CS457 Functional Programming
Mark Jones Winter 2012
Homework 8

Russell Miller

March 8, 2012

QUESTION 1

Prove the following law holds for all f, e, and g.

```
foldr f e . map g = foldr (f . g) e
```

I'm going to rewrite this law using a variable xs to represent the list argument for each side. We'll say that this law is P(xs).

```
P(xs) = (foldr f e . map g) xs = (foldr (f . g) e) xs
```

Similar to the proof we did in class, we will need to prove 3 cases: `P([])`, `P(⊥)`, and `P(xs)` $\Rightarrow$ `P(x:xs)`, where $\perp$ is execution that does not terminate properly.

First we need the definition of `foldr`:

```
foldr f z []     = z                         (foldr.0)
foldr f z (x:xs) = f x (foldr f z xs)        (foldr.1)
```

(found in the Prelude using Hugs's `:f` command.)

The definition of map we defined in class.

```
map f []           = []                      (map.0)
map f (x:xs)       = f x : map f xs          (map.1)
```

Great! `foldr` and `map` are defined for `[]` and `x:xs`. Now we need to come up with laws about `map` and `foldr` for the case of $\perp$.

In class we talked about `map f ⊥`.

```
map f ⊥       = ⊥ (map.⊥)
```

By looking at the definition of `foldr`, it is clear that it will work the same. It does something to each element of a list, and recursively works through the list the same way map does just that. Thus:

```
foldr f z ⊥  = ⊥ (foldr.⊥)
```

Now we're ready to prove the property for the 3 cases talked about earlier.

P([]):
We want to show that
(foldr f e . map g) [] = foldr (f . g) e []

```
LHS = (foldr f e . map g) []
    = foldr f e (map g [])               {definition of .}
    = foldr f e []                       {by map.0}
    = e                                  {by foldr.0}
RHS = foldr (f . g) e []
    = e                                  {by foldr.0}
LHS = RHS
```

P(⊥):
We want to show that
(foldr f e . map g) ⊥ = foldr (f . g) e ⊥

```
LHS = (foldr f e . map g) ⊥
    = foldr f e (map g ⊥)                {definition of .}
    = foldr f e ⊥                        {by map.⊥}
    = ⊥                                  {by foldr.⊥}
RHS = foldr (f . g) e ⊥
    = ⊥                                  {by foldr.⊥}
LHS = RHS
```

P(xs) ⇒ P(x:xs):
We want to show that
(foldr f e . map g) (x:xs) = foldr (f . g) e (x:xs)

```
LHS = (foldr f e . map g) (x:xs)
    = foldr f e (map g (x:xs))           {definition of .}
    = foldr f e (g x : map g xs)         {by map.1}
    = f (g x) (foldr f e (map g xs))     {by foldr.1}
RHS = foldr (f . g) e (x:xs)
    = (f . g) x (foldr (f . g) e xs)     {by foldr.1}
    = (f . g) x ((foldr f e . map g) xs) {induction, P(xs)}
    = f (g x) (foldr f e (map g xs))     {definition of .}
LHS = RHS
```

■

Practical application of this law? Well on the left side of this law is a foldr and a map. In order to apply functions f and g it goes over the input list twice. The better version on the right, which we have shown to be equivalent, only goes through the list once and applies both functions to each element.

QUESTION 2

Using the definition of `stretch` and `rotate` (below), prove the following law holds for all values of θ and `m`.

```
rotate θ . stretch m = stretch m . rotate θ
```

The functions `stretch` and `rotate`, as defined by Mark in class:

```
stretch m src = \(u,v) -> src (u/m, v/m)
rotate θ src  = \(u,v) -> src (c*u - s*v, s*u + c*v)
                          where c = cos θ
                                s = sin θ
```

In order to prove this we'll add an argument to the law, on both sides.

```
(rotate θ . stretch m) src = (stretch m .rotate θ) src
```

```
LHS = (rotate θ . stretch m) src
    = rotate θ (stretch m src)           {definition of .}
    = rotate θ (\(u,v) -> src (u/m, v/m)) {definition of stretch}

    = (\(w,x) -> src (c*w - s*x, s*w + c*x)) (\(u,v) -> (u/m, v/m))
          where c = cos θ
                s = sin θ                {definition of rotate}
    = \(w,x) -> src (c*(w/m) - s*(x/m), s*(w/m) + c*(x/m))
          where c = cos θ
                s = sin θ                {by applying the \(u,v) function}
RHS = (stretch m . rotate θ) src
    = stretch m (rotate θ src)           {definition of .}
    = stretch m (\(u,v) -> src (c*u - s*v, s*u + c*v))
          where c = cos θ
                s = sin θ                {definition of rotate}
    = (\(w,x) -> src (w/m, x/m)) (\(u,v) -> src (c*u - s*v, s*u + c*v))
          where c = cos θ
                s = sin θ                {definition of stretch}
    = (\(w,x) -> src ((c*w - s*x)/m, (s*w + c*x)/m))
          where c = cos θ
                s = sin θ                {by applying the \(u,v) function}
    = (\(w,x) -> src ((c*w)/m - (s*x)/m, (s*w)/m + (c*x)/m))
          where c = cos θ
                s = sin θ                {math! (distribute the (1/m))}
    = (\(w,x) -> src (c*(w/m) - s*(x/m), s*(w/m) + c*(x/m)))
          where c = cos θ
                s = sin θ                {math! (associativity of * and /)}
LHS = RHS
```

■

Would this law be valid if stretch and rotate were being applied to rectangular grids of pixels? Yes. The thing we're applying these functions to is an arbitrary collection of Points. Regardless of the shape, the stretch and rotate are associative because of the associativity of the multiplication and division happening to the points.

QUESTION 3

Rewrite the Image functions from the class slides, as a data type.
We have a definition of each shape given on the slides:

```
rectangle          :: Float -> Float -> color -> Image color
rectangle h w col = ...
```

We're calling our new data type ImageD.

```
data ImageD color = Rectangle Float Float color
                  | Square Float color
                  | Circle Float color
                  | Semi color
                  | Over (ImageD color) (ImageD color)
                  | Mask (ImageD color) (ImageD color)
                  | Translate Point (ImageD color)
                  | Stretch Float (ImageD color)
                  | XStretch Float (ImageD color)
                  | YStretch Float (ImageD color)
                  | XReflect (ImageD color)
                  | YReflect (ImageD color)
                  | Rotate Float (ImageD color)
```

Show how we can convert an ImageD to an Image.

```
render                    :: ImageD color -> Image color
render (Rectangle h w col) = \(u,v) -> if u>=0 && u<=w && v>=0 && v<=h
                                        then Just col
                                        else Nothing
```

Or, assuming we have the definition of the `rectangle` function...

```
render (Rectangle h w col) = rectangle h w col
```

I'll demonstrate a combinator by writing `over`. Again we use the original function. This requires that `top` and `bot` be converted from `ImageD` to `Image`, which is exactly what `render` does!

```
render (Over top bot)      = over (render top) (render bot)
```

Lastly, a transformation. We'll rewrite `stretch`. The pattern is the same.

```
render (Stretch m src)     = stretch m (render src)
```

Strengths and weaknesses of `data` versus functions?
Well it appears there is a simple pattern to add functionality to this `ImageD` data type we've defined. That works well because we already had the functions written. Had we not, it would have been a lot of work to operate on the `color` values.
Defining a show function for these would only need to be done once. However, adding a new primitive would require expanding the definition of the `ImageD` data type and also (possibly) writing a function for render to call. Which, of course, means we'd need to add a pattern to the render function.