BIM (Basic Image Manipulator)
Un eDSL orientado a la manipulación de mapas de bits escrito en Haskell

AGUSTÍN MISTA
Universidad Nacional de Rosario
Análisis de Lenguajes de Programación
Rosario, 28 de Enero de 2016

Introducción

La manipulación digital de imágenes es usualmente un proceso gráfico, ésto es, si consideramos que las fotos de nuestras últimas vacaciones necesitan algún retoque, las abrimos en algún software específico de manipulación de imágenes con interfaz gráfica, navegamos entre ventanas de diálogo, movemos controles deslizantes, y rellenamos tantas cajas de texto como sea necesario hasta encontrar el fino equilibrio que buscamos, siendo éste un trabajo que puede llevar un tiempo considerable.

En éste informe veremos un enfoque distinto del proceso de manipulación digital de imágenes, orientado a la programación de pequeños scripts mediante BIM, un lenguaje de dominio específico embebido en Haskell, que nos permite describir el proceso de retocar o crear una nueva imagen de manera sencilla, con una sintaxis concisa y una semántica composicional diseñada para encadenar operaciones fácilmente.

Analizaremos también las decisiones importantes que se tomaron a la hora de implementar este software, sus limitaciones, y aspectos que podrían mejorarse en un futuro.

¿Cómo funciona?

Para describir el funcionamiento de este software, lo primero que debemos considerar es como se representaron los conceptos fundamentales sobre los que trabajamos:

- El proceso de manipulación de imágenes es representado por una computación que puede tener éxito, devolviendo una imagen, o fallar, devolviendo un mensaje con información del error sucedido. (`Result`)
- Un mapa de bits (`Bitmap`) es representado como una matriz de píxeles.
- Un `píxel` es representado como una tupla de valores enteros (R,G,B) con valores permitidos entre 0-255 (8 bits).
- Un punto en el plano (`Point2D`) es representado como una tupla de valores enteros (X,Y). Convenimos que el origen de coordenadas se encuentra en la esquina superior izquierda de una imagen.

Por otro lado, se abstrajeron las operaciones fundamentales que una representación de imágenes adecuada `a` debe poseer (`Image a`):

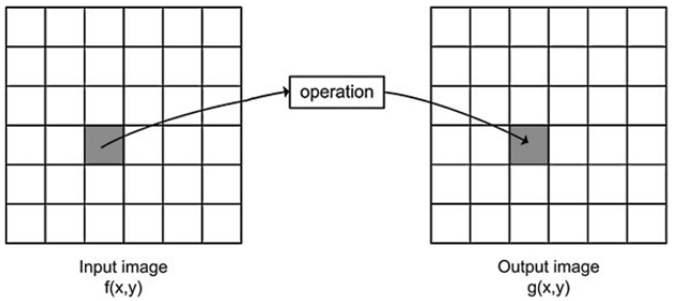
Combinadores:

- Una función para crear imágenes, a partir de un tamaño, y una función que asigne un píxel a cada punto dentro de la misma:

```
create :: (Point2D -> Pixel) -> Point2D -> Result a
```

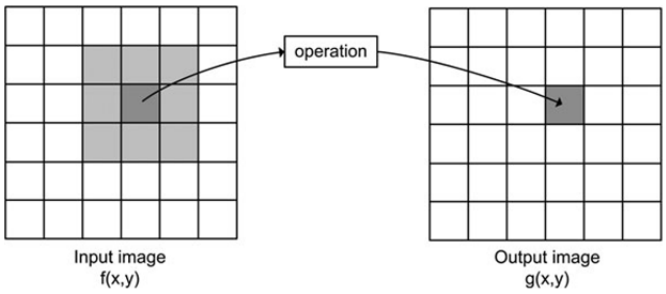
- Una función para manipular imágenes puntualmente, que a partir de (quizá) una imagen, y una función que mapea píxeles en píxeles, devuelva (quizá) una nueva imagen:

```
pixelTrans :: (Point2D -> Pixel -> Pixel) -> Result a -> Result a
```



