Every Animation Should Have a Beginning, a Middle, and an End

A Case Study of using a Functor-based Animation Language

Kevin Matlage and Andy Gill

Information Technology and Telecommunication Center
Department of Electrical Engineering and Computer Science
The University of Kansas
2335 Irving Hill Road
Lawrence, KS 66045
{kmatlage,andygill}@ku.edu

Abstract. Animations are sequences of still images chained together to tell a story. Every story should have a beginning, a middle, and an end. We argue that this advice leads to a simple and useful idiom for creating an animation Domain Specific Language (DSL). We introduce our animation DSL, and show how it captures the concept of beginning, middle, and end inside a Haskell applicative functor we call `Active`. We have an implementation of our DSL inside the image generation accelerator, ChalkBoard, and we use our DSL on an extended example, animating a visual demonstration of the Pythagorean Theorem.

1 Introduction

Consider the problem of specifying the corners of a rotating square that is also moving from one location to another. There are two fundamental things happening over time: rotation and translation. The location of the corners is simply the combination of both movements, without interaction or interference. When describing more complex animations, however, we want to model simple interactions, and more generally, causality. Specifically, we want to introduce concepts like termination and sequentiality, and be able to describe interactions as one thing happening *after* another. In this paper, we discuss a *composable* solution to this description challenge which uses a Domain Specific Language (DSL) on top of Haskell [1] to express values that change over time, and also have a beginning and an end.

The fundamental question when crafting any type-based DSL is figuring out the key types and the primitives for these types. When we look at our target application, educational animations, and also look at animation tools in PowerPoint and Keynote, we make the following two basic observations. First, animations take a finite length of time, with a specific start and end point. In a sense, animations have a presence inside time, in the same way as a square can be present

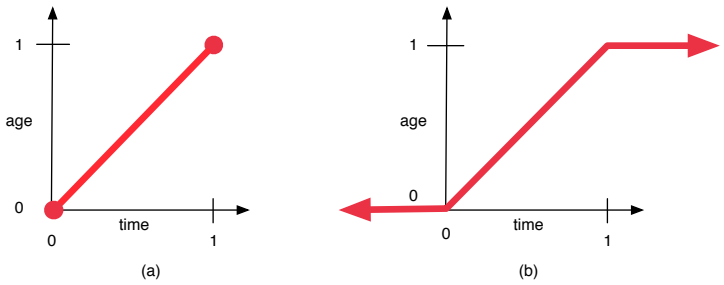

Fig. 1. The `age` combinator

on a 2D plane. We postpone considering infinite dynamic animations with our DSL, because we are explicitly attempting to build a language for scripting finite animations. Second, animations also often contain static, infinite elements, perhaps background images, that do not change for the duration of an animation. From these simple observations, we propose two primitives in our DSL, one that changes over time and is finite, and one that is static and infinite.

This paper presents these primitives, and a combinator-based language for creating dynamic animation using the primitives. We use an applicative functor [2] structure to implement this abstraction and incur other advantages, such as the clean and easy composition of animations (Section 2). We also provide a number of helper combinators and predefined functions for quickly creating functional animations (Section 4). Finally, we show how our language can be used to easily create practical, non-trivial animations (Section 5).

2 The Active Language

Our solution to this animation problem is the `Active` language. The conceptual framework behind the `Active` language is that all animations have a beginning, a middle, and an end. For every object in our language, we normalize the time component of an animation such that the value starts from 0 (the beginning of animation time), and ending at 1 (the end of animation time). This can be illustrated using the figure 1(a), where the dots are the beginning and end, and the line is the progression, or `age`, of an animation. The user of `age` does not need to be concerned about when each animation runs in the global time scale, but can instead build each animation with the assumption that it will act over its own 0 to 1 progression, and compose them later.

What happens before something is animated, or after animation? We *choose* to have an `Active` value be a constant before and after animation. Consider animating a object traveling; it is in one place before the animation, in transition during the animation, and one another place after the animation. We therefore choose our basic `Active` object to be of unit size in time (from 0 to 1), but also have a value before and after any animation. Figure 1(b) illustrates our realization of the representation in 1(a).

