HIP and OHIIO: Functional Image Processing in Haskell

Phaedon Sinis
CS240h, Fall 2011

Abstract.

Image processing in Haskell is fragmented. Decoders are limited to a few formats, or lack support for features such as tiles and mipmapping that are essential to professional workflows. Image manipulation and filtering in Haskell is available via wrappers to libraries written in other languages. HIP (Haskell Image Processing), inspired by Conal Elliot's *Pan* system for functional image synthesis [1], reimplements and extends *Pan* to create the foundations for a future standalone DSL. For image loading and saving, OHIIO (Open Haskell Image IO) provides Haskell bindings to OIIO, an open source image decoding library with an emphasis on high performance.

Introduction.

This project emerged from background reading for a new research project in Pat Hanrahan's lab to create an image processing DSL. A pure functional approach is useful when formalizing the semantics of a language, because the relationships between operators and data structures are immutable. Theorems can be developed more easily in this framework, which in turn can be used later to generate and optimize code in the later stages of the DSL compiler.

In 2001, Conal Elliot (Microsoft Research) created *Pan,* a functional library and optimizing compiler for image synthesis in Haskell [1]. The library introduced an algebraic approach to creation and manipulation of images, with some generalizations. All images are functions from geometric space to color space. For example, a color photograph can be viewed as a function from a 2D coordinate system to an RGBA color value. The library was small and elegant, implementing a minimal set of image processing operations while ignoring common tasks like file loading, convolution, histograms and filtering.

The goal of this project is to create a new implementation of a functional library for image processing, inspired by *Pan;* to extend this implementation with some of the most frequently needed tools in image processing; and to explore the transition from library to language by applying techniques from Oleg Kiselyov's work on typed tagless-final interpreters [7]. The result is a prototype library that allows photographs to be treated as Haskell functions, with the elegance of the tagless-final approach. In addition, the basic representation of an image processing expression as a tree of operations with regions of interest [5] allows for the deferment of certain operations, and the realization of others whose computation is order-dependent, such as fast algorithms for median filtering.

OHIIO.

OpenImageIO (OIIO) [2] is an image decoding library that supports high-performance, professional workflows, including image conversion, texture access, and efficient caching. The OIIO library was created by Larry Gritz of Sony Pictures Imageworks, which has released many of its back-end tools as open-source projects under the New BSD license [3].

The first part of this project was to enable simple image loading and saving by creating Haskell bindings to this library. The library, baptized "OHIIO", is available at https://github.com/phaedon/ohiio.

C wrappers

OIIO is written in C++, but interfaces between Haskell and C++ are fragile and experimental [4]. However, the Haskell Foreign Function Interface (FFI) to C is reliable, simple, and well-established. The first step was to create C-compatible wrappers for the C++ classes in OIIO, and to export these symbols using the "extern" keyword. These C wrappers are wholly contained in the files ohiio.h/cpp and can be compiled according to the instructions in the README.

Haskell bindings

The second step was to create a set of Haskell bindings to these C functions (`src/bindings.hs`). These type signatures match the C function prototypes exactly and provide the necessary information to run `hsc2hs`, which creates the *OhiioBindings* module. This module can then be imported by modules with friendlier and (somewhat) more Haskell-like APIs.

High-level interface

Three such modules are currently available in OHIIO: *Ohiio.ImageInput, Ohiio.ImageSpec*, and *Ohiio.ImageOutput*. Although the functions found in these modules do expose raw pointers via the *Foreign.Ptr* module, it is expected that higher-level APIs (such as those found in *Hip.ImageLoader,* discussed below) will completely hide these pointers behind a clean monadic interface.

HIP.

Domain

Given the definition of an image as a function, the domain of an image is defined in the module *Hip.PointSpace*, where *Point* is a typeclass and *Point2d* is an instance of the typeclass. Other instances may be defined in the future. For example, a *Point3d* could represent volumetric textures, or a *Point2dInt* could restrict coordinate accesses to integer grid points. All points must implement certain basic operations, including addition, scalar multiplication, and linear interpolation.