- Una función para manipular imágenes localmente, que a partir de (quizá) una imagen, y una función que mapea el área alrededor de un píxel (con determinado radio) en un nuevo píxel, y devuelva (quizá) una nueva imagen:

```
localTrans :: Int -> (a -> Pixel) -> Result a -> Result a
```



Observaciones:

- Una función que a partir de una imagen y una dimensión (X o Y), devuelva el ancho o alto respectivamente de la misma:

```
(~>) :: a -> Dim -> Int
```

- Una función que dada una imagen, y una posición en el plano devuelva el píxel en esa posición (o error en caso de que la posición esté fuera de rango):

```
(!) :: a -> Point2D -> Pixel
```

- Una función que permita consumir una imagen retornando un valor, recorriendo píxel por píxel y combinándolos mediante una función y un elemento *"neutro"*:

```
fold :: (Pixel -> b -> b) -> b -> a -> b
```

Luego, dimos una instancia de imagen para nuestra representación de mapas de bits (`Image Bitmap`), con esto conseguimos una forma de implementar operaciones sobre imágenes sin tener en cuenta la implementación subyacente, y permitiendo cambiar ésta por una de mejor performance de ser necesario, sin preocuparnos por lo implementado encima de ella.

Lo anterior describe un centro de computaciones puras, es decir, no tiene acceso al mundo real. Para poder operar con imágenes guardadas en disco, y poder guardar las nuevas imágenes, implementamos operaciones que interactúen con el sistema de archivos:

- Carga (quizá) una imagen a partir de la ruta a la misma, parseando las cabeceras y desempaquetando la información almacenada de manera continua en una matriz de píxeles (`Bitmap`):

```
load :: String -> IO (Result Bitmap)
```

- Guarda una imagen a disco, generando cabeceras y empaquetando en forma continua los píxeles de la misma, o bien, informa de un error ocurrido mientras se manipulaba la misma:

```
save :: String -> Result Bitmap -> IO ()
```

Finalmente, para darle utilidad real a nuestro lenguaje, derivamos (a partir de las operaciones fundamentales) varios grupos de operaciones de uso cotidiano que operan sobre una o más imágenes:

- Aritméticas: suma, resta, multiplicación, mezcla, etc..
- Geométricas: cortar, unir, rotar, reflejar, etc..
- Puntuales: canales, modificar, negativo, conversión a B&W, etc..
- Filtros: gaussiano, detección de bordes, enfoque, grabado, etc..
- Paleta: lista de colores dominantes, paleta de colores.
- Histograma: guardar a disco, plotear en gnuplot.

Estas operaciones se describen en detalle al final de este informe

Instalación

Antes de instalar BIM, es necesario contar previamente con:

1. Haskell Platform
2. Git (opcional)
3. gnuplot (opcional)

Este software puede instalarse a partir de su código fuente alojado en el repositorio de GitHub del mismo:

```
> git clone https://github.com/agustinmista/bim.git
> cd bim
> cabal install
```

Manual de uso

El uso de éste software es similar al de cualquier librería de Haskell, una vez instalado podemos importarlo, y crear procesos de manipulación de imágenes fácilmente usando notación `do`. Usualmente, el proceso de manipular imágenes inicia cargando una o varias imágenes asignándoles un nombre:

```
test = do
   lena   <- load "img/lena.bmp"
   baboon <- load "img/baboon.bmp"
   ...
```

y termina realizando una o más operaciones de salida sobre alguna imagen obtenida usando las operaciones sobre las mismas:

```
test = do
   lena <- load "img/lena.bmp"
   ...
   ...
   plotHist lena
   save "lenaNegative.bmp" (negative lena)
```

Si nuestras expresiones se vuelven extensas, podemos darles nombre usando la expresión `let` logrando mayor claridad en el código:

```
test = do
   lena   <- load "img/lena.bmp"
   baboon <- load "img/baboon.bmp"
   ...
   let lena2 = sharpen (mono lum lena)
       lena3 = modify (+50) (median lena)
       left  = baboon#>R <|> baboon#>G
                         </>
               baboon#>B <|> (baboon + lena)
       right = lena2 </> negative lena3
   ...
   save "output.bmp" (left <|> right)
```