Our implementation of the `Active` language accomplishes this timing abstraction using a data type `Active` and a few primitive functions. The `Active` data type is defined (in Haskell) as:

```
data Active a                    -- Dynamic Animation
        = Active Rational        -- start time
                 Rational        -- stop time
                 (Rational -> a) -- what to do in this time frame
        | Pure a                 -- Static Animation
```

The first two `Rationals` are used to hold timing information, specifically the start and stop time of the current object. The function takes a `Rational` argument, representing time values between the start and stop times, and returns an animated value in the context of the time. The alternative constructor, `Pure`, is the way we represent an `Active` that is constant over time. The most primitive value in the `Active` DSL is `age`:

```
age :: Active UI
age = Active 0 1 f
    where f n | n < 0     = error $ "age value negative" ++ show n
              | n > 1     = error $ "age value above unit (1)" ++ show n
              | otherwise = fromRational n
```

`age` represents the most basic `Active`, which has a start time of 0 and a stop time of 1, as discussed above. This `Active` object also holds within it a basic function that returns the input `Rational` time value as a `UI`. A `UI` is simply a type synonym for `Float`, but is used to represent only those values on the interval [0,1]. Because the function stored within `age` returns a `UI`, `age` is of the type `Active UI`. Actions can then be mapped over this returned `UI`, but in order to do this, we must first define `Active` as a functor, given below for the curious reader. We also provide applicative functor capabilities. Specifically, applicative functors as used here allow for the declaration of a constant (static, infinite) `Active` value and the combination two `Active` values:

```
instance Functor Active where
    -- fmap :: (a -> b) -> Active a -> Active b
    fmap f (Active start stop g) = Active start stop (f . g)
    fmap f (Pure a) = Pure (f a)

instance Applicative Active where
    -- pure :: a -> Active a
    pure a = Pure a

    -- (<*>) :: Active (a -> b) -> Active a -> Active b
    (Pure a) <*> b = fmap a b
    (Active start0 stop0 f0) <*> (Pure a) =
            Active start0 stop0 (\ i -> (f0 i) a)
    a0@(Active start0 stop0 f0) <*> a1@(Active start1 stop1 f1) =
            Active (min start0 start1) (max stop0 stop1)
                   $ \ i -> f0 (boundBy a0 i) (f1 (boundBy a1 i))
```

When applying two animations, using the applicative functor `<*>` combinator, the interesting case is when combining two non-static animations. The first argument is a *function* which changes over time, the second is a value that changes over time, and the result is the application of the function to the argument, at every point in time. We choose to make this combined animation start at the earliest beginning of the two arguments, and finish at the last ending.

These definitions are particularly helpful in creating and combining animations. For example, the `<*>` operator allows for multiple animation functions to easily be applied to the same initial object. This ability can be really useful if, for instance, we wish to move an object while simultaneously scaling it. `Active` being an applicative functor is also helpful in creating combinators and predefined functions, as we will see in Section 4.

`age` is the primary method of creating an `Active` object. Once we have created an `Active`, all we have to do to get values over time is `fmap` a function over it. Generally for animation, this function would return an image so that we could display the returned images over time, creating an animation. The function can actually return any value, however, as shown by this definition for linear interpolation over time between two points:

```
simpleLerp :: (Float,Float) -> (Float,Float) -> Active (Float, Float)
simpleLerp (x1,y1) (x2,y2) = fmap (\ui -> lerp ui (x1,y1) (x2,y2)) age
        where lerp ui (x1,y1) (x2,y2) = ( x1+ui*(x2-x1) , y1+ui*(y2-y1) )
```

This `Active` will return values ranging linearly from (x1,y1) to (x2,y2) over time (though `lerp` would typically be a predefined library function). We can also begin to see some of the abstraction the Active DSL provides. Notice how the creation of this `Active` is completely independent from any timing information other than its own personal time progression. This same `Active` can be used to create a `lerp` that takes 1 second to complete or 100 seconds. The timing can be applied to each `Active` object separately, using either basic functions or built-in combinators. The primitive `Active` functions for handling timing effects are `scale`, `mvActive`, and `after`:

```
scale :: Float -> Active a -> Active a
scale _ (Pure a)              = Pure a
scale u (Active start stop f) = Active (scale u start) (scale u stop)
                                       $ \ tm -> f (tm / toRational u)

mvActive :: Float -> Active a -> Active a
mvActive _ (Pure a)              = Pure a
mvActive d (Active start stop f) = Active (toRational d + start)
                                          (toRational d + stop)
                                          $ \ tm -> f (tm - toRational d)

after :: Active a -> Active b -> Active a
after act@(Active low _ _) (Active _ high _) =
                                mvActive (fromRational (high - low)) act
```

When applied to an `Active` object, `scale` will stretch or shrink the amount of time that the object acts over. This can be used to make certain animations longer or shorter. It should be noted that this definition is actually an instance of a previously-defined `Scale` type class. This is not critical to understanding the details of `scale` except that it explains the call to `scale` within the body of the definition. This is a call to `scale`'s previously-defined `Rational` instance (which simply multiplies the two numbers).

