ChalkBoard: A Functional Image Description Language and Its Practical Applications

BY

Kevin J. Matlage

Submitted to the graduate degree program in Electrical Engineering & Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Date Defended: April 21, 2011

The Thesis Committee for Kevin J. Matlage
certifies that this is the approved version of the following thesis:

ChalkBoard: A Functional Image Description Language and Its Practical Applications

Chairperson Dr. Andy Gill

Date Approved: April 21, 2011

Abstract

ChalkBoard is a domain specific language (DSL) for describing images. The ChalkBoard language is uncompromisingly functional and encourages the use of modern functional idioms. Unlike many similar systems, ChalkBoard uses off-the-shelf graphics cards to speed up the rendering of these functional descriptions. The speed gained from this efficient rendering has allowed the addition of animation to the language. The Active extension to ChalkBoard is another DSL, built on top of ChalkBoard, that captures the concept of animation inside a Haskell applicative functor. This allows for a clean, compositional approach to animation in ChalkBoard. Given both the efficiency and functional style of this animation, there are many possible applications. One of these is a system called Active Transformations, which uses the animation capabilities of ChalkBoard to animate code (and other tree-based) transformations. The Active Transformations system uses ChalkBoard animation in order to show *how* certain transformations and optimizations occur, instead of just providing before and after snapshots, making it an extremely useful educational tool. In this thesis, the design of each of these systems is described and explained, from the core ChalkBoard language, to the Active extension to ChalkBoard, and finally the Active Transformation system built on top of them.

Acknowledgements

I would like to thank the members of CSDL at KU, and especially my advisor, Dr. Andy Gill for all the help and support they have show me, both while working on this thesis and throughout my time at KU.

This thesis describes the ChalkBoard project and related components, which I have been working on over the last two years with Dr. Andy Gill.

Contents

Chapter 1

Introduction

Options for image generation abound. Functional languages and image generation have been courting for decades. Describing the *mathematics* of images in functional languages like Haskell [26] is straightforward. Yet there is no clear choice for describing images functionally and then *efficiently* rendering them.

There certainly are many image generation choices in Haskell. The popular cairo [1], for example, is an efficient image language, based on imperatively drawing shapes onto a canvas, with a Haskell IO port of the API. This thesis is interested in exploring *purely functional* representations of images, however, and seeks to understand if they can be made efficient.

The ChalkBoard project, presented in this thesis, 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. Though systems like cairo offer the ability to use created images as new components, the difference here 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—efficiency. Unless the work of computing these images can be offloaded onto efficient execution engines, then the nice abstractions become tremendously expensive.

Part of this thesis describes a successful attempt to target one such functional image description language to the widely supported OpenGL standard in order to achieve this efficiency. Much of this work on ChalkBoard has been described in and is based upon my original ChalkBoard paper [22].

After the success of ChalkBoard in efficiently rendering functionally-described images, the next step was to see if this generation was efficient enough to allow for real-time animation. Initial, rudimentary tests appeared to indicate that it would be, and so work began on an extension to ChalkBoard that would allow it to easily create animations, called Active.

The key to this extension was keeping the animation and timing effects abstracted from the actual ChalkBoard drawing functions. In addition, Active must also be flexible and robust, as well as maintain the functional style of ChalkBoard. This new set of challenges faced in creating Active are expanded upon in Section 2.2. As with ChalkBoard, much of this work is based upon my original paper on the subject [23].

The final piece of this thesis is a practical example of a complex system built on top of ChalkBoard and Active, called Active Transformations. This system takes advantage of many of the features of ChalkBoard and Active mentioned above, as well as others expanded upon throughout this thesis. This Active Transformations system is a transformation animation framework that can be used to create animations for code transformations almost automatically. Although the system is intended for displaying code transformations, such as those that might be found inside a compiler, it is actually a lot more general than that. Active Transformations can be used to animate nearly any sequence of changes to nearly any tree, given certain requirements are fulfilled by the user.

In addition to demonstrating the usefulness of ChalkBoard in a practical, complex example, however, this transformation animation system actually encounters many interesting challenges of its own. It must wrestle with problems such as how to represent a user's data structure internally, capturing incremental changes to this structure, and knowing how each of these changes should be correctly animated. These problems, among others, are expanded upon in Section 2.3. The work for this portion of the thesis, unlike the others, has yet to be published elsewhere, but will likely be so in the future.

Overall, through each different piece of this thesis, the goal remains roughly the same. That goal is to show how functional languages and functional design patterns can be combined effectively and efficiently with computer graphics in order to create new and interesting opportunities for both. Inherent in this goal is that the new opportunities generated must also have practical applications in the real world.

Chapter 2

Problem Specifics

2.1 Functional and Efficient Rendering

In order to understand some of the problems in achieving an efficient rendering of a functional image specification, we must first understand roughly how this transliterating from a functional image description language to a imperative-style library such as OpenGL may work. Figure 2.1 gives the basic architecture of ChalkBoard. Our image specification language is an embedded Domain Specific Language (DSL). An embedded DSL is a style of library that can be used to capture and cross-compile DSL code, rather than interpret it directly. In order to do this and allow use of a polygon-based back-end, we have needed to make some interesting compromises, but the ChalkBoard language remains pure, has a variant of functors as a control structure, and has first-class images. We compile this language into an imperative intermediate representation that has first-class *buffers*—regular arrays of colors or other entities. This language is then interpreted by macro-expanding each intermediate representation command into a set of OpenGL commands. In this way, we leverage modern graphics cards to do the heavy lifting of the language.

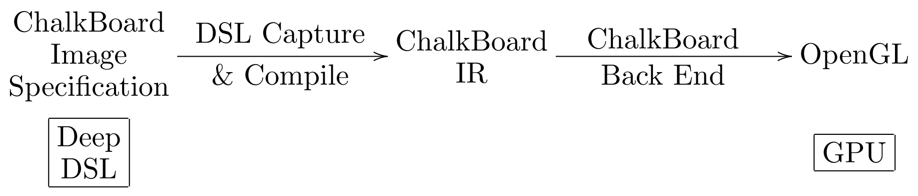

Figure 2.1. The ChalkBoard Architecture

Both subsystems of ChalkBoard are written in Haskell, and are compiled using GHC [2]. ChalkBoard could use the same Intermediate Representation (IR) to target other back ends besides OpenGL, but this first part of the thesis focuses on how we use OpenGL to achieve fast *static* image generation from a purely functional specification. Specifically, our work on ChalkBoard makes the following contributions.

- We pursue an efficient functional representation of images. In order to do this, we build a simple image generation DSL, modeled on Elliott's Pan [6], but with an abstract principal data type to facilitate introspection.

- To allow DSL capture, we need to impose some restrictions on the form of expressions. In particular, we identify challenges with capturing maps over functors and introduce our solution, the observable `O` datatype, which should be reusable in other DSLs.

- Having captured our DSL, we need a vehicle to experimentally verify our image generation ideas. We describe the design of our ChalkBoard accelerator and give some initial performance results for our ChalkBoard compiler and runtime system that demonstrate that ChalkBoard has sufficient performance to carry out these future experiments.

The intent with the technology discussed here is that it will be of immediate applicability, as well as serve as a basis for our dynamic image generation and processing tools discussed later, all of which will be executing specifications written in functional languages.

2.2 Animation

This thesis also describes our endeavor to construct a useful abstraction for ChalkBoard animations, as well as animations in general. Our early attempts at animation used functions to create changing images. By looping and passing slightly varied arguments to one of these functions, we could create a quick progression of slowly-changing images, or an animation. While this approach did successfully create animations, it was by no means a very sophisticated solution. It required an argument for each aspect of an image that we wished to change, so an appropriate sequence of values had to be generated for each argument every time an animation was created. This solution was also very intertwined with the specific animation created. Every animation had to be built from scratch, without much in the way of reusable code.

This lack of abstraction seemed unnecessary, as the drawing functions and how to use these functions over time are two inherently different things. At least the basics of an animation language seemed like it should be abstracted from the image creation language. In addition, many of these timing-related tasks are also repeated over and over in creating an animation. We therefore wanted an animation language that would allow for this abstraction from the drawing language, but that would also lend itself well to creating useful combinators for common, repeated tasks. The Active animation language is our solution to this problem, and is built as an extension to ChalkBoard (though it could work in any similar drawing system). In this thesis, we show how the Active extension:

- Uses an applicative functor structure to gain many advantageous properties, such as abstracting the animation and timing away from the image description and drawing
- Takes a non-standard timing approach to help provide further abstraction and ease the creation of composable animation pieces
- Provides many helpful combinators and predefined functions for quickly creating functional animations
- Can be used to easily create practical animations with real-world applications

The fundamental goal of Active is to have a simple animation language, parameterized over time, which keeps the desired final effects and the timing of those effects independent. We also want to make sure that this timing and how it is applied to a given drawing system is abstracted and flexible, both so that the same general structures can be reused and also so that combinators can easily be built up for common actions.

2.3 Practical Applications and Other Problems

The last topic covered by this thesis is the practical application of ChalkBoard in a system for animating code transformations. Pretty printing an abstract syntax tree is something functional languages are really good at. By using a small

Domain Specific Language [15] [21], a clear and clean syntax rendering mechanism can be constructed for any representable syntax in a few lines of code. In a sense, these prettified renditions are stand-alone snapshots of observable, static syntax trees. In the programming language community, however, we consider languages to be first class citizens. Code changes over time, and pretty printers are ill-equipped to do anything but re-render a syntax tree from scratch.

Consider explaining a program transformation, using a specific example. We can show the (static) code before and after, in a visually aesthetic form, and hope that the relationship between the two pieces of code brings out the differences in a way that the transformation can be observed and understood. Or we can *animate* the transformation, showing the difference between two examples using dynamically morphing code, and being explicit about the change. The final part of this thesis discusses the issues faced in building this animation capability; a pretty printer that is aware of its commitments to time, as well as usage of whitespace.

An overarching design decision of this system is to allow, as much as possible, the structure and data types used in an original transformation to be preserved when using our pretty printer. This enables the user to take a transformation system they have already created and animate it straightforwardly, with only a few changes/additions. Most of the information needed to animate a transformation is already present inside the implementation of that transformation. Our final solution, therefore, has the user augment their transformation with *some* additional information and parameterization, which allows the rest of the work to be done for them.

The leading questions in this work are:

- What structural representation allows the capture and the extraction of changes caused by a user's transformation?

- If the transformation is recursive with nested changes, how can we break this recursion, and capture the intermediate steps to animate them correctly?

These questions are interesting in and of themselves, in addition to the overall system being a suitably complex example that demonstrates the usefulness of ChalkBoard. I will therefore be exploring these issues in further depth as well in the later parts of the thesis, and displaying how this system effectively makes use of ChalkBoard.

Chapter 3

Background

3.1 Haskell

The ChalkBoard domain specific language is written in Haskell, a pure, functional programming language [26]. Because of this, understanding the basics of Haskell and how it works is therefore very helpful in comprehending this thesis.

3.1.1 Types

One of the first things that must be discussed when trying to briefly explain the basics of Haskell is types. Haskell is a language that relies heavily on types and a comprehensive type system. The basic types in Haskell are pretty much the same as you would find in most language. There are `Int`, `Float`, `Bool`, `String`, and many other fundamental types commonly found in most languages.

In Haskell, however, these types can be combined and extended in a large variety of ways. The simplest forms of combining types are pairs, triples, and so on, which use the "tuple" syntax. These structures simply combine two types together to form one type, with a field for each of the sub-types. An example of this would be the type:

```
(Bool, Float)
```

This type is simply a combination of a `Bool` and a `Float`, where values of both types must be passed around together. The values can also be accessed individually, but never changed, because values in Haskell are *immutable*. In order to change the values, a new pair must be constructed with the new desired values.

Another way to group values together in Haskell is with lists. Lists in Haskell are homogeneous structures, meaning that they can hold *any* type, but *only one* type per instance. For example, a list of `Int` can be created and a list of `Bool` can be created, but once they are created, no other types besides `Int` and `Bool` can be used in these lists, respectively. Because of how commonly the list structure is used, it is given a simple syntax in Haskell:

```
foo :: [Int]  -- List of Ints
foo = [1,2,3,4]
```

This example also shows a couple other features in Haskell. In this example, `foo` is specified to have the type `[Int]`, or list of ints, by the `::` operator. This operator ascribes a type to an object, and can be thought of as saying "has the type", meaning that in this instance `foo` "has the type" list of ints. We also see in this example that `--` is the syntax used to add comments, as well as a basic way of creating lists. This way of creating lists is simply to place the desired list values inside square brackets, separated by commas. In this case, `foo` has been assigned the list "1, 2, 3, 4".

This same list (of 1 through 4) could also be created using the syntax `[1..4]`, which says to make a list of all the values from 1 to 4, counting by 1. This syntax can also be used to count by other values than 1 if an example is given, such as [2,4..10] or [1,1.1..2], which will construct lists from 2 to 10, counting by 2, or 1 to 2, counting by 0.1, respectively.

There are also other, more complex ways of creating lists, such as list comprehensions:

```
listComp1 = [i*i | i <- [1..4]]
listComp2 = [(i,j) | (i,j) <- zip [1..] [-1,-2..]]
```

In the first list comprehension, each element of the returned list is equal to i*i, where i is "drawn from" the list [1..4]. What this means is that, for the first element of the returned list, the i varible will represent the first element of the list [1..4], or 1. In the returned list, then, the first element will therefore be 1*1. The rest of the list will follow the same pattern, eventually returning the final result [1,4,9,16].

The second list comprehension is much the same except that the zip function is used to zip two lists together into a new list of tuples. These tuples are then drawn, one at a time, to represent the variables i and j. The two lists that are zipped in this example are also infinite, one counting upwards from 1, and the other counting downwards from negative 1. This list comprehension will therefore also return an infinite list. It will return an infinite list of pairs of integers, starting with "(1,-1), (2,-2), (3,-3) ..." and continuing onwards.

Because Haskell is a *lazy*, language, the values of this list are not actually generated until they are needed. This is what allows infinite lists to be represented. The first few values of the list can then be accessed by using the take function to grab, for instance, the first 10 values from the list (take 10 listComp2), or the first 1000 values from the list (take 1000 listComp2). The user must be careful with infinite lists, however, as using them in a recursive call can obviously result in an infinite loop.

Although lists in Haskell have all this syntactic sugar to make them easier to use, they are actually a good example of a Haskell data structure as well. The Haskell declaration of a data structure representing lists may look something like:

```
data List a = Cons a (List a)
            | Nil
```

There are actually quite a few new pieces of Haskell to examine in this short code snippet. First, let's begin with the data structure declaration itself. The `data` key word is used here to declare a new datatype, `List a`. Before worrying about what this `a` means, let's first worry about the rest of the datatype definition. A datatype in Haskell is kind of similar to a grammar. On the right hand side of the equals sign, there is a set of "constructors." These constructors are the different options used to create instances of the datatype. An object of type `List a`, therefore, must either be a `Cons a (List a)`, or a `Nil`.

If it is `Nil`, then we are done and the object has been created. If it is `Cons a (List a)`, on the other hand, then there are some fields that need filling in. The first is this `a` that has been ignored up until now. This `a` is a type variable that designates the type of elements contained within the list. For instance, if we are creating a list of ints, then the full type would be `List Int`, or if we were creating a list of booleans, then the full type would be `List Bool`. Coming back to the `Cons` constructor then, the `a` field is going to hold an element of the list, either an `Int`, a `Bool` or anything else. The final `List a` field, then, is how lists are created of an arbitrary length. This field holds another list of the same type, creating an explicit recursion. In this way, the `Cons` constructor can be used over and over to create a list of whatever length before it is eventually terminated by the `Nil` constructor. An example of our original list created using this data structure would be:

```
list1 :: List Int
list1 = Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))
```

3.1.2 Functions

Having explored the basic style of types in Haskell, the next fundamental structure is a function. In Haskell, functions are actually treated like values, such as `Int`s, and therefore can be passed around as arguments to other functions or have almost anything else done to them that is possible to do with normal values (such as creating a list of functions, for instance). The type and structure of a basic function in Haskell is:

```
successor :: Int -> Int
successor i = i + 1
```

In this example, the `successor` function is of type `Int -> Int`, or `Int` "goes to" `Int`. What this essentially means is that the function takes an `Int` values as a parameter, and then returns an `Int` as a result as well. In this case, the argument passed in is `i` and the returned value is `i + 1`.

A more complicated example of a function might be:

```
bar :: Int -> String -> String
bar 0 str = str
bar i str = newStr ++ show i
        where newStr = str ++ str
```

This useless code shows a couple new other features of Haskell. The most obvious is the `++` operator, which is simply an operator used to append strings or lists together. More notably, we see how multiple argument functions are done in

Haskell. A new type is simply inserted into the type signature followed, meaning that this function takes an `Int` and a `String` and returns a `String`. Functions in Haskell can only return one value, although this value can be a complex type, such as a tuple.

Another useful addition is the "where" clause, which is simply a way to name and store intermediate values. While this isn't particularly necessary in this case, it can be highly useful in more complicated functions. Perhaps one of the most useful additions seen in `bar` is the idea of pattern matching. Pattern matching is the matching of a variable to specific cases or patterns, which determines how to proceed. In this instance, if the input `Int` is a zero, then the first definition of `bar` is used, simply returning the old string. If, however, a different `Int` is passed in, then the second definition is used. In this definition, the string is concatenated with itself (in the `where` clause), and then with `show i`. `show` is a built-in function that turns the given `Int` into a `String` of that integer. As can be seen here, application in Haskell is done simply by placing the arguments after the function, with nothing but white space in between.

If this value of `i` is not needed in the current function, the `_` symbol can be used to designate that anything will match, but that we do not need to know what the value is. This could be used if the second definition was instead something like:

```
bar _ str = str ++ str
```

Before it was mentioned that functions can actually be passed around as arguments to other functions, or returned from those functions. This can done in the following way:

```
curryFn :: (Int -> Int -> String) -> Int -> (Int -> String)
curryFn fn i = let newFn = fn $ i - 1
                 in
               newFn
```

This code again has lots of new Haskell pieces in it. The first of these can be observed in the type signature. The `(Int -> Int -> String)` is the type of the function that is passed in. In addition, another `Int` value is passed in, and then finally, a function of the type `(Int -> String)` is returned.

Looking at the actual function, the first thing we see is the `let ... in` clause. This clause is pretty much the same as the `where` described above, except that values defined inside it can only be used after the `let`, in the expression following `in`. Values defined inside a `where`, on the other hand, can be used anywhere inside that function.

Next, the `$` operator used in this function is basically the same as putting parenthesis around `i - 1`. It is used to designate that the rest of the current line is all one argument, being passed to the function `fn`. Speaking of `fn`, this function is of the type `Int -> Int -> String`, and here we are passing a single `Int` to the function (`i - 1`), and then returning it. This is called "currying", or partial application of the function. We are passing one argument to the function now, and the other `Int` must be passed to it later in order to get the resulting `String`. In essence, the function is now storing the value `i - 1` inside it and only needs one more `Int` argument before it returns a resultant `String`, based on the two `Int`s.

We can sometimes increase the number of arguments to functions as well:

```
newArgs :: (Int -> Bool) -> (Int -> Int -> Bool)
newArgs fn = (\ x y -> fn (x*y ‘mod‘ 10))
```

In this example, a "lambda" function is used. A lambda function is essentially an unnamed function, or a function that isn't bound to a symbol, such as `newArgs`. In this case, by calling `newArgs` of a function of type `Int -> Bool`, we return a new function that takes two `Int` arguments instead. In this new function, we multiply the two `Int` arguments by each other, modulus by 10, and then pass the result to the original function, which finally returns a `Bool`.

Note the usage of the back-ticks around the `mod` function. If a function in Haskell takes two arguments, it can be used as an infix function instead of a prefix function by simply placing back-ticks before and after the name of the function. This can be useful in some instances to help make code more legible, or to use functions in the same way as we think about them day to day, such as with the `mod` function.

3.1.3 Classes and Useful Monads

Other useful features of Haskell that are utilized in this thesis are "classes" and "monads." Without getting too much into the implementation details and how they work, let's first look at some things that monads allow us to do in Haskell.