Adicionalmente, si necesitamos hacer alguna observación de una imagen para poder operar, podemos desempaquetar la misma del entorno de manipulación:

```
test = do
   lena <- load "img/lena.bmp"
   let colors = lena >>= \img -> palette (img~>X, img~>X %> (1/6)) 6 lena
   save "paletteLena.bmp" (lena </> colors)
```

o bien,

```
test = do
   lena <- load "img/lena.bmp"
   let cropped = do img <- lena
                    crop (10, 10) (img~>X - 10, img~>Y - 10) lena
   save "paletteLena.bmp" cropped
```

Organización

A continuación se describe la jerarquía de archivos que conforman este software, como así también el propósito de cada uno de ellos:

- **doc/**: Contiene éste informe y las imágenes incrustadas en él.
- **img/**: Contiene imágenes de prueba.
- **tests/**: Contiene tests a cada operación, usando las imágenes de prueba.
- **src/**: Contiene el código fuente de este software, organizado en los siguientes módulos de Haskell:
  - **Pixel**: Define la implementación de los píxeles, y las operaciones relacionadas a la manipulación de los mismos.
  - **Colors**: Define 16 colores básicos, que se corresponden con aquellos predefinidos en **HTML**.
  - **Image**: Define el TAD **Image**, la manipulación de imágenes tolerante a fallas (**Result**) y reexporta los dos módulos vistos arriba.
  - **Bitmap**: Define la implementación de los mapas de bits y las operaciónes de entrada/salida sobre los mismos.
  - El directorio **src/Operations/** describe los módulos que implementan operaciones sobre imágenes derivadas a partir de las primitivas:
    - **Arithmetic**: Implementa operaciones aritméticas.
    - **Geometric**: Implementa operaciones geométricas.
    - **Point**: Implementa operaciones que usan transformaciones puntuales.
    - **Filters**: Implementa operaciones que usan transformaciones locales, e implementa convolution kernels (a.k.a. filtros genéricos).
    - **Palette**: Implementa operaciones para obtener colores dominantes y generar paletas de los mismos.
    - **Histogram**: Implementa operaciones para calcular el histograma de una imagen, y aquellas necesarias para interactuar con Gnuplot usando el wrapper de la librería EasyPlot
  - **src/BIM.hs**: Instancia los mapas de bits a **Image**, implementando las operaciones primitivas sobre imágenes, y reexporta todos los módulos vistos anteriormente, siendo éste el módulo principal de nuestro software, y el único que necesita importarse para ser usado por completo.

Decisiones de diseño

Durante el período de diseño e implementación de BIM debieron tomarse algunas decisiones que resultan importantes en el resultado final del mismo, a continuación algunas de ellas:

- Se decidió que las operaciones primitivas sobre imágenes deberían abstraerse mediante una clase **Image**, dando lugar a múltiples posibles implementaciones de las mismas (cada una con sus pros y contras), y operaciones derivadas que usen la interfaz de la misma, independientemente de la implementación usada.
- Para la lectura y escritura de las imágenes a disco, se usó la librería **Codec.BMP** que implementa funciones útiles para el parseo y generación de cabeceras.
- Para la representación de matrices de píxeles, se usó la librería **Data.Matrix** que cuenta con operaciones dedicadas a operar eficientemente sobre matrices.
- Debido a que el formato `bmp` almacena los píxeles en filas desde abajo hacia arriba, una conversión directa de la imagen a una matriz de píxeles utilizando las operaciones primitivas de **Codec.BMP** y **Data.Matrix** resulta en una imagen volteada horizontalmente. Dado esto, se decidió que la operación de lectura invierta el orden de las filas con el fin de representarse en la matriz de píxeles de la manera intuitiva, volviéndose a invertir a la hora de ser guardadas nuevamente a disco.
- Para el caso del cálculo del histograma de una imagen, se decidió usar la librería EasyPlot y el software de ploteo numérico Gnuplot, ya que esto daría mayor versatilidad al mismo, pudiéndose visualizar en una ventana interactiva, o bien ser guardado a disco en múltiples formatos de imagen.
- Para la representación de los píxeles, podría haberse usado el tipo **Word8** para los valores de cada canal, sabiendo que los valores permitidos están entre 0-255 (8 bits), pero surgieron dos cuestiones:
  - Al operar aritméticamente sobre píxeles puede darse el caso de que el nuevo valor no caiga dentro de los valores permitidos, dándose lo que se conoce como *píxel overflow*. Para resolver esto, se debe escoger una técnica adecuada según el comportamiento que deseamos modelar, siendo las más comunes las de *wrapping* y *saturation*. En la primera, los valores se comportan de manera