`mvActive` is used for translating time values. When applied to an `Active` object, `mvActive` moves an animation forwards or backwards in time with regards to the rest of the scene. It can be used to put parts of an animation in the right place or offset animations to start at slightly different times.

The last basic timing function is the `after` function. It takes two `Active`'s as parameters and changes the time values of the first so that it will occur immediately after the second one finishes. This function is especially important for building up combinators to manage the ordering of animations in a scene, as we will see in Section 4.

3 ChalkBoard

The ChalkBoard project is an attempt to bridge the gap between the clear specification style of a language with first-class images, and a practical and efficient rendering engine. We will use ChalkBoard as an engine to display images generated using `Active`. The hook for ChalkBoard is that with the first-class status offered by pure functional languages comes clean abstraction possibilities, and therefore facilitated construction of complex images from many simple and compossible parts. This first-class status traditionally comes at a cost though—efficiency. Unless the work of computing these images can be offloaded onto efficient execution engines, then the nice abstractions become tremendously expensive. ChalkBoard was designed to bridge this gap by creating a functional image description language that targeted the OpenGL standard.

In order to understand the specifics of the ChalkBoard language, we need to think about types. In ChalkBoard, the principal type is a `Board`, a two dimensional plane of values. So a color image is a `Board` of color, or `RGB`. A color image with transparency is a `Board` of `RGBA`. A region (or a plane where a point is either in a region or outside a region) can be denoted using `Board` of `Bool`. Table 1 lists the principal types of `Board`s used in ChalkBoard.

The basic pattern of image creation begins by using regions (`Board Bool`) to describe primitive shapes. ChalkBoard supports unit circles and unit squares, as well as rectangles, triangles, and other polygons. The primitive shapes provided to the ChalkBoard user have the following types:

```
circle    :: Board Bool
square    :: Board Bool
rectangle :: Point -> Point -> Board Bool
triangle  :: Point -> Point -> Point -> Board Bool
polygon   :: [Point] -> Board Bool
```

`Board RGB`	Color image
`Board RGBA`	Color image with transparency
`Board Bool`	Region
`Board UI`	Grayscale image of Unit Interval values
`type R = Float`	Represent real numbers
`type Point = (R,R)`	2D coordinate or point

Table 1. Boards and Type Synonyms in ChalkBoard

To "paint" a color image, we map color over a region. Typically, this color image would be an image with the areas outside the original region being completely transparent, and the area inside the region having some color. This mapping can be done using the combinator `choose`, and the `<$>` operator:

```
choose (withAlpha 1 blue) transparent <$> circle
```

We choose the color blue with an alpha value of 1 for inside the region, and transparent for outside the region. The `<$>` operator is a map-like function which lifts a specification of how to act over individual points into a specification of how to translate an entire board. The types of `choose` and `<$>` are

```
choose :: O a -> O a -> O Bool -> O a
(<$>)  :: (O a -> O b) -> Board a -> Board b
```

where `O a` is an observable version of `a`.

As well as translating point-wise, ChalkBoard supports the basic spatial transformation primitives of scaling, moving, and rotating, which work over *any* `Board`.

```
scale   :: R     -> Board a -> Board a
move    :: (R,R) -> Board a -> Board a
rotate  :: R     -> Board a -> Board a
```

Although there are many more functions and possibilities available in ChalkBoard, we should now know enough to begin talking about its use within the context of the Active DSL. Any additional required ChalkBoard information will be explained as needed, but for a better background understanding, see the original paper on ChalkBoard [3].

4 Active Combinators

Now that we have some of the most important functions in the Active language, we want to make using them with ChalkBoard easier. One natural way to do this is to create combinators that integrate common Active and ChalkBoard tasks. The first, and perhaps most essential, of these is the `over` function:

```
over :: Over a => Active a -> Active a -> Active a
over a1 a2 = fmap (\ (a,b) -> a `over` b) (both a1 a2)

both :: Active a -> Active b -> Active (a,b)
both a b = pure (,) <*> a <*> b
```

The `over` function takes two `Active` parameters and combines them so that both animations are displayed one on top of the other (but not necessarily at the same time). `over` is actually an instance of the ChalkBoard `Over` type class, which helps explain the second reference to `over` in the body of the definition. This uses the ChalkBoard version of `over` to overlay two static objects, most notably boards with transparency, `Board RGBA`.