The first of these monads is the IO monad. The IO monad allows a purely functional program to still have effects, such as interactions with the user. It also allows functional programs to use "`do`-notation" in order to provide a different style of syntax/structure that appears to be more imperative and stateful. A typical usage of the IO monad may be:

```
main :: IO ()
main = do
        str <- getLine
        print str
```

This code sample simply reads a line of text from the terminal into a string, and then prints that string back out to the terminal. Notice the sequential ordering of commands inside the `do`-notation, unlike traditional Haskell where a function is a single expression. This sequence of commands is actually still turned into a single expression and remains purely functional, but the `do`-notation and IO monad allow the user to program imperatively like this when necessary.

Other useful monads include the State monad and the Writer monad. As the name would suggest, the State monad simply allows for a state to be maintained throughout the monad, so that certain variables can be accessed from any function that remains inside that State monad. In reality, a simplified explanation of how this monad works would be that it just passes the state around as a hidden argument to each function inside the monad. In this way, the current state can be accessed from anywhere inside the monad.

The Writer monad is similar, but instead an *accumulation* of certain "write" commands is maintained. For example, if we wished for some function(s) to accumulate a list of `Int`s, we could use the Writer monad to do this. Each time we add to the list, all subsequent calls and/or functions will now take the new list and add more elements from there. We can then retrieve this list of `Int`s after these functions have completed. The Writer monad just supplies a simpler means to do this than by passing around our own list explicitly.

Moving on from monads, the final Haskell topic covered here that might be useful in reading this thesis is the topic of classes. This includes both classes that are built into Haskell, and creating our own new classes.

Classes in Haskell, generally take the form of:

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

While the Functor class is already defined in Haskell, user-created classes are also declared in this same way. What this class does is make a claim about whatever type is filled in for f. It says that for each f type that this class applies to, the following functions must be implemented (in this case, fmap). What this does is allow for all members of the class, Functor, to have some similar properties, namely that the function fmap applies to them in a given way. This can be useful in creating certain polymorphic functions, as described a bit later.

First, however, let's look at how a type f that this class applies is declared. This association of a given type with a given class in Haskell is called an "instance." A basic instance of Functor for our lists we created earlier would be:

```
instance Functor (List a) where
    fmap _  Nil        = Nil
    fmap fn (Cons e l) = Cons (fn e) (fmap l)
```

This code again uses the pattern-matching feature of Haskell discussed earlier. It says that for a given element of the list that is not the end (Cons constructor), the function being fmapped (fn) will be applied to the element (e), and this same mapping will be done for the rest of the list as well, recursively (fmap l). Once the end of the list is reached (Nil), the recursion will stop. In essence, all fmap does is apply the given function to each element in the given list. This can be seen in the following example:

```
squares = fmap sq list
    where list = [1,2,3,4]
          sq i = i*i
```

The squares function in this case will produce the list [1,4,9,16], applying the sq function to every element of the list. This example is obviously done with the built-in lists, but behaves exactly the same. In fact, the built-in lists are already an instance of the `Functor` class (though the `map` function performs the example same task, explicitly for lists only). This brings up an interesting point. In what situations are classes like this actually better than just a single definition for a certain type we are interested in?

One example of such an instance would be the following:

```
squareAll :: (Num a, Functor f) => f a -> f a
squareAll functor = fmap (\ x -> x*x) functor
```

Before we get to why this function is useful, there is first a new aspect of how classes work to be discussed. This is the class constraints that are placed on the `squareAll` function. The `(Num a, Functor f) =>` part of the definition says that `a` and `f` are type variables, but that `a` must be an instance of the `Num` class, and that `f` must be an instance of the `Functor` class. These constraints guarantee that the functions `*` and `fmap` will, respectively, apply to the given types `a` and `f` whenever the function is used. Without these constraints, there would be no way to know that the `fmap` function would apply to the structure we pass in.

Now that the types of the function have been sorted out a little bit, let's look at what it actually does. The function takes in some kind of `Functor` that contains elements that are instances of `Num`. Because the variable `functor` must be a `Functor`, the `fmap` function can be applied to it, and because the elements inside `functor` must be instances of `Num`, the `*` operation can be used on them as the lambda function is mapped over each element. This function, therefore, squares every element contained inside the functor. This functor could be a list, as defined above, or it could be a tree, a graph, or any other structure that holds elements. The elements inside this structure can also be of type `Int`, `Float`, `Double`, or any other type that is an instance of the `Num` class. This function can be used on any of these data structures in order to accomplish the same task, squaring every element contained inside a functor.

This type of polymorphism is a powerful feature of Haskell. It allows for simple functions such as the one above to be created once and applicable to many different types and structures. Any class can be constructed as needed, and many functions and libraries created to utilize these classes to make efficient, general code.

Other classes used prominently in this thesis besides `Functor` include `Foldable`, `Applicative`, and `Traversable`.

`Foldable` is another basic class, like `Functor`. Instead of applying a function to each element of a structure in place, however, `Foldable` combines the elements of a structure together. For instance, if we wish to sum together all the numbers in a structure, `Foldable` would let us do this. We are combining the elements together, using the `+` operator to create one element out of two. In this way, each number of the structure can be added to the current number, finally resulting in a total. The `Foldable` class allows these type of combining functions to be used over a structure, resulting in one output element (and throwing away the structure).

The `Applicative` class basically allows us to hold and apply functions within a functor (`Functor` with application). What this means is that we could have different functor elements that are of general types `f a` and `f (a->b)` and use them together to get an `f b`. With `Applicative`, we can both hold functions inside a functor (`f (a->b)`, etc) and apply those functions while staying inside the functor (using `f (a->b)` to change `f a` to `f b`).

The `Traversable` class is a little bit more complicated, combining parts of all the previously mentioned classes (`Functor`, `Foldable`, and `Applicative`), but for now let's just say that it allows us to traverse a data structure while also applying actions at each different element. This allows us to perform actions that might normally only be allowed in the IO monad, mentioned above, using the elements of our data structure as input (with `Functor`, we could only modify the contents inside the functor, not perform actions on them).

3.2 OpenGL

OpenGL also plays a role in ChalkBoard, and therefore some knowledge of this system as well as basic concepts in computer graphics is helpful. Thankfully, not a lot of OpenGL experience is needed to understand this thesis, however, as the OpenGL implementation details are not required to comprehend most of the concepts and contributions presented.

The first concept that may be helpful in reading this paper is how images are represented and displayed on computers. Digital colors are often thought of as having three main components: a red value, a green value, and a blue value. Using these three color values, or RGB, most colors can be recreated. Often these values will be either in the 0 to 255 range or the 0 to 1 range. In either case, lower numbers for a value signify that less of the color is used, while higher numbers signify that more of the color is used.

Images are then represented by two-dimensional arrays of these RGB color values, where each array element represents a either pixel or the smallest discrete dot of an image. These arrays are sometimes used directly to store images (such as in the PPM image format), and in general are just how digital images are often thought of.

Sometimes a fourth value is added to this set, however. This value is called an "alpha" value. What alpha represents is the transparency of an image, with 0 being completely transparent, and 1 or 255 being completely opaque. This value is exceedingly useful in representing objects such as glass or water, or placing one image on top of another. If parts of an object or image in the foreground are partially transparent, then the resulting color at a pixel may be influenced by objects or images further back in the scene. There will be examples of this throughout the thesis.

Some image formats support the saving of images with alpha values (such as PNG), and others do not (such as PPM). All *displayed* images, however, must have some RGB representation. This is because transparency is not a color, and therefore monitors cannot represent it. Transparency is simply a property that may affect the final color of light as it travels towards the eye. Displaying images with transparency is usually achieved, then, by adding an opaque background of white, black, or a checkerboard (often white and gray). The final image using this method will have an RGB color, but the effect of transparency can still be seen.

The final computer graphics topic to be discussed has a lot more to do with OpenGL itself. It is the topic of textures. In OpenGL, a texture is an array of color, as described above, that is pinned to an object. Take, for instance, the drawing of a wall in OpenGL. The wall itself can be constructed simply of polygons, placed next to each other in the shape of a wall. These polygons can have certain color properties and be displayed easily by OpenGL, allowing us to see our wall on the screen. What if we want this wall to be a brick wall, however? While color properties of polygons can be described in OpenGL, there is no magic button to make a polygon look like a brick wall, instead textures can be used.

Textures are basically images themselves. In our brick wall example, if we had a picture of a brick wall, or a small pixel art image of a brick pattern, we could map these images onto our polygons in order to make a much more convincing brick wall. These textures are basically pasted onto the polygon in order to create the effect of the image being part of the polygon itself. OpenGL allows us to tile these images across the polygon, stretch them, or many other cool things. While this is not exactly what textures are used for in ChalkBoard, the same mechanics and methods are still used.

Chapter 4

ChalkBoard

4.1 Functional Image Generation

As a first example of using ChalkBoard, consider drawing a partially-transparent red square over a partially-transparent green circle. The image we wish to draw looks like Figure 4.1.

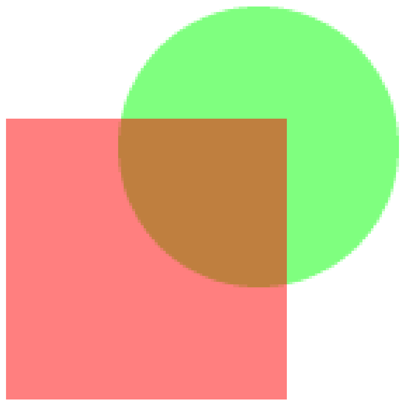

Figure 4.1. Square over Circle

This image can be described using the following ChalkBoard specification:

```
board = unAlphaBoard (boardOf white) (sq1 ‘over‘ cir1)          1
  where                                                         2
    cir1 =  move (0.2,0.2)                                      3
         $  choose (withAlpha 0.5 green) transparent            4
        <$> circle                                              5
    sq1  =  move (-0.2,-0.2)                                    6
         $  choose (withAlpha 0.5 red) transparent              7
        <$> square                                              8
```

This code snippet specifies each aspect of the image. A circle (line 5) is colored green with 50% transparency and placed on top of a transparent background (line 4). This circle is then moved slightly up and to the right (line 3). A square (line 8) is colored red with 50% transparency placed on top of a transparent background as well (line 7). This square is then moved slightly down and to the left (line 6). Finally, the red square is placed on top of the green circle and all transparency is removed by placing the scene on top of an opaque white background (line 1).

In order to understand the ChalkBoard language, we need to first think about types. In ChalkBoard, the principal type is a `Board`, which represents a two-dimensional plane of values. A color image is therefore 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 inside a region or outside a region) can be denoted using `Board` of `Bool`. Table 4.1 lists the fundamental types of `Board`s used in ChalkBoard.

Generally, image creation in ChalkBoard begins with using regions (`Board Bool`) to describe basic shapes. ChalkBoard supports unit circles and unit squares, as well as rectangles, triangles, and other polygons.

`Board RGB`	Color image
`Board RGBA`	Color image with transparency
`Board Bool`	Region
`Board UI`	Grayscale image
`type UI = Float`	Values between 0 and 1
`type R = Float`	Floating point coordinate
`type Point = (R,R)`	2D Coordinate or point

Table 4.1. Boards and Type Synonyms in ChalkBoard

The primitive shapes provided in ChalkBoard 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
```

To construct a basic color image in ChalkBoard, a color can be mapped over one of these regions. Typically, this color image will have the area outside the region be completely transparent, and the area inside the region be a specific color. This mapping can be done using the `choose` combinator, and the `<$>` operator:

```
        choose (alpha blue) transparent <$> circle
```

In this code segment, we choose `alpha blue` to replace the `true` values inside the region, and `transparent` to replace the `false` values outside the region. In this way, a blue circle is created on a transparent background. The `alpha` function lifts `blue` (normally an `RGB` value) into the `RGBA` space, giving it an alpha value of 1, or completely opaque. 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
```

`choose` is a `bool`-like combinator that we partially apply, and `<$>` is an `fmap`-like combinator. The type `O a` is an *observable* version of `a`. Although `O` is actually an abstract type, it can be considered to have this trivial definition:

```
data O a = O a   -- working definition; to be refined.
```

This type for `O` will be redefined and its implementation examined in Section 4.5.

For the user, ChalkBoard provides all point-wise functions and primitives already lifted over `O`. For example, the colors, and functions like `alpha`, have the types:

```
red                     :: O RGB
green                   :: O RGB
blue                    :: O RGB
transparent             :: O RGBA
alpha                   :: O RGB -> O RGBA
```

Another important feature of a modularized system like ChalkBoard is the ability to combine images together. Our boards of `RGBA`, or images with transparency, can be combined (overlaid) into new boards of `RGBA`:

```
        (choose (alpha blue) transparent <$> circle)
                            `over`
        (choose (alpha green) transparent <$> square)
```

The `over` combinator is used here to lay one `Board` on top of the other. It's type is simply:

```
over :: Board a -> Board a -> Board a
```

These boards of `RGBA` can also be transformed into `Board RGB`, or true color images without transparency. This translation is done using the `unAlphaBoard` function and any default background board (here we choose a blank, white board):

```
   unAlphaBoard (boardOf white) $
      (choose (alpha blue) transparent <$> circle)
                             `over`
      (choose (alpha green) transparent <$> square)
```

unAlphaBoard removes the alpha (transparency) component of a Board RGBA by laying it on top of an opaque (RGB) background board. This leaves a Board RGB image without transparency.

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

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

Finally, ChalkBoard also contains a primitive for constructing a (conceptually infinite) Board of a constant value, which has the type:

```
boardOf :: O a -> Board a
```

With all of these features, ChalkBoard users can begin to construct images by combining primitives and translating them both in *space* and *representation*, ultimately building a Board RGB to be displayed. ChalkBoard does contain many other helpful features that we don't have time to explore in this thesis, however, such as importing of images as Board RGBA and font support.

4.2 An Example: Drawing Lines

Now that we have our fundamental primitives and combinators, we can begin to build more interesting, complex combinators. A straightline combinator can be constructed, for instance, which takes two points and a width value, using them to construct a line region:

```
straightline :: (Point,Point) -> R -> Board Bool
straightline ((x1,y1),(x2,y2)) width =
          move (x1,y1)
        $ rotate (pi /2 - th)
        $ rectangle ((-width/2,0),(width/2,len))
  where
          (xd,yd)  = (x2 - x1,y2 - y1)
          (len,th) = toPolar (xd,yd)
```

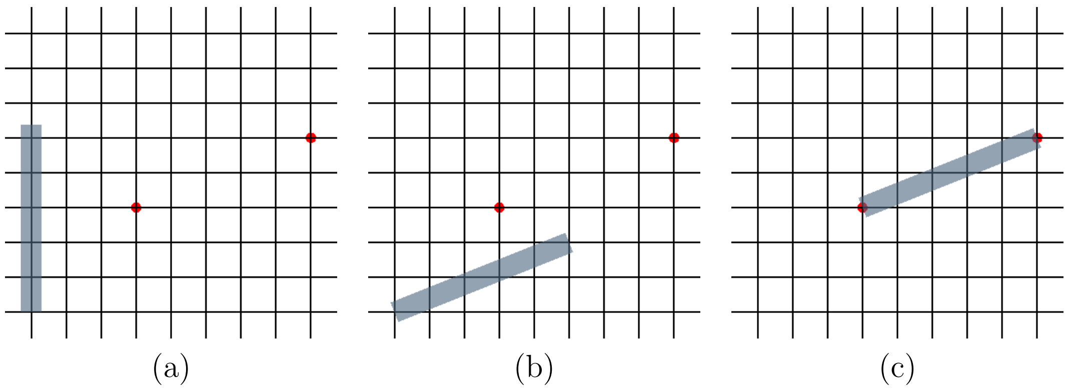

Figure 4.2. How `straightline` works

Figure 4.2 shows how the `straightline` function works. Assuming the dots are the start and end points of our line, and the bottom left intersection is (0,0), we can draw a rectangle of the right size and width (a), rotate it (b), then move it to the correct location (c). There are other ways that `straightline` could be written, but this way is compositional, a style that works well in ChalkBoard.

Now that we can draw lines of arbitrary thicknesses between arbitrary points, curved lines can also be simulated using many of these straight segments together. To do this, the `outerSteps` function is used, which counts a specified number of steps between 0 and 1, inclusive:

```
> outerSteps 5
[0.0,0.2,0.4,0.6,0.8,1.0]
```

The result in this instance is a set of 6 values, representing the 5 steps of size 0.2 between 0 and 1. Using `outerSteps`, a function that represents a curved line can be *sampled*. The curved line is emulated by drawing straight lines between each pair of sample points. Any infidelities at the joints of these lines are filled in with small dots the size of the width of the lines:

```
functionline :: (UI -> Point) -> R -> Int -> Board Bool
functionline line width steps = stack
                [ straightline (p1,p2) width
                | (p1,p2) <- zip samples (tail samples)
                ] ‘over‘ stack
                        -- not the first or last point
                [ dotAt p | p <- tail (init samples) ]
    where
        samples = map line (outerSteps steps)
        dotAt p = move p $ scale width circle
```

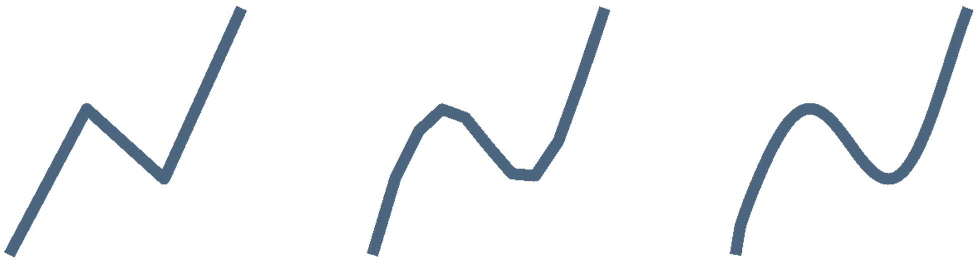

Figure 4.3. Examples of `functionline`

Figure 4.3 gives an example of using `functionline` on a function with 3, 10, and 50 straight line segments. The figure clearly shows how with a higher number of samples, the quality of rendering the curved line improves.

Although presented here as an example of the style of ChalkBoard, both of these functions are actually already defined in the ChalkBoard library. They were chosen for this example because they clearly show how ChalkBoard compromises between continuous boards, and discrete components on these boards. Collectively, these ChalkBoard combinators and the many others like them give a clean and productive system for scripting images.

4.3 Considerations in Compiling ChalkBoard

ChalkBoard has a relatively simple semantic model. A `Board` of α is a field of α-values over $\mathcal{R}^2$, where $\mathcal{R}^2$ is a floating point coordinate for two dimensions.

$$\texttt{Board}\ \alpha = (R, R) \rightarrow\ \alpha$$

This is the same model used in Pan [6], on which the ChalkBoard language is based. Pan uses this model of a function directly to implement a `Board`-like object. Although ChalkBoard shares this same semantic model, however, we desire a different implementation.

In ChalkBoard, a `Board` is *abstract*, specifically to allow the possibility of our rendering optimizations. In Pan, the equivalent of `Board` is implemented as an explicit function, directly guided by the semantic model. Our choice of abstraction limits the language to using only the built-in combinators for `Board` transformations. This is a definite restriction, especially when compared to the full expressiveness of `Pan`, but there are almost always ways of accomplishing the same tasks, given our combinators, or certain tricks that can be played when there are not.

Our choice does allow for more rendering efficiency than the original functional representation, however. ChalkBoard is intended as a system for constructing complex images, consisting of perhaps tens of thousands of individual components.

The original functional representation normally prevents this from being efficiently rendered, though techniques like the worker/wrapper transformation [13] could perhaps be used to translate an explicit function into something like our abstract representation.

Consider the ChalkBoard image way back in Figure 4.1. This example can be used to illustrate a number of the challenges in optimizing a chalkboard image specification. Figure 4.1 was generated by first building a `Board Bool` for each of the two basic shapes and using the `<$>` operator on each board to convert it into a `Board RGBA`. `move` was then used to move the boards to the desired locations. Finally, the two boards are overlaid, using `over`, and the alpha channel is removed for rendering the scene as a color image.

