An ArrowLoop of Music

Hans Höglund

October 16, 2011

We want to have a generalized musical representation. There are many "standard" formats, but none of these are really generalized. In fact, they are more or less derived from traditional musical notation.

Typical representation such as MusicXML, GUIDO or ABC notation has these flaws:

- Assumptions on pitch names, tuning and rhytm
- Assumptions on event types, treating notes as the common case, ignoring continous values.
- Assumptions synchronicity, making it difficult to represent "free" time.

It would be desirable to define a model that does not suffer from these shortcomings, yet still is able to represent everything the above models can represent.

We want the traditional models emerge as a special case. We also want the auditory model to emerge as a special case.

Strategy

- Separate time semantics from other semantics
- Support discrete and continous values
- Support synchronous and asynchronous time

Relation to FRP

Functional Reactive Programming (FRP) is based on the notion that time-varying values can be described in a purely functional language, as long as the internal state of each particular value is not exposed.

Classical FRP systems are based on the notion of *Signals* (also called *Behaviours*), which are time-varying values with are semantically equivalent to functions in the time domain (we use `::=` to denote meaning).

```
type Behaviour a ::= Time -> a
```

The main trick here is that the `Time -> a` part of the definition is not actually exposed. Instead, behaviours are manipulated using combinators such as:

```
addS :: Signal Double -> Signal Float -> Signal Float
cosS :: Signal Double -> Signal Float -> Signal Float
```

As demonstrated by Elliot, such signals are semantically well behaved with respect to the `Functors` and `Applicative` type classes. This captures the intuitions of static functions on varying values and varying functions on varying values respectively.

```
mapS   :: (a -> b) -> Signal a -> Signal b
constS :: a -> Signal a
applyS :: Signal (a -> b) -> Signal a -> Signal b
```

Besides signals, FRP also defines a type called *Event*, which is taken to mean an ordered list of states.

```
type Event a ::= [(Time, a)]
```

This choice of name is probably confusing to computer musicians, who would use the word event to refer to a single occurance, not to a list of occurances (which would be an event list). The reason is that FRP are always used as streams, so that the intuition of the "current state" correspond to the head of the stream, and "future states" to its tail.

The main problem with FRP events is that they do not lend themself to parallel composition.

```
parallelE :: Event a -> Event a -> Event a
-- Not possible until one of the two events has occured
```

Arrowized FRP (Yampa, Animas, Euterpia) take advantage of the `Arrow` type class to model temporal values. Just as in classical FRP, the semantics of behaviours are functions of time, but rather than representing signals directly, AFRP use arrows to represent functions on signals. This turn out to generalize the functor and applicative operations.

```
type SF a b ::= (Time -> a) -> (Time -> b)
addS :: SF Double Double
cosS :: SF Double Double

mapS   :: (a -> b) -> SF a b
constS :: a -> SF b a
applyS :: SF (SF b c, b) c
```

The main idea of this paper is to employ the arrow representation not only for signals, but also for events. This generalizes the algebraic properties of music outlined by Hudak.

Preliminaries

```
module Music.Model.General where
import Prelude hiding (seq, id)
import Control.Category
import Control.Arrow
```

Composition

- Sequence
  - This option combines two events so that they occur in sequence.
- Parallel
  - This option combines two events so that they occur simultaneously.
- Repetition
  - This option takes an event and repeats it infinitely.

```
type Time = Double

class (Arrow t) => Temporal t where
    duration :: t a b -> Time
    rest     :: Time -> t a b
```

Laws:

```
duration id        = 0
duration (rest n)  = n
duration (f >> g)  = duration f + duration g
```

```
duration (f || g)  = duration f `max` duration g
duration (left f)  = duration f
duration (right f) = duration f
duration (loop f)  = duration f
```

Events

The primitive events:

- Constant
  - This event is simply a constant value
- Discrete
  - This event is a step of values, which changes at the given sample rate
- Continous
  - This event represents an arbitrary change in time

Obviously, continous events generalize constant and discrete ones. We use them for clarity and optimizations.

To represent an event bounded in time we use the Finite type. This simply specifies a duration for the event. Note that the discrete constructor could also be used for this purpose, as the list of sampled events may be finite.

```
data Signal a = Constant  a
              | Discrete  Time [a]
              | Continous (Time -> a)

data Event a = E Time (Signal a)
newtype EF a b = EF (Event a -> Event b)

makeContinous :: Signal a -> Signal a
makeContinous (Constant x)     = Continous $ const x
makeContinous (Discrete sr xs) = Continous $ (\t -> xs !! floor (t / sr))
makeContinous (Continous f)    = Continous $ f

instance Category (EF) where
   id  = EF (\x -> x)
   (EF f) . (EF g) = EF (\ (E t s) -> f (g (E t s)))
```

```
instance Arrow (EF) where
   arr f    = undefined
   first f  = f *** id
   second f = id *** f
   (EF f) *** (EF g) = EF (\ ((E t s), (E t' s')) -> ((E t (f s)), (E t' (g s'))))

instance Temporal (EF) where
   duration = undefined
   rest = undefined
```