While `over` and our current timing functions let us combine animation pieces and display them in order, it can be verbose to specify a long sequence of animations that should all be overlaid and displayed at times relative to each other. This led us to create one of the main code structures that we have used repeatedly to manage our scenes. The main version of this structure uses the `flicker` and `taking` functions, though multiple derivatives of `flicker` have been created for managing time in different ways. The type of these functions and the general code structure can be seen here:

```
flicker :: Over a => [Active a] -> Active a
taking  :: Float -> Active a -> Active a

let anim = flicker [ animStep1
                   , taking 3 animStep2
                   , taking 0.5 animStep3
                   ]
```

The `flicker` function takes a list of `Active`'s and combines them into one `Active` object, with each animation in the list occurring immediately after its predecessor. Each successive animation is also placed on top of the previous ones, so parts of a scene can be built independently but displayed together. This is helpful in increasing the amount of abstraction in building a scene. Constructing each part separately allows for greater flexibility in changing certain aspects of a scene without affecting others, and managing the ordering of the scene without affecting what happens during each part.

`taking`, on the other hand, helps control the amount of *time* it takes to execute each of the individual animations. The `taking` function stretches or shrinks an `Active` so that it occurs in the amount of time specified by the `Float` argument. Generally, `taking` is easiest to use in close conjunction with the `flicker` function, as shown above, though it does not have to be. This just keeps most of the timing information in one place, even if one does not directly affect the other.

Now that we can manage the ordering and timing of an animation pretty well, we can start looking at some good combinators for common animation tasks. To help create many of these combinators, we use the `addActive` function:

```
addActive :: (UI -> a -> b) -> Active a -> Active b
addActive fn act = (fmap fn age) <*> act
```

This is a simple function we use to create many animation functions. Typically for animation, the `a` and `b` types are `Board`'s of some variety. The function argument is then a representation of how we want to change a `Board` over time, and the `Active` argument contains a `Board` we want to change (though it may already be changing in other ways as well). `addActive` is especially helpful in adding new animations to existing ones, allowing us to avoid the systematic coding overhead of placing each new function into an `Active` and then applying it to the previous `Active`.

We use `addActive` to help create many of our predefined animation functions, including the standard 2D transformation functions from ChalkBoard (`move`, `scale`, and `rotate`) applied over time. As an example of this usage, the predefined move-over-time function in Active is:

```
activeMove :: (R,R) -> Active (Board a) -> Active (Board a)
activeMove (x,y) = addActive $ \ui -> move (ui*x,ui*y)
```

This function takes the ChalkBoard `move` command and turns it into a function over time as well. The `move` command in ChalkBoard simply moves a `Board` from its current position by a specified amount in the x and y directions. These amounts are given, respectively, in the ordered pair `(R,R)`. The `Active` version of this function does the same thing, but applies this move over time. It will treat the input `UI` time value as a percentage and move the `Board` inside the `Active` argument step by step as the time value increases from 0 to 1, finally ending up displaced by a total amount of (x,y).

Other common actions defined using `addActive` are the remaining transformation functions (`activeScale` and `activeRotate`), as well as functions for making an `Active` appear/disappear (`activeAppear`, `activeTempAppear`, and `activeDisappear`). All of the Active versions of the ChalkBoard transformations (`move`, `scale`, and `rotate`) are versions of those functions that are applied over time. The appear/disappear functions tell a given `Active` whether it should only be visible once its time value is great than 0 (`activeAppear`), when its time value is in between 0 and 1 (`activeTempAppear`), or from the start of execution up until its time value is 1 (`activeDisappear`). Unless one of these functions is applied, all `Active`'s will remain visible for the duration of the scene, regardless of when their animations execute (since they will still be receiving time values of 0 or 1). Example usage of these functions is the subject of the next section.

5 Case Study

While testing the current features and usability of Active, we decided to recreate an existing animation. This was done both to see how close we could get to the original, as well as how difficult it would be to do so. The animation we chose for this experiment was an animated proof of the Pythagorean Theorem that can

be found on Wikipedia at http://en.wikipedia.org/wiki/Pythagorean_theorem. This example was visually pleasing, served a useful purpose, and was exactly the type of animation we wanted to create easily in ChalkBoard. It also was complicated enough that we felt like it would be a good test of ChalkBoard's features, without being too complicated as to prevent new users, who haven't seen any of these feature before, from following along.

In building this and other examples, a general structure for ChalkBoard animations using Active has begun to appear. It looks something like the following:

```
let animStep1  = ...
    animObject = ...
    animStep2  = ... f animObject ...
    animStep3  = ... g animObject ...

let wholeAnim = flicker [ animStep1, animStep2, animStep3 ]
```

First, the individual pieces of the animation are constructed. This stage consists of building all the separate `Active Board`'s that will be the parts of the final scene. These could be such things as an object moving, rotating, changing colors, or a ton of other possibilities. The second stage of construction is stringing all of these smaller pieces together into a coherent whole using functions such as `flicker`. After the animation is complete, it can then be played back, saved, or manipulated however the user wishes. While creating animations using this structure is by no means the only way to do so, it has proven to be effective for the examples we have built thus far. Therefore, this case study will follow the same structure, explaining how each stage was completed and some of the functions that were used.