cíclica, volviendo a 0 si se sobrepasa el valor máximo; en la última, cualquier resultado que exceda los límites permitidos queda atada a valores mínimos y máximos. En nuestro caso deseamos que los píxeles tengan el comportamiento esperable, es decir, si agregamos mucho brillo a una imagen ésta resulta sobreexpuesta, pero nunca con valores oscuros resultado un comportamiento cíclico, siendo este último comportamiento del tipo **Word8** ante un overflow.

  - La segunda cuestión, aunque menos importante, resulta ser la gran cantidad de conversiones entre tipos de datos que deberían aplicarse si se usase el tipo **Word8**, ya que los valores numéricos usados en las operaciones son generalmente de tipo **Int**

  Dado esto, decidimos usar un tipo de datos que soporte operaciones aritméticas de mayor tamaño (**Int**), y validar los resultados mediante una función auxiliar (**validate**), todo esto a costa de un mayor consumo de memoria.

- A la hora de implementar las transformaciones locales de imágenes, debimos elegir como representaríamos los píxeles que quedasen fuera de la imagen, pero dentro del radio de la transformación, fenómeno que sucede en los bordes de la misma. Existen varias aproximaciones para resolver este problema conocido como *padding*:

  - Computar la transformación sólo en aquellos píxeles cuyos vecinos están bien definidos, sin agregar bordes, resultando en una imagen más pequeña que la original.
  - **Zero padding**: se extiende la imagen usando píxeles nulos (negros). Sencillo de implementar, pero produce transformaciones que oscurecen los bordes.
  - **Mirror extension**: al llegar a un borde, se usan los píxeles más cercanos del borde contrario. Puede producir efectos no deseados si los bordes opuestos de una imagen son muy distintos.
  - **Border extension**: se extiende la imagen con un borde del radio de la transformación, donde cada píxel del mismo replica a su vecino más cercano dentro de la imagen original. Relativamente sencillo de implementar y produce buenos resultados en la mayoría de los casos, por lo que fue el método que usamos para implementar el padding en nuestro software.

Limitaciones

La mayor limitación de BIM es que, actualmente, sólo puede operar sobre imágenes en formato `bmp`, por lo que intentar cargar imágenes en cualquier otro formato resultaría en un error de parseo. Ésto sería lo más importante a resolver en un futuro cercano.

Otra limitación apreciable es que si bien el formato `bmp` admite el manejo de transparencia, mediante píxeles **RGBA** de 32bits, **Codec.BMP** no soporta la escritura de éste tipo de imágenes. Por lo que actualmente el canal *alpha* es ignorado.

Bibliografía

- **Image Processing Learning Resources**.
  Robert Fisher, Simon Perkins, Ashley Walker, Erik Wolfart
  University of Edinburgh
  `http://homepages.inf.ed.ac.uk/rbf/HIPR2/hipr_top.htm`

- **Fundamentals of Image Processing**.
  Ian T. Young, Jan J. Gerbrands, Lucas J. van Vliet
  Delft University of Technology
  `http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/TUDELFT/FIP2_3.pdf`

- **Basic image processing methods**.
  Marcin Kiełczewski
  Poznań University of Technology
  `http://etacar.put.poznan.pl/marcin.kielczewski/VBC5.pdf`