The basic plan of attack is to augment the representation of a `Board` internally, replacing the shape primitives with more complex information about what is being rendered, and attempting to translate our tree of operations into the drawing of polygons. Rendering polygons is something OpenGL does extremely efficiently, so this translation will hopefully be able to lend the power of OpenGL to our system.

4.4 Capturing the ChalkBoard DSL

ChalkBoard is a language for describing boards, which are similar to functors. The language provides mechanisms for describing the creation of boards, using spatial transformations and applying a functor-style `map`. In this section, we consider how to express all three of these aspects with a deep embedding of the ChalkBoard DSL.

Constant boards are captured using a `Constant` constructor, inside `Board`:

```
data Board a where
        Constant :: O a -> Board a
        ...
```

Here the GADT [27] syntax is used for `Board` because of the ability to declare constructors that are specialized to monomorphic instances, and each of the principal constructors is presented individually.

Primitives shapes, like circles and squares, are regions, or `Board Bool`. They are represented inside the `Board` data structure by a list of points, which mark a convex boundary around the region. For these shapes, the `Polygon` constructor is used:

```
data Board a where
        Polygon :: (...) -> Board Bool
        ...
```

Representing squares or like shapes in this style is easy, using the corner points, but what about the points around a circle? There are infinitely many points on a circle. That is, there are infinitely many points (x,y) that solve the equation:

$$\sqrt{x^2 + y^2} = r$$

Graphical rendering systems approximate circles using a small number of these points on a small circle and a larger number of them on a large circle. At this point, we appeal to a common idiom of functional programming and *defer* the decision of how many points to use in approximating a circle with a simple function. The full type of the `Polygon` constructor is therefore:

```
data Board a where
        Polygon :: (Float -> [Point]) -> Board Bool
        ...
```

The `Float` argument to this function is the "resolution" of the final polygon. Specifically, it is an approximation of how many pixels a line of unit length should affect. Knowing this, the ChalkBoard functions of `square` and `circle` can both now be defined:

```
square :: Board Bool
square = Polygon (const [(-0.5,-0.5),(-0.5,0.5),(0.5,0.5),(0.5,-0.5)])

circle :: Board Bool
circle = Polygon $ \ res ->
        let ptcount = max (ceiling res) 3
        in [ (sin x/2,cos x/2)
           | x <- map (* (pi/(2 * fromIntegral ptcount)))
                      (take ptcount [0..])
           ]
```

`sin` and `cos` are used in `circle` to find the `x` and `y` points on a unit circle (after scaling), and the number of points is dictated by the size of the final circle. The point count formula used here generates reasonable images, but remains open to further tuning. In addition, we may consider using the OpenGL functions to generate circles in the future, offloading this fine tuning onto the experts.

Spatial transformations are also handled using a single `Board` constructor, which combines all the relevant information:

```
data Board a where
        Move   :: (Float,Float) -> Board a -> Board a
        Scale  :: (Float,Float) -> Board a -> Board a
        Rotate :: Float         -> Board a -> Board a
        ...

move :: (Float,Float) -> Board a -> Board a
move (x,y) = Move (x,y)

scale :: Float -> Board a -> Board a
scale w = Scale (w,w)
```

```
rotate :: Radian -> Board a -> Board a
rotate r = Rotate r
```

Finally, we also have our functor map (or `fmap`) like operators. Consider the following attempt at a `fmap` constructor:

```
data Board a where
        Fmap :: forall b . (b -> a) -> Board b -> Board a -- WRONG
        ...

(<$>) = Fmap
```

This constructor could be used to successfully *typecheck* a ChalkBoard-like language, but we run into problems when trying to walk the `Board` tree during compilation. Here, `b` can be any type; the type information about what it was has been lost. An `<$>` over a `Board Bool` will have completely different behavior than an `<$>` over a `Board RGB`. When walking the tree and performing our attribute grammar interpretation, we get stuck.

This problem is addressed in ChalkBoard by assuming a pointwise function is a function over our observable type, `O`, giving the corrected:

```
data Board a where
        Fmap :: forall b . (O b -> O a) -> Board b -> Board a
        ...

(<$>) = Fmap
```

ChalkBoard requires the `O` type to hold runtime type information, as described in Section 4.5. It may first appear that using `O` just postpones the problem, and does not solve it. By forcing pointwise manipulations to be expressed in the `O` world, however, the user's intention can be observed without requiring that every function be translated into a `Board` equivalent.

With these constructors, We can now construct basic abstract syntax trees for ChalkBoard, using the `Board` data type. For example:

```
scale 2 (choose red green <$> square)
```

This ChalkBoard specification represents a scaled version of a red square on a green background. It constructs the `Board` tree:

```
Scale (2,2) (Fmap (...) (Polygon (...))
```

The specific polygon points contained inside the `Polygon` constructor can be retrieved when compiling for OpenGL because we know the size of the image in context. The challenge, then, is how to extract the first argument to `Fmap`. In order to do so, the observable type, `O`, must be used.

4.5 `O`, the Observable

The data type `O`, nicknamed *the observable*, is a mechanism used to observe interesting values. The idea is that an observable can simultaneously have both a shallowly and deeply embedded interpretation of the same expression. The shallow interpretation can be used to directly extract the value of any `O` expression, while the deep interpretation can be examined to find out how that result was constructed. Specifically, the data definition for `O` is:

```
data O a = O a E        -- abstract in ChalkBoard API
```

`E` in this definition is a syntax tree of possible `O` expressions. In limiting the ways of building `O`, we allow `O` expressions to only be constructed out of primitives we know how to compile. In ChalkBoard, `E` has the definition:

```
data E = E (Expr E)
data Expr e
       = Var Int        | Lit Float       | Choose e e e
       | O_Bool Bool    | O_RGB RGB       | O_RGBA RGBA
       | Alpha UI e     | UnAlpha e e
       ...
```

Implicit recursion is used inside `Expr` so that this functor-style representation of `Expr` can be shared between the expression inside `O` and the compiler, for reasons explained in [12].

As a basic example of how primitives are constructed, `choose` is defined as:

```
choose :: O a -> O a -> O Bool -> O a
choose (O a ea) (O b eb) (O c ec) = O (if c then a else b)
                                      (E $ Choose ea eb ec)
```

Primitive `O` values can also be built using the `Obs` class:

```
class Obs a where
         o :: a -> O a
```

Only instances of `Obs` can construct objects of type `O a`. ChalkBoard uses this class to provide a means of taking a value of `Bool`, `RGB`, `RGBA`, or `Float` and lifting it into the `O` structure using the `o` function. In many ways, this is similar to a restricted version of `return` for monads, or `pure` for applicative functors [24].

So how is a function actually observed? This is done by first giving a dummy argument and then observing the resulting expression. The `Expr` type above contains a `Var` constructor specifically for this purpose. If we take the example:

```
choose red green :: O Bool -> O RGB
```

We can pass in the argument '`O ⊥ (Var 0)`' to this function, and get the result:

```
O ⊥ (E (Choose
        (E (O_RGB (RGB 1 0 0)))
        (E (O_RGB (RGB 0 1 0)))
        (E (Var 0))))
```

The structure of the `choose` and the arguments are completely explicit. Using this trick, the function argument to the functor can now be observed because the `Fmap` constructor of `Board` requires the argument and result type to both be of type `O`. Ignoring the type change between the function argument to `Fmap` and its tree representation, our earlier example from the end of Section 4.4 can be parsed into:

```
Scale (2,2) (Fmap (E (Choose
                        (E (O_RGB (RGB 1 0 0)))
                        (E (O_RGB (RGB 0 1 0)))
                        (E (Var 0))))
                   (Polygon (...))
             )
```

This result can now be translated into the ChalkBoard Intermediate Representation.

4.6 The ChalkBoard IR

Both inspection of the `Board` object itself and observation of `O` structures are used to construct our ChalkBoard abstract syntax tree. From here, compilation is a matter of implementing the attribute grammar interpretation over this tree in a way that leverages OpenGL's polygon rendering abilities. These trees are translated by the compiler into ChalkBoard Intermediate Representation (CBIR), and then the CBIR is interpreted on the fly into OpenGL commands.

Figure 4.4 gives the syntax of the CBIR. There are two main commands:

- **allocate**, which allocates a new, fixed-sized buffer in graphics memory.
- **buffersplat**, which takes a specified polygon from one buffer and renders it onto another buffer.

Statement	stmt	::=	`allocate` dest (x,y) back	Allocate Buffer
		|	`buffersplat` dest src pointmaps	Splat Texture
		|	`colorsplat` dest col points	Splat Color
		|	`delete` src	Deallocate
		|	`save` src filename	Write to file
		|	`exit`	
Background	back	::=	col	Background Color
		|	Ptr	Pointer to an Image
Color	col	::=	RGB	RGB Constant
		|	RGBA	RGBA Constant
	dest,src	::=	buffer-id	
	pointmap	::=	(point,point)	
	pointmaps	::=	pointmap$_1$, pointmap$_2$, …, pointmap$_n$	$n \geq 3$
	point	::=	(u,v)	
	points	::=	point$_1$, point$_2$, …, point$_n$	$n \geq 3$
	x,y	::=	int	
	u,v	::=	float	

Figure 4.4. ChalkBoard Intermediate Representation

buffersplat takes a source buffer, a destination buffer, and a sequence of point maps, each of which is a mapping from a point on the source board to a point on the destination board. This mapping capability is both powerful and general. Among many other things, it can be used to simulate scaling, translation, or rotation. This is the command that does the majority of the rendering work inside ChalkBoard.

As well as the two principal instructions, there are also commands in the CBIR for deallocation of buffers, saving buffers to disk images, and **colorsplat**, a specialized version of **buffersplat** where the source is a single color instead of a buffer.

4.7 Compiling ChalkBoard to ChalkBoard IR

When compiling ChalkBoard, the AST is traversed in much the same way as the attribute grammar example above, but instead of passing in the inherited attribute (x, y) many times, we only walk over the graph *once*, providing as inherited attributes:

- a basic "quality of the required picture" argument, which in effect tells a `Board` how much it is contributing to the final image;
- any rotations, translations, or scaling performed *above* the current node;
- and an identifier for a previously-allocated *target* buffer.

A set of compilation schemes for each `Board` type are then used. In general, the following compile steps are performed:

- `Constant Board`s are compiled into a single CBIR splat onto the target board.
- `Move`, `Scale`, and `Rotate` update the inherited attribute context, recording the movement required, and then the sub-board is compiled with this new context.
- `Over` causes its child boards to be interpreted according to the type of `Board`:
  - For `Board Bool` and `Board RGBA`, `Over` draws the back (second) board, and then draws the first board on top of it.
  - For `Board RGB`, `Over` simply compiles the first board (since it is opaque).

- For `Fmap`, the type of the map is inferred by observing the type of the functional argument to `Fmap`, using the capabilities provided by `O`. The bridging code for the `Fmap`, compiled from the reified functional argument, is then emitted and the relevant compilation scheme for the sub-board is called.

The compilation scheme for a `Board Bool` has one extra inherited attribute, the colors to use for `True` and `False` values. The primitive `Polygon`, which is always of type `Board Bool`, is translated into a **colorsplat** of this `True` color onto a backing board that is initialized to the `False` color.

The key to the compiler is the compilation of an `Fmap` that translates a `Board Bool` into a `Board RGB` (or `Board RGBA`). For example:

```
( Fmap f (Polygon (...) :: Board Bool) ) :: Board RGB
```

`f` in this example has the type `O Bool -> O RGB`. To compile the inner `Board Bool` syntax tree, the `True` (or foreground) color, and `False` (or background) color must be computed. In order to find these colors, `f` is simply applied to `True`, and then to `False`, giving the two colors present on the board.

4.8 Interpreting ChalkBoard IR

The ChalkBoard IR is interpreted by the ChalkBoard Back End (CBBE). This CBBE is ran in a separate thread from the rest of ChalkBoard. After it has been initialized, it waits on an `MVar` (a type of concurrency “mailbox” used in concurrent Haskell programs) for lists of CBIR instructions from the compiler. These CBIR instructions are then expanded and executed inside OpenGL. After these instructions are executed, a specified final board is printed out onto the screen by the CBBE. A new set of instructions can then be passed to the CBBE in order to repeat the process. Any of the created boards can also be saved to a file using the `save` CBIR instruction.

The concept of a `Board` in ChalkBoard translates roughly into an OpenGL texture inside the CBBE. For each new buffer that is allocated in the CBIR instructions, a new OpenGL texture is created in the CBBE. These new textures can have a variety of internal formats based on the color depth needed by the board (Luminance, RGB, or RGBA) and either have an initial color specified by the CBIR instruction or an initial image that is read in from an image file.

These textures can then be texture-mapped onto one another in order to create the effects of `buffersplat` in the CBIR. The preferred way to do this is using the OpenGL Framebuffer object, or FBO. The FBO saves a lot of overhead by allowing images to be rendered straight into a texture instead of onto the screen, from which images would need to be copied back into a texture. When splatting one board onto another, the back or destination texture is attached to the current color attachment point of the FBO, and then the front or source texture is simply texture-mapped on top of it using the `pointmaps` specified in the CBIR instruction. The resulting image is automatically rendered into the destination texture. There is no additional copying necessary because the effects have already been stored in the destination texture directly.

To support older graphics cards and drivers, an alternate method to using FBOs is also implemented. This alternative method is not as well supported, however, so its usage should try be avoided if possible. The main difference between the methods is that drawing in the alternative method must be done to the default screen Framebuffer and then copied back out into the appropriate destination texture using `glCopyTexImage`. Because the CBBE use OpenGL's double buffering, the images are drawn to the buffer and then copied out without ever being swapped onto the screen. In this way, the actual step-by-step drawing is still invisible to the user but will take considerably longer than when using FBOs because the resulting image structure must be copied back out into the destination texture.

As an example of the performance difference between the two methods, a small micro-benchmark, called ChalkMark, was written to stress test `splatbuffer` by rendering 5000 triangles onto a buffer 100 times. When running ChalkMark on a OSX 10.5 with an NVIDIA GeForce 8600M GT running OpenGL 2.0, the CBBE currently achieves about 38,000 `splatbuffer` commands per second when using an FBO, versus about 11,000 `splatbuffer` commands per second when using the alternative `glCopyTexImage`. Even as the CBBE is further tuned, we expect the difference between the two methods to remain this significant. Thankfully, most systems in use today should have graphics cards with FBO support, with the `glCopyTexImage` method providing only backwards compatibility.

The `colorsplat` CBIR instruction also uses these two methods for its implementation in the CBBE. It works in much the same way as the `buffersplat` instruction except that a simple, colored polygon is drawn onto the destination texture instead of mapping a second texture onto it. This removes the needless overhead of allocating extra 1x1 textures just to use as basic colors.

Although there remains considerable scope for optimizing the use of the OpenGL pipeline, and many improvements could be made to our ChalkBoard compiler as well, the current performance of ChalkBoard has been more than sufficient for rendering simple animations in real time.

Chapter 5

Active

5.1 The Active Language

The conceptual framework behind the Active language is that all animations have a beginning, a middle, and an end. The language treats the time component of an animation as a value that increases from 0 to 1. Before each animation begins, its time value is considered to be 0. When the animation starts, the time value begins to increase from 0 to 1. Finally, once the animation has finished, its time value will remain 1. This idea of having a pre-life value of 0 and a post-life value of 1 enables animations to have a specific start and end state that occurs for these values.

This system also allows us to have separate time progressions for separate animations, instead of just one global progression. Each independent animation can be built assuming it has its own, unique 0 to 1 time progression. These individual animations can then be composed together. For instance, they could be combined to occur at the same time, one right after the other, or with times explicitly given by the user but independent from the creation of the animation.

The timing relative to other animations is kept completely abstracted from how an animation actually behaves. This is because each animation function will always be given time values between 0 and 1. The only things that change are *when* the program starts to give it values above 0 and *how many* values within this range it is given (50, 100, 1000, etc). Abstracting the global timing of an animation in this way allows for much greater flexibility. Animations can be created once and then used in multiple places within a program, or at different animations speeds. Animations can also be constructed compositionally by building up smaller pieces to be combined together in interesting ways.

The implementation of the Active language accomplishes this time abstraction using the `Active` data type and a few primitive functions. The `Active` data type is defined as:

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

For the `Active` constructor, the two `Rationals` are used to hold timing information, the start and stop time of the current object. The function takes a `Rational`, representing time values, and returns an object corresponding to that time value, such as a ChalkBoard `Board` for an animation. While generally the type of an `Active` used for animation is `Active (Board RGBA)`, any type can actually be used (to generate values of that type over time). Getting back to the definition, the `Pure` constructor is how objects that should remain constant are represented in the `Active` world. If an object doesn't change over the course of a program, it is stored as `Pure`.

Active is also an applicative functor, which, among other things, is particularly helpful in applying multiple animation functions to the same initial board all at once. This ability allows, for instance, an object to be moved over time while simultaneously being scaled (made larger or smaller). In theory, any number of animation functions can be applied to a board like this, though obviously there are practical limits. Active being an applicative functor is also useful in creating combinators and predefined functions, as can be seen in Section 5.2. The instance definitions for Functor and Applicative are given below:

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

instance Applicative Active where
    pure a = Pure a
    Pure a <*> b = fmap a b
    Active start stop f <*> Pure a = Active start stop (\ i -> (f 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))
```

Given these instances and the datatype above, primitive functions can now be created to use these structures. One of the first and most primitive members of 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 is actually a simple Active object, with a start time of 0 and a stop time of 1. This Active also stores a basic function, which takes a Rational, does some error checking, and returns it as a UI. A UI in ChalkBoard is simply a type synonym for `Float`, but is used to represent only values on the interval [0,1]. Because ChalkBoard uses the `UI` type, and all values returned should be between 0 and 1, the function stored within `age` is made to return a `UI` by giving `age` the type annotation of `Active UI`.

Using `age` is the primary method of creating an `Active` object. Once an `Active` has been created using `age`, all that must be done to create a basic animation is `fmap` a function over it. This function can do just about anything, as long as it takes a `UI` parameter, representing time. For example, a simple function for animating a ChalkBoard `rotate` over time could be written as:

```
-- A simple Active rotate (not actually implemented this way)
activeRotate' :: R -> Board a -> Active (Board a)
activeRotate' radians brd = fmap (\ui -> rotate (ui*radians) brd) age
```

This function takes a `Board a` and an amount of radians to rotate this board by over time. It uses the `UI` from `age` as a percentage of how far to rotate a given frame towards the final goal of `radians` (because time goes from 0 to 1 once the animation begins). Because the lambda function used in the `fmap` over `age` returns a `Board a`, the type of the `Active` has changed from `UI` to `Board a`. This function, therefore, produces an `Active` that returns images, and can successfully be played back as an animation. Playback is described at the end of Section 5.3.2.

So why isn't this definition used to implement rotation-over-time in the Active library? This will be discussed more shortly, but the main reason can be found in the type of `activeRotate'`. `activeRotate'` takes a `Board`, but produces an `Active`. Often in animation, many transformations, as well as other types of animation, are performed at the same time. If an `Active` is created that scales an object, for instance, and that object should also rotate at the same time, `activeRotate'` cannot be used. This function only allows creation of a new

Active, not additions to an existing Active. While two sets of functions could be constructed to accomplish each of these tasks separately (creation vs. addition), this would be a waste of the advantages gained from using an applicative functor. The cleaner system, using these applicative functor properties, is described in Section 5.2.

Looking at the activeRotate' function again, the abstraction the Active DSL provides can begin to be seen. Notice how the creation of this animation is completely independent from any timing that will eventually be applied to it. This same function can be used to create rotations that take 1 second or 100 seconds, at the beginning of a program or the end. The timing is applied to each Active object later, using either basic functions or built-in combinators.

Two of these primitive Active functions for handling timing effects are scale and mvActive:

```
instance Scale (Active a) where
    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)
```

When applied to an Active object, scale will stretch or shrink the animation time of the object. This can be used to make certain animations longer or shorter. mvActive, on the other hand, is used for translating time values. When applied to an Active object, mvActive can move an animation forwards or backwards in time, with regards to the rest of the scene. This function, therefore, can be used to put parts of an animation in order, or to begin animations at slightly offset times.

Two of the last basic `Active` functions are `over` and `after`:

```
instance Over a => Over (Active a) where
    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

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

These functions both take two `Active`'s as parameters and specify their relation. `over` is another instance of the ChalkBoard function with the same name. It takes the two `Active` parameters and combines them into one, with both animations occurring one of top of the other. This requires that the `a` and `b` of `Active a` and `Active b` must be types that are compatible with the ChalkBoard version of `over` (namely, `Board`'s). The `after` function, on the other hand, simply changes the time values of the first `Active` so that it will occur immediately after the second `Active` finishes. Both of these functions are especially important for building up combinators to manage the ordering of animations in a scene.