5.1 Stage 1: Building Animation Pieces

In beginning the Pythagorean example, we start by creating all of the different `Active` animation pieces that will be used in the scene. The first of these is the animation's background, which we just build to make about the same color as the Wikipedia animation. The `pure` function is then applied to this background board to lift it into the `Active (Board a)` space so that it can be combined with the other `Active Board` objects we create for the animation.

Next, we build up a basic triangle in the middle of the screen, with code that looks something like the following:

```
let (x,y)        = (0.2,0.15)
    (a,b,c)      = ((-x,y),(-x,-y),(x,-y))
    triangle345  = triangle a b c
    triLines     = pointsToLine [a, b, c, a] 0.004
    mainTriangle = (choose (alpha black) transparent <$> triLines)
                   `over`
                   (choose (alpha yellow) transparent <$> triangle345)
```

In doing this, we first create a 3-4-5 triangle by giving three points to the `triangle` constructor. This creates a `Board Bool` of our triangle. We also want a black outline around it in order to match the original animation. To do this, we use the `pointsToLine` function, which takes a list of points and a line width and draws a line between all adjacently listed points. Both `Board Bool`'s are then given their colors by using the `choose` function as shown. This makes the lines black over a transparent background (so we can see the triangle behind them) and the triangle yellow with a transparent background (to see the animation's background behind it).

While this code does create a simple triangle, the triangle itself is never actually displayed in the animation. Instead, this triangle is transformed in different ways to create the displayed triangles. For instance, the initial triangle shown in the animation is achieved by scaling `mainTriangle` by 1.5. The animation for shrinking and moving this new triangle into its final position is achieved by adding `Active` functions, as shown below:

```
let movingTriangle = activeMove (y,x) $ activeScale (2/3) $
                     pure $ scale 1.5 $ mainTriangle
```

First, the triangle is lifted into the `Active` world using `pure`. Then we start to add animation functions to it. In this instance, we apply an `activeScale` and an `activeMove`. This creates an animated triangle that shrinks slightly while also moving slightly up and to the right. Images of this resulting animation are in Figure 2.

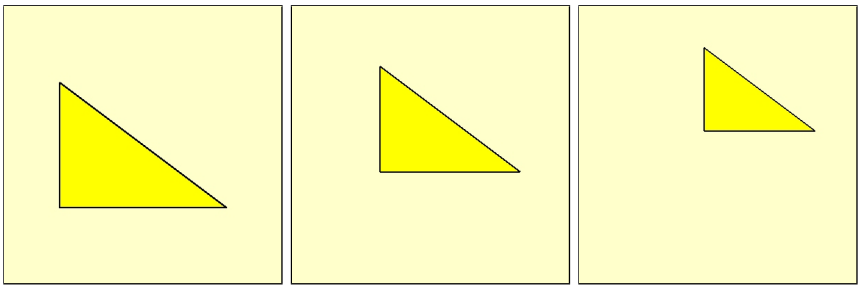

Fig. 2. movingTriangle animation

As a note, all of the text for this animation was actually added in last, separate from the geometry. In this case study, we will only be covering the creation of the geometric animation, and not the insertion of font. This is due to space constraints and because the only interesting font problem that involves the Active DSL is when to make the pieces appear and/or disappear (which we will already cover).

Moving on with the example, the next step is to create three identical but rotated triangles as displayed in the Wikipedia graphic. These are created using

the list comprehension in `otherTriangles` (defined below), which simply rotates a moved version of the original `mainTriangle`:

```
let movedTriangle = move (y,x) $ mainTriangle
    otherTriangles = [ rotate (-i*pi/2) $ movedTriangle | i <- [1..3] ]
    addOtherTriangles = foldl1 over [ mvActive i $ activeAppear $ pure $ t
                                    | (t,i) <- zip otherTriangles [1..] ]
```

These triangles are then made to appear when their animations start using the `activeAppear` function as described in Section 4. The next step, however, is getting them to appear *individually*. While they could each be treated as an independent animation piece and listed separately in the `flicker` portion of the program, we instead choose to apply the `mvActive` function to each of these new triangles in order to save time and make our code cleaner. As described in Section 2, this function (not to be confused with `activeMove`) simply moves the actions of a given `Active` backwards or forwards in time by the given value. Using the list comprehension in `addOtherTriangles` above, each new triangle is made to appear a little later in time than the previous. Finally, the list of `Active Board` objects, each element representing one new triangle appearing, is compressed into a single `Active Board` using `foldl1` with the `over` function. Figure 3 shows each of the new triangles being added individually to the animation.

