Efficient Image Manipulation via Run-time Compilation

Conal Elliott, Oege de Moor*, and Sigbjorn Finne

November, 1999

Technical Report

MSR-TR-99-82

Microsoft Research

Microsoft Corporation

One Microsoft Way

Redmond, WA 98052

Efficient Image Manipulation via Run-time Compilation

Conal Elliott, Oege de Moor, and Sigbjorn Finne

Abstract

An image manipulation system can be thought of as a domain-specific programming language: by composing and manipulating pictures, the user builds up an expression in such a programming language. Each time the picture is displayed, the expression is evaluated. To gain the required efficiency of display, the expression must be optimized and compiled on the fly. This paper introduces a small language that could form the basis of an image manipulation system, and it describes a preliminary implementation. To compile an expression in the language, we inline all function definitions and use algebraic laws of the primitive operations to optimize composites. We apply a form of code motion to recover (as far as possible) the sharing lost in the inlining phase. Finally, we generate intermediate code that is passed to a JIT compiler.

1 Introduction

A modern image manipulation application like Adobe Photoshop™ or Microsoft PhotoDraw™ provides its users with a collection of image operations, and implements these operations by means of a fixed set of precompiled code. This code must be written to handle a variety of image formats and arbitrary combinations of image operations. It is also often augmented by special-purpose code to handle some common cases more efficiently than done by the more general algorithms. Special-casing can only go so far, however, because the user can perform any of an infinite set of combinations, and because the more code the product includes, the less reliable it is and the more demanding of its users' resources. Consequently, the user gets sub-optimal time and space performance.

For example, a user might import an 8-bit gray-scale image and a 32-bit image with per-pixel partial transparency, and then overlay one onto the other. Most likely, the system does not have code specifically for this combination, so it will first convert the gray-scale image to 32-bit format, and then combine them using code written for 32-bit images. This solution suffers in both time and space. First, the space requirement for the first image quadruples, and time must be spent converting all of the pixels. Second, the overlay computation must access and process all of this extra data. The increased space use can seriously compound the speed problem by multiplying reads from secondary cache, system RAM, or even disk (perhaps for paging in virtual memory or memory-mapped files).

The space overhead of image conversion can be avoided by adopting an object-oriented (or higher-order functional) interface to pixel reading, so that conversion from gray-scale to true color can be done during each pixel read by the overlay computation. Extending this idea leads to the notion of a "pixel interpreter". Direct implementation of this solution is elegant but grossly impractical, because the method-dispatching overhead is too great to be paid per pixel. More sophisticated variations would allow batching up more pixels at once and could improve speed while complicating the programming of such operations, especially considering affine transformations, warps, etc.

We propose an alternative implementation technique that combines the strengths of the two approaches described above. The idea is to use the "pixel interpreter" approach as a programming model, but to specialize the pixel interpreter with respect to each combination tree it is about to execute. This process of specialization amounts to a compiler for combination trees. Since the combinations come into existence at run-time (from the authoring application's point of view), we must do run-time compilation. The program being edited and compiled is the composition of image importations and manipulations indicated by the end-user, and denotes a function over 2D space. To carry out this idea, we propose to present a conventional authoring UI that is implemented as a simple programming environment. Operator trees are represented as syntax trees in a suitable language, and image editing steps are reflected as syntactic edits. Novice and casual users need never be exposed to the linguistic nature of the implementation. Advanced users can benefit significantly, through the powerful scripting that will be available with minimal extra effort. In typical use, the system should have a very fluid feel, not the edit-compile-run cycle of C programming or even the edit-run cycle of interpreters.

Because we adopt a rather abstract model of the basic operations on images, it is very easy to extend these ideas to time-varying pictures. For such compositions, efficiency of the display routine is even more important than for static images, where the main reason for efficiency is instantaneous feedback as the user is editing a composition.