5.2 Active Combinators

Having explored the primitive functions of the Active language, how can these operations be made easier for the user? For starters, many timing combinators have been added to make organizing animations much simpler. Perhaps the most important of these is the `flicker` function, of type:

```
flicker :: (Over a) => [Active a] -> Active a
```

This function takes a list of `Active` objects and combines them into one `Active` object, with each animation in the list being executed one right after the other.

Each of the successive animations is also placed on top of the previous ones, so that parts of a scene can be built up independently and still displayed together. This is again helpful in terms of increasing the amount of abstraction in a scene. Now, independent object animations can be built up completely separate, which allows for greater flexibility in changing certain aspects of a scene without affecting others, or managing the ordering of the scene without affecting what happens during each of the individual animations.

In addition to ordering the animations, the amount of time each takes to animate is also essential to a scene. The way this is primarily controlled in `Active` is with the `taking` function, of type:

```
taking :: R -> Active a -> Active a
```

The `taking` function stretches or shrinks the length of an animation based on the `R` value. This `R` value can be thought of for now roughly as the amount of seconds the animation should take. If this animation is combined with other animations and then the `taking` function is used again, however, then the old values are just used to determine what percentage of the new time should be spent on the different sub-animations. Generally, `taking` is easiest to use in close conjunction with the `flicker` function, though it doesn't have to be. This keeps most of the timing information in one place, even if one doesn't directly affect the other. A typical example of how these functions are used might be:

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

In addition to timing, many other types of combinators are also included in Active to help make common animation tasks simpler. The function used to create many these combinators is the `addActive` function:

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

This function is a simple abstraction which helps to create many of the standard animation functions in Active. It takes a function of type `UI -> a -> b`, which represents how to change an object over time. Typically for animation, these `a`'s and `b`'s are boards of some type. The `Active a`, in that case, would contain the previous animations on the `Board`. The input function will use the `UI` time values to modify the `Board` over time. `addActive` is especially helpful in adding new animations to existing ones (already an `Active`) without having to explicitly turn each function into an `Active` first. What happens, however, when starting with just a `Board` and an initial animation to be applied? Because `Active` is an applicative functor, the `pure` function can be used to simply lift the `Board` into the `Active` world (with the `Pure` constructor). `addActive` can then be used like normal to start adding animations to it.

`addActive` is also helpful in creating animation combinators. It is used to create many predefined functions which combine elements of both Active and ChalkBoard, in order to make some common animation tasks simpler. For instance, moving boards over time is a very common animation task. The user shouldn't need to create their own version of this same animation function over and over in every project. Instead, the Active extension to ChalkBoard provides this move-over-time combinator, using the ChalkBoard `move` function:

```
activeMove :: (R,R) -> Active (Board a) -> Active (Board a)
activeMove (x,y) = addActive $ \ui -> move (ui*x,ui*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 greater than 0 (`activeAppear`), when its time value is between 0 and 1 (`activeTempAppear`), or up until its time value is 1 (`activeDisappear`). Unless one of these functions is applied, all `Active`'s will be visible for the duration of the scene, regardless of when their animations actually occur. Example usage of these functions can be seen in Section 5.3.1.

5.3 Case Study

In order to test the features and usability of Active, a pre-existing animation was recreated in the Active/ChalkBoard system. This was done both to see how close Active and ChalkBoard could get to the original, as well as how difficult it would be to do so. The animation chosen 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 looked nice, served a useful purpose, and was exactly the type of animation that should be easy to create in the Active/ChalkBoard system. It also was complicated enough to be a good test of the system's features, without being too complicated as to prevent new users, who haven't seen any of these features before, from following along. In addition to the included snippits, the full source code of this example can be found in Appendix A.

In creating this and other examples, a general structure for ChalkBoard animations using Active has begun to appear. First, the individual pieces of the animation are constructed. This stage includes building each of the separate `Active (Board a)` objects that will make up a part of the final scene. These individual animation chunks could be such things as objects 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 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. In general, a basic template for creating an animation in this way looks like the following:

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

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

5.3.1 Stage 1: Building Animation Pieces

In starting the Pythagorean example, construction is first begun on all of the individual animation pieces that will be used in the scene. The first of these is a basic 3-4-5 triangle in the middle of the screen:

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

The 3-4-5 triangle is constructed by giving the points (-0.2,0.15), (-0.2,-0.15) and (0.2,-0.15) to the `triangle` constructor. This creates a `Board Bool` of the triangle. A black outline around this triangle is also needed to match the original animation. To do this, the `pointsToLine` function is used, which takes a list of points and a line width, drawing a line between all adjacently listed points. Finally, both `Board Bool` objects are then given their colors by using the `choose` function. This makes the lines black over a transparent background (so the triangle can be seen behind them) and the triangle yellow with a transparent background (to see the other objects in the scene).

While this code does create a simple triangle, the triangle itself is never actually displayed in the animation. Instead, this base triangle is transformed in many different ways to create all of the other triangles that *are* displayed in the scene. 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:

```
let movingTriangle = activeMove (0.15,0.2) $ activeScale (2/3) $
                     pure $ scale 1.5 $ mainTriangle
```

First, the triangle is lifted into the `Active` world using `pure`. Then, animation functions can be added to it. In this instance, an `activeScale` of 2/3 and an `activeMove` of (0.15, 0.2) are applied. Snapshots of the resulting animation are shown in Figure 5.1.

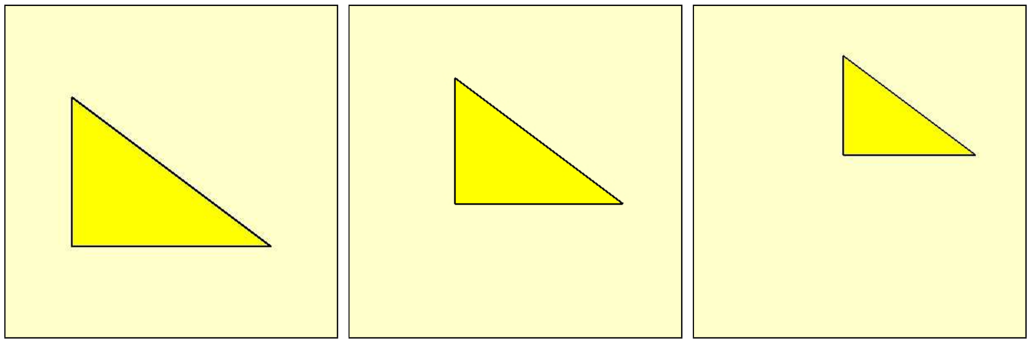

Figure 5.1. movingTriangle animation

As a quick note, all of the font for this animation was actually added in last and is contained within the second `let` clause in the source code. For this case study, only the creation of the actual, shape-based animation will be covered, and not the insertion of font. This is because the only interesting problem involving the font is when to make the font labels appear and/or disappear. How to do this is already shown numerous times in the shape-based animation, however, and so the repetition was left out.

Moving on the with example, the next step is to create three identical but rotated triangles, as displayed in the Wikipedia graphic:

```
let otherTriangles = [ activeAppear $ pure $ position i $ mainTriangle
                     | i <- [1..3] ]
    position i = rotate (-i*pi/2) $ move (0.15,0.2)
```

These three triangles are created using the list comprehension in `otherTriangles`, which simply rotates a moved version of the original `mainTriangle` using the `position` function. These triangles are then made to appear when their animations start using `activeAppear`. This function makes it so that these animations are only displayed when they begin animating, instead of for the entire duration of the scene. Figure 5.2 shows each of the new triangles being added individually to the animation.

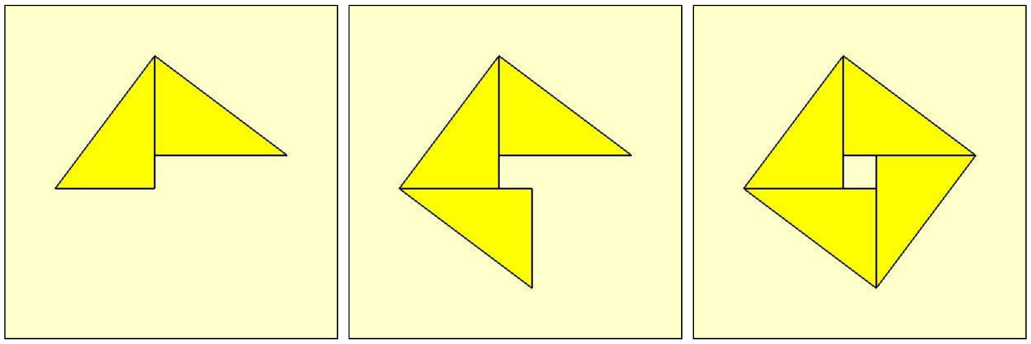

Figure 5.2. otherTriangles animation

The next part of the scene is simply adding in a missing piece so that the full area can be clearly identified. A small yellow square is added to the middle so that the larger square can be seen to have a size of *cxc*. 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 5.3.

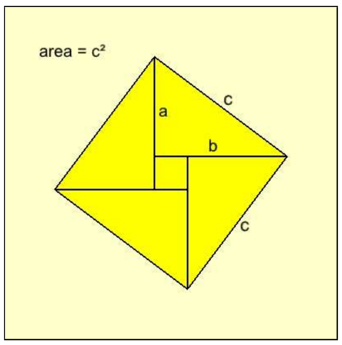

Figure 5.3. fillSquare animation

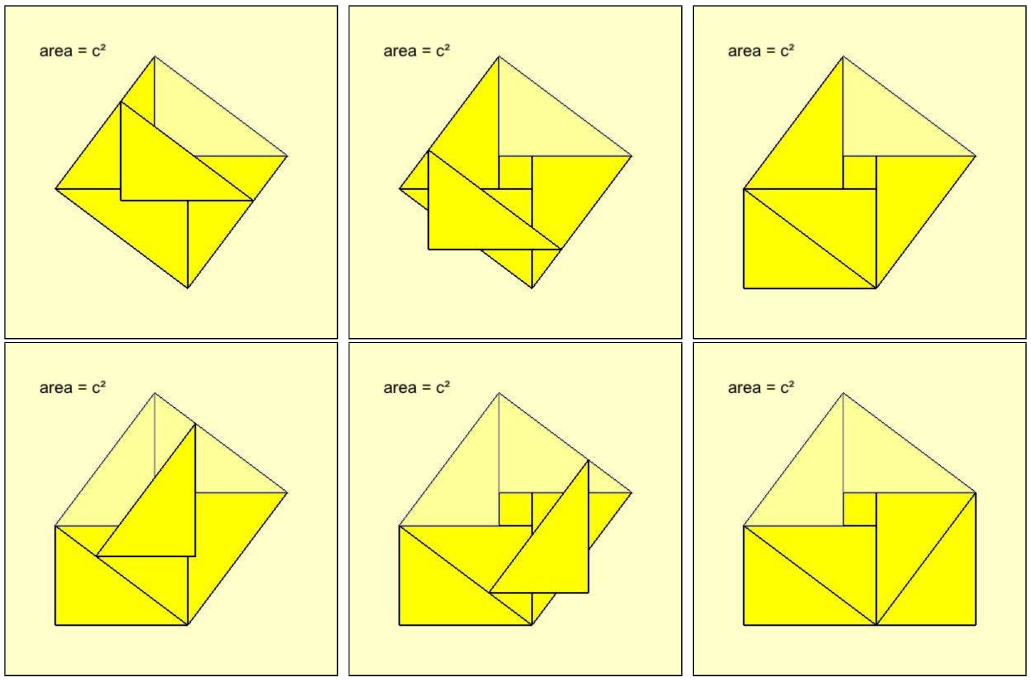

Figure 5.4. slideLeft and slideRight animations

Next, the top triangles need to slide down to match up with the lower triangles, as seen in Figure 5.4. An outline of the old triangles also needs to remain behind so their starting positions can still be see (like in the original on Wikipedia). This is done in two parts. 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:

```
let fadedTris = [ position i $
                    choose (withAlpha 0.6 white) transparent <$> triangle345
                | i <- [0,1] ]
    slideLeft = activeAppear $
                (activeMove (-0.3,-0.4) $ pure $ position 0 $ mainTriangle)
                `over` (pure $ head fadedTris)
    slideRight = activeAppear $
                 (activeMove (0.4,-0.3) $ pure $ position 1 $ mainTriangle)
                 `over` (pure $ last fadedTris)
```

The first part is done similarly to the creation of `otherTriangles` above. In this instance, however, white triangles with alpha values of 0.6 are placed over the two triangles so that they will appear faded. For the second part, the sliding triangles must be created. This is done by moving copies of the original `mainTriangle` to the initial positions, where the triangles were before. `activeMove` is then applied to each to animate the movement of the triangles down to their final locations, one on the left side and one on the right side.

The final step 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, the same general strategy as fading out the two triangles in the last step is used. The main differences are that this time yellow squares with alpha values of 0.9 are used so that the new squares will be a darker yellow instead of a lighter one, and that lines are also drawn around the new squares in order to make them more clear:

```
let newSquares = (move (0.15, -0.15) $ scale 0.4 $ square)
                 `over` (move (-0.2, -0.2) $ scale 0.3 $ square)
    newLines   = pointsToLine [(-0.05,-0.35), ... , (-0.05,-0.05)] 0.004
    fadeInSquares = (fadeIn 1 black newLines)
                    `over` (fadeIn 0.9 yellow newSquares)

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

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 predefined `fadeIn` function (included for completeness). This function takes an RGB color, an alpha value, and a `Board Bool`, such as the newly created squares. It then creates an `Active (Board RGBA)` which fades in the shape defined in the `Board Bool` from transparent to the given RGB and alpha values. 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 5.5.

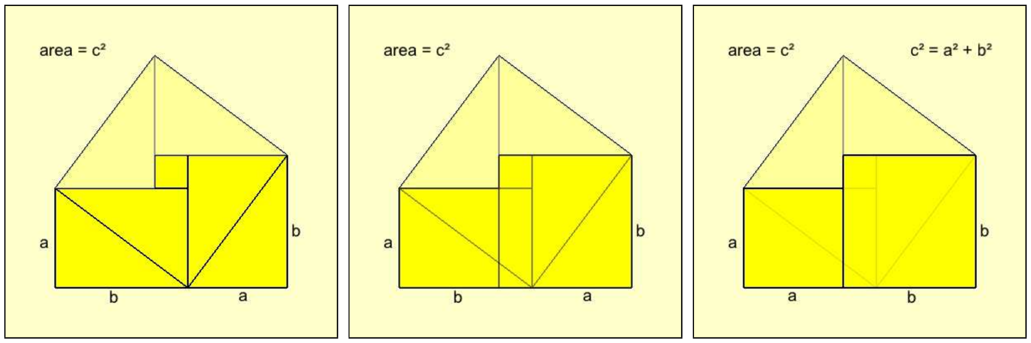

Figure 5.5. fadeInSquares animation

5.3.2 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 other triangles used in the scene. While longer `Active`'s can definitely be created directly using the `mvActive` function defined in Section 5.1, it is generally much cleaner and easier to construct simple animations and then organize them into a series using one of the predefined combinators, such as `flicker`.

This usage of functions like `flicker` is the second major stage in creating an animation. With `flicker`, animations can be strung together, one after the other, stacking newer parts onto the older ones. The length of time each individual animation component takes to be performed can be specified using the `taking` function, often inside the list of animations given to `flicker`, as described in Section5.2. The general structure, using this case study as an example, often looks something 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 usage of `flicker` and `taking` manages the majority of the ordering and timing for animations. It returns a single `Active (Board a)`, which can then be used to display the animation, or reused to create an even bigger animation, hierarchically. In terms of displaying the animation, this will largely be done the same way for most animations:

```
playObj <- byFrame 29.97 anim

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

First, the `Active (Board a)` 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 repeatedly 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 also saved into a video file with `frameChalkBoard`. 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, as shown, which extracts and then displays the returned frame. In the future, Active will hopefully include many of these basic cases as predefined functions so that animations can be played automatically, though obviously there are many other actions that can be taken with the returned `Board`, if the user desires.

Finally, the complete animation created in this case study can be seen online at http://www.youtube.com/watch?v=UDRGhTFu17w. It was produced as a video by simply saving each of the returned boards to a video file using the `frameChalkBoard` command after opening a default ChalkBoard write stream.

Chapter 6

Active Transformations

6.1 Design Considerations

One of the main challenges in animating transformations is being able to generically represent the various classes of transformations. A transformation animation system needs to be able to render arbitrary Abstract Syntax Trees (ASTs) and animate transformations over that AST. Additional information from the user beyond their original transformation system will be needed, but the task in hand is to minimize this necessary information and require as simple of changes as possible. The key issue is how to observe and track what is happening to the user's AST during their transformations, and especially how that relates to the printed version of the AST so that these changes can be appropriated animated.

One way of accomplishing this observation is by indexing the nodes of the AST. Indexing gives the system a way to look inside the AST and track changes as they are made. Every iteration of our Active Transformations system has taken advantage on this concept.

One basic form of indexing is explicitly adding an index to each node:

```
data Expr = Lit ID Int
          | Add ID Expr Expr
          | Mul ID Expr Expr
```

This simple indexing has changed and improved in our system through different iterations, but the idea of increasing the visibility inside the AST and tracking the changes as they are made has always been the same.

One sub-system that takes advantage of indexing is the ChalkDoc system. A ChalkDoc is a data structure used to hold all of the information necessary to print a given AST to the screen, as well as some additional data to help animate changes to that AST. A ChalkDoc must be generated for every AST instance to be displayed. This generation requires slightly modifying the AST's `show` function, a concept that will be addressed later in Sections 6.3 and 6.4.

One especially important piece of information ChalkDocs contain is which part of the AST generated each string present inside it. This allows us to treat the printed strings much like AST nodes, since the generating AST node is always known. The strings printed by the same node can then often be grouped up and treated as one element. The system can then use this ability to see exactly where a certain part of an AST was before and after a transformation step, and animate the changes between the two states. For instance, if a variable `v` is being replaced with the expression `1 + 1`, this new expression is larger and will require additional screen space. If there is other text surrounding the original variable, `v`, this text must be expanded to fit the new expression. The AST index and other information stored inside the ChalkDocs allow us to see that the text around the replacement is the same in both the before and after ChalkDoc, regardless of its new position. This text can then be animated, moving into its correct position as a less-emphasized 'text displacement' rather than as the main transformation.

The same abilities of this system are also exploited for the actual transformations. In the example above, say the expression `1 + 1` is being inlined from a `let` clause. The expression should be copied from there and moved to its final location, but how can the system know this from just the before and after ChalkDocs? The inlined expression and the original expression in the `let` clause, though identical, must have different indexes. This is necessary for maintaining an AST without repeated indexes, so that nodes can be told apart. It makes knowing when a node is copied much more difficult, however. A naive thought could be to analyze the new expression and see if it appears anywhere else in the AST. This approach is obviously flawed though, because having two identical expressions does not guarantee that one was inlined, or that they are related at all. Our system, by the necessity of being generic, has no idea that inlining is being performed, or even what inlining is. If it required knowledge of the transformation in order to know what to do, it would only work for those transformations that it had knowledge of. Instead, our solution to this problem is to introduce a generic Transformation Algebra.

This Transformation Algebra helps to clarify AST changes that may otherwise be ambiguous. The goal of this Algebra is to describe any transformation a user could want in as few terms as possible. The Transformation Algebra and how it works is discussed more thoroughly in Section 6.2.

With this new Transformation Algebra, solving our initial problem is straightforward. If the user is somehow allowed to insert these algebra terms into their transformation code, the system can then be told that the expression `1 + 1` is being copied to a new location and perform the appropriate animation.

The Transformation Algebra again takes advantage of the AST indexing to identify which nodes should be copied, or have other such actions performed on them. Using these indexes, the applicable pieces of the ChalkDoc can be identified and animated according to each term in the algebra.

In general, it is the combination of these three features (Indexed ASTs, ChalkDocs, and the Transformation Algebra) that allows the Active Transformations system to correctly animate user transformations. Some of the implementation details for these features can be found in Section 6.5, but first let us explore the specifics of these systems, difficulties faced in creating them, and improvements that were made to them.

6.2 Transformation Algebra

When thinking about the operations the Active Transformation system needs to animate, it is helpful to first consider the structure being animated. In essence, we wish to animate a tree. Different kinds of trees may have different ways in which they are printed, or different operations that are performed on them, but the Active Transformation system attempts to animate all of these possibilities. As a generic system, it must therefore be able to animate nearly any change that can occur over nearly any tree. This is an enormous space, however. There are many different operations that could be performed on a tree, depending on the specific tree and situation. What we need to do is determine a basic set of *fundamental* operations over trees that can be combined to create any of the other operations. For our purposes, this most-basic set of operations will be called our Transformation Algebra, where the operations form an algebra over all possible transformations that can be applied to a tree. It is important to remember, however, that a transformation is a *change* to a tree, which therefore doesn't include traversals or other *observation* operations.

When thinking about a generic tree, what are the most basic operations that can be performed on that tree? The two most-basic operations that may come to mind are adding elements and deleting elements, or `Add` and `Delete`. As long as the proper connections are made, a node can be added or deleted anywhere inside a tree. When certain additions are legal or not, and how these connections are changed, is dependent on the type of tree. The role of the Active Transformation system, however, is simply to be able to animate any such change that might be made.

In theory, all changes to a tree can be made in terms of `Add` and `Delete`. When swapping two subtrees of a parent, for instance, both trees could be deleted and then added again on the opposite side. Expressing this change in terms of `Add` and `Delete`, however, misses out on the point of the operation. It completely ignores the fact that the trees being added are the same as the ones that were deleted, and that they switched places. This information is lost and doesn't play a role in the modification if it is only expressed in terms of `Add` and `Delete`.

In animating changes to an AST, this loss of information is important. Because we are animating a move from one state to the next, we care about *how* these changes are made, not just the final result. If the system is trying to animate a tree-balancing operation, and the two subtrees of a given node are supposed to swap, simply deleting both and then inserting them again is not going to give the audience the correct impression of what is happening. In order to understand this algorithm, and how the changes are being made to the AST, it is important to see that the subtrees are actually swapping. Not only should our system be able to provide an animation to represent a change to an tree, it should be able to provide an animation that shows *how* this change is happening.

This leads to the next fundamental operation for animating tree changes, the `Move` operation. With this operation, how the above changes are occurring can be described more accurately. In swapping two subtrees of a node, the left subtree is moving to the right subtree position, and the right subtree is moving to the left subtree position. This same `Move` operation can be used in many other ways, moving different nodes and subtrees all around a tree in different situations.

The `Move` operation still isn't quite enough, however, in some situations where movement is involved. Take, for instance, a node of a binary tree with only one left subtree. What if we want to copy this left subtree and use it for the right subtree also? In this type of situation, a `Copy` operation is needed.

The difference between a `Move` and a `Copy` is whether the original node is completely moved to a new location, or instead whether a copy of that node is moved to the new location without affecting the original. While the situation above describes why this might be useful, the last sentence may hint at another possible way of implementing our algebra. What if, instead of a `Move` operation, we simply used a `Copy` followed by a `Delete`? Unfortunately, this is not ideal for much the same reason as why `Move` shouldn't be expressed in terms of `Add` and `Delete`.

While the reader may note that, for a given step from one tree to the next, a `Move` could easily be created by doing a `Copy` and a `Delete`, the distinction is necessary because we are again not just interested in the final state of the tree, but in how it got to that final state. The point of this Transformation Algebra is to describe the actual transformation itself and how it moves from one state to the

`Add`	Add a subtree to the AST
`Delete`	Delete a subtree from the AST
`Move`	Move a subtree from one location in the AST to another
`Copy`	Create a copy of a subtree inside the AST and move it to another location

Table 6.1. Initial Transformation Algebra

next, not simply to create the next state. If the system always does a `Copy` and then a `Delete`, this will not have the same look or effect in an animation as when explicitly using the `Move` operation. The difference is visible in the animation, and therefore the `Move` term is necessary in order to show *how* the user's transformation actually works.

The four operations discussed so far make up the initial Transformation Algebra, summarized in figure 6.1.

While this version of the Transformation Algebra appears to work and be correct, it is not the same as the version implemented in our system. The Active Transformation system can automatically detect and animate all `Add` and `Delete` operations, so these do not need to be included in our implementation of the algebra. How this automatic detection is done is described in Section 6.5.

In addition, a `Break` operation is also included in our implementation of the algebra. What this `Break` operation does is tell the system to end the current animation step. For instance, the system could animate an `Add` and `Delete` at the same time, or it could break them apart into two separate animations when the `Break` command is given. The `Break` command also stores a snapshot of the current AST. These intermediate ASTs are used to search for updated subtrees, as described in Section 6.3, as well as to help automatically detect adds and deletes.

The system inserts some `Break` operations automatically, however, so additional input from the user is only required for customizing animations.

The implementation of this Animation Algebra is described in Section 6.5.

6.3 Indexed AST Improvements

Given the general architecture for the Active Transformation system, presented in Section 6.1, one important decision to be made was how to index a user's generic AST. The first approach was a little simplistic, but worked for a lot of situations and was the easiest to implement, making it a decent place to start for an initial prototype while working on other features. The main idea was to simply add an index to each node of a user's AST. It required the user to create an alternate AST datatype that incorporated this index, and then provide a function for indexing from their original AST into this new indexed AST.

The thought behind this version of the indexing was simple. Given these indexes, the system tried to avoid the problem of storing an internal representation of the user's AST at all. While this would eventually prove not to be possible, the indexing itself has still proven useful, as described in Section 6.1, and enabled the creation of other systems based around indexing.

The biggest problem with this initial approach comes up in the `Move` and `Copy` terms of the Transformation Algebra. While the system knows which top-level *nodes* are affected, based on the information provided in the Transformation Algebra, it has no way of finding the complete *subtrees* of these nodes. Consider the following example:

```
let a = (\x -> (+ x 1)) 3 in
    a
```

In this case, the expression to be inlined is an application, but the only string printed by that application node is likely a single space. There is no way from the ChalkDocs alone to figure out which strings belong to the subtrees of that application (namely, the lambda expression and the constant, 3). This observation quickly led to the conclusion that the Active Transformation system needed to have an internal representation of the user's data structure.

A first attempt at this internal representation was to reuse the work done for Data Reify [12]. This package already had the ability to generically create graphs that could represent our structure, given the user implement a rather simple class instance. This graph structure was quickly found to be non-ideal for our purposes, however, given that this system only uses trees. Having to look up the next node of an AST in an association list for every step down the AST made traversal $O(n^2)$ instead of $O(n)$, and deletion of a node once that node has been found $O(m^2)$ instead of $O(1)$ (where m is the number of nodes in the subtree).

For the final, more-efficient attempt, a similar approach to Data.Reify is used, but this approach is based even more on the work of Generics.Regular [25]. Instead of using either package explicitly, however, the Active Transformation system requires a new class instance of its own that is slightly easier for the user to implement (and will also not require any additional packages). This final approach is to convert the user's data structure into an explicitly recursive type using functors. In terms of user implementation, this is quite straightforward. All the user has to supply is a mapping between their initial data structure and a new `Functor` version of that structure. Implementing this doesn't even require the user to know that their `Functor` type is being used recursively. All they need to do is create a new version of their data structure with a functor hole in every place that the original structure was recursive. They then complete a `Traversable` instance for the *new* data structure and the new `MuRef` class for their *original* data structure:

```
class MuRef a where
  type DeRef a :: * -> *
  toFunctor :: a -> DeRef a a
```

The `DeRef` type function in this class says that the original datatype can be converted into the new `Functor` version of the datatype, and `toFunctor` provides this conversion (much like `from` in the `Regular` class). This conversion converts one node of the old structure into one node of the new `Functor` version. Internally, this function is used recursively to construct a private representation of the AST for tracking changes. This internal representation of the AST is created by placing the user's `Functor` type inside a structure very similar to the standard `Mu` type commonly used as a fixed point for explicitly recursive types [19]. The traditional `Mu` is defined as:

```
data Mu f = In (f (Mu f))
```

The Active Transformation version of `Mu` is essentially the same thing, but with an extra field used for indexing. The name used for the new structure is therefore `IndexedAST` instead of `Mu`, since placing an AST inside the structure results in an indexed version of that AST.

To solidify exactly what is required when implementing the `MuRef` class and the indexed AST system in general, an example instance for a list data structure can be found in Section 6.4.

With this final system in place, how has the user implementation changed from the original indexing of ASTs? For one, the need for users to index their own structures has been removed. This is rather helpful, especially in cases where multiple ASTs may need to be indexed or new AST nodes added. It allows the user to avoid implementing a system to keep track of all these indexes. Replacing this index function with a `toFunctor` function simplifies what the user has to implement, and means there is less opportunity for mistakes to be made because all of the indexing work is done internally.

More importantly, the ability to store an internal representation of the user's AST has been gained. This allows the system to track changes to the AST and make those same changes to the internal representation. This is important in our project for two related reasons.

The first reason is that it solves the problem of finding a subtree from a given node index. Now, if the inlining of the application described at the beginning of this section is displayed, the system can find the correct subtrees and all the indexes contained within them. From there, the system can easily find the strings that correspond to these indexes in the ChalkDoc, which will allow it to successfully move the subtrees to their new locations on the screen.

The second reason is that it allows us to take snapshots of the AST at different points in the transformation, and track incremental changes as they are made to the AST. This is absolutely essential in recursive transformations (as most code transformations are) because, in addition to simply finding AST subtrees, the system needs these subtrees to be up to date. In recursive transformations, one AST is passed into the function, and one AST is returned. All that is visible are the initial and final versions of the AST. In an animation system, however, we are interested not only in the result, but in how we got to that result. If there are a lot of small steps made by the transformation, these individual steps are normally lost. This can be especially detrimental when changes are nested and might not make sense without seeing the incremental stages. Take, for instance, the following code segment:

```
let a = let v = 1 in
          v in
    a
```

Applying an inlining transformation to this code segment results in the following stages:

```
let a = let v = 1 in
          1 in
    a

let a = let v = 1 in
          1 in
    let v = 1 in
        1
```

In this example, the `v` is inlined first, followed by the `a`. If the system doesn't have an updated version of the AST after the `v` inlining occurs, however, we run into problems. If the system looks in the original AST for the subtree to be inlined for `a`, it won't include the newly inlined `1`. The animation would then fail, likely moving everything correctly except the `1`, but therefore displaying a highly inaccurate animation result, such as:

```
let a = let v = 1 in
          1 in
    let v = 1 in
```

Keeping track of incremental changes to the AST internally solves this problem and allows us to find correct subtrees at any given point during animation. It allows us to capture information that is normally lost in a recursive transformation and use that data to create correct animations of all the individual steps.

6.4 List Example

A list data structure is a good starting example for the overall system. It's simple to follow and has a few differences from other, code-based examples that make it worth looking at. Assuming the user already has a Haskell transformation system that operates over lists, they will likely have a datatype similar to:

```
data List a = Cons a (List a)
            | Nil
```

The user would also likely have some sort of `Show` function written, their transformation functions (such as append, etc.), and perhaps some example instances of their lists and transformations on them. These functions will all need to be slightly modified, so starting with a copy of the originals would be a good idea if the user wishes to retain both functionalities.

The alternate, `Functor` version of the list data structure (required by `MuRef`), would look something like:

```
data L a x = C a x
           | N
```

In our implementation, the type variable `x` will be used as a recursive hole, but the user need not worry about that. They only need to implement the `MuRef` class for `List` and `Traversable` for `L` (as well as `Functor` and `Foldable`, but these are trivial given `Traversable`). The instance for `MuRef` would look like:

```
instance MuRef (List a) where
        type DeRef (List a) = L a
        toFunctor list = case list of
                Cons a l -> C a l
                Nil      -> N
```

Here, `L a` is identified to be the `Functor` version of `List a`, and a means of converting from `List a` to `L a` is given. This is very similar to the `Regular` class in Generics.Regular [25], but note that only converting *to* the `Functor` is required. Converting the other direction is an optional second class, and only necessary if the user ever wishes to unindex an indexed AST for some reason.

Using this `MuRef` instance, the system is able to convert and index the user's AST internally, turning it into an `IndexedAST (L a)`, as described in Section 6.3. In order to completely manipulate this new `Functor` version of the user's data structure, however, it must be given additional instances:

```
instance Functor (L a) where fmap = fmapDefault
instance Foldable (L a) where foldMap = foldMapDefault
instance Traversable (L a) where
        traverse fn list = case list of
                C e l -> liftA (C e) (fn l)
                N     -> pure N
```

Note that `traverse` isn't recursive and only applies the function one level deep. The user should just assume that `L a x` isn't recursive and implement `traverse` as such.

Now that the data structures are set up properly, the user can move on to the other functions that must be adapted. The first of these is the `Show` function. The Active Transformation system needs this function to return `[ASTSymbol]` instead of a `String`. Most of these changes are very simple however. The user must simply associate an AST index with each string that is printed. In this example, the user may have an original print function similar to:

```
myPrint :: Show a => List a -> String
myPrint l = case l of
        Nil       -> "[]"
        Cons e l2 -> "[" ++ show e ++ print' l2 ++ "]"
    where print' l = case l of
              Nil      -> ""
              Cons e l -> ", " ++ show e ++ print' l
```

The extra top level is needed here to add the outside brackets normally seen around lists and to print the first element without a comma. The new, adapted version of this function will be rather similar with a couple of main exceptions. In addition to the normal changes, however, this particular example also has a special case. Extra characters are printed that don't directly correspond to any node of the data structure, but are simply used for printing at the top level. Because of this, a new index must be generated for these strings that isn't used anywhere inside the AST. The new function might look something like:

```
makePrint :: Show a => IO (IndexedAST (L a) -> [ASTSymbol])
makePrint = do
    topID <- newID
    let gTopID = giveID topID

    let print iast  = case getAST iast of
            N     -> gTopID "[]"
            C e l -> gTopID "[" ++ giveID (getID iast) (show e)
                       ++ print' l ++ gTopID "]"
        print' iast = let gID = giveID (getID iast) in
                      case getAST iast of
            N     -> gID ""
            C e l -> gID ", " ++ gID (show e) ++ print' l

    return print
```

As you can see, `makePrintFn` actually *generates* the needed print function. This is done in order to create a new index with `newID`, which requires the IO monad. Once this index is created, the rest of the function can just be created like normal and returned, giving us our modified print function.

Now as for the function itself, notice that it takes an `IndexedAST (L a)` as an argument. It must therefore use the `getAST` and `getID` functions in order to grab the AST portion or index portion, respectively. The other main difference (apart from some cleanup functions to make the code shorter), is the use of the `giveID` function. This is the function that should be used to change a `String` into an `[ASTSymbol]`. It is used anywhere a string is normally returned in order to associate an index with that string. While this does make the code a little bit bulkier, it is usually a very straightforward change that can be made systematically.

One small caveat with this system, however, can sometimes appear when strings are combined that may need to be treated separately. For instance, the string `", "` and the string generated by `show e` must remain separate and both be given the same index separately. This is because it is possible that the strings may need to be animated independently. If that element becomes the front of the list, for instance, the comma will need to disappear. If the strings are combined, however, then they must be treated the same and the system will display a slightly weird result. In general, it is safer to keep strings separate unless it is known that they will always be displayed together.

Another note, returned to in Section 6.6, is that the built-in function for newlines should be used at all times. This function, `newline`, takes an index for the newline to be associated with and begins a new line in the output. It is important that this function is used instead of newline escape sequences inside the strings because the strings are never actually printed, they are simply used to create the graphics that are displayed on the screen.

With the modified version of the user's print function, the last modification that must be made is to the actual transformation functions. Take, for instance, the following `append` transformation:

```
append2 :: List a -> List a -> List a
append2 f s = case f of
        Cons e l -> Cons e (append2 l s)
        Nil      -> s
```

Turning this function into a version accepted by our animation engine may result in something like:

```
appT :: IndexedAST (L a) -> IndexedAST (L a) -> Transform (L a)
appT f s = case getAST f of
        C e l -> changeM f $ liftM (C e) (appT l s)
        N     -> replace f s
```

As indicated by the type signature, the lists should all be in their indexed, functor forms, and the new transformation must now use the `Transform` monad. Again, the `getAST` function is used to return the data structure portion of this `IndexedAST`. Because the transformation is now in a monad, `liftM` is used to complete the structure of `L a` inside the monad. The two main `return`-style functions of the `Transform` monad are `change` and `replace` (with monadic versions `changeM` and `replaceM`).

The `change` function is used to indicate that subtrees of the node may be modified, but that the node itself is basically the same and should retain the same index. In this example, the current node, `f`, is modified to `liftM (C e) (appendT l s)`. The current node remains the same as it was (with element `e`), but other nodes further down the list are changed. Here, the monadic version, `changeM`, is used simply to keep the code in one line, making it look closer to the original. The same code could also be written using `change` as:

```
C e l -> do
           rest <- appendT l s
           change f (C e rest)
```

The `replace` function is the other main `return`-style function of the `Transform` monad. It is used when the current subtree should be completely replaced by another subtree. In this example, once the end of the first list, `f`, has been reached, the nil node should be replaced with the list being appended, `s`. Looking at the original transformation, `append2`, shows that both of these cases in `appT` are actually quite similar to their original forms, except that they require extra DSL functions with the current subtree as a first argument.

This last change completes all necessary modifications to the user's code. A `main` function could now be written to create and run specific animations:

```
main = do
    let ex1 = Cons 1 $ Cons 2 $ Cons 8 $ Cons 1 Nil
        ex2 = Cons 5 $ Cons 6 Nil
        ex3 = Cons 10 $ Cons 11 $ Cons 13 $ Cons 12 Nil

    printFn <- makePrint
    iex2 <- index ex2
    iex3 <- index ex3
    let anims = [flip appT iex2, flip appT iex3]

    animate "Arial.ttf" printFn ex1 anims
```

The core library function is this snippet is the `animate` function, which begins the animation of the user's transformations. It takes a `FilePath` to a font to use, the print function created above, a starting AST on which to perform transformations, and finally, a list of transformations to apply. Notice, that these transformations must all be of type `IndexedAST f -> Transform f`, where in this case `f` is `L a`. Therefore, `ex2` and `ex3` must be indexed (using the library's index function) and passed as arguments to the appT functions. In this way, each instance of appT is partially applied and only needs its final argument, the list to be transformed. The call to animate will then begin animations with ex1 as the list to be transformed, first appending ex2 to it, followed by ex3.

6.5 Implementation

Given all of the changes the user must implement, the next question becomes how our system takes advantage of these modifications in order to animate the transformations. The first step is to take the user's starting AST and transform it into our internal, indexed representation. This conversion can be done rather succinctly, given the new datatype and class instances supplied by the user, using our IndexedAST type and index function:

```
data IndexedAST f = IndexedAST ID (f (IndexedAST f))

index :: (MuRef a, Traversable (DeRef a)) => a -> IO (IndexedAST (DeRef a))
index ast = putIDs $ toFunctor ast
      where putIDs f = traverse (putIDs . toFunctor) f >>= newIAST

newIAST :: f (IndexedAST f) -> IO (IndexedAST f)
newIAST ast = do
        id <- newID
        return (IndexedAST id ast)
```

The only function in this definition that hasn't been introduced yet is the newID function. This function simply returns the next unique id available and keeps track of all the indexing information behind the scenes, similar to newUnique or other such functions.

Working through the index function, nodes of the user's initial type are recursively turned into their Functor counterparts. The current Functor style node is then passed through using `traverse`, as defined by the user, and the process is repeated. On the way back up the recursion, all of the nodes are placed inside an `IndexedAST`, which both acts as a fixed point for the recursive type and indexes each of the nodes. This process creates an indexed version of the user's AST, which can be freely traversed by the system in order to apply changes made during transformations.

6.5.1 The Transform Monad

The changes to the AST will be made inside of the `Transform` monad, which all user transformations must be written in. An example of using this monad was given in Section 6.4, but the specific functions and internals of the monad have not been defined. The type of the monad is:

```
type Transform f  = Transform' f (IndexedAST f)
type Transform' f = StateT (IndexedAST f) (WriterT [Transformation f] IO)
```

The `Transform` monad is therefore a state + writer monad on top of IO. The state in this instance is the internal representation of the AST, which is updated whenever the user makes changes to it in their transformations. The writer monad, then, is a listing of the Transformation Algebra terms that have been applied over the course of the transformations, including the order in which they were applied. The Transformation Algebra is represented internally as the type:

```
data Transformation f = Move ID
                      | Copy ID ID
                      | Break (IndexedAST f)
```

The `Break` term contains an `IndexedAST`. This `IndexedAST` corresponds to the current state of the internal AST at a given point in time. The `Break` term, therefore, will hold snapshots of the AST as the transformation progresses, so that the system can retain information about all the incremental modifications. Using these AST snapshots, the system prevents data about the transformation from being lost in order to animate all of the individual steps correctly and in the right order.

The functions that add these algebra terms into a user's transformation are:

```
addMove :: ID -> Transform’ f ()
addMove id = tell $ [Move id]

addCopy :: ID -> ID -> Transform’ f ()
addCopy id1 id2 = tell $ [Copy id1 id2]

addBreak :: Transform’ f ()
addBreak = do
        ast <- get
        tell $ [Break ast]
```

Any of these functions can be inserted, where appropriate, into a user's transformation in order to add the Transformation Algebra terms necessary to correctly display the animation. In general, only the `Move` and `Copy` terms will need to be inserted into a user's transformation when nodes of an AST are being moved or copied from one place to another. This is because a `Break` is already inserted by default whenever the `replace` function is used.

This behavior can be seen in the definitions for the `return`-style `Transform` functions, given below:

```
change :: Functor f => IndexedAST f -> f (IndexedAST f) -> Transform f
change oldNode newAST = return $ setID (getID oldNode) newAST

replace :: Functor f => IndexedAST f -> IndexedAST f -> Transform f
replace old new = do
        modify $ changeSt (getID old) new
        addBreak
        return new
```

```
changeSt :: Functor f => ID -> IndexedAST f -> IndexedAST f -> IndexedAST f
changeSt oldID new st = if (oldID == getID st)
        then new
        else setID (getID st) $ fmap (changeSt oldID new) (getAST st)
```

Here, the two `return`-style functions of the `Transform` monad, `replace` and `change`, are defined. If the node returned should completely replace the node that was there previously, the `replace` function is used. If the node being returned is basically the same as it was before (and should therefore retain the same index), but perhaps with some of its sub-nodes changed, then the `change` function is used. The only undefined function in this snippet is the `setID` function, which simply gives the current `Functor` node an `ID` in order to become an `IndexedAST`.

There are alternate versions of both of these functions, called `changeM` and `replaceM` respectively, which can be used when the second argument is already inside the `Transform` monad. The examples below and in Section 6.4 show when either of these alternates might be useful.

The last `Transform` command that may be useful is the `newNode` function:

```
newNode :: f (IndexedAST f) -> Transform f
newNode ast = liftIO $ newIAST ast
```

This function is used when a new node should be created, and will therefore require a new index. Take, for instance, working with a list of `Int`'s. A `cons` transformation can be created that that takes an `Int` and a previous list, and returns the new list. Inside the transformation function, the `newNode` command would be used in order to create the new `IndexedAST` node:

```
cons :: Int -> IndexedAST (L Int) -> Transform (L Int)
cons i l = replaceM l $ newNode (C i l)
```

Note that although `newNode` is of type `Transform f`, just like `replace` and `change`, it should not be used as a `return`-style function because it does not modify the internal state of the AST. One reason for this is because multiple nodes could be built up with `newNode`, which could then be combined and returned. The system doesn't know at what position in the AST to put the new node initially, so it doesn't modify the state yet.

6.5.2 Creating Animations

Using all of the `Transform` monad functions together, the user is able to write transformations that enable the system to internally retain essential information about the transformations. This extra data is all contained within the list of Transformation Algebra terms created with the writer part of the `Transform` monad. This list is then broken apart at every `Break` term, and each sub list is used to create the animation for a single step of the (often recursive) transformation.

Creating these single step animations is divided roughly into three parts:

- Constructing the ChalkDocs of both the before and after AST for a transformation step
- Automatically detecting the adds, deletes, and "text displacement" between the two ChalkDocs
- Animating any other terms that appear in the Transformation Algebra.

The first bullet begs perhaps a more basic question of what a ChalkDoc actually is. A ChalkDoc is simply a combination of pieces that are useful in printing an AST to the screen. Its type is simply:

```
type ChalkDoc = [(String, Board UI, (Float,Float), ID)]
```

The `String` term is fairly self explanatory, as the `ChalkDoc` is simply a list of all the `String` segments that are used in printing out the AST. The `ID` is also straightforward, as it is the index of the AST node that generated the `String`. The `Board UI`, then, is the first somewhat interesting term.

As mentioned in Chapter 4, a `Board` is the fundamental type of our drawing system, ChalkBoard [22] [23]. In ChalkBoard, the principal type, `Board`, is a two-dimensional plane of values. A color image in ChalkBoard, therefore, is a `Board` of color, or `Board RGB`. In this instance, the `Board` is holding values of type `UI`, which represents only those values on the unit interval between 0 and 1 (inclusive). This can be thought of as an image of our current `String` in grayscale.

The last term of the `ChalkDoc` type is the tuple, `(Float,Float)`. It represents the (x,y) location of where the `Board UI` will be printed on the screen. This can be highly useful information when comparing and matching different ChalkDocs to each other, and is absolutely essential when determining how to move a `Board` across the screen during certain animations.

So, given this structure, how is a `ChalkDoc` created? This is done with the user's modified print function. This print function uses `giveID` to associate an AST index with every string printed. The definition of `giveID` is:

```
giveID :: ID -> String -> [ASTSymbol]
giveID id str = [(str,id)]
```

These pieces of information can then be used to gather the other two fields of the `ChalkDoc`. The `Board UI` is generated by including the ChalkBoard font package, Graphics.ChalkBoard.Font, and using the provided `label` function to generate a `Board UI` from an input string and font. The position of the `Board UI` on the screen must then be discovered. This can be done by pretending to line up the `Board UI` graphics across a line on the screen, one after the other, until the end of the line or a newline string is reached. The next line is then started and handled in the same way.

After constructing the ChalkDocs for the current transformation step, the next item at hand is to automatically detect any adds or deletes that might have occurred during that step. While the internal AST is updated with this information automatically, a slightly different approach is taking for printing these changes. This is because parts of an AST that remain unchanged can sometimes print differently based on other factors, such as their sub-nodes (the comma in comma-separated lists, for instance). While examples of this are somewhat uncommon, the best way to make sure the system catches all of these printing changes is to examine them on the string output level instead of the AST level.

First, all AST nodes (and subtrees) included in the list of Transformation Algebra terms must be removed from the ChalkDoc. These elements are dealt with explicitly by the animation system, so they are not included in the automatic detection phases. This removal is performed by the following functions:

```
removeTree :: Foldable f => IndexedAST f -> ID -> ChalkDoc -> ChalkDoc
removeTree iast id cdoc = let maybeIAST = getSubAST id iast in
    case maybeIAST of
        Nothing    -> cdoc
        Just iast' -> let ids = getAllIDs $ iast' in
                      filter (\(_,_,_,id') -> not $ elem id' ids) doc

getAllIDs :: Foldable f => IndexedAST f -> [ID]
getAllIDs iast = F.foldl fn [(getID iast)] (getAST iast)
    where fn ids iast' = F.foldl fn ((getID iast'):ids) (getAST iast')

getSubAST :: Foldable f => ID -> IndexedAST f -> Maybe (IndexedAST f)
getSubAST id iast = if (id == getID iast)
    then Just iast
    else F.foldl fn Nothing (getAST iast)
        where fn mb iast' = if (id == getID iast')
            then Just iast'
            else F.foldl fn mb (getAST iast')
```

In these functions, `F.foldl` corresponds to the `Foldable` version of `foldl` instead of the prelude version. The functions take advantage of the fact that the user's `Functor` version of the structure is `Foldable` in order to find the subtree to remove and the indexes of the nodes in that subtree. Elements with an `ID` contained in this list are then filtered out of the `ChalkDoc`, resulting in a `ChalkDoc` with the given subtree removed.

Once all subtrees contained in the list of Transformation Algebra terms have been removed, the system can examine the remaining before and after `ChalkDoc` structures for differences. Any elements that are present in the first `ChalkDoc` but not the second are deletes, and elements that are present in the second but not the first are adds. The final part of this step is to discover all of the text that has been displaced. This is done by examining the two `ChalkDoc` structures for elements that are present in both, but that have a different location in one versus the other. The difference between these locations can then be used to animate the text moving from its current location to its final location. The adds and deletes can also be animated simply, by fading the involved `Board` in or out, respectively. This set of animations (fades and moves) can be constructed very easily with predefined functions in ChalkBoard. They can even be constructed independently and combined later, making the actual animation process quite trivial once the correct boards have been identified.

Finally, the last step in creating animations with the Active Transformation system is animating all of the terms of the Transformation Algebra. This is rather similar to the last problem. Instead of finding all the indexes and then removing those elements from the `ChalkDoc`, however, we want to find them and keep *only* those elements. This is done separately for each term in the Transformation Algebra, so that each can be animated according to the given term and indexes, and then combine all of the animations together. ChalkBoard is extremely helpful here, because it facilitates this modularized building of animations. The Active Transformation system can simply walk through the list of algebra terms for the current transformation step, animate each one, and then combine them all together. For example, the following function is used to construct the animation for any given `Move` or `Copy` term in the list:

```
moveAnim :: Traversable f => IndexedAST f -> IndexedAST f -> ID -> ID ->
                             ChalkDoc -> ChalkDoc -> Active (Board UI)
moveAnim iast1 iast2 id1 id2 cdoc1 cdoc2 =
    foldr over (pure $ boardOf 0) moves
        where moves            = map anim $ zip3 brds locs1 locs2
              (_,brds,locs1,_) = unzip4 $ findTree iast1 id1 cdoc1
              (_,_,locs2,_)    = unzip4 $ findTree iast2 id2 cdoc2
              anim (b,l1,l2)   = activeMove (locDiff l1 l2) (pure b)
              locDiff (x1,y1) (x2,y2) = (x2-x1,y2-y1)

findTree :: Foldable f => IndexedAST f -> ID -> ChalkDoc -> ChalkDoc
findTree iast id cdoc = let maybeIAST = getSubAST id iast in
    case maybeIAST of
        Nothing   -> []
        Just iast -> let ids = getAllIDs $ iast in
                     filter (\(_,_,_,id1) -> elem id1 ids) cdoc
```

The `moveAnim` function constructs a `Move` when passed the same `ID` for `id1` and `id2`, or a `Copy` when the indexes are distinct (there are a couple other differences between the two in different parts of the system, however, such as in removal from a ChalkDoc, since the original node of a `Copy` must stay in place). In this function, `findTree` is used to find the original ChalkBoard `Board`'s and their locations before and after the transformation (given the before and after `ChalkDoc`'s, `cdoc1` and `cdoc2`). Each `Board` can then be animated by taking the difference of its two locations and using the Active command `activeMove`, which will move each `Board` into its new location over time. These separate animations for each `Board` are then combined using `foldr` and the ChalkBoard `over` function. The return type, `Active (Board UI)`, shows that all of the separate images have now been combined into a single animation. When multiple Transformation Algebra terms are present for a given step in a transformation, the resulting animations for each term will be combined in much the same way as these animations are here.

6.6 Code Example

Given the implementation details, let us turn now to a slightly more complicated example to see how this system works in practice, using a programming language AST instead of a list. This will hopefully showcase the capabilities and utility of the system a bit more, as well as display a much more substantial case where the system has proven useful.

Let us use, for this example, a simplified version of the Haskell Core language (simplified for the sake of showing the implementation details, not out of necessity). The AST for the language may look something like:

```
data Expr = Ap Expr Expr
          | Lam Var Expr  -- single argument lambdas
          | Case Expr (Alts Expr)
          | Let (Bind Expr) Expr
          | Con Con [Expr]
          | Prim Prim [Expr]
          | Var Var
          | Lit Literal

data Bind x = Bind Var x  -- non-recursive bindings
data Literal = LitI Int
             | LitF Float

type Var = String
type Con = String
type Prim = String
```

In this original system, `Traversable` instances for `Bind` and `Alts` already exist for ease of use. These instances and the rest of the implementation can all be found in Appendix B. If the instances had not already existed and were instead explicit over `Expr`, they could easily be changed to the versions above and given such instances trivially.

From this original structure, a new, `Functor` version of the data structure must be created:

```
data Expr' x = Ap' x x
             | Lam' Var x
             | Case' x (Alts x)
             | Let' (Bind x) x
             | Con' Con [x]
             | Prim' Prim [x]
             | Var' Var
             | Lit' Literal
```

The new version of `Expr` reuses the original structures for `Bind` and `Alts`, as well as many of the other sub-structures, as there is no reason for these to be recreated. Given this new data structure, we must now provide both a `MuRef` instance for `Expr` and a `Traversable` instance for `Expr'`:

```
instance MuRef Expr where
    type DeRef Expr = Expr'
    toFunctor expr = case expr of
            Ap e1 e2     -> Ap' e1 e2
            Lam v e      -> Lam' v e
            Case e alts  -> Case' e alts
            Let bind e2  -> Let' bind e2
            Con c es     -> Con' c es
            Prim p es    -> Prim' p es
            Var v        -> Var' v
            Lit l        -> Lit' l
```

```
instance Functor Expr’ where fmap = fmapDefault
instance Foldable Expr’ where foldMap = foldMapDefault
instance Traversable Expr’ where
  traverse f expr = case expr of
    Ap’ e1 e2    -> liftA2 Ap’ (f e1) (f e2)
    Lam’ v e     -> liftA (Lam’ v) (f e)
    Case’ e alts -> liftA2 Case’ (f e) (traverse f alts)
    Let’ b e2    -> liftA2 Let’ (traverse f b) (f e2)
    Con’ c es    -> liftA (Con’ c) (traverse f es)
    Prim’ p es   -> liftA (Prim’ p) (traverse f es)
    Var’ v       -> pure $ Var’ v
    Lit’ l       -> pure $ Lit’ l
```

These instances are rather straightforward and not complicated to write. They complete all of the necessary data structure level changes to the code.

Next up is the `Show` function. The simple `Show` function for the original system, defined in Appendix B, is modified slightly to become the necessary print function in the Active Transformation system. The new function may look something like:

```
tshow :: Int -> IndexedAST Expr’ -> [ASTSymbol]
tshow depth iast = case (getAST iast) of
  Ap’ e1 e2  -> gID "(" ++ tshow (depth+1) e1 ++ gID " " ++ tshow (depth+1) e2 ++ gID ")"
  Lam’ v e   -> gID ("(\\" ++ v ++ " -> ") ++ tshow (depth+1) e ++ gID ")"
  Case’ e as -> gID "case " ++ tshow (depth+1) e ++ gID " of" ++ newline id
                  ++ tshowAlts (depth+1) as id
  Let’ b e   -> gID "let " ++ sBind b ++ gID " in" ++ newline id ++ tabs (depth+1) id
                  ++ tshow (depth+1) e
  Con’ c es  -> gID ("(" ++ c) ++ concatMap sList es ++ gID ")"
  Prim’ p es -> gID ("(" ++ p) ++ concatMap sList es ++ gID ")"
  Var’ v     -> gID v
  Lit’ l     -> gID (show l)
    where sBind (Bind v e) = gID (v ++ " = ") ++ tshow (depth+1) e
          sList e = gID " " ++ tshow depth e
          gID = giveID id
          id = getID iast

tshowAlts :: Int -> Alts (IndexedAST Expr’) -> ID -> [ASTSymbol]
```

In this instance, the changes may appear to be a little tedious, but are at least rather straightforward. The main difference between this function and the original is the use of `gID` (a shorthand version of `giveID (getID iast)`). This is inserted to associate the current index with the strings that the current AST node is outputing. The other changes are a little smaller, such as using the new `Expr'` constructors in the case instead of the `Expr` ones, changing the types of the functions, and using the `newline` function to create new lines instead of the escape sequence. The `tabs` function used here is another built-in function that takes the number of tabs desired and an `ID` to associate them with.

While this function is relatively easy to construct using a pre-existing `Show` function, it is certainly an area where improvement could be made in the future by including pretty printing combinators. This inclusion would definitely make the system easier to use, and allow users to reuse previously constructed pretty printers. The interesting work in pretty printing, however, has already been done [18] [30] and therefore has not been a large focus for the Active Transformation system yet.

With the modified `Show` function ready to go, the next task is to adapt the transformation functions used in our system. One such transformation may be the removal of dead code, given here in its original form:

```
rmDeadCode :: Expr -> Expr
rmDeadCode expr = case expr of
    (Ap e1 e2)  -> Ap (rmDeadCode e1) (rmDeadCode e2)
    (Lam v e)   -> Lam v (rmDeadCode e)
    (Case e as) -> Case (rmDeadCode e) (fmap rmDeadCode as)
    (Let b e)   -> let (Bind v _) = b in
                   if (varUsed v e)
                     then Let (fmap rmDeadCode b) (rmDeadCode e)
                     else rmDeadCode e
    (Con c es)  -> Con c (map rmDeadCode es)
    (Prim p es) -> Prim p (map rmDeadCode es)
    _           -> expr

varUsed :: Var -> Expr -> Bool
```

This is a pretty standard definition of dead code removal for functional programs. The only unknown function in this definition is `varUsed`, which simply determines whether a given variable is used in the given expression or not.

This transformation, in order to work in the Active Transformation system, needs to be adapted slightly to operate inside the `Transform` monad, and therefore retain all of the information needed for animating. This updated version, written similarly to the original, may look like:

```
rmDC :: IndexedAST Expr' -> Transform Expr'
rmDC iast = case (getAST iast) of
    (Ap' e1 e2)  -> changeM iast $ liftM2 Ap' (rmDC e1) (rmDC e2)
    (Lam' v e)   -> changeM iast $ liftM (Lam' v) (rmDC e)
    (Case' e as) -> changeM iast $ liftM2 Case' (rmDC e) (traverse rmDC as)
    (Let' b e)   -> let (Bind v _) = b in
                    if (varUsed v e)
                      then changeM iast $ liftM2 Let' (traverse rmDC b) (rmDC e)
                      else replaceM iast $ rmDC e
    (Con' c es)  -> changeM iast $ liftM (Con' c) (mapM rmDC es)
    (Prim' p es) -> changeM iast $ liftM (Prim' p) (mapM rmDC es)
    _            -> return iast
```

Though it may not appear so at first, this code is actually structured very similar to the original. The `liftM` and `changeM` functions are used in much the same way as they were in the list example in Section 6.4, `liftM` letting us retain a similar shape while operating inside the monad, and `changeM` being used to designate a node where only its sub-nodes are changed. As might be expected, all instances of `Expr` are changed to `Expr'`, and `traverse` and `mapM` must also be used over `fmap` and `map` because of the monad. These straightforward changes are the large majority of modifications that need to be made. It should perhaps be noted, however, that all instances of `Expr` in `varUsed` must also be changed to `Expr'`, but otherwise it remains the same (since it does not modify the data structure).

Notice the use of replaceM, which works as discussed in Section 6.5. This is the only place where a modification is made to the data structure. In this case, the current let node is replaced with its in expression because the let variable is unused within that expression. In all other cases, the function is simply recursing on all available subtrees of the current node. If we wish, we can take advantage of this behavior to make the code cleaner by using the built-in Transform function, recurse:

```
rmDC :: IndexedAST Expr’ -> Transform Expr’
rmDC iast = case (getAST iast) of
  Let’ (Bind v _) e | not (varUsed v e) -> replaceM iast $ rmDC e
  _                                     -> recurse iast rmDC
```

This code makes explicit when changes are actually occurring to the AST. It tells the system when to replace the current node with another, and when to simply keep recursing. Although this new incarnation does not strictly resemble the original function, it can be useful and make implementation much quicker. recurse is especially helpful in transformations where only one or two cases will result in a change to the AST, and has the secondary benefit of making these changes quite explicit without needing to worry about or focus on excess recursive calls.

Although recurse is non-fundamental to the way our system works, it can certainly make using the system easier. recurse is defined as:

```
recurse::Traversable f=> IndexedAST f->(IndexedAST f->Transform f)->Transform f
recurse iast f = changeM iast $ traverse f (getAST iast)
```

Having completed dead code removal, this new transformation system can already create animations, but before we do so, let’s add in another transformation to make the example a little bit more interesting. A simple form of inlining for the language (which always inlines everything) could be:

```
inline :: IndexedAST Expr' -> Transform Expr'
inline iast = case (getAST iast) of
    Let' (Bind v e1) e2 -> do
                              ie1 <- inline e1
                              ie2 <- inline e2 >>= replaceVar v ie1
                              change iast $ Let' (Bind v ie1) ie2
    _                   -> recurse iast inline

replaceVar :: Var -> IndexedAST Expr' -> IndexedAST Expr' -> Transform Expr'
replaceVar var val iast = case (getAST iast) of
  Lam' v e | v == var -> return iast
  Case' e as          -> changeM iast $ liftM2 Case' (rep e) (repAlts as)
  Let' (Bind v e1) e2
        | v == var    -> do
                           re1 <- rep e1
                           change iast $ Let' (Bind v re1) e2
  Var' v | v == var   -> do
                           newIAST <- liftIO (reindex val)
                           addCopy (getID val) (getID newIAST)
                           replace iast newIAST
  _                   -> recurse iast rep
    where rep = replaceVar var val
```

Much of the bulkiness in `replaceVar`, defined fully in Appendix B, comes from checking all the cases where a variable might occur and cause recursion to stop. Like dead code removal, most changes from the original versions of these functions are simply due to being inside a monad.

`recurse` is also useful again, limiting the number of cases that need to be specified. If no changes are being made and recursion doesn't stop, then all of those cases can be lumped together.

This example also shows how to use more of the `Transform` monad functions. The `reindex` function is used to reindex the given `IndexedAST` so that it can be copied into another part of the AST than it appeared originally. This keeps indexes from repeating, which is essential in the correctness of animations. The `addCopy` function is also used in order to indicate to the system that the new subtree has actually been copied from another location.

As a final piece, the two converted transformations can now be used in a `main` function such as this one:

```
main = do
    let start = Let (Bind "a"
                        (Let (Bind "v" (Lit (LitI 1)))
                             (Var "v")))
                    (Var "a")
        anims = [inline, rmDC]

    animate "Arial.ttf" (tshow 0) start anims
```

This `main` function will apply the transformations of inlining and dead code removal, in that order, to the starting AST that is given, animating all of the changes on the screen. The starting AST can be any AST in the language, and the transformations can be any list of transformations converted into the `Transform` monad, of type `IndexedAST Expr' -> Transform Expr'`. This particular starting AST is one of the examples mentioned in Section 6.3, and snapshots from the resulting animation can be see in Figure 6.1.

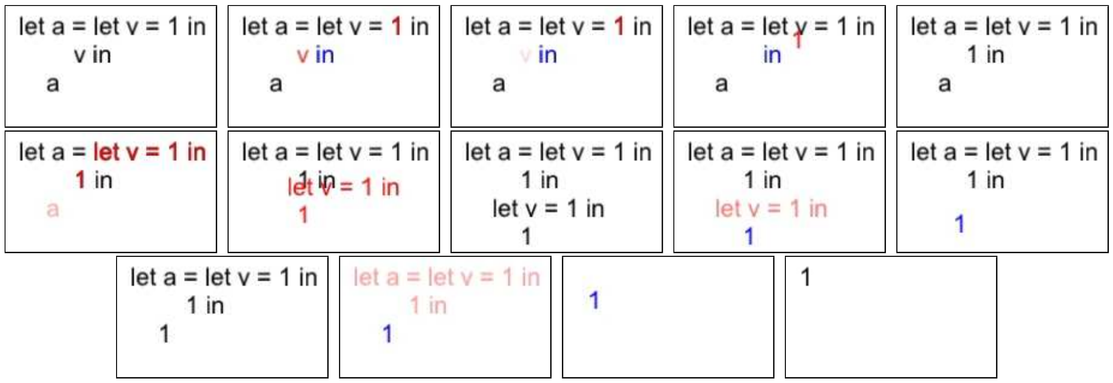

Figure 6.1. Code Transformation Animation

Chapter 7

Related Works

7.1 ChalkBoard

Functional image generation has a rich history, and there have been many previous image description DSLs for functional languages. Early work includes Reade [28], where he illustrates the combinational nature of functional programming using a character picture DSL in ML, resulting in ASCII art, Peter Henderson's functional geometry [14], Kavi Arya's functional animation [3], and more recently Findler and Flatt's slide preparation toolkit [11]. Functional languages are also used as a basis for a number of innovative GUI systems, the most influential one being the Fudgets toolkit [4]. ChalkBoard instead concerns itself with image generation and not GUIs, and intentionally leaves unaddressed the issues of interactivity and interactivity abstractions.

Elliott has been working on functional graphics and image generation for many years resulting in a number of systems, including TBAG [10], Fran [5], Pan [8] and Vertigo [7]. The aims of these projects are all aligned with ChalkBoard—making it easier to express patterns (sometimes in 2D, sometimes in 3D) using functional programs and embedded domain specific languages, and to aggressively optimize and compile these embedded languages. Elliott's ongoing work has certainly been influential to us, and ChalkBoard starts from the basic combinators provided in Pan. The main difference from the user's point of view is the adoption of the ability to aggressively optimize and compile these EDSLs for faster execution.

There are a number of imperative-style interfaces to graphic systems in Haskell. Hudak [16] used the HGL graphics Library, which exposes the basic imperative drawing primitives of Win32 and X11, allowing students to animate basic shapes and patterns. On top of this imperative base, Hudak shows how to build purely functional graphics and animations. OpenGL, GLUT and other standard graphics systems are also available to Haskell programmers, through FFI layers provided on `hackage.haskell.org`. The issue remains that these libraries behave like imperative graphics libraries.

7.2 Active

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 above in Section 7.1.

In terms of animation and the `Active` DSL, some similar systems that have been created are Slideshow [11] and the functional system presented by Kavi Arya [3]. 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 [17], 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) [9], 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.

7.3 Active Transformations

This Active Transformation system draws on a lot of different work from a variety of topics. One of the most fundamental influences was work done on pretty printing, such as that by John Hughes [18] and Philip Wadler [30]. While our print functions don't take advantage of their combinators yet in terms of output capabilities, the idea behind the work is similar. The Active Transformation system attempts to provide a mechanism and combinators for printing over *time*. The goal is to make the creation of these animations easy to integrate with a user's code in order to provide a simple yet powerful means of expressing the ideas conveyed in that code, much like how a pretty printer is used.

Another aspect of this project that was heavily influenced by the work of others was the data structure portion of the system. As described in Section 6.3, there were many different iterations of how users needed to adapt their current transformation system to fit into our framework, and each of these was at least partially influenced by the work of others. One of the first things we discovered was that maintaining an internal copy of the data structure was essential to capturing all of the intermediate transformation steps needed to animate a recursive transformation. Noticing this, we first looked at Data Reify in order to maintain this internal data structure [12]. This Data Reify implementation proved to be rather inefficient, however, when all of the structures we needed to represent were trees. Data Reify didn't allow us to build anything but graphs internally, so we began to look more into the work of Generics.Regular [25]. Using this style of approach, the user simply creates a mapping from their original datatype to a new `Functor` datatype. Data Reify, in essence, is a combination of this mapping and `traverse` (with some extra requirements), but since we require an instance of `Traversable`, it is easier to abstract this mapping out. Doing so makes it simpler to implement and allows tree structures to be constructed internally instead of graphs, in effect decreasing the user burden while increasing the system's efficiency.

There is also some overlap between the Active Transformation system and functional Strategic Programming [20]. While we do not support many strategies in our own work as of yet, we do use a similar generic traversal framework to strategic programming and so could potentially provide more of these in the future. This would allow for much quicker creation of transformation functions inside the framework of our system. We do have some rudimentary work in this area already, however, with functions such as `recurse`, as described in Section 6.6.

Chapter 8

Conclusions and Future Work

8.1 ChalkBoard

We have developed an OpenGL-based accelerator for a simple domain specific language for describing images. The language supports basic shapes, transparency, and color images, and our implementation also provides import and export of images in popular image formats. Our system generates images successfully and quickly, giving a many-fold improvement over our previous implementations of ChalkBoard.

In order to capture our DSL, we needed to invent our observable object `O`, and create a small, functor-like algebra for it. This idiom appears to be both general and useful, and merits further study. Lifting this idea into the space of applicative functors [24] is an obvious next step.

We intentionally chose OpenGL as a well-supported target platform. Most modern graphics cards are independently programmable beyond what is offered in OpenGL, through interfaces like OpenCL or CUDA. We use OpenGL because it offers the hardware support for what we specially want—fast polygon rendering—

rather than using general computation engines for polygon pushing. In the future, it would be interesting to consider in what ways these additional computational offerings could be used while at the same time retaining fast polygon support.

We believe that ChalkBoard is a viable and useful research platform for experimenting with applied functional programming. Most of the diagrams in this paper were rendered using ChalkBoard. A precursor to the version of ChalkBoard discussed in this paper is available on the Haskell package server, hackage, and development continues on the next version, which should be released soon.

8.2 Active

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 more precise combinators for the cueing and timing of animations. While the current structures have proven extremely useful, there are some instances in which the current `Active` API could be 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. In addition, the ChalkBoard Back End already has the capability to store and reuse these textures, the tricky part is just improving the compiler to incorporate this cross-compilation sharing of boards.

8.3 Active Transformation

As can be seen in Sections 6.4 and 6.6, we've implemented a couple different transformation systems inside the Active Transformation framework already. While not all of the implemented transformations are included in this thesis, every required part of both systems has been given and explained. This required code is relatively short as well as straightforward to implement, often looking very similar to the original code, or being a direct translation from one to the other. Given the expressive capabilities of the output animations, and how little the user has to know about animation to create them, the system appears to have quite a high ratio of usefulness to user burden. This is one of the main goals we were striving for when we first began the project, and we believe we have put forth a very solid first effort in this regard.

There are, however, still some improvements that could be made in this area. The simplest would be the creation of additional combinators to clean up some of the user code, both for the print function and their transformations. These seem to be two of the areas where the code can sometimes get messy with all of the different requirements placed on the user. Although the current versions are much, much cleaner than the original Active Transformation system required, there is definitely still work that could be done in this regard.

In a similar vein, most of the implementation that the user has to do is actually so straightforward and tedious that it could potentially be done automatically using a system like Template Haskell [29]. Using Template Haskell, we could likely generate the `Functor` version of a user's data structure for them, as well as all of the class instances. We could also take a stab at translating the user's print and transformation functions as well, though they may still need some addition input from the user (such as adding in Transformation Algebra functions). Overall, the power of Template Haskell may let us reduce the user burden for our system significantly further, to the point where we merely require a few lines to be added after a new .hs file is generated.

In other areas of practical system use, we are happy with the current look and functionality of the system in regards to correctly displaying animations. We feel that the current display does a pretty good job of differentiating between text that remains the same, transformations that are occurring, and other "text displacement" that must happen in order for the animation to occur. We would, however, like to offer more customizability for some of these aspects in the future. A lot of this customizability can be achieved nearly for free by simply abstracting out the current values we use and placing them into some sort of options list. These simple changes could allow the user greater control over both what is shown and how it is animated, and are areas we plan to address in the future.

The efficiency of our system also appears to be sufficient in practice so far. We have been able to animate all of the transformations we have attempted in nearly real-time, as well as create videos from these animations without loosing any speed. While we have noticed slight slow-downs when a large number of different strings need to be animated at once, this is mostly due to how the system interacts with Active specifically and some lingering inefficiencies that need to be worked out of Active itself. These slow-downs are not very noticeable, however, and all animations still run in nearly real-time.

8.4 Overall

Overall, through each different piece of this thesis, the goal has remained roughly the same: to show how functional languages and functional design patterns can be combined effectively and efficiently with computer graphics in order to create new and interesting opportunities for both. These opportunities should also have practical applications in the real world.

Beginning with ChalkBoard and Active, I believe this goal has mostly succeeded. A new, unique system has been created for expressing graphics and animation problems in a very functional and mathematical fashion. This type of expression can be useful in many situations, such as the Active Transformation system, and allows for certain types of images and animations to be scripted up very quickly.

I do believe there are some drawbacks as well. I believe that this system requires a big emphasis on this functional way of thinking, as well as a pretty in-depth knowledge of ChalkBoard, in order to benefit from this quick scripting of animations. Although I believe the idea of bringing graphics to functional programmers is sound, I still feel that the system could deal with some improvements to its usability, and a much easier learning curve.

In terms of efficiency, I feel that ChalkBoard is compares well to most other functional graphics systems. I do still feel there are some improvements to be made in this area, however, and especially when it comes to combining this system with Active. Active and ChalkBoard are built up separately for the most part, and I believe the two would benefit greatly from a closer connection. The ability to reuse parts of ChalkBoard images, as mentioned above, could speed up animation considerably.

Moving on to the Active Transformation system, I feel that this system really does show the effectiveness that functional graphics programming can have on practical, real-world applications. Compilers and interpreters are often written in a functional style, mirroring the grammars that they implement. Being able to quickly and easily attach animations to these functional systems could be really helpful, both in terms of learning the concepts and in terms of debugging.

Building libraries such as this on top of ChalkBoard and Active seems to be the really big use case for the system. Because ChalkBoard is functional, functional programmers can easily build libraries on top of the system that perform certain sets of scriptable animations. In this way, high-level users don't need to understand the specifics of ChalkBoard while the writers of the libraries can take the type to learn ChalkBoard in order to take advantage of its quick implementation time as well as efficient generation of animations.

Overall, I believe this system shows that new opportunities have been created by efficiently combining functional programming with computer graphics, and that these systems can have some highly useful applications.

Appendix A

```haskell
module Main where

import Prelude as P
import Graphics.ChalkBoard as CB
import Graphics.ChalkBoard.Font
import Control.Applicative (pure)

main = do
    font <- initFont "Arial.ttf" 0
    let (w,h) = (400,400)
    startChalkBoard [BoardSize w h] (\cb ->  animMain cb font 0.01 (w,h))

animMain cb font sz (w,h) = do
    --Set up the font labels
    (aLabel,aSP) <- label font sz ("a")
    (bLabel,bSP) <- label font sz ("b")
    (cLabel,cSP) <- label font sz ("c")
    (areaLabel,areaSP) <- label font sz ("area = c{^2}")
    (formulaLabel,formulaSP) <- label font sz ("c{^2} = a{^2} + b{^2}")

    --Set up the different parts of the animation
    let triangle345 = triangle (-0.2,0.15) (-0.2,-0.15) (0.2,-0.15)
        triLines = pointsToLine [(-0.2,0.15), (-0.2,-0.15), (0.2,-0.15), (-0.2,0.15)] 0.004
        mainTriangle = (choose (alpha black) transparent <$> triLines) `over`
                       (choose (alpha yellow) transparent <$> triangle345)
        movingTriangle = activeMove (0.15,0.2) $ activeScale (2/3) $
                         pure $ scale 1.5 $ mainTriangle

        position x b = rotate (-x*pi/2) $ move (0.15,0.2) b
        otherTriangles = [ activeAppear $ pure $ position i $ mainTriangle | i <- [1..3] ]
        fillSquare = activeAppear $ pure $ scale 0.095 $
                     choose (alpha yellow) transparent <$> square

        fadedTriangles = [ position i $ choose (withAlpha 0.6 white) transparent <$> triangle345
                         | i <- [0,1] ]
        slideLeft = activeAppear $ (activeMove (-0.3,-0.4) $ pure $ position 0 $ mainTriangle)
                                   `over` (pure $ head fadedTriangles)
        slideRight = activeAppear $ (activeMove (0.4,-0.3) $ pure $ position 1 $ mainTriangle)
                                    `over` (pure $ last fadedTriangles)
        newSquares = (move (0.15, -0.15) $ scale 0.4 $ square) `over`
                     (move (-0.2, -0.2) $ scale 0.3 $ square)
        newLines = pointsToLine [(-0.05,-0.35), (0.35,-0.35), (0.35,0.05), (-0.05,0.05),
                                (-0.05,-0.35), (-0.35,-0.35), (-0.35,-0.05), (-0.05,-0.05)] 0.004
        fadeInSquares = (fadeIn black 1 newLines) `over` (fadeIn yellow 0.9 newSquares)

    --The font parts of the animation
    let a = move (-0.25,-0.03) $ makelbl aSP aLabel
        b = move (-0.04,-0.195) $ makelbl bSP bLabel
        c = move (0.005,0.005) $ makelbl cSP cLabel
        positions2 aPos bPos = (move aPos a) `over` (move bPos b)
        positions3 aPos bPos cPos = (move aPos a) `over` (move bPos b) `over` (move cPos c)

        firstABC = activeTempAppear $ pure $ positions3 (-0.1,0) (-0.03,-0.08) (0.005,0.005)
        areaEq = activeAppear $ pure $ move (-0.4,0.35) $ makelbl aSP areaLabel
        secondABC = activeTempAppear $ pure $ move (0.15, 0.2) $
                    positions3 (0.06,0) (0,0.06) (0.05,-0.38)
        thirdABC = activeTempAppear $ pure $
                   positions2 (-0.15,-0.2) (-0.15,-0.2) `over` positions2 (0.45,-0.36) (0.4,0)

        finalABC = activeAppear $ pure $
                   position2 (-0.15,-0.2) (0.18,-0.2) `over` position2 (0.03,-0.36) (0.4,0)
        formula = activeAppear $ pure $ move (0.15,0.35) $ makelbl aSP formulaLabel

    --Set up the animation ordering/timing
    let anim = flicker [ wait 0.5
                       , taking 1 $ firstABC
                       , taking 1 $ movingTriangle
                       , wait 0.5
                       , taking 0.75 $ otherTriangles !! 0
                       , taking 0.75 $ otherTriangles !! 1
                       , taking 0.75 $ otherTriangles !! 2
                       , taking 1.5 $ fillSquare `over` secondABC `over` areaEq
                       , taking 1 $ slideLeft
                       , wait 0.5
                       , taking 1 $ slideRight
                       , taking 1 $ thirdABC
                       , taking 1 $ fadeInSquares `over` thirdABC
                       , taking 3 $ finalABC `over` formula
                       ]

    --Pick the animation you would like to see and turn it into a play object
    playObj <- byFrame 29.97 anim

    --Start the video write stream
    sid <- startWriteStream cb $ ffmpegOutCmd "pythagorean-test.avi"
```

```
    --Run the animation
    let loop = do
            mbScene <- play playObj
                case mbScene of
                    Just scene -> do
                        -- To screen:
                        drawChalkBoard cb $ unAlphaBoard (boardOf (o (RGB 1 1 0.8))) scene
                        -- To file:
                        frameChalkBoard cb sid
                        loop
                    Nothing  -> return ()

    loop

    --Close the video write stream and exit
    endWriteStream cb sid
    exitChalkBoard cb



--Create a font board at the right size
makelbl :: Float -> Board UI -> Board (RGB -> RGB)
makelbl size lbl =  color black $ scale (0.7) $ scale (1/(size*25)) $ lbl

color :: O RGB -> Board UI -> Board (RGB -> RGB)
color rgb brd = (\ ui -> withAlpha ui rgb) <$> brd
```

Appendix B

```
{-# LANGUAGE TupleSections #-}
{-# LANGUAGE TypeFamilies #-}

import Control.Monad (liftM, liftM2)
import Control.Monad.Trans (liftIO)
import Data.Foldable (Foldable, foldMap)
import Data.Traversable (Traversable, traverse, fmapDefault, foldMapDefault)
import Data.Functor
import Control.Applicative

import IndexedAST
import Transform
import ChalkDoc
import Animate


-----------------------------------------------------------------------------------------------
--------------------------------- Original Core-Like Language ---------------------------------
-----------------------------------------------------------------------------------------------

data Expr = Ap Expr Expr
          | Lam Var Expr  -- single argument lambdas
          | Case Expr (Alts Expr)
          | Let (Bind Expr) Expr
          | Con Con [Expr]
          | Prim Prim [Expr]
          | Var Var
          | Lit Literal
instance Show Expr where
        show = show' 0

type Var = String
type Con = String
type Prim = String

data Bind x = Bind Var x  -- non-recursive bindings

data Literal = LitI Int
             | LitF Float
```

```
instance Show Literal where
        show (LitI int) = show int
        show (LitF float) = show float

data Alts x = C [Calt x]
            | L [Lalt x]
            | CwDef [Calt x] (Default x)
            | LwDef [Lalt x] (Default x)

type Calt x = (Con, [Var], x)
type Lalt x = (Literal, x)
type Default x = (Var, x)


instance Functor Bind where fmap = fmapDefault
instance Foldable Bind where foldMap = foldMapDefault
instance Traversable Bind where traverse f (Bind v x) = Bind v `liftA` f x
instance Functor Alts where fmap = fmapDefault
instance Foldable Alts where foldMap = foldMapDefault
instance Traversable Alts where
  traverse f as = case as of
    C cas -> C `liftA` cs cas
    L las -> L `liftA` ls las
    CwDef cas defalt -> liftA2 CwDef (cs cas) (def defalt)
    LwDef las defalt -> liftA2 LwDef (ls las) (def defalt)
      where
        cs = traverse (\ (con, vs, x) -> (con, vs,) `liftA` f x)
        ls = traverse (\ (lit, x) -> (lit,) `liftA` f x)
        def (v, x) = (v,) `liftA` f x


------------------------------------------------------------------------------------------------
------------------------------------ Functor-Type Additions ------------------------------------
------------------------------------------------------------------------------------------------

data Expr' x = Ap' x x
             | Lam' Var x
             | Case' x (Alts x)
             | Let' (Bind x) x
             | Con' Con [x]
             | Prim' Prim [x]
             | Var' Var
             | Lit' Literal

instance Functor Expr' where fmap = fmapDefault
instance Foldable Expr' where foldMap = foldMapDefault
instance Traversable Expr' where
        traverse fn es = case es of
                Ap' e1 e2    -> liftA2 Ap' (fn e1) (fn e2)
                Lam' v e     -> liftA (Lam' v) (fn e)
                Case' e alts -> liftA2 Case' (fn e) (traverse fn alts)
                Let' b e2    -> liftA2 Let' (traverse fn b) (fn e2)
                Con' c es    -> liftA (Con' c) (traverse fn es)
                Prim' p es   -> liftA (Prim' p) (traverse fn es)
                Var' v       -> pure $ Var' v
                Lit' l       -> pure $ Lit' l

instance MuRef Expr where
        type DeRef Expr = Expr'
        toFunctor expr = case expr of
                Ap e1 e2          -> Ap' e1 e2
                Lam v e           -> Lam' v e
                Case e alts       -> Case' e alts
                Let bind e2       -> Let' bind e2
                Con c es          -> Con' c es
                Prim p es         -> Prim' p es
                Var v             -> Var' v
                Lit l             -> Lit' l


------------------------------------------------------------------------------------------------
--------------------------------------- Original Printer ---------------------------------------
------------------------------------------------------------------------------------------------

show' :: Int -> Expr -> String
show' depth e = case e of
    Ap expr1 expr2  -> "(" ++ show' (depth+1) expr1 ++ " " ++ show' (depth+1) expr2 ++ ")"
    Lam var expr    -> "(\\" ++ var ++ " -> " ++ show' (depth+1) expr ++ ")"
    Case expr alts  -> "case " ++ show' (depth+1) expr ++ " of\n" ++ showAlts (depth+1) alts
    Let bind expr   -> "let " ++ sBind bind ++ " in\n" ++ tabs (depth+1) ++ show' (depth+1) expr
    Con con exprs   -> "(" ++ con ++ concatMap (\x -> " " ++ (show' 0 x)) exprs ++ ")"
    Prim prim exprs -> "(" ++ prim ++ concatMap (\x -> " " ++ (show' 0 x)) exprs ++ ")"
    Var var         -> var
    Lit lit         -> show lit
        where tabs d = concat ["    " | i<-[1..d]]
              showBind (Bind var expr) = var ++ " = " ++ show' (depth+1) expr

showAlts :: Int -> Alts Expr -> String
showAlts depth alts = case alts of
    C calts -> concatMap (showCalt depth) calts
    L lalts -> concatMap (showLalt depth) lalts
    CwDef calts (var, expr) -> concatMap showCalt calts ++ tabs ++ var ++ " -> " ++ tabs
                               ++ show' (depth+1) expr ++ "\n"
    LwDef lalts (var, expr) -> concatMap showLalt lalts ++ tabs ++ var ++ " -> " ++ tabs
                               ++ show' (depth+1) expr ++ "\n"
        where tabs = concat ["    " | i<-[1..depth]]
              showCalt (con, vars, expr) = tabs ++ con ++ " " ++ show vars ++ " -> "
                                         ++ show' (depth+1) expr ++ "\n"
              showLalt (lit,expr) = tabs ++ show lit ++ " -> " ++ show' (depth+1) expr ++ "\n"
```

```
------------------------------------------------------------------------------------------------
--------------------------------------- Modified Printer ---------------------------------------
------------------------------------------------------------------------------------------------

tshow' :: Int -> (IndexedAST Expr') -> [ASTSymbol]
tshow' depth iast = case (getAST iast) of
  Ap' e1 e2  -> gID "(" ++ tshow' (depth+1) e1 ++ gID " " ++ tshow' (depth+1) e2 ++ gID ")"
  Lam' v e   -> gID ("(\\" ++ v ++ " -> ") ++ tshow' (depth+1) e ++ gID ")"
  Case' e as -> gID "case " ++ tshow' (depth+1) e ++ gID " of" ++ newline id
                      ++ tshowAlts (depth+1) as id
  Let' b e   -> gID "let " ++ sBind b ++ gID " in" ++ newline id ++ tabs (depth+1) id
                      ++ tshow' (depth+1) e
  Con' c es  -> gID ("(" ++ c) ++ concatMap (\x -> gID " " ++ tshow' depth x) es ++ gID ")"
  Prim' p es -> gID ("(" ++ p) ++ concatMap (\x -> gID " " ++ tshow' depth x) es ++ gID ")"
  Var' v     -> gID v
  Lit' l     -> gID (show l)
    where gID = giveID id
          id = getID iast
          sBind (Bind v e) = gID (v ++ " = ") ++ tshow' (depth+1) e


tshowAlts :: Int -> Alts (IndexedAST Expr') -> ID -> [ASTSymbol]
tshowAlts depth alts id = case alts of
  C calts -> concatMap sCalt calts
  L lalts -> concatMap sLalt lalts
  CwDef calts (v,e) -> concatMap sCalt calts ++ sDef v e
  LwDef lalts (v,e) -> concatMap sLalt lalts ++ sDef v e
    where gID = giveID id
          sDef v e = tabs depth id ++ gID (v ++ " -> ") ++ tshow' (depth+1) e ++ newline id
          sCalt (c,vs,e) = tabs depth id ++ gID (c ++ " " ++ show vs ++ " -> ")
                              ++ tshow' (depth+1) e ++ newline id
          sLalt (l,e) = tabs depth id ++ gID (show l ++ " -> ") ++ tshow' (depth+1) e
                              ++ newline id


------------------------------------------------------------------------------------------------
----------------------------------- Dead Code Removal Original --------------------------------
------------------------------------------------------------------------------------------------

rmDeadCode' :: Expr -> Expr
rmDeadCode' expr = case expr of
    Ap e1 e2  -> Ap (rmDeadCode' e1) (rmDeadCode' e2)
    Lam v e   -> Lam v (rmDeadCode' e)
    Case e as -> Case (rmDeadCode' e) (fmap rmDeadCode' as)
    Let b e   -> let (Bind v _) = b in
                 if (varUsed' v e)
                     then Let (fmap rmDeadCode' b) (rmDeadCode' e)
                     else rmDeadCode' e
    Con c es  -> Con c (map rmDeadCode' es)
    Prim p es -> Prim p (map rmDeadCode' es)
    _           -> expr
```

```
varUsed' :: Var -> Expr -> Bool
varUsed' var expr = case expr of
    Ap expr1 expr2  -> (varUsed' var expr1) || (varUsed' var expr2)
    Lam v expr      -> if (var == v)
                           then False
                           else varUsed' var expr
    Case expr alts  -> (varUsed' var expr) || (varUsedAlts var alts varUsed')
    Let b expr2     -> let (Bind v expr1) = b in
                       if (var == v)
                           then varUsed' var expr1
                           else (varUsed' var expr1) || (varUsed' var expr2)
    Con con exprs   -> or (map (varUsed' var) exprs)
    Prim prim exprs -> or (map (varUsed' var) exprs)
    Var v           -> var == v
    Lit l           -> False

varUsedAlts :: Var -> Alts a -> (Var -> a -> Bool) -> Bool
varUsedAlts v alts vuFn = case alts of
    C calts -> or [if (elem v vs) then False else (vuFn v e) | (c,vs,e) <- calts]
    L lalts -> or [vuFn v e | (_,e) <- lalts]
    CwDef calts (dv,de) -> (or [if (elem v vs) then False else (vuFn v e) | (c,vs,e) <- calts])
                               || if (v == dv) then False else (vuFn v de)
    LwDef lalts (dv,de) -> (or [ (vuFn v expr) | (lit, expr) <- lalts])
                               || if (v == dv) then False else (vuFn v de)


-----------------------------------------------------------------------------------------------
------------------------------------ Dead Code Removal Modified --------------------------------
-----------------------------------------------------------------------------------------------

-- With "recurse" function
rmDC :: IndexedAST Expr' -> Transform Expr'
rmDC iast = case (getAST iast) of
    Let' (Bind v _) e | not (varUsed v e) -> replaceM iast $ rmDC e
    _                                     -> recurse iast rmDC


-- Without "recurse" function
rmDC :: IndexedAST Expr' -> Transform Expr'
rmDC iast = case (getAST iast) of
    (Ap' e1 e2)  -> changeM iast $ liftM2 Ap' (rmDC e1) (rmDC e2)
    (Lam' v e)   -> changeM iast $ liftM (Lam' v) (rmDC e)
    (Case' e as) -> changeM iast $ liftM2 Case' (rmDC e) (traverse rmDC as)
    (Let' b e)   -> let (Bind v _) = b in
                    if (varUsed v e)
                        then changeM iast $ liftM2 Let' (traverse rmDC b) (rmDC e)
                        else replaceM iast $ rmDC e
    (Con' c es)  -> changeM iast $ liftM (Con' c) (mapM rmDC es)
    (Prim' p es) -> changeM iast $ liftM (Prim' p) (mapM rmDC es)
    _            -> return iast
```

```
varUsed :: Var -> IndexedAST Expr' -> Bool
varUsed var iast = case (getAST iast) of
    Ap' expr1 expr2  -> (varUsed var expr1) || (varUsed var expr2)
    Lam' v expr      -> if (var == v)
                            then False
                            else varUsed var expr
    Case' expr alts  -> (varUsed var expr) || (varUsedAlts var alts varUsed)
    Let' b expr2     -> let (Bind v expr1) = b in
                        if (var == v)
                            then varUsed var expr1
                            else (varUsed var expr1) || (varUsed var expr2)
    Con' con exprs   -> or (map (varUsed var) exprs)
    Prim' prim exprs -> or (map (varUsed var) exprs)
    Var' v           -> var == v
    Lit' l           -> False


-----------------------------------------------------------------------------------------------
--------------------------------------- Inlining Original -------------------------------------
-----------------------------------------------------------------------------------------------

inline' :: Expr -> Expr
inline' code = case code of
    Ap expr1 expr2            -> Ap (inline' expr1) (inline' expr2)
    Lam var expr              -> Lam var (inline' expr)
    Case expr alts            -> Case (inline' expr) (inlineAlts alts)
    Let (Bind var expr1) expr2 -> Let (Bind var (inline' expr1))
                                     (replaceVar' var (inline' expr1) (inline' expr2))
    Con con exprs             -> Con con (map inline' exprs)
    Prim prim exprs           -> Prim prim (map inline' exprs)
    _                         -> code
      where inlineAlts alts = case alts of
                C calts -> C [(con, vars, (inline expr)) | (con, vars, expr) <- calts]
                L lalts -> L [(lit, (inline expr)) | (lit, expr) <- lalts]
                CwDef calts (dv, de) -> CwDef [(c,vs,inline' e) | (c,vs,e) <- calts]
                                              (dv,inline' de)
                LwDef lalts (dv, de) -> LwDef [(lit,inline' e) | (lit,e) <- lalts]
                                              (dv,inline' de)


replaceVar' :: Var -> Expr -> Expr -> Expr
replaceVar' var val expr = case expr of
    Ap expr' atom        -> Ap (replaceVar' var val expr') (replaceVar' var val atom)
    Lam v expr'          -> if (v == var)
                                then Lam v expr'
                                else Lam v (replaceVar' var val expr')
    Case expr' alts      -> Case (replaceVar' var val expr') (replaceAlts var val alts)
    Let (Bind v e) expr' -> if (v == var)
                                then Let (Bind v (replaceVar' var val e)) expr'
                                else Let (Bind v (replaceVar' var val e))
                                         (replaceVar' var val expr')
    Con con exprs        -> Con con (map (replaceVar' var val) exprs)
    Prim prim exprs      -> Prim prim (map (replaceVar' var val) exprs)
    Var v                -> if (v == var)
                                then val
                                else Var v
    Lit l                -> Lit l
    where replaceAlts v val alts = case alts of
              C calts -> C [(c,vs, if (elem v vs) then e else (replaceVar' v val e))
                           | (c,vs,e) <- calts]
              L lalts -> L [(lit, replaceVar' v val e) | (lit, e) <- lalts]
              CwDef calts (dv,de) -> CwDef [(c,vs, if elem v vs then e else replaceVar' v val e)
                                               | (c,vs,e) <- calts]
                                           (dv, if (v == dv) then de else replaceVar' v val de)
              LwDef lalts (dv,de) -> LwDef [(lit, replaceVar' v val e) | (lit, e) <- lalts]
                                           (dv, if (v == dv) then de else replaceVar' v val de)


------------------------------------------------------------------------------------------------
---------------------------------------- Inlining Modified -------------------------------------
------------------------------------------------------------------------------------------------

inline :: IndexedAST Expr' -> Transform Expr'
inline iast = case (getAST iast) of
    Let' (Bind v e1) e2 -> do
                              ie1 <- inline e1
                              ie2 <- inline e2 >>= replaceVar v ie1
                              change iast $ Let' (Bind v ie1) ie2
    _                   -> recurse iast inline


replaceVar :: Var -> IndexedAST Expr' -> IndexedAST Expr' -> Transform Expr'
replaceVar var val iast = case (getAST iast) of
    Lam' v e | v == var            -> return iast
    Case' e as                     -> changeM iast $ liftM2 Case' (rep e) (repAlts as)
    Let' (Bind v e1) e2 | v == var -> do
                                        re1 <- rep e1
                                        change iast $ Let' (Bind v re1) e2
    Var' v | v == var              -> do
                                        newIAST <- liftIO (reindex val)
                                        addCopy (getID val) (getID newIAST)
                                        replace iast newIAST
    _                              -> recurse iast rep
        where rep = replaceVar var val
              repAlts alts = case alts of
                  C calts           -> liftM C $ traverse fCalt calts
                  CwDef calts (v,e) -> liftM2 CwDef (traverse fCalt calts) (fDef v e)
                  L lalts           -> liftM L $ traverse fLalt lalts
                  LwDef lalts (v,e) -> liftM2 LwDef (traverse fLalt lalts) (fDef v e)
                      where fLalt (l,e) = liftM (l,) $ rep e
                            fCalt (c,vs,e) = if elem var vs
                                                 then return (c,vs,e)
                                                 else liftM (c,vs,) $ rep e
                            fDef v e = if v == var
                                           then return (v,e)
                                           else liftM (v,) $ rep e
```

```
------------------------------------------------------------------------------------------------
---------------------------------------- Main Function ----------------------------------------
------------------------------------------------------------------------------------------------

main = do
        let start = Let (Bind "a"
                        (Let (Bind "v" (Lit (LitI 1)))
                             (Var "v")))
                    (Var "a")
            anims = [inline, rmDC]

        animate "Arial.ttf" (tshow' 0) start anims
```

References

[1] cairo. http://www.cairographics.org/.

[2] The Glasgow Haskell Compiler. http://haskell.org/ghc/.

[3] K. Arya. Processes in a functional animation system. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 382–395, New York, NY, USA, 1989. ACM.

[4] M. Carlsson and T. Hallgren. Fudgets: a graphical user interface in a lazy functional language. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 321–330, New York, NY, USA, 1993. ACM.

[5] C. Elliott. From functional animation to sprite-based display. In *Practical Aspects of Declarative Languages*, 1999.

[6] C. Elliott. Functional images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, Mar. 2003.

[7] C. Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004.

[8] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

[9] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[10] C. Elliott, G. Schechter, R. Yeung, and S. Abi-Ezzi. TBAG: A high level framework for interactive, animated 3D graphics applications. In *SIGGRAPH*, 1994.

[11] R. B. Findler and M. Flatt. Slideshow: functional presentations. *J. Funct. Program.*, 16(4-5):583–619, 2006.

[12] A. Gill. Type-safe observable sharing in Haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, Sep 2009.

[13] A. Gill and G. Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251, March 2009.

[14] P. Henderson. Functional geometry. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 179–187, New York, NY, USA, 1982. ACM.

[15] P. Hudak. Modular domain specific languages and tools. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[16] P. Hudak. *The Haskell school of expression : learning functional programming through multimedia*. Cambridge University Press, New York, 2000.

[17] P. Hudak. An algebraic theory of polymorphic temporal media. In *Practical Aspects of Declarative Languages*, pages 1–15, 2004.

[18] J. Hughes. The design of a pretty-printing library. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 53–96, London, UK, 1995. Springer-Verlag.

[19] M. P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 97–136, London, UK, 1995. Springer-Verlag.

[20] R. Lämmel and J. Visser. Design Patterns for Functional Strategic Programming. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, 5 2002. ACM Press. 14 pages.

[21] D. Leijen and E. Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, Oct. 1999.

[22] K. Matlage and A. Gill. ChalkBoard: Mapping functions to polygons. In *Proceedings of the Symposium on Implementation and Application of Functional Languages*, Sep 2009.

[23] K. Matlage and A. Gill. Every animation should have a beginning, a middle, and an end. In *Proceedings of Trends in Functional Programming*, May 2010.

[24] C. McBride and R. Patterson. Applicative programing with effects. *Journal of Functional Programming*, 16(6), 2006.

[25] T. V. Noort, A. Rodriguez, S. Holdermans, J. Jeuring, and B. Heeren. A lightweight approach to datatype-generic rewriting. In *ACM SIGPLAN Workshop on Generic Programming*, 2008.

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

[27] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 50–61, New York, NY, USA, 2006. ACM.

[28] C. Reade. *Elements of functional programming.* Addison-Wesley, Wokingham, England, 1989.

[29] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In M. M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, Oct. 2002.

[30] P. Wadler. A prettier printer. In *Journal of Functional Programming*, pages 223–244. Palgrave Macmillan, 1998.