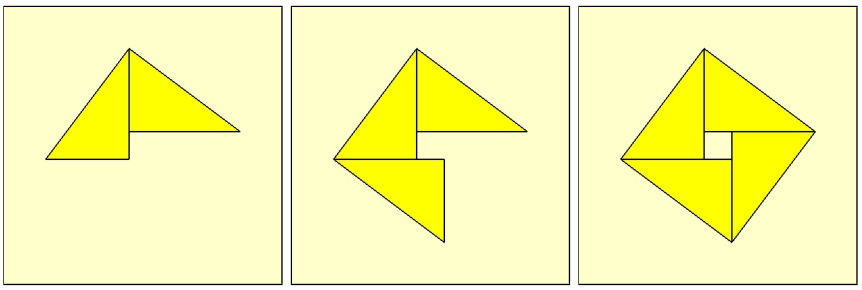

Fig. 3. addOtherTriangles animation

The next part of the animation is just adding in a couple missing pieces to the image so that the full area can be clearly seen. A small yellow square is added to the middle so that the larger square can be seen to have a size of $c \times c$. This larger square therefore has an area of c^2, as indicated by the accompanying text. The result of this small portion of the animation can be seen in Figure 4.

Next, we need to slide the top triangles down to match up with the lower triangles, as seen in Figure 5. We also want an outline of the old triangles to remain behind so we can see where they started (like in the original on Wikipedia). This is done in two parts, both of which are defined below:

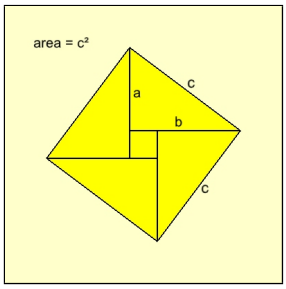

Fig. 4. fillSquare animation

```
let fadedTris = [ rotate (-i*pi/2) $ move (y,x) $
                  choose (withAlpha 0.6 white) transparent <$> triangle345
                | i <- [0,1] ]
    slideLeft = activeAppear $
                (activeMove (-2*y,-2*x) $ pure $ movedTriangle)
                `over` (pure $ head fadedTris)
    slideRight = activeAppear $
                 (activeMove (2*x,-2*y) $ pure $ head otherTriangles)
                 `over` (pure $ last fadedTris)
```

The first part is to fade the existing triangles to leave behind as outlines, and the second is to create the new triangles that will actually move. The first part is done by placing white triangles with alpha values of 0.6 over the two existing triangles so that they will appear faded. For the second part, we create the first slide by reusing `movedTriangle` (top right triangle) and applying an `activeMove` down to its final position on the bottom left. We do pretty much the same thing with the second slide, but grab the initial triangle from the head of `otherTriangles` (first rotated triangle on the top left) and slide it right.

The final part of the animation is simply changing the organization of the resulting shapes. Now that the triangles are in their final positions, two new squares can be drawn that cover the entire area. These squares have side lengths of a and b, and thus areas of a^2 and b^2. This in effect concludes the proof that $a^2 + b^2$ equals the original area of c^2. In order to animate this part, we use the same general strategy as fading out the two triangles in the last step:

```
let newSquares = (move (y, -y) $ scale 0.4 $ square)
                 `over` (move (-x, -x) $ scale 0.3 $ square)
    (s1, s2) = (x-y, x+y)
    newLines = pointsToLine [(-s1,-s2), (s2,-s2), ..., (-s1,-s1)] 0.004
    fadeInSquares = (fadeIn black 1 newLines)
                    `over` (fadeIn yellow 1 newSquares)

fadeIn :: O RGB -> UI -> Board Bool -> Active (Board (RGBA -> RGBA))
fadeIn rgb a brd = fmap fn age
      where fn ui = choose (withAlpha (o (ui*a)) rgb) transparent <$> brd
```