We have not yet implemented the envisioned syntax-free authoring environment, but have constructed a usable prototype based on a Haskell-like [20] syntax. Although we borrow Haskell's notation, we do not adopt its semantics: the language of pictures is strict. Also, all function definitions are non-recursive. An unusual feature of our implementation is a radical approach to inlining: *all* derived operations are unfolded to a small set of primitives, regardless of code bloat or duplication of work. The resulting expression is rewritten using simple optimization rules. Finally, we recover (as much as possible) the sharing lost in the inlining by a code motion algorithm that abstracts common subexpressions, and that hoists invariant expressions out of loops. Traditional optimizing compilers are very conservative about inlining. We have found that, in this particular application, such conservatism inhibits the application of many important optimizations.

The contributions of this work are the following:

- The introduction of a language that can serve as the basis for an image manipulation system.
- An on-the-fly compilation scheme that specializes a picture interpreter to a given image composition.
- The implementation of such specialization through greedy inlining, rewriting, and code motion.

In the remainder of this paper, we present: a simple semantic model for images and an extensible set of image operators specified in terms of this model (Section 2), including examples of images and animations constructed with these operators; optimization of compositions of ("expressions" containing) the specified operators into space- and time-efficient implementations (Section 3); and then discusses related and future work (Sections 4 and 5).

2 Semantic model

For images, the model is simply functions from continuous 2D space to colors with partial opacity. Although the domain space is infinite, many images are transparent everywhere outside of a bounding region[1]. In the notation of Haskell, one might capture the definition of images as follows:

```
type Image  = Point2 -> Color
type Point2 = (Double, Double)
```

Here `Double` is the type of double-precision floating point numbers – admittedly this is only an approximation of 2D space, and we shall have more to say about that below. It is useful to generalize the semantic model of images so that the range of an image is not necessarily `Color`, but an arbitrary type. For instance, Boolean-valued "images" can be used to represent spatial regions for complex image masking, and real-valued images for non-uniform color adjustments. For this reason, `Image` is really a type constructor:

```
type Image c = Point2 -> c
```

It can also be useful to generalise the domain of images, from points in 2D space to other types (such as 3D space or points with integer coordinates), but we shall not exploit that generality in this paper.

2.1 Colors

Our model of colors will be a quadruple of real numbers, with the first three for red, green, and blue (RGB) components, and the last for opacity (traditionally called "alpha"):

```
type Color    = (Fraction, Fraction, Fraction, Fraction)
type Fraction = Double
```

There are also the constraints that (a) all four components are between zero and one inclusive, and (b) each of the first three is less than or equal to the alpha. That is, we are using "pre-multiplied alpha" [22]. These constraints are not expressible in the type system of our language, but to remind ourselves that the arguments of `Color` must lie in the interval [0..1], we introduce the type synonym `Fraction`. Given the constraints on colors, there is exactly one fully transparent color:

```
invisible = (0, 0, 0, 0)
```

[1] These lines define the type `Image` to be a function from 2D points to colors, `Point2` to be a pair of real numbers (rectangular coordinates), and `Color` to be a data structure consisting of real values between 0 and 1, for red, green, blue, and opacity.

We are now in a position to define some familiar (completely opaque) colors:

```
red       = (1, 0, 0, 1)
green     = (0, 1, 0, 1)
...
```

It is often useful to interpolate between colors, to create a smooth transition through space or time. This is the purpose of (`cLerp w c1 c2`). The first parameter `w` is a fraction, indicating the relative weight of the color `c1`. The weight assigned to the second color `c2` is `1-w`:

```
cLerp :: Fraction -> Color -> Color -> Color
cLerp w (r1, g1, b1, a1) (r2, g2, b2, a2) = (h r1 r2, h g1 g2, h b1 b2, h a1 a2)
  where h x1 x2 = w * x1 + (1 - w) * x2
```

A similar operation is color *overlay*, which will be used later to define image overlay. The result is a blend of the two colors, depending on the opacity of the top (first) color. A full discussion of this definition can be found in [22]:

```
cOver (r1, g1, b1, a1) (r2, g2, b2, a2) =
  (h r1 r2, h g1 g2, h b1 b2, h a1 a2)
  where h x1 x2 = x1 + (1 - a1) * x2
```