- **GIMP Documentation**.
  `http://docs.gimp.org/2.6/en/`

- **Image Kernels explained visually**.
  Victor Powell
  `http://setosa.io/ev/image-kernels/`

- **Padding of Borders**.
  Czech Technical University in Prague
  `http://radio.feld.cvut.cz/matlab/toolbox/images/linfilt4.html`

Apéndice

Descripción de operadores según el módulo al que pertenecen:

Image

- `create`: crea una imagen a partir de una función.

```
create :: (Point2D -> Pixel)  -- Generator function
       -> Point2D             -- Size
       -> Result a
```

- `pixelTrans`: transforma una imagen mediante una función puntual. Ésta además recibe el tamaño de la imagen para mayor versatilidad.

```
pixelTrans :: (Point2D -> Pixel ->  Pixel) -- Transforming function
           -> Result a
           -> Result a
```

- `localTrans`: transforma una imagen localmente mediante una función que recibe una imagen de la zona alrededor de cada pixel, con determinado radio.

```
localTrans :: Int           -- Radius
           -> (a -> Pixel)  -- Transforming function
           -> Result a
           -> Result a
```

- `(~>)`: Devuelve una dimensión (X o Y) de una imagen.

```
(~>) :: a -> Dim -> Int
```

- `(!)`: Obtiene el pixel en determinada posición de una imagen. En caso de que la posición no sea válida, lanza una excepción.

```
(!) :: a -> Point2D -> Pixel
```

- `fold`: Consume una imagen devolviendo un elemento de tipo `b` mediante una función y un elemento neutro.

```
fold :: (Pixel -> b -> b) -> b -> a -> b
```

Pixel

- `pixelMap`: mapea una función sobre un píxel.

```
pixelMap :: (Int -> Int) -> Pixel -> Pixel
```

- `(%>)`: multiplica un valor entero por una constante real.

```
(%>) :: Int -> Float -> Int
```

Bitmap

- `load`: carga una imagen desde un archivo, en caso de error devuelve un mensaje descriptivo.

```
load :: String -> IO (Result Bitmap)
```

- `save`: guarda una imagen a disco, en caso de que la imagen sea inválida, informa el error que ésta acarrea.

```
save :: String -> Result Bitmap -> IO ()
```

Operations.Arithmetic

- `(+),(-),(*)`: operaciones aritméticas sobre imágenes.

```
(+), (-), (*) :: Result a -> Result a -> Result a
```

- `(*=)`: Multiplica una imagen por una constante.

```
(*=) :: Image a => Result a -> Float -> Result a
```

- `blend`: Mezcla dos imágenes con una proporción lineal entre 0.0 $\sim$ 1-0. Falla en caso de una proporción inválida.

```
blend :: Image a => Float -> Result a -> Result a -> Result a
```

Operations.Point

- `modify`: modifica cada valor de cada píxel mediante una función.

```
modify :: Image a => (Int -> Int) -> Result a -> Result a
```

- `modifyCh`: modifica cada valor de cada pixel de un determinado canal (R,G o B) mediante una función.

```
modifyCh :: Image a => Channel -> (Int -> Int) -> Result a -> Result a
```

- `(#>)`: devuelve un canal.

```
(#>) :: Image a => Result a -> Channel -> Result a
```

- `negative`: negativo de una imagen.

```
negative :: Image a => Result a -> Result a
```

- `mono`: conversión a monocromo, mediante algún método de conversion (lum: Luminancia, lig: Luminosidad, avg: Promedio).

```
mono :: Image a => ConvMethod -> Result a -> Result a
```

- `threshold, binary`: Threshold color y binario, mediante algún método de conversión.

```
threshold, binary :: Image a => ConvMethod -> Float -> Result a -> Result a
```

Operations.Filters

- `erosion, dilation`: erosión y dilatación de imágenes, con determinado radio. Internamente usa conversión por luminancia para comparar píxeles, por lo que en imágenes a color tiene un efecto de erosionar o dilatar los píxeles más brillantes.

```
erosion, dilation :: Image a => Int -> Result a -> Result a
```