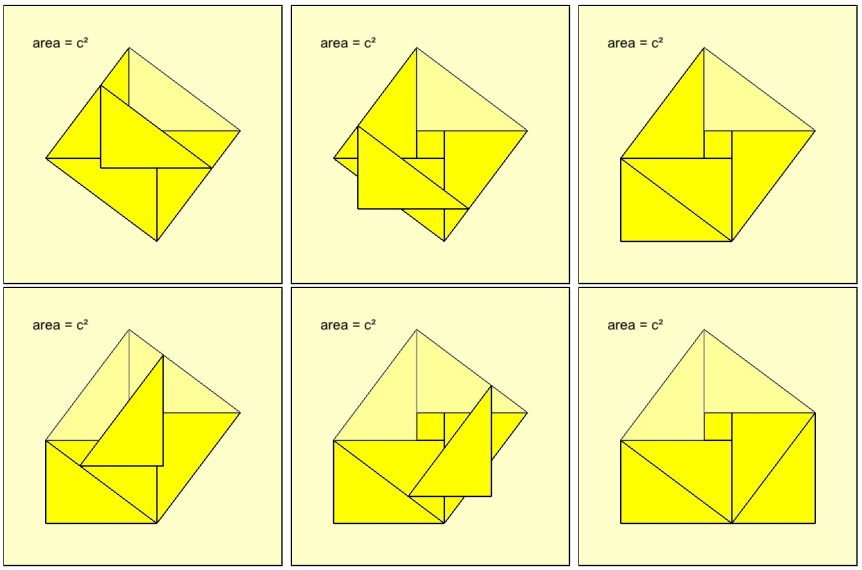

Fig. 5. slideLeft and slideRight animations

The main differences are that this time we use yellow squares with alpha values of 0.9 so that the new squares will be a darker yellow instead of a lighter one, and that we also draw lines around the new squares to make them clearer. The squares to be faded in are created as `Board Bool` shapes in ChalkBoard, like normal, and moved to the right locations. They are then faded in over time using the `fadeIn` function (predefined in `Active`, but included here for reference). This function simply creates an `Active` that fades a `Board RGBA` in from transparent to the given RGB and alpha value. The lines around the squares are also faded in over the squares at the same time, using the same function. This final piece of the animation is shown in Figure 6.

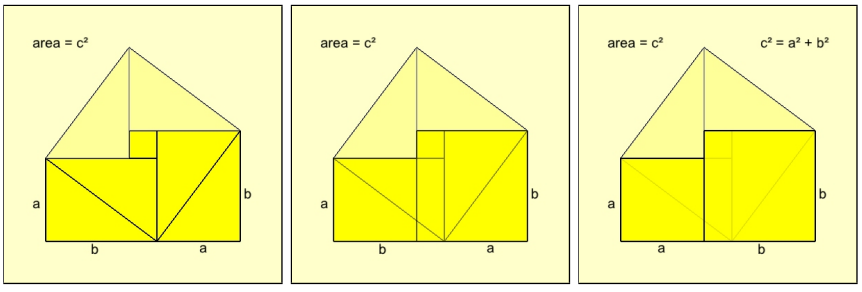

Fig. 6. fadeInSquares animation

5.2 Stage 2: Combining Animation Pieces

In this example, each part of the animation is created separately. The smaller animation pieces often use some of the same basic structures repeatedly, and this piecemeal construction strategy lends itself well to reuse. For instance, the originally defined `maintriangle`, which is never directly displayed, is rotated and moved around to create most of the triangles in the scene. While longer animations can be created directly using the `mvActive` function, we have found that it is generally much cleaner and easier to organize simple animations into a series using one of our combinators, such as `flicker`.

Using the `flicker` function in this way is the second major stage we discussed for creating an animation. With the `flicker` function, animations can be strung together, one after the other, stacking newer parts onto older ones. The time each individual animation component takes to be performed can be specified using the `taking` function, as described earlier. Our general structure looks like:

```
let anim = flicker [ taking 0.5 $ background
                   , taking 1 $ firstABC
                   , taking 1 $ movingTriangle
                   ...
                   , taking 1 $ fadeInSquares `over` thirdABC
                   , taking 3 $ finalABC `over` formula
                   ]
```

This use of `flicker` and `taking` is what we use to manage the majority of our ordering and timing for animations. It returns a single `Active Board` that can then be used to display the whole animation, or reused in turn to create an even bigger animation, hierarchically. In terms of displaying the animation, this will largely be done the same way for most animations:

```
sid <- startDefaultWriteStream cb "pythagorean.mp4"
playObj <- byFrame 29.97 anim

let loop = do mbScene <- play playObj
       case mbScene of
              Just scene -> do
                     drawChalkBoard cb $ unAlphaBoard (boardOf white) scene
                     frameChalkBoard cb sid
                     loop
              Nothing -> return ()
loop
```

First, the `Active Board` must be turned into a `Player` using the `byFrame` function (which also takes a desired frame rate). The `Player` is then passed to the `play` function to retrieve the next image of the animation (or `Nothing`, if the animation is finished). Finally, this retrieved image can be used in any way that ChalkBoard can use a `Board`. Traditionally, the image is displayed on the screen using `drawChalkBoard` or saved into a video file with `frameChalkBoard`

(or both). After this, the process of calling `play` on the `Player` must be repeated to extract the next image. This is usually placed into a simple loop that extracts and then displays the returned frame, as shown above. We used this method to produce a video of the full animation created in this case study. The video can be seen online at http://www.youtube.com/watch?v=UDRGhTFu17w.