2.2 Spatial transforms

In traditional computer graphics, spatial transforms are represented by matrices, and hence are restricted to special classes like affine or projective. Application of transformations is implemented as a matrix/vector multiplication, and composition as matrix/matrix multiplication. In fact, this representation is so common that transforms are often thought of as *being* matrices. A simpler and more general point of view, however, is that a transform is simply a space-to-space function.

```
type Transform2 = Point2 -> Point2
```

It is then easy to define the common affine transforms. For instance, we might define:[2]

```
translate :: (Double, Double) -> Transform2
translate (dx, dy) = \ (x,y) -> (x + dx, y + dy)
```

That is, given a pair of displacements `(dx,dy)`, the function `translate` returns a spatial transform. A transform is itself a function, which takes a point `(x,y)` and adds the displacements to the respective coordinates. The function `scale` is very similar to `translate`. It takes a pair of scaling factors (one for the `x` coordinate, and another for the `y` coordinate), and multiplies by corresponding coordinates. For convenience, `uscale` performs uniform scaling:

```
scale :: (Double,Double) -> Transform2
scale (sx, sy) = \ (x,y) -> (sx * x, sy * y)

uscale :: Double -> Transform2
uscale s = scale (s,s)
```

Another useful transform is rotation:

```
rotate :: Double -> Transform2
rotate ang = \ (x,y) -> (x * c - y * s, y * c + x * s)
             where c = cos ang
                   s = sin ang
```

In addition to these familiar transforms, one can define any other kind of space-to-space function, limited only by one's imagination. For instance, here is a "swirling" transform. It takes each point `p` and rotates it about the origin by an amount that depends on the distance from `p` to the origin. For predictability, this transform takes a parameter `r`, which gives the distance at which a point is rotated through a complete circle (2π radians).

[2] The notation "`\ p -> E`" stands for function that takes an argument matching the pattern `p` and returns the value of `E`. A pattern may be simply a variable, or a tuple of patterns.

```
swirling r = \ p -> rotate (distO p * (2.0 * pi / r)) p
             where distO (x,y) = sqrt (x * x + y * y)
```

Below we shall see an example where swirl is used to generate an interesting image (Figure 1).

2.3 Images

The simplest image is transparent everywhere. Its specification:

```
empty :: Image Color
empty = \ p -> invisible
```

The first line of the specification above says that `empty` is a color-valued image. The second line says that at each point `p`, the color of the empty image is invisible. More generally, we can define an image of constant value, by "lifting" the value into a constant function, using Haskell's `const` function:

```
const :: c -> (p -> c)
const a = \ x -> a
```

Note that the type of `const` is quite polymorphic. We have used the type variable names "`c`" and "`p`" to suggest `Color` and `Point2`, for intuition, but they may be any types at all. Given this definition, we can redefine the empty image:

```
empty = const invisible
```

As an example of a non-trivial synthetic image, consider a checkerboard. As a building block, first define a Boolean image `checker` that alternates between true and false on a one-pixel checkerboard. The trick is to convert the pixel coordinates from floating point to integer (using `floor`) and test whether the sum is even or odd:

```
checker :: Image Bool
checker = \ (x,y) -> (floor x + floor y) `mod` 2 == 0
```

Now we can define our checkerboard image function. It takes a square size `s` and two colors `c1` and `c2`. It chooses between the given colors, depending on whether the input point, *scaled down* by `s` falls into a true or false square of `checker`.

```
checkerBoard :: Double -> c -> c -> Image c
checkerBoard s c1 c2 = \ (x,y) -> if checker (x/s, y/s) then c1 else c2
```

2.3.1 Applying spatial transforms

In the checkerboard example, recall that we scaled *up* a given image (`checker`) by scaling *down* the input points. In general, to transform an image, it suffices to inversely transform sample points before feeding them to the image being transformed. Using an infix dot (`.`) to denote function composition, we may write:

```
applyTrans :: Transform2 -> Image -> Image
applyTrans xf im = im . inverse xf
```

