\
\text{none } 0 &= \text{none } 0 & \text{– fold } takeM \\
\text{none } 0 &= \text{none } 0 & \text{– fold } min \\
\end{align*}
\]

\[
\begin{align*}
takeM d_1 (takeM d_2 (Prim x)) &= takeM d_1 (Prim (takeM d_2 x)) & \text{– unfold } takeM \\
Prim (takeM d_1 (takeM d_2 x)) &= Prim (takeM d_1 (takeM d_2 x)) & \text{– unfold } takeM \\
Prim (takeM (min d_1 d_2) x) &= Prim (takeM (min d_1 d_2) x) & \text{– assumption} \\
\end{align*}
\]

Induction steps:
\textit{takeM} \ d_1 \ (\textit{takeM} \ d_2 \ (m_1 ++: m_2))

If \( d_2 \leq \text{dur} \ m_1 \):
\[
= \text{takeM} \ d_1 \ (\textit{takeM} \ d_2 \ m_1) \quad \text{– unfold \textit{takeM}}
\]
\[
= \text{takeM} \ (\min \ d_1 \ d_2) \ m_1 \quad \text{– ind. hyp.}
\]
\[
= \text{takeM} \ (\min \ d_1 \ d_2) \ (m_1 ++: m_2) \quad \text{– fold \textit{takeM}}
\]
If \( d_2 > \text{dur} \ m_1 \):
\[
= \text{takeM} \ d_1 \ (m_1 ++: \text{takeM} \ (d_2 - \text{dur} \ m_1) \ m_2) \quad \text{– unfold \textit{takeM}}
\]
If \( d_1 \leq \text{dur} \ m_1 \):
\[
= \text{takeM} \ d_1 \ m_1 \quad \text{– unfold \textit{takeM}}
\]
\[
= \text{takeM} \ (\min \ d_1 \ d_2) \ m_1 \quad \text{– fold \textit{min}}
\]
\[
= \text{takeM} \ (\min \ d_1 \ d_2) \ (m_1 ++: m_2) \quad \text{– fold \textit{takeM}}
\]
If \( d_1 > \text{dur} \ m_1 \):
\[
= m_1 ++: \text{takeM} \ (d_1 - \text{dur} \ m_1) \ (\text{takeM} \ (d_2 - \text{dur} \ m_1) \ m_2) \quad \text{– unfold \textit{takeM}}
\]
\[
= m_1 ++: \text{takeM} \ (\min \ (d_1 - \text{dur} \ m_1) \ (d_2 - \text{dur} \ m_1)) \ m_2 \quad \text{– ind. hyp.}
\]
\[
= \text{takeM} \ (\min \ d_1 \ d_2 - \text{dur} \ m_1) \ m_2 \quad \text{– arithmetic}
\]
\[
= \text{takeM} \ (\min \ d_1 \ d_2) \ (m_1 ++: m_2) \quad \text{– fold \textit{takeM}}
\]

For equation (2):

Base cases:

If \( d_2 = 0 \):
\[
dropM \ d_1 \ (\text{dropM} \ d_2 \ m)
\]
\[
= \text{dropM} \ d_1 \ m \quad \text{– unfold \textit{dropM}}
\]
\[
= \text{dropM} \ (d_1 + d_2) \ m \quad \text{– arithmetic}
\]

If \( d_1 = 0 \):
\[
dropM \ d_1 \ (\text{dropM} \ d_2 \ m)
\]
\[
= \text{dropM} \ d_2 \ m \quad \text{– unfold \textit{dropM}}
\]
\[
= \text{dropM} \ (d_1 + d_2) \ m \quad \text{– arithmetic}
\]

\[
dropM \ d_1 \ (\text{dropM} \ d_2 \ (\text{Prim} \ x))
\]
\[
= \text{dropM} \ d_1 \ (\text{Prim} \ (\text{dropM} \ d_2 \ x)) \quad \text{– unfold \textit{dropM}}
\]
\[
= \text{Prim} \ (\text{dropM} \ d_1 \ (\text{dropM} \ d_2 \ x)) \quad \text{– unfold \textit{dropM}}
\]
\[
= \text{Prim} \ (\text{dropM} \ (d_1 + d_2) \ x) \quad \text{– assumption}
\]
\[
= \text{dropM} \ (d_1 + d_2) \ (\text{Prim} \ x) \quad \text{– fold \textit{dropM}}
\]