Range

The module *Hip.ColorSpace* represents the range of an image function. Addition is the only required operation for colors (which may be numbers, Boolean values, or possibly something else.)

A typeclass *CompositeColor* allows us to define colors that implement the compositing operations defined in the classic paper by Porter and Duff [6]. *ColorRGBA* is the specific instance that forms the current foundation of the HIP library. Each channel is represented as a Double whose range is in [0, 1]. Overflow is handled by saturating arithmetic, and conversion functions enable the representation of colors as *ColorRGBA8*, in which each channel is an 8-bit unsigned integer (i.e. the common practice of placing each channel in the range [0, 255]).

DSL syntax

In the module *Hip.Image*, a core group of six operators is defined in a typeclass, *ImageSYM*. These operators should be visualized as nodes in an expression tree: *leaf* is a leaf node, *unary* and *binary* are color transformations, *spatial* is a geometric transformation, *crop* selects a region of interest, and *reduce* combines all of the color values in a local neighborhood into a single value.

The typeclass simply defines the syntax and types of these functions but is agnostic as to what they actually perform. Following Kiselyov's examples [7], multiple interpreters can be written with great ease. For example,

```
instance ImageSYM (Point2d -> ColorRGBA) where ...
```

interprets expressions such that leaf nodes contain image functions. Evaluating one of these expressions at a coordinate returns an RGBA color. An alternate interpreter may be written as:

```
instance ImageSYM [Char] where ...
```

which composes a string representation of the expression tree.

The first of these two interpreters is most useful for actual image processing, and defines the core of this image processing DSL.

File I/O

Representing algorithmically defined images as functions is a trivial task. *Hip.Generator* contains one such function that generates checkerboards. Such images are defined over an infinite, real-valued domain, so no bounds-checking is required.

Loading a photograph from a file and representing it as a function is less obvious but nearly as simple. In the *Hip.ImageLoader* module, the *mkImageFnFromFile* function takes a filename and calls the OHIIO library to load the image into a memory buffer. That memory buffer is then wrapped in a partially-evaluated closure and returned to the user as a *(Point2d -> ColorRGBA)* function, and thus behaves identically to synthetic images. In this case, the function converts the coordinate into an array offset and looks up the color value in the memory buffer. The function does the required bounds checking and returns the empty color if the coordinate is out of bounds. In addition, the image expression that is returned contains a *crop* node that can be later referenced to avoid unnecessary work when filtering or saving the image.

Stencils

A stencil operation is an repeated operation over a neighborhood of a point. In the module *Hip.Stencil,* the function *convolve* computes a 2D convolution at a point by creating an image expression that consists of the operations *reduce, binary,* and *spatial.* While this is the simple implementation, not the optimized version, it shows how a minimal set of language primitives can be used to implement a relatively common image processing filter.

Histograms

Histograms are interesting to photographers as a visualization of the brightness distribution of a region in an image. But they are also important as inputs to algorithms. For example, the naive approach to median filtering is to compute the median from scratch for each pixel's neighborhood; this is extremely slow, but fortunately most of the computations are redundant. To accelerate median filtering, one can use local histograms to compute the median of a neighborhood. A further optimization recognizes that histograms of adjacent pixels are nearly identical, so it is desirable to keep a cache of column histograms that can slide up and down while minimizing redundant computation [8].

The *Hip.Histogram* module contains an implementation of this data structure. Because histograms over small regions will typically contain only a few colors, an array representation of the histogram would be sparse, and therefore wasteful and slow. The standard library's *Data.Map* structure is a simple alternative.

Median filtering

The fast median filtering algorithm [8] requires the image to be traversed in a specific order in order to ensure maximal overlap of region histograms. Therefore, the approach in *Hip.Filter.Median* is to evaluate the filtered image into a new buffer and to present it back to the user in a new function.