While this definition is simple and general, it has the serious problem of requiring inversion of arbitrary spatial mappings. Not only is it sometimes difficult to construct inverses, but also some interesting mappings are many-to-one and hence not invertible. In fact, from an image-centric point-of-view, we *only* need the inverses and not the transforms themselves. For these reasons, we simply construct the images in inverted form, and do not use `applyTrans`. Because it may be mentally cumbersome to always think of transforms as functions and transform-application as composition, we provide a friendly vocabulary of image-transforming functions:

```
move     (dx,dy) im = im . translate (-dx, -dy)
stretch  (sx,sy) im = im . scale     (1/sx, 1/sy)
ustretch s       im = im . uscale    (1/s)
turn     ang     im = im . rotate    (-ang)
swirl    r       im = im . swirling  (-r)
```

As a visual example of transform application, Figure 1 shows a swirled checkerboard. This picture was generated by compiling the expression

```
swirl 100 (checkerboard 10 black white)
```

2.3.2 Pointwise lifting

Many image operations result from pointwise application of operations on one or more values. For example, the overlay of one image on top of another can be defined in terms of `cOver` (defined earlier) which applies that operation to a pair of colors:[3]

```
over :: Image Color -> Image Color -> Image Color
top `over` bot = \ p -> top p `cOver` bot p
```

Another example of pointwise lifting is the function `cond` which chooses on a pointwise basis from two images:

```
cond :: Image Bool -> Image c -> Image c -> Image c
cond  b c1 c2 = \ p -> if b p then c1 p else c2 p
```

With `cond` and spatial transformation, we obtain a pleasingly short rephrasing of our checkerboard image:

```
checkerBoard s c1 c2 =
  ustretch s (cond checker (const c1) (const c2))
```

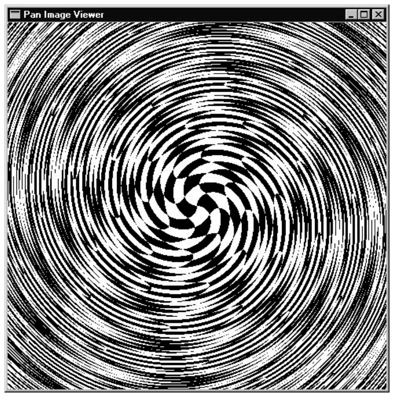

Figure 1. Swirled checkerboard

2.4 Bitmaps

Image importation must make up two differences between our “image” notion and the various “bitmap” images that can be imported. Our images have infinite domain and are continuous, while bitmaps are finite and discrete. For easy analysis, we represent bitmaps as an abstract array of colors. Our notion of array consists of dimensions and a subscripting function:

```
data Array2 c = Array2 Int Int ((Int, Int) -> c)
```

That is, `(Array2 n m f)` represents an array of `n` columns and `m` rows, and the valid indices of `f` are in the range `{0..n-1}×{0..m-1}`.

The heart of the conversion from bitmaps to images is captured in the `reconstruct` function. Sample points outside of the array’s rectangular region are mapped to the invisible color. Inner points generally do not map to one of the discrete set of pixel locations, so some kind of filtering is needed. For simplicity with reasonably good results, we will assume bilinear interpolation (`bilerp`), which performs a weighted average of the four nearest neighbors. Bilinear interpolation is conveniently defined via three applications of linear interpolation — two horizontal and one vertical:

```
bilerp :: Color -> Color -> Color -> Color -> Image Color
bilerp ll lr ul ur = \ (dx,dy) -> cLerp dy (cLerp dx ll lr) (cLerp dx ul ur)
```

Because of the type invariant on colors, this definition only makes sense if `dx` and `dy` fall in the interval [0,1].

---

[3] More generally, we can “lift” a binary function to a binary image function:

```
lift2 :: (a -> b -> c) -> (Image a -> Image b -> Image c)
lift2 op top bot = \ p -> top p `op` bot p
```

Now functions like “`over`” can be defined very simply:[3]

```
over = lift2 cOver
-- pointwise interpolation of two images:
iLerp = lift3 cLerp
```

In fact, the type of `lift2` is more general, in the style of `const` (which might be called “`lift0`”):

```
lift2 :: (a -> b -> c) -> ((p -> a) -> (p -> b) -> (p -> c))
```

Similarly, there are lifting functions for functions of any number of arguments.