Induction steps:

\[
dropM \ d_1 \ (\text{dropM} \ d_2 \ (m_1 ++: m_2))
\]
If \( d_2 \leq \text{dur} \ m_1 \):
\[
= \text{dropM} \ d_1 \ (\text{dropM} \ d_2 \ m_1 ++: m_2) \quad \text{– unfold \textit{dropM}}
\]
If \( d_1 \leq \text{dur} (\text{drop}_M d_2 m_1) \):
\[
= \text{drop}_M d_1 (\text{drop}_M d_2 m_1) :+ m_2 \quad \text{-- unfold \text{drop}_M}
\]
\[
= \text{drop}_M (d_1 + d_2) m_1 :+ m_2 \quad \text{-- ind. hyp.}
\]
\[
= \text{drop}_M (d_1 + d_2) (m_1 :+ m_2) \quad \text{-- fold \text{drop}_M *}
\]
* because:
\[
d_1 \leq \text{dur} (\text{drop}_M d_2 m_1) \quad \text{-- Theorem 4.3}
\]
\[
\Rightarrow d_1 \leq \text{dur} m_1 - d_2 \quad \text{-- arithmetic}
\]
\[
\Rightarrow d_1 + d_2 \leq \text{dur} m_1 \quad \text{-- arithmetic}
\]
If \( d_1 > \text{dur} (\text{drop}_M d_2 m_1) \):
\[
= \text{drop}_M (d_1 - \text{dur} (\text{drop}_M d_2 m_1)) m_2 \quad \text{-- Theorem 4.3}
\]
\[
= \text{drop}_M (d_1 + d_2 - \text{dur} m_1) m_2 \quad \text{-- arithmetic}
\]
\[
= \text{drop}_M (d_1 + d_2) (m_1 :+ m_2) \quad \text{-- fold \text{drop}_M *}
\]
* because:
\[
d_1 > \text{dur} (\text{drop}_M d_2 m_1) \quad \text{-- Theorem 4.3}
\]
\[
\Rightarrow d_1 > \text{dur} m_1 - d_2 \quad \text{-- arithmetic}
\]
\[
\Rightarrow d_1 + d_2 > \text{dur} m_1 \quad \text{-- arithmetic}
\]
If \( d_2 > \text{dur} m_1 \):
\[
= \text{drop}_M d_1 (\text{drop}_M (d_2 - \text{dur} m_1) m_2) \quad \text{-- unfold \text{drop}_M}
\]
\[
= \text{drop}_M (d_1 + d_2 - \text{dur} m_1) m_2 \quad \text{-- ind. hyp.}
\]
\[
= \text{drop}_M (d_1 + d_2) (m_1 :+ m_2) \quad \text{-- fold \text{fold}_M}
\]
\[
\text{drop}_M d_1 (\text{drop}_M d_2 (m_1 := m_2))
\]
\[
= \text{drop}_M d_1 (\text{drop}_M d_2 m_1 := \text{drop}_M d_2 m_2) \quad \text{-- unfold \text{drop}_M}
\]
\[
= \text{drop}_M d_1 (\text{drop}_M d_2 m_1 := \text{drop}_M d_1 (\text{drop}_M d_2 m_2)) \quad \text{-- unfold \text{drop}_M}
\]
\[
= \text{drop}_M (d_1 + d_2) m_1 := \text{drop}_M (d_1 + d_2) m_2 \quad \text{-- unfold \text{drop}_M}
\]
\[
= \text{drop}_M (d_1 + d_2) (m_1 := m_2) \quad \text{-- fold \text{drop}_M}
\]

For equation (3):

Base cases:
If \( d_2 = 0 \):
\[
\text{take}_M d_1 (\text{drop}_M d_2 m)
\]
\[
= \text{take}_M d_1 m \quad \text{-- unfold \text{take}_M}
\]
\[
= \text{take}_M (d_1 + d_2) m \quad \text{-- arithmetic}
\]
\[
= \text{drop}_M d_2 (\text{take}_M (d_1 + d_2) m) \quad \text{-- fold \text{drop}_M}
\]

If \( d_1 = 0 \):
\[
\text{take}_M d_1 (\text{drop}_M d_2 m)
\]
\[
= \text{none} 0 \quad \text{-- unfold \text{take}_M}
\]
\[
= \text{drop}_M d_2 (\text{take}_M d_2 m) \quad \text{-- Theorems 4.2 and 4.3}
\]
\[
= \text{drop}_M d_2 (\text{take}_M (d_1 + d_2) m) \quad \text{-- arithmetic}
\]
\[
\text{take}_M d_1 (\text{drop}_M d_2 (\text{Prim } x))
\]
\[
= \text{take}_M d_1 (\text{Prim (drop}_M d_2 x)) \quad \text{-- unfold \text{drop}_M}
\]
\[
= \text{Prim (take}_M d_1 (\text{drop}_M d_2 x)) \quad \text{-- unfold \text{take}_M}
\]
= \text{Prim} \left( \text{dropM}\ d_2 \ (\text{takeM} \ (d_1 + d_2) \ x) \right) \quad \text{-- assumption}
= \text{dropM} \ d_2 \ (\text{Prim} \ (\text{takeM} \ (d_1 + d_2) \ x)) \quad \text{-- unfold} \ \text{dropM}
= \text{dropM} \ d_2 \ (\text{takeM} \ (d_1 + d_2) \ (\text{Prim} \ x)) \quad \text{-- unfold} \ \text{takeM}