Crop regions

A compelling example of the power of the tagless final form is in *Hip.Image*. A "crop context" data type is defined which either carries a bounding box, or indicates that there is no crop region. An interpreter for our DSL can then be written which pushes crops through the expression down to the leaves (see the *push_crop* function). This action actually reorders the nodes of the expression tree, yet the relaxation of the DMR allows us to take the result and treat it as an ImageRGBA, a String, or anything else that has been defined as an instance of the ImageSYM typeclass.

Another interpreter can then be written which converts the expression tree into a bounding box (see *boxify)*. This works by simply retrieving the bounding box from the leaves of the tree, and is particularly interesting for two reasons. First, it shows how we can pattern-match in the tagless final form, without using the data constructor. Second, it allows us to extract the bounding box to save on computation time, for example by avoiding slow operations like filtering on a mostly empty region of interest.

Performance (a.k.a. Future Work)

Because the emphasis on this project was on prototyping and on the elegance of the tagless final approach, little time was spent on optimizing performance (except for the histogram-based median filtering algorithm). Much work could be done on minimizing copying of data; recognizing symmetric convolution kernels and implementing them as sequential 1D convolutions; speeding up merging of column histograms; and using OIIO's facilities for scanlines and mipmaps to load only the number of pixels that are relevant for the output image.

The choice to convert all images from Word8 to Double, and then back to Word8, was motivated by the need to minimize rounding error, particularly on spatial transformations such as rotations and color adjustment operations such as darkening. Quantization can have particularly deleterious effects on 8-bit values, so Doubles are a safe approach. Unfortunately, this is extremely wasteful if the user only wants to generate a simple image and save it to a file. But detecting simple expression trees and choosing the datatype based on the operations used required greater generality in the code, and was not feasible for this deadline.

Because of the lack of emphasis on performance, a few indicative benchmarks are provided and do not claim to be exhaustive:

- Generation & saving of 640x480 image: 0.4 seconds
- 2D Convolution of a 640x480 checkerboard image with a 7x7 Gaussian kernel: 12 seconds
- Median filter of a 640x360 photograph with a kernel radius of 1 pixel: 3.5 seconds
- Median filter as above, kernel radius of 3 pixels: 16 seconds

Conclusion.

It is not clear whether future work on the DSL research project will continue in Haskell. While Haskell is ideal for this algebraic approach to library design, thanks to strong typing, lazy evaluation, partial application, and a clean aesthetic, recent discussions in the research group suggest that we might take a hybrid approach that allows for more direct intervention in the memory access patterns while restricting images to regular, finite 2D grids. This decision may be driven by fellow researchers working on stencil computation DSLs in a variety of contexts, including physical simulation [9].

Regardless of the future direction of the imaging research project, HIP has proven to be a useful prototype by showing that it is possible to implement some non-naive, optimized algorithms in the algebraic framework, while cleanly supporting a wide range of filtering operations including convolution.

To appeal to a wider audience, future work on OHIIO should focus on a friendlier, pointer-free API. If accomplished, it may have a promising future as a standard image decoder for the Haskell community.

Bibliography.

[1] Elliott, C. Functional Image Synthesis. *Proceedings of Bridges 2001*.

[2] https://sites.google.com/site/openimageio/

[3] http://opensource.imageworks.com/?p=all

[4] http://www.haskell.org/haskellwiki/Cxx_foreign_function_interface

[5] Shantzis, Michael A. (1994). A model for efficient and flexible image computing. *Proceedings of the 21st annual conference on Computer graphics and interactive techniques - SIGGRAPH '94*, 147-154. New York, New York, USA: ACM Press. doi: 10.1145/192161.192191

[6] Compositing Digital Images. Porter, T., & Duff, T. (1984). *Computer Graphics*. Computer, 18(3), 253-259.

[7] http://okmij.org/ftp/tagless-final/course/index.html

[8] Perreault, S., & Hébert, P. (2007). Median filtering in constant time. *IEEE transactions on image processing: a publication of the IEEE Signal Processing Society*, *16*(9), 2389-94.

[9] *http://liszt.stanford.edu/*

Images.

Fig 1. "Lambros", original photo. 738x370 PNG

Fig 2. "Lambros", median filter, krad=3. Cropped to 640x360