We can extend `bilerp` from four pixels to an entire array of them as follows. Given any sample point `p`, find the four pixels nearest to `p` and `bilerp` the four colors, using the position of `p` relative to the four pixels. Recall that `f2i` and `i2f` convert from floating point numbers to integers (by rounding towards zero) and back, so the condition that `dx` and `dy` are fractions is satisfied, provided `x` and `y` are positive:

```
bilerpArray2 :: ((Int, Int) -> Color) -> Image Color
bilerpArray2 sub = \ (x,y) ->
 let
     i  = f2i x
     dx = x - i2f i
     j  = f2i y
     dy = y - i2f j
 in
     bilerp (sub (i, j  )) (sub (i+1, j  ))
            (sub (i, j+1)) (sub (i+1, j+1))
            (dx, dy)
```

Finally, we define reconstruction of a bitmap into an infinite extent image. The reconstructed bitmap will be given by `bilerpArray2` inside the array's spatial region, and empty (transparent) outside. For convenience, the region is centered at the origin:

```
reconstruct  :: Array2 Color -> Image Color
reconstruct (Array2 w h sub) =
  move (- i2f w / 2, - i2f h / 2)
       (cond (inBounds w h) (bilerpArray2 sub) empty)
```

The function inBounds takes the array bounds (width `w` and height `h`), and checks that a point falls within the given array bounds. Note that the use of `inBounds` in the above definition guarantees that the arguments of `bilerpArray2 sub` are non-negative, and so the type invariant of colors will not be violated. The definition of `inBounds` is as follows:

```
inBounds :: Int -> Int -> Image Bool
inBounds w h = \ (x,y) ->
               0 <= f2i x && f2i x <= w-1 && 0 <= f2i y && f2i y <= h-1
```

2.4.1 Color Conversions

So far, we have imagined colors to be represented using real numbers. This fiction simplifies our semantic model and composability in much the same way as the model of infinite resolution images. In fact, these two simplifications are exactly dual to each other, since space is the domain and color is the range in our semantic model.

To account for finite color resolution at reconstruction and sampling, we introduce a family of color conversion functions. For instance, the following converters apply to 32-bit integers made up of one byte for each of red, green, blue, and alpha:

```
type ColorRef = Int

fromRGBA32 :: ColorRef -> Color
toRGBA32   :: Color -> ColorRef
```

One aspect of these conversions is switch between a floating point number and a byte. These bytes use the range [0, 255] to represent the real interval [0.0, 1.0], so we can think of them as being implicitly divided by 255. This understanding is captured by the following definitions:

```
type ByteFrac = Int  -- One byte

fromBF :: ByteFrac -> Float
fromBF bf = i2f bf / 255.0
```

```
toBF :: Float -> ByteFrac
toBF x = f2i (x * 255.0)
```

The other aspect of `ColorRef/Color` conversions is the packing and unpacking of quadruples of byte fractions. For these operations, the following functions are useful.

```
pack :: Int -> Int -> Int
pack a b = (a <<< 8) .|. b

unpack :: Int -> (Int,Int)
unpack ab = (ab >>> 8, ab .&. 255)

packL :: [ByteFrac] -> ColorRef
packL = foldl pack 0
```

The color conversions are now simple to define.

```
fromRGBA32 rgba32 =
  (fromBF r8, fromBF g8, fromBF b8, fromBF a8)
 where
   (rgb24,a8) = unpack rgba32
   (rg16,b8)  = unpack rgb24
   (r8,g8)    = unpack rg16

toRGBA32 :: Color -> ColorRef
toRGBA32 c = packToBF [a,r,g,b]
  where (r, g, b, a) = c
```

