ChalkBoard: Mapping Functions to Polygons

Kevin Matlage and Andy Gill

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

Abstract. ChalkBoard is a domain specific language for describing images. The ChalkBoard language is uncompromisingly functional and encourages the use of modern functional idioms. ChalkBoard uses off-the-shelf graphics cards to speed up rendering of functional descriptions. In this paper, we describe the design of the core ChalkBoard language, and the architecture of our static image generation accelerator.

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 [1] 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[2], for example, is an efficient image language, based on imperatively drawing shapes onto a canvas, with a Haskell IO port of the API. We are explicitly interested in exploring purely functional representations of images and want to understand if they can be made efficient.

The ChalkBoard project is an attempt to bridge the gap between the clear specification style of a language with first-class images, and a practical and efficient rendering engine. Though systems like cairo offer the ability to use created images as new components, the hook 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. This paper describes a successful attempt to target one such functional image description language to the widely supported OpenGL standard.

Figure 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.

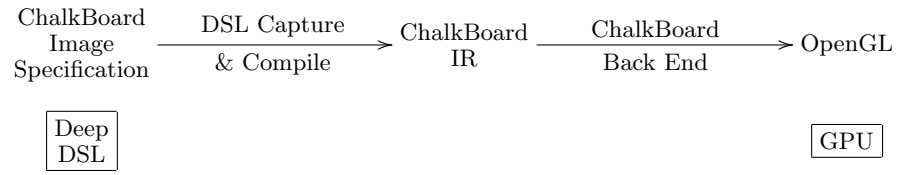


Fig. 1. The ChalkBoard Architecture

Both subsystems of ChalkBoard are written in Haskell, and are compiled using GHC [3]. We have plans for other back ends for ChalkBoard that share the same Intermediate Representation (IR), but in this paper we focus on how we use OpenGL to achieve fast *static* image generation from a purely functional specification. Specifically, this paper 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 [4], but with an abstract principal data type to facilitate introspection.
- To allow DSL capture, we needed 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`, 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.

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

2 Functional Image Generation

As a first example of ChalkBoard, consider drawing a partially-transparent red square over a partially-transparent green circle. The image we wish to draw looks like Figure 2.

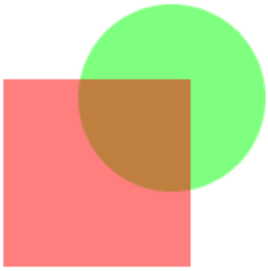


Fig. 2. Square over Circle

We can describe the picture in Figure 2 using the following ChalkBoard specification.

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

This fragment of code specifies that

- A circle (line 5), colored green with 50% transparency (line 4), is moved slightly up and to the right (line 3).
- A square (line 8), colored red and also with 50% transparency (line 7), is moved slightly down and to the left (line 6).
- Both these shapes are placed on a transparent background (line 4 & 7).
- The red square is placed on top of the green circle, which is on top of a white background (line 1).

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

`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 1. Boards and Type Synonyms in ChalkBoard

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

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

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

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

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

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

`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`. We can consider `O` to have this trivial definition, though `O` is actually an abstract type.

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

We will redefine our `O` type and consider its implementation in section 6.

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

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

Our boards of `RGBA`, or images of transparent color, can be combined (overlaid) into new boards of `RGBA`.

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

The combinator `over` is used to lay one `Board` onto another `Board`.

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

Also, these boards of `RGBA` can be (value) transformed into `Board RGB`, true color images. Again, this translation is done using our map-like operator, `<$>`, and a point-wise `unAlpha`.

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

`unAlpha` removes the alpha component of `RGBA`, leaving `RGB`.

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

This is a significant restriction over the generality provided in Pan, and one we intend to lift in a future version of ChalkBoard.

Finally, we also have a primitive for constructing a (conceptually infinite) `Board` of a constant value, which has the type

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

Using these combinators, ChalkBoard constructs images by combining primitives and translating them in both *space* and *representation*, ultimately building a `Board RGB`. ChalkBoard also supports importing of images as `Board RGBA`, and other `Board` creation primitives, for example font support, are planned.

3 ChalkBoard Example: Drawing Lines

Now that we have our fundamental primitives and combinators, we can build more interesting, complex combinators. We can build a `box` combinator which takes two points and constructs a region box between them.

```
box :: (Point,Point) -> Board Bool
box ((x0,y0),(x1,y1)) = polygon [(x0,y0),(x1,y0),(x1,y1),(x0,y1)]
```

Using this `box` function, we can build a straight line region builder.

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

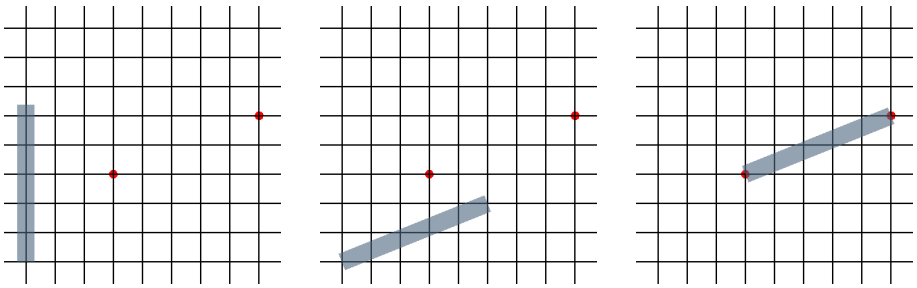


Fig. 3. How `straightline` works

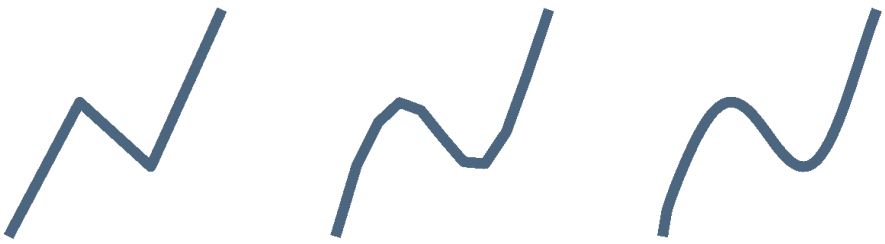
Figure 3 illustrates how `straightline` operates. Assuming the dots are the target start and end points of our straight line, and the bottom left intersection is (0,0), we draw a box of the correct size and width (a), rotate it (b), then move it to the correct location. There are other ways of writing `straightline`, but this is compositionally written.

Now we can draw lines between arbitrary points of arbitrary thicknesses, and we can simulate curved lines using many of these straight segments together. To do this, we have a function, `outerSteps`, 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 here is a set of 6 values, or the 5 steps of size 0.2 between 0 and 1. Using `outerSteps`, we *sample* a function that encodes the curved line. We draw the curved line by drawing straight lines between the sample points, and filling in any infidelity at the joins between these lines 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
```



3 segments 10 segments 50 segments

Fig. 4. Examples of `functionline`

Figure 4 gives examples of `functionline` being used on a function with 3, 10, and 50 segments. Figure 4 shows how with a higher number of samples the quality of rendering the curved line improves.

In fact, all these functions are already defined with the ChalkBoard library, but are given here as an illustration of the flavor of ChalkBoard and how it compromises between continuous boards, and discrete components on these boards. Collectively, ChalkBoard combinators give a clean and productive environment for scripting images.

4 Considerations with Compiling ChalkBoard

ChalkBoard has a 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 exactly the model used in Pan [4], on which the ChalkBoard language is based. Pan uses this semantic model of a function directly to implement a `Board`, but though we too share the same semantic model, we want a different implementation.

- In ChalkBoard, `Board` is *abstract*, specifically to admit the possibility of future representation 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` transformation. This is a significant restriction, especially when compared to the full expressiveness of `Pan`, and one we intend to revisit in the future.
- ChalkBoard is intended as a system for constructing complex images, consisting of perhaps tens of thousands of individual components. The functional representation precludes this being efficiently rendered, though techniques like the worker/wrapper transformation [5] could be used to translate an explicit function into something like our abstract representation.

Consider the ChalkBoard image in Figure 2. We will use this example to illustrate a number of the challenges with optimizing a chalkboard specification of an image. Figure 2 was generated by first building a `Board Bool` for each of the two basic shapes, using our `<$>` on each board to convert it to a `Board RGBA`, and then using `move` to move these boards to the desired locations. Finally, these two boards are overlaid, using `over`, and the alpha channel is removed for rendering as a color image.

Our plan of attack is to augment the representation of `Board` internally, replace the shape primitives with more complex information about what is being rendered, and attempt to translate our tree of operations into the drawing of polygons. Rendering polygons is something OpenGL does extremely efficiently. We will allow fallback onto the slower pixel sampling if needed. The longer term goal is to allow most user-written ChalkBoard specifications to be compiled efficiently, and tell the user when a board is being rendered using the fallback sampling. Currently, however, the ChalkBoard language discussed in this paper is limited enough that no fallback is ever actually needed.

5 Capturing the Domain Specific Language ChalkBoard

ChalkBoard is a language that describes boards, and these boards are functor-like. The language provides mechanisms for (1) describing the creation of boards, and for creating boards from boards, using (2) spacial transformations and (3) a functor-style `map`. In this section, we will consider how to express all three of these language aspects in a deep embedding of the ChalkBoard language.

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

```
data Board a where
        Constant :: 0 a -> Board a
        ...
```

Here we use GADT [6] syntax for `Board` because of the ability to declare constructors that are specialized to monomorphic instances, and we present each of the principal constructions individually.

Primitives shapes, like circles and squares, are regions, or `Board Bool`. They are represented inside the data structure `Board` using a list of points that mark a convex boundary of the region.

So for primitives we have the constructor

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

We can see the use of the monomorphic constructor here.

Representing squares is easy using the four corner points, but what about the points around a circle? There are infinitely many corner points on a circle, that is, there are infinitely many points that match, for example, the equation

$$\sqrt{x^2 + y^2} = 0.5$$

Graphic rendering systems approximate the circle using a small number of corners on a small circle and a larger number of corners on a larger circle. At this point, we appeal to classical functional programming and *defer* the decision about how many points to use to approximate the circle using a function. The type of the `Polygon` constructor is

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

The `Float` argument is the resolution of the final polygon. Specifically, it is an approximation of how many pixels a line of unit length would affect. We can now give a definition for `square` and `circle`.

```
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 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.

Spacial transformations are also handled using a single `Board` constructor, which combines all the relevant transformations.

```
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 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 can be used to successfully *typecheck* a ChalkBoard-like language and *construct* a representation of `Board`, but we run into a fatal problem when we try to walk the `Board` tree during compilation. Here `b` can be any type; we have lost the type information about what it was. We can expect an `<$>` over a `Board Bool` to have a completely different operational behavior than an `<$>` over a `Board RGB`! When walking our tree and performing our abstract attribute grammar interpretation, we get stuck.

We address this problem by assuming our pointwise function is a function over our observable type `O`. So we have the corrected

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

(<$>) = Fmap
```

and we also require our `O` type to hold runtime type information, as described in section 6. It looks like using `O` just postpones the problem, and does not solve it. By forcing pointwise manipulations to be expressed in the `O` language, we can observe what the user intended, without requiring that *every* function be directly promoted into a `Board` equivalent.

We can now construct basic abstract syntax trees for ChalkBoard, using our `Board` data type. For example

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

represents the creation of a red square on a green background. It constructs the `Board` tree

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

We can extract the specific polygon points contained inside the `Polygon` constructor when we are compiling for OpenGL because in context we know the size of the target image. The challenge is how do we extract the first argument to `Fmap`? To do so, we use our observable type, `O`.

6 `O`, the Observable

The data type `O`, which we nickname *the observable*, is the mechanism we use to observe interesting values. The idea is that an observable can simultaneously have a shallowly and deeply embedded interpretation of the same constructed expression. We can use the shallow interpretation to directly extract the value of any `O` expression, and we examine the deep interpretation to observe how a result was constructed. Specifically, the data definition for `O` is

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

`E` is a syntax tree of possible `O` expressions. By limiting the ways of building `O`, we allow `O` expressions to be built only 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
       ...
```

We use implicit recursion inside `Expr` so we can share this functor-style representation of `Expr` between our expression inside `O` and the internal compiler; see [7] for a detailed description of this design decision.

To illustrate how we build our primitives, the definition of `choose` is

```
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)
```

We can also build primitive `O` values using our `Obs` class.

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

That is, only instances of `Obs` can construct objects of type `O a`. ChalkBoard uses this to provide a way of taking a value of `Bool`, `RGB`, `RGBA`, or `Float` and lifting it using the `o` function into the `O` structure. In many ways, this is a restricted version of `return` for monads, or `pure` for applicative functors [8].

So how do we actually observe a function? By giving a dummy argument and 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))))
```

where the structure of the `choose` and the arguments are completely explicit. Using this trick, we can observe the function argument to our functor because we require 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 can be parsed into

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

thus allowing compilation.

7 ChalkBoard IR

We use both the inspection of the `Board` structure itself and the observation of `O` objects 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 pushing abilities. We translate into a ChalkBoard Intermediate Representation (CBIR), then interpret CBIR on the fly into OpenGL commands.

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	::= $\text{pointmap}_1, \text{pointmap}_2, \ldots, \text{pointmap}_n$	$n \geq 3$
	point	::= (u,v)	
	points	::= $\text{point}_1, \text{point}_2, \ldots, \text{point}_n$	$n \geq 3$
	x,y	::= int	
	u,v	::= float	

Fig. 5. ChalkBoard Intermediate Representation

Figure 5 gives the syntax of CBIR. There are two main commands.

- **allocate**, which allocates a new, fixed-sized buffer in graphics memory.
- **buffersplat**, which takes a polygon from one buffer and renders it onto another buffer. **buffersplat** takes a source and 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. It can be used to simulate scaling, translation, or rotation. This is the command that does the majority of the work inside ChalkBoard.

As well as the two principal instructions, there are also commands 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.

8 Compiling ChalkBoard to ChalkBoard IR

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

- a basic quality of the required picture argument. In this way, a specific `Board` can know how much it is contributing to the final image;
- any rotations, translations, or scaling performed *above* this node;
- and an identifier for a pre-allocated *target* buffer.

We then have a set of compilation schemes for each possible `Board` type. In general we perform the following compile steps

- `Constant Board`s compile into a single splat onto the target board.
- `Move`, `Scale` and `Rotate` update the inherited attribute context, recording the movement required, and then compile the sub-board with this new context.
- For `Over`, we interpret the child boards 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. For `Board RGB`, `Over` simply compiles the first board.
- For `Fmap` we infer the type of the map by inferring the type of the functional argument to `Fmap`, using the capabilities provided by `O`. We then emit the bridging code for the `Fmap`, compiled from the reified functional argument, and call the relevant compilation scheme for the sub-board's type.

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

The key trick in the compiler is compiling a `Fmap` which translates a `Board Bool` into a `Board RGB` (or `Board RGBA`). For example

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

`f` here has type `O Bool -> O RGB`. To compile the inner `Board Bool` syntax tree, we need to compute the `True` (or foreground) color, and `False` (or background) color. To find these colors, we simply apply `f` to `True`, and also apply `f` to `False`, giving the two colors present on the board.

9 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 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 a specified 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. 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 we use 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, we wrote a small micro-benchmark, called ChalkMark, 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 we further tune the CBBE, we expect the difference between the two methods to remain this significant. Thankfully, most systems today that would use ChalkBoard 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 over the destination texture instead of texture-mapping a second source texture onto it. This removes the needless overhead of allocating extra 1x1 texture boards just to use as basic colors.

Though there remains considerable scope for optimizing the use of the OpenGL pipeline, and there also are many improvements that could be made to our ChalkBoard compiler, we have found the current performance is more than sufficient for simple animations.

10 Related Work

Functional image generation has a rich history, and there have been many previous image description languages for functional languages. Early work includes Reade [9], 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 [11], Kavi Arya's functional animation [12], and more recently Findler and Flatt's slide preparation toolkit [13]. Functional languages are also used as a basis for a number of innovative GUI systems, the most influential one being the Fudgets toolkit [14]. 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 [15], Fran [16] Pan[17] and Vertigo [18]. The aims of these projects are 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 by 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 [10] 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.

11 Conclusion and Future Work

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 implementation 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 [8] is an obvious next step.

Our implementation has many possible improvements and enhancements. In the current ChalkBoard compilation scheme, we do not actual make use of the `buffersplat` primitive, but we expect to do so in the near future as we continue to enhance our compiler. Specifically, we will use data reification [7] to allow the observation of sharing of boards and other intermediate values, and `buffersplat` will allow us to compile this sharing into CBIR. There are a number of interesting compilation tradeoffs to explore here.

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, we will consider in what way we can use these additional computational offerings while at the same time retaining the fast polygon support.

We want to use ChalkBoard for drawing educational animations. Right now, a `Board` is a two-dimensional abstract object. Could we make it a three-dimensional object with time as the third dimension, and build on the ideas from functional reactive programming (FRP) [19]? If we limit ourselves to only animations, and not reactivity, could we simplify the current complexities and challenges surrounding FRP events, and build a small animation language on top of ChalkBoard? In particular, we are interested in describing animations that are performed on streaming video sources.

We believe that ChalkBoard is a viable and useful research platform for experimenting with applied functional programming. All the diagrams in this paper that did not require font support were rendered using ChalkBoard. The version of ChalkBoard discussed in this paper is available on the Haskell package server hackage, and development continues on the next version.

Acknowledgments

We would like to thank the members of CSDL at KU, and the IFL referees for useful and detailed feedback. We would especially like to thank Conal Elliott, who both provided a starting point for this work, and extensive feedback and comments about the paper and the research direction.

References

1. Peyton Jones, S., ed.: Haskell 98 Language and Libraries – The Revised Report. Cambridge University Press, Cambridge, England (2003)
2. : cairo. http://www.cairographics.org/
3. : The Glasgow Haskell Compiler. http://haskell.org/ghc/
4. Elliott, C.: Functional images. In: The Fun of Programming. "Cornerstones of Computing" series. Palgrave (March 2003)
5. Gill, A., Hutton, G.: The worker/wrapper transformation. Journal of Functional Programming **19**(2) (March 2009) 227–251
6. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for gadts. In: ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM (2006) 50–61
7. Gill, A.: Type-safe observable sharing in Haskell. In: Proceedings of the 2009 ACM SIGPLAN Haskell Symposium. (Sep 2009)
8. McBride, C., Patterson, R.: Applicative programing with effects. Journal of Functional Programming **16**(6) (2006)
9. Reade, C.: Elements of functional programming. Addison-Wesley, Wokingham, England (1989)
10. Hudak, P.: The Haskell school of expression : learning functional programming through multimedia. Cambridge University Press, New York (2000)
11. Henderson, P.: Functional geometry. In: LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming, New York, NY, USA, ACM (1982) 179–187
12. Arya, K.: Processes in a functional animation system. In: FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture, New York, NY, USA, ACM (1989) 382–395
13. Findler, R.B., Flatt, M.: Slideshow: functional presentations. J. Funct. Program. **16**(4-5) (2006) 583–619
14. Carlsson, M., Hallgren, T.: Fudgets: a graphical user interface in a lazy functional language. In: FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture, New York, NY, USA, ACM (1993) 321–330
15. Elliott, C., Schechter, G., Yeung, R., Abi-Ezzi, S.: TBAG: A high level framework for interactive, animated 3D graphics applications. In: SIGGRAPH. (1994)
16. Elliott, C.: From functional animation to sprite-based display. In: Practical Aspects of Declarative Languages. (1999)
17. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. Journal of Functional Programming **13**(2) (2003)
18. Elliott, C.: Programming graphics processors functionally. In: Proceedings of the 2004 Haskell Workshop, ACM Press (2004)
19. Elliott, C., Hudak, P.: Functional reactive animation. In: International Conference on Functional Programming. (1997)