\text{1st induction step:}
\text{takeM} \ d_1 \ (\text{dropM} \ d_2 \ (m_1 :+: m_2))
\text{If } d_2 \leq \text{dur} \ m_1 = \text{dm1}:
\quad = \text{takeM} \ d_1 \ (\text{dropM} \ d_2 \ m_1 :+: m_2) \quad \text{-- unfold} \ \text{dropM}
\text{If } d_1 \leq \text{dur} \ (\text{dropM} \ d_2 \ m_1) = \text{dm1} \ominus d_2 = d':
\quad = \text{takeM} \ d_1 \ (\text{dropM} \ d_2 \ m_1) \quad \text{-- unfold} \ \text{takeM}
\quad = \text{dropM} \ d_2 \ (\text{takeM} \ (d_1 + d_2) \ m_1) \quad \text{-- induction hypothesis}
\quad = \text{dropM} \ d_2 \ (\text{takeM} \ (d_1 + d_2) \ (m_1 :+: m_2)) \quad \text{-- unfold} \ \text{takeM}, \ b/c \ d_1 + d_2 \leq \text{dm1}
\quad \text{Else } d_1 > d':
\quad = \text{dropM} \ d_2 \ m_1 :+: \text{takeM} \ (d_1 - d') \ m_2 \quad \text{-- unfold} \ \text{takeM}
\quad = \text{dropM} \ d_2 \ (m_1 :+: \text{takeM} \ (d_1 - d') \ m_2) \quad \text{-- fold} \ \text{dropM}
\quad = \text{dropM} \ d_2 \ (\text{takeM} \ (d_1 - d' + \text{dm1}) \ (m_1 :+: m_2)) \quad \text{-- fold} \ \text{takeM}
\quad = \text{dropM} \ d_2 \ (\text{takeM} \ (d_1 + d_2) \ (m_1 :+: m_2)) \quad \text{-- fold} \ \text{takeM}
\quad \because d_1 - d' + \text{dm1} = d_1 - (\text{dm1} \ominus d_2) + \text{dm1} = d_1 + d_2
\text{Else } d_2 > \text{dm1}:
\quad = \text{takeM} \ d_1 \ (\text{dropM} \ (d_2 - \text{dm1}) \ m_2) \quad \text{-- unfold} \ \text{dropM}
\quad = \text{dropM} \ (d_2 - \text{dm1}) \ (\text{takeM} \ (d_1 + d_2 - \text{dm1}) \ m_2) \quad \text{-- induction hypothesis}
\quad = \text{dropM} \ d_2 \ (m_1 :+: \text{takeM} \ (d_1 + d_2 - \text{dm1}) \ m_2) \quad \text{-- fold} \ \text{dropM}
\quad = \text{dropM} \ d_2 \ (\text{takeM} \ (d_1 + d_2) \ (m_1 :+: m_2)) \quad \text{-- fold} \ \text{takeM}
\quad \because d_2 - \text{dm1} + d_1 + d_2 = d_2

\text{2nd induction step:}
\text{takeM} \ d_1 \ (\text{dropM} \ d_2 \ (m_1 :+: m_2))
\quad = \text{takeM} \ d_1 \ (\text{dropM} \ d_2 \ m_1 :+: \text{dropM} \ d_2 \ m_2) \quad \text{-- unfold} \ \text{dropM}
\quad = \text{takeM} \ d_1 \ (\text{dropM} \ d_2 \ m_1 :+: \text{takeM} \ d_3 \ (\text{dropM} \ d_2 \ m_2)) \quad \text{-- unfold} \ \text{takeM}
\quad = \text{dropM} \ d_2 \ (\text{take} \ (d_1 + d_2) \ m_1) \quad \text{-- induction hypothesis (twice)}
\quad = \text{dropM} \ d_2 \ (\text{take} \ (d_1 + d_2) \ m_1 :+: \text{take} \ (d_1 + d_2) \ m_2) \quad \text{-- fold} \ \text{dropM}
\quad = \text{dropM} \ d_2 \ (\text{take} \ (d_1 + d_2) \ m_1 :+: m_2) \quad \text{-- fold} \ \text{takeM}

\text{For equation (4):}
\text{Base cases:}
\text{If } d_2 = 0:
\text{dropM} \ d_1 \ (\text{takeM} \ d_2 \ m)
\quad = \text{dropM} \ d_1 \ (\text{none}) \quad \text{-- unfold} \ \text{takeM}
\quad = \text{none} \ 0 \quad \text{-- Theorem 4.2}
\quad = \text{takeM} \ 0 \ (\text{dropM} \ d_1 \ m) \quad \text{-- fold} \ \text{takeM}
\quad = \text{takeM} \ (d_2 \ominus d_1) \ (\text{dropM} \ d_3 \ m) \quad \text{-- arithmetic}
\text{If } d_1 = 0:
\text{dropM} \ d_1 \ (\text{takeM} \ d_2 \ m)
\quad = \text{takeM} \ d_2 \ m \quad \text{-- unfold} \ \text{dropM}
1st induction step:
\[ \text{dropM } d_1 \text{ (takeM } d_2 \text{ (Prim } x)) \\]

- **If** \( d_2 \leq \text{dur} m_1 = \text{dm1} \):
  \[ = \text{dropM } d_1 \text{ (takeM } d_2 \text{ m_1}) \]
- **Else** \( d_2 > d_1 \):
  \[ = \text{takeM } (d_2 \oplus d_1) \text{ (dropM } d_1 \text{ m_1}) \]

\[ \text{dropM } d_1 \text{ (takeM } d_2 \text{ (Prim } x)) \]

2nd induction step:
\[ \text{dropM } d_1 \text{ (takeM } d_2 \text{ m_1 :=: m_2)) \]

- **If** \( d_1 \leq \text{dm1} \):
  \[ = \text{dropM } d_1 \text{ m_1 :=: takeM } d_2 \text{ (m_1 :=: m_2) } \]
  \[ = \text{takeM } (d_2 \oplus d_1) \text{ (dropM } d_1 \text{ (m_1 :=: m_2))} \]
  \[ = \text{fold dropM} \]
- **Else** \( d_1 > \text{dm1} \):
  \[ = \text{takeM } (d_2 \oplus d_1) \text{ (dropM } d_1 \text{ m_1 :=: m_2) } \]
  \[ = \text{arith} \]

- **If** \( d_1 \leq \text{dm1} \):
  \[ = \text{dropM } d_1 \text{ m_1 :=: takeM } d_2 \text{ (m_1 :=: m_2) } \]
  \[ = \text{fold dropM} \]
- **Else** \( d_1 > \text{dm1} \):
  \[ = \text{takeM } (d_2 \oplus d_1) \text{ (dropM } d_1 \text{ m_1 :=: m_2) } \]
  \[ = \text{arith} \]

- **If** \( d_1 < \text{dur} m_1 = \text{dm1} \):
  \[ = \text{dropM } d_1 \text{ m_1 :=: takeM } d_2 \text{ (m_1 :=: m_2) } \]
  \[ = \text{fold dropM} \]
- **Else** \( d_1 > \text{dm1} \):
  \[ = \text{takeM } (d_2 \oplus d_1) \text{ (dropM } d_1 \text{ m_1 :=: m_2) } \]
  \[ = \text{arith} \]

- **If** \( d_1 < \text{dur} m_1 = \text{dm1} \):
  \[ = \text{dropM } d_1 \text{ m_1 :=: takeM } d_2 \text{ (m_1 :=: m_2) } \]
  \[ = \text{fold dropM} \]
- **Else** \( d_1 > \text{dm1} \):
  \[ = \text{takeM } (d_2 \oplus d_1) \text{ (dropM } d_1 \text{ m_1 :=: m_2) } \]
  \[ = \text{arith} \]

- **If** \( d_1 < \text{dur} m_1 = \text{dm1} \):
  \[ = \text{dropM } d_1 \text{ m_1 :=: takeM } d_2 \text{ (m_1 :=: m_2) } \]
  \[ = \text{fold dropM} \]
- **Else** \( d_1 > \text{dm1} \):
  \[ = \text{takeM } (d_2 \oplus d_1) \text{ (dropM } d_1 \text{ m_1 :=: m_2) } \]
  \[ = \text{arith} \]
A.2 Proof of Theorem 7.2

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

\[
\text{takeM } d \ (m_1 \vdash m_2) \vdash \text{dropM } d \ (m_1 \vdash m_2)
\]

If \( d \leq \text{dur} \ m_1 \):

\[
= \text{takeM } d \ m_1 \vdash (\text{dropM } d \ m_1 \vdash m_2)
\]

- unfold \text{takeM} and \text{dropM}

\[
= (\text{takeM } d \ m_1 \vdash \text{dropM } d \ m_1) \vdash m_2
\]

- associativity axiom

\[
= m_1 \vdash m_2
\]

- ind. hyp.

If \( d > \text{dur} \ m_1 \):

\[
= (m_1 \vdash \text{takeM } (d - \text{dur} \ m_1) \ m_2) \\
= m_1 \vdash (\text{takeM } (d - \text{dur} \ m_1) \ m_2) \vdash \text{dropM } (d - \text{dur} \ m_1) \ m_2
\]

- unfold \text{takeM} and \text{dropM}

\[
= m_1 \vdash (\text{takeM } (d - \text{dur} \ m_1) \ m_2) \vdash \text{dropM } (d - \text{dur} \ m_1) \ m_2
\]

- associativity axiom

\[
= m_1 \vdash m_2
\]

- ind. hyp.

For the second induction step:

\[
\text{takeM } d \ (m_1 \vdash m_2) \vdash \text{dropM } d \ (m_1 \vdash m_2)
\]

\[
= (\text{takeM } d \ m_1 \vdash \text{takeM } d \ m_2) \vdash (\text{dropM } d \ m_1 \vdash \text{dropM } d \ m_2)
\]

- unfold \text{takeM} and \text{dropM}

\[
= (\text{takeM } d \ m_1 \vdash \text{dropM } d \ m_1) \vdash (\text{takeM } d \ m_2 \vdash \text{dropM } d \ m_2)
\]

- serial/parallel axiom

\[
= m_1 \vdash m_2
\]

- ind. hyp.

A.3 Proof of Lemma 7.1

Base cases:

\[
\text{norm } d \ t \ (\text{none } d')
= \text{none } d
= \text{none } t \vdash \text{none } d' \vdash \text{none } (d - t - d')
\]

\[
\text{norm } d \ t \ (\text{Prim } x)
= \text{none } t \vdash \text{Prim } x \vdash \text{none } (d - t - \text{dur } x)
= \text{none } t \vdash \text{Prim } x \vdash \text{none } (d - t - \text{dur } (\text{Prim } x))
\]

For the first induction step, let \( d_1 = \text{dur} \ m_1, d_2 = \text{dur} \ m_2, \text{and } d_{12} = \text{dur} (m_1 \vdash m_2). \) Then:

\[
\text{norm } d \ t \ (m_1 \vdash m_2)
= \text{norm } d \ t \ m_1 \vdash \text{norm } d \ t \ m_2
\]
\[
\equiv ( \text{none } t \mapsto m_1 \mapsto \text{none } (d - t - d_1)) \mapsto (\text{none } t \mapsto m_2 \mapsto \text{none } (d - t - d_2))
\]
\[
\equiv (\text{none } t \mapsto \text{none } t) \mapsto (m_1 := m_2) \mapsto (\text{none } (d - t - d_1)) := (\text{none } (d - t - d_2))
\]
\[
\equiv \text{none } t \mapsto (m_1 := m_2) \mapsto (\text{none } (d - t - d_1)) := (\text{none } (d - t - d_2))
\]
\[
\equiv \text{none } t \mapsto (m_1 := m_2) \mapsto \text{none } (d - t - d_1)
\]

For the second induction step, let \( d_1 = \text{dur } m_1 \), \( d_2 = \text{dur } m_2 \), and \( d_{12} = \text{dur } (m_1 \mapsto m_2) \). Then:

\[
\text{norm } d \ t (m_1 \mapsto m_2) = \text{norm } d \ t m_1 \mapsto \text{norm } d \ (t + d_1) m_2
\]
\[
= (\text{none } t \mapsto m_1 \mapsto \text{none } (d - t - d_1))
\]
\[
= (\text{none } t \mapsto \text{none } d_1 \mapsto m_2 \mapsto \text{none } (d - t - d_1 - d_2))
\]
\[
= (\text{none } t \mapsto \text{none } t \mapsto (m_1 := \text{none } d_1))
\]
\[
\equiv (\text{none } t \mapsto (m_1 \mapsto m_2) \mapsto (m_2 \mapsto \text{none } (d - t - d_1 - d_2))
\]
\[
\equiv \text{none } t \mapsto (m_1 := m_2) \mapsto \text{none } (d - t - d_{12})
\]

References