2.4.2 Bitmap importation

We now address how to convert bitmap data into the form needed by the `reconstruct` function. Bitmaps are represented via the following data type, which defines them to consist of three components: the width, the height and a block of bits:

```
data BMP = BMP Int Int BMPBits
```

The nature of the "block of bits" will depend on the particular format. The following data type supports a few common cases. Note that at this point, the pixel data is completely unstructured, appearing as a pointer to the start of the data.

```
data BMPBits = BitsRGBA32  Addr
             | BitsRGB24   Addr
             | BitsGray8   Addr
             | BitsMapped8 Addr Addr
```

Conversion from a BMP to a color array consists mainly of forming a subscripting function, because we already have the width and height of the array:

```
fromBMP :: BMP -> Array2 Color
fromBMP (BMP w h bits) = Array2 w h (fromBits bits w)

fromBits :: BMPBits -> Int -> (Int,Int) -> Color
```

The particulars of `fromBits` vary with the file format. The 32-bit case is simple, because a whole pixel can be extracted in one memory reference. For a width of *w* and pixel position *(i,j)*, the sought pixel will be at an offset of *w * j + i* four-byte words from the start of the data. The extracted integer is then converted to a color, via `fromRGBA32`:

```
fromBits (BitsRGBA32 bptr) w (i,j) =
  fromRGBA32 (deref32 (addrToInt bptr + 4 * (w * j + i)))
```

The 24-bit case is trickier. Pixels must be extracted one byte at a time, and then assembled them into a color. As an added complication, each row of pixels is padded to a multiple of four bytes: (the "`.&.`" operator is bitwise conjunction)

```
fromBits (BitsRGB24 bptr) w (i,j) =
  Color (fetch 2) (fetch 1) (fetch 0) 1.0
  where wbytes  = (3 * w + 3) .&. (-4)
        addr    = addrToInt bptr + wbytes * j + 3 * i
        fetch n = fromBF (deref8 (addr + n))
```

Note the reversal of the calls to `fetch`: this is because the bitmap file is in BGR order, and not RGB. We first compute the width in bytes (`wbytes`), aligned to the next multiple of four. This quantity is then used to calculate what address the data should be fetched from (`addr`).

The `importBMP` function takes care of opening files and finding bitmap data:

```
importBMP :: String -> BMP
```

At first glance `importBMP`, would appear to be an impure primitive, breaking referential transparency of our little language. However, the bitmap file is read in at compile time, so it no less pure than a module importation.

Finally, we define a bitmap importation function that takes a file name, accesses the bitmap it contains with `importBMP`, maps it into a color array with `fromBMP`, and reconstructs a continuous image:

```
importImage :: String -> Image Color
importImage = reconstruct . fromBMP . importBMP
```

2.5 Animations

A rather pleasing consequence of our semantic model of image manipulation is that one can easily extend it to images that vary over time. Such animations can be modeled as functions from time to images: for each instant, the function returns the image that is to be displayed. We therefore define

```
type Time        = Double
type Animation c = Time -> Image c
```

A still image can be turned into an animation by ignoring the time argument. Here is an example of a true animation, namely a rotating checkerboard:

```
checkturn :: Animation Color
checkturn t = turn t (checkerboard 10 red blue)
```

In fact, following Fran [12][10] we can generalize further to support "behaviors", which are time-varying values of any type:

```
type Behavior a = Time -> a
```

To speed up a behavior, one can simply multiply its argument by a constant factor:

```
speedup :: Double -> Behavior a -> Behavior a
speedup x a t = a (t*x)
```

so for instance (`speedup 2 checkturn`) is a checkerboard that turns twice as fast as the original.

The generalization from images to animations not only illustrates the advantages of our semantic model, it also makes a strong case for our implementation approach, which is to specialize a pixel interpreter. The more traditional bitmap-caching representations (as discussed in the introduction) do not work so well for animation, because the cache gets less reuse.

2.6 Display

Given a color-valued image, one can display it in a window simply by sampling at a finite grid of pixel locations, converting each pixel color to an integer for the display device. (For a faithful presentation, images need to be antialiased, but that topic is beyond the scope of the present paper.) This approach was indeed our first implementation, but it leaves room for improvement. For one thing, it requires an indirect function call per pixel. More seriously, it prevents any optimization across several pixels or rows of pixels. To address these shortcomings, we made the two-dimensional iteration that samples storage of pixel values visible to the optimizer, by expressing that iteration in the language.

At the top level, an animation is converted into a “display function” that is to be invoked just once per frame. The argument of such a function is a tuple consisting of the time, XY pan, zoom factor, window size, and a pointer to an output pixel array (represented as an `Int`):

```
type DisplayFun = (Float, Float, Float, Float, Int, Int, Int) -> Action
```

The `Action` type represents an action that yields no value, much like Haskell’s type `IO ()`. It is the job of the viewer applet to come up with all these parameters and pass them into the display function code.

For reasons explained below, our main display function takes a `ColorRef`-valued image, rather than a color-valued one. Also, it handles not just static images, but time-varying images, i.e., 2D animations.

```
display :: Animation ColorRef -> DisplayFun
display imb (t,panX,panY,zoom,width,height,output) =
  loop height (\ j ->
    loop width (\ i ->
      setInt (output + 4 * j * width + 4 * i)
             (imb t ( zoom * i2f (i - width  `div` 2) + panX
                    , zoom * i2f (j - height `div` 2) + panY ))))
```

This definition uses two functions for constructing actions. The first takes an address (represented as an integer) and an integer value, and it performs the corresponding assignment. Its type is

```
setInt :: Int -> Int -> Action
```

The second function for constructing actions is like a for-loop. It takes an upperbound, and a loop body that is a function from the loop variable to an action. The loop body is executed for every value from zero up to (but not including) the upperbound:

```
loop :: Int -> (Int -> Action) -> Action
```

Note that the address calculation in the code for `display` is similar to the one used in the 32-bit case of `fromBits` above. Aside from calculating the destination memory address, the inner loop body samples the animation at the given time and position. The spatial sampling point is computed from the loop indices by taking account of the dynamic zoom and pan, and placing the image’s origin in the center of the window.

For flexibility and convenience, we make it easy to convert various image-like types into displayers.

```
class Displayable d where
  -- Convert to DisplayFun and say whether possibly time-varying
  toDisplayer :: d -> (DisplayFun, Bool)
```

The simplest case is a display function, which “converts” trivially:

```
instance Displayable DisplayFun where
  toDisplayer disp = (disp, True)
```

More interesting is a `ColorRef`-valued image animation:

```
instance Displayable (ImageB ColorRef) where
  toDisplayer imb = (display imb, True)
```

Given a color-valued image animation, we must convert the resulting colors to 24-bit integers, discarding the alpha (opacity) channel:

```
instance Displayable (ImageB Color) where
  toDisplayer imb = (display (\ t (x,y) -> toRGB24 (imb t (x,y))), True)
```

Various “static” (non-time-varying) image types are handled by converting to constant animations, converting, and then overriding the time-varying status.

```
instance Displayable (Image Color) where
  toDisplayer im = (disp, False)
   where
     (disp, _) = toDisplayer (const im :: ImageB Color)
```

```
instance Displayable (Image ColorRef) where
  toDisplayer im = (disp, False)
   where
     (disp, _) = toDisplayer (const im :: ImageB ColorRef)
```

Finally, it is sometimes convenient to define images over discrete temporal and spatial domains. The spatial indices are taken to be pixels, while the temporal index is milliseconds.

```
-- Discrete in space and color
type DImage  = (Int, Int) -> ColorRef
-- and in time (milliseconds)
type DImageB = Int -> DImage

instance Displayable DImageB where
  toDisplayer dimb =
    toDisplayer (\ t (x,y) -> dimb (f2i (t * 1000)) (f2i x, f2i y))

instance Displayable DImage where
  toDisplayer im = (disp, False)
   where
     (disp, _) = toDisplayer (const im :: DImageB)
```

We have decided not to include an explicit loop for time in the display function. That has the advantage of simplicity, but the disadvantage that we cannot expose successive time values (and independence of those time values) to the optimizer. Consequently, an image is re-displayed completely for each successive time sample. A more sophisticated implementation of `display` would make time-stepping explicit.

This completes a programmer's view of our image manipulation language. In the authoring application we envisage, only advanced users (who wish to do their own scripting) will take this view. In normal usage, expressions in our language will be generated through a graphical user interface, with the end-user being unaware of the linguistic nature of his actions.

Some readers may wish to pause at this point, and take a look at a web page that shows a number of example images and animations, to get a better feel for the expressiveness of the language we have introduced [11]. As we shall see below, scripting new composite operations does not require any knowledge of the implementation, or the optimizations that the implementation applies.

3 Compilation

Compilation is performed in several phases: inlining, simplification, common sub-expression elimination, VM code generation, and then JITing to machine code. Each of these phases, except the last, has been implemented in Haskell.

3.1 Inlining

The type, value, and function definitions given above are all taken literally, and inlined (and beta-reduced) at their uses. The result of this inlining is to reduce all image and animation expressions to low-level terms. The remaining types include only integers, floats, Booleans, and actions. The remaining expression is essentially first order, with the only function-typed subexpressions being the bodies of loops. Moreover, there are not even any data structures so there is no need for run-time (i.e., display-time) allocation or garbage collection.

It is clearly not the case that this inlining preserves the efficiency of the original program: work may be duplicated, so we shall need to try to eliminate repeated computations in a later phase. Also, the size of the program may increase rather dramatically. This is not such a serious problem, however, if terms are represented as directed acyclic graphs. That representation will also allow us to speed up later phases.

3.2 Simplification

Because inlining eliminates all high-level notions like colors, points, transforms, or images, the simplification phase need only handle numbers and Booleans. Our first prototype implementation used the program transformation system Mag [1] to process a collection of rewrite rules over these simple types. We are now in the process of producing a more specialized implementation. This new implementation takes advantage of sharing in the compilation process, representing terms as directed acyclic graphs.

Rules encode simple identities on the basic types, such as numbers

```
0.0 + x = x
0.0 * x = 0.0
1.0 * x = x
```

Booleans:

```
not (not a)             = a
if True  then a else b = a
if False then a else b = b
```

or integers as bit-strings, using Boolean and shift operators:

```
b .|. 0          = b
b .&. 0          = 0
b << 0           = b
0 >> b           = 0
(b << s1) << s2 = b << (s1+s2)
```

The rules are applied bottom-up, until no more apply. While this strategy is admittedly rather limited, it has the advantage of giving predictable results. Furthermore, the bottom-up strategy is very easy to implement: the rewriting can be done while building up expressions. Whenever a new tree constructor is applied, we apply all rules that have the constructor as their head.

3.3 Common subexpression elimination

Full inlining (followed by beta-reduction) causes replication of function argument expressions. Similarly, all “let” expressions are fully expanded, substituting the expression bound to a variable at each of the variable’s occurrences. This replication is sometimes beneficial, since it allows context-specific optimizations for each use of an argument. It is, however, then necessary to collect up all remaining repeated subexpressions, which we do by means of a common subexpression elimination (CSE) phase. Although our compiler performs inlining for function and non-function types, CSE only extracts non-functions, in order to avoid an expensive and nondeterministic higher-order matching operation.

Naturally one must be careful not to introduce unwarranted evaluation by abstracting multiple occurrences. For example, it would be quite wrong to transform

```
if a then b/c else (if d then b/c else e)
```

into the let expression

```
let x = b/c in if a then x else (if d then x else e)
```

When both `a` and `d` evaluate to `False`, `b/c` is evaluated in the transformed expression, but not in the original. Our CSE algorithm does a simple dataflow analysis to ensure that such erroneous abstractions do not happen.

3.4 Code generation

The back-end of our compiler is a simple stack-based virtual machine (VM), so it is easy to compile from the simple expression language into VM code. It should be noted that we use *strict evaluation* for all but conditional