Haskell and OpenCV: theory and practice

Francesco Mazzoli <f@mazzo.li>

October 2016

A year ago, I talked about a problem...

- To get things done in many fields you need access to well-established libraries.
- Accessing these libraries from Haskell is cumbersome, if at all possible.
- Thus, prototyping and iterating on Haskell code that uses foreign code is annoying.

OpenCV

- Kitchen-sink library for computer vision.
- If you need some algorithm in that space, OpenCV probably has it.
- From standard image filters, to features detection, to face recognition, to more practical utilities such as decoding images from files or processing a webcam feed.

OpenCV's main type: `Mat`

- OpenCV's main type is `cv::Mat`. It is used to represent both images and matrices used to express transformations.
- The Haskell bindings encode a great deal of information about a `cv::Mat` at the type level, which is very helpful for both safety and documentation.

OpenCV's main type: `Mat`

```
data Mat shape channels depth
```

- `shape`: the shape of the matrix, for example `[3, 3]` for a 3 by 3 matrix.
- `channels`: how many channels the matrix has, for example 3 for an RGB image.
- `depth`: the type of the scalars in the matrix, for example `Double` or `Word8`

OpenCV's main type: `Mat`

```
data Mat
  (shape :: [Nat])
  (channels :: Nat)
  (depth :: *)
```

What do we do if we don't know some of the parameters at compile time?

OpenCV's main type: Mat

```
-- | 'D'ynamically or 'S'tatically known values
data DS a
   = D   -- ^ Something is dynamically known
   | S a -- ^ Something is statically known

data Mat
  (shape :: DS [DS Nat])
  (channels :: DS Nat)
  (depth :: DS *)
```

OpenCV's main type: Mat

```
-- RGB image of some dimension
Mat (S [D, D]) (S 3) (S Word8)

-- RGBA image of known dimension
Mat (S [S 480, S 680] (S 4) (S Word8)

-- Affine transformation matrix
Mat (S [S 2, S 3]) (S 1) (S Double)

-- Array of floats
Mat (S [D]) (S 1) (S Float)
```

Example: blurring images

```
gaussianBlur ::
     (depth `In` '[Word8, Word16, Float, Double])
  => V2 Int32 -- ^ Kernel size
  -> Mat (S [h, w]) channels (S depth)
  -- ^ Input matrix
  -> CvExcept (Mat (S [h, w]) channels (S depth))
```

`gaussianBlur` takes a 2-dimensional image with an arbitrary number of channels – the blurring is applied per-channel.

Example: blurring images

```
gaussianBlur ::
     (depth `In` '[Word8, Word16, Float, Double])
  => V2 Int32 -- ^ Kernel size
  -> Mat (S [h, w]) channels (S depth)
  -- ^ Input matrix
  -> CvExcept (Mat (S [h, w]) channels (S depth))
```

The shape of the image is preserved in the output.

Example: blurring images

```
gaussianBlur ::
     (([depth `In` '[Word8, Word16, Float, Double])
  => V2 Int32 -- ^ Kernel size
  -> Mat (S [h, w]) channels (S depth)
  -- ^ Input matrix
  -> CvExcept (Mat (S [h, w]) channels (S depth))
```

The depth of the image is restricted to what OpenCV can work with for this operation.

Example: blurring images

```
gaussianBlur ::
     ([depth `In` '[Word8, Word16, Float, Double])
  => V2 Int32 -- ^ Kernel size
  -> Mat (S [h, w]) channels (S depth)
  -- ^ Input matrix
  -> CvExcept (Mat (S [h, w]) channels (S depth))
```

Finally, the function is pure (no `IO`/`ST`), but runs in an error monad – `CvExcept`.

Example: edge detection

```
canny ::
     Mat (S [h, w]) channels (S Word8)
  -> CvExcept (Mat (S [h, w]) (S 1) (S Word8))
```

- In this case shape and depth are preserved...

Example: edge detection

```
canny ::
     Mat (S [h, w]) channels (S Word8)
  -> CvExcept (Mat (S [h, w]) (S 1) (S Word8))
```

- In this case shape and depth are preserved...
- ...but the channels aren't: the output is only needs one channel because it represents a mask over the original image, with 0 where there is no edge and 255 where there is no edge.

Mutable matrices

- Matrices can also be mutable, to allow in-place operation.
- Every `Mat shape channels depth` type can be turned into its mutable version with the `Mut` type constructor.
- Mutable matrices work in `IO` and `ST`, much like `Vector`s and `Array`s.

Mutable matrices

```
thaw ::
      (PrimMonad m)
  => Mat shape channels depth
  -> m (Mut
          (Mat shape channels depth)
          (PrimState m))

freeze ::
      (PrimMonad m)
  => Mut (Mat shape channels depth) (PrimState m)
  -> m (Mat shape channels depth)
```

Mutable matrices: drawing circles

```
circle ::
      (PrimMonad m, ToScalar color)
  => Mut
       (Mat (S '[h, w]) channels depth)
       (PrimState m)
  -- ^ Matrix to draw on
  -> V2 Int32 -- ^ Center of the circle
  -> Int32 -- ^ Radius of the circle
  -> color
  -> Int32 -- ^ Thickness of the outline
  -> CvExceptT m ()
```

Live demo!

let's hope it works...

What's in a binding?

```
canny ::
     Mat (S [h, w]) channels (S Word8)
  -> CvExcept (Mat (S [h, w]) (S 1) (S Word8))
canny src = unsafeWrapException $ do
    dst <- newEmptyMat
    handleCvException (pure $ unsafeCoerceMat dst) $
      withPtr src $ \srcPtr ->
      withPtr dst $ \dstPtr ->
        [cvExcept|
          cv::Canny(
            *$(Mat * srcPtr), *$(Mat * dstPtr),
            // TODO let user set the parameters
            30, 200, 3, false);
        |]
```

Questions?