6 Related Work

There have been numerous image description DSLs using functional languages, many of them capable of animation. A lot of the image description languages similar to ChalkBoard are described in our earlier ChalkBoard paper [3].

In particular, the work of Conal Elliott had one of the largest influences on ChalkBoard. Elliott has been working on functional graphics and image generation for many years and has produced a number of related systems. These include Fran [4], Pan [5], and Vertigo [6]. ChalkBoard was heavily influenced by Pan and started from the same basic set of combinators provided in Pan.

In terms of animation and the `Active` DSL, some similar systems that have been created are Slideshow [7] and the function system presented by Kavi Arya [8]. One of the major differences between the `Active` animation system and these, however, is the treatment of time. Slideshow is predominately frame-based because of its goal of generating slides for presentations. Arya's system, meanwhile, can cue animations relative to one another or to object interactions. The `Active` DSL, on the other hand, is time-based. It allows the user to create functions mapped over a known time progression and then affect the time management of animations separately. While this management often includes cueing animations relative to others, similar to the two languages mentioned, it can also include stretching or shrinking animations and moving them forwards or backwards in time. A few of the `Active` combinators can also help provide a simple framework for reordering animations.

The closest related work to our `Active` DSL is Hudak's temporal media DSL [9], which was also used to specify change over time in a pre-determined manner, but was used to generate music, not images, and also did not codify the ability to use applicative functors. The `Active` DSL is also conceptually close to Functional Reactive Programming (FRP) [10], even though `Active` does not attempt to be reactive in the same sense as FRP. Both `Active` and (one implementation form of) `FRP` are mappings from time to value, however `Active` does not implement FRP Events, but rather an `Active` object has a start and an end. With `Active` being designed for presentations and similar educational animations, all of the actions in the `Active` DSL are explicitly specified ahead of time by the user, although they can be in relation to other animations.

Of course, there are many other animation languages and systems. `Active` is an attempt to combine the concept of first class functions over time (from FRP), width in time (like the temporal media DSL), and the idiom of packing such functions over time (as an analog to stacking boxes in space) to provide a clean starting idiom for animation specification.

7 Conclusions and Future Work

The `Active` language is a mathematically-based system where actions are the results of mapping functions over time values progressing from 0 to 1. It provides substantial abstraction for the different pieces that go into creating an animation, such as the drawing, timing, and ordering, and is useful in practice.

The biggest improvement we hope to make to the `Active` DSL in the future is the inclusion of some more precise combinators for the cueing and timing of animations. While the current structures have proven useful, there are some instances in which the current `Active` API could have been improved. Specifically, we hope to work on structures that will allow users to specify *when* animations should be visible. In this type of structure, the default may be for animations to only appear when they are currently active (progressing from 0 to 1), and have means of specifying which objects should be visible at other times.

Another improvement we hope to make is to increase the amount of internal sharing that is done by the ChalkBoard compiler in order to more efficiently create the animations it generates. In our animations, a lot of the same boards are often reused, just at slightly different positions on the screen. Because ChalkBoard treats each of these boards as a texture, the potential for reuse of these textures in animation is very high, they often just need to be remapped onto the scene at a slightly different location or size.

References

1. Peyton Jones, S., ed.: Haskell 98 Language and Libraries – The Revised Report. Cambridge University Press, Cambridge, England (2003)
2. McBride, C., Patterson, R.: Applicative programing with effects. Journal of Functional Programming **16**(6) (2006)
3. Matlage, K., Gill, A.: ChalkBoard: Mapping functions to polygons. In: Proceedings of the Symposium on Implementation and Application of Functional Languages. (Sep 2009)
4. Elliott, C.: From functional animation to sprite-based display. In: Practical Aspects of Declarative Languages. (1999)
5. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. Journal of Functional Programming **13**(2) (2003)
6. Elliott, C.: Programming graphics processors functionally. In: Proceedings of the 2004 Haskell Workshop, ACM Press (2004)
7. Findler, R.B., Flatt, M.: Slideshow: functional presentations. J. Funct. Program. **16**(4-5) (2006) 583–619
8. Arya, K.: Processes in a functional animation system. In: FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture, New York, NY, USA, ACM (1989) 382–395
9. Hudak, P.: An algebraic theory of polymorphic temporal media. In: Practical Aspects of Declarative Languages. (2004) 1–15
10. Elliott, C., Hudak, P.: Functional reactive animation. In: International Conference on Functional Programming. (1997)