- `mean`: filtro de promedio, con determinado radio.

```
mean :: Image a => Int -> Result a -> Result a
```

- `median`: filtro de mediana, con determinado radio. Mejor reducción de ruido *salt and pepper* que `mean`, pero bastante más lento.

```
median :: Image a => Int -> Result a -> Result a
```

- `convolution`: aplica un convolution kernel de determinado radio, representado mediante una lista de valores. Retorna error si el radio no se corresponde con el tamaño de la lista.

```
convolution :: Image a
            => Int      -- Radius
            -> [Float]  -- Kernel list
            -> Result a
            -> Result a
```

- `sobel, laplace`: detección de bordes, de radio 1.

```
sobel, laplace :: Image a => Result a -> Result a
```

- `smooth`: desenfoque simple, de radio 1.

```
smooth :: Image a => Result a -> Result a
```

- `gaussian`: desenfoque gaussiano, de radio 1.

```
smooth :: Image a => Result a -> Result a
```

- `motionBlur`: desenfoque de movimiento, de radio 1, usando una orientación.

```
data Orientation = H  -- Horizontal
                 | V  -- Vertical
                 | A  -- Ascending ( / )
                 | D  -- Descendig ( \ )

motionBlur :: Image a => Orientation -> Result a -> Result a
```

- `sharpen`: efecto de enfoque, de radio 1.

```
sharpen :: Image a => Result a -> Result a
```

- `emboss`: efecto de grabado, de radio 1.

```
emboss :: Image a => Result a -> Result a
```

Operations.Geometric

- `solid`: crea una imagen de color sólido de determinado tamaño.

```
solid :: Image a => Pixel -> Point2D -> Result a
```

- `crop`: recorte entre dos puntos. Si los puntos están fuera de la imágen, el recorte llega hasta el borde.

```
crop :: Image a => Point2D -> Point2D -> Result a -> Result a
```

- `overlap`: superpone una imagen sobre otra en determinada posición.

```
overlap :: Image a
        => Point2D   -- Overlap position
        -> Result a  -- Front image
        -> Result a  -- Back image
        -> Result a
```

- `scale`: escalado proporcional usando método del vecino más cercano.

```
scale :: Image a => Float -> Result a -> Result a
```

- `scale2D`: escalado en dos dimensiones, usando método del vecino más cercano.

```
scale2D :: Image a
        => (Float, Float)  -- (X factor, Y factor)
        -> Result a
        -> Result a
```

- `scaleTo`: escalado a un tamaño determinado, usando método del vecino más cercano.

```
scaleTo :: Image a => Point2D -> Result a -> Result a
```

- `rot90, rot180, rot270`: rotaciones.

```
rot90, rot180, rot270 :: Image a => Result a -> Result a
```

- `reflect`: reflecciones sobre algún eje (X o Y).

```
reflect :: Image a => Dim -> Result a -> Result a
```

- `(<|>)`: unión de dos imágenes, una a lado de la otra. Retorna error si el alto de las imágenes no coinciden.

```
(<|>) :: Image a => Result a -> Result a -> Result a
```

- `(<->)`: unión de dos imágenes, una encima de la otra. Retorna error si el ancho de las imágenes no coinciden.

```
(<|>) :: Image a => Result a -> Result a -> Result a
```

Operations.Histogram

- `plotHist`: muestra un histograma en una ventana de Gnuplot.

```
plotHist :: Image a => Result a -> IO ()
```

- `saveHist`: guarda un histograma a disco, usando un perfil de salida. Retorna error si el perfil de salida no se corresponde a un tipo de imágen.

```
TerminalType:= (Constructor Path)
Constructors:= PS,EPS,PNG,PDF,SVG,GIF,JPEG,Latex

saveHist :: Image a => TerminalType -> Result a -> IO ()
```

Operations.Palette

- `dominants`: retorna los n colores más dominantes de una imagen.

```
dominants :: Image a => Int -> a -> [Pixel]
```

- `palette`: crea una paleta determinado tamaño, con los n colores más dominantes de una imagen.

```
palette :: Image a => Point2D -> Int -> Result a -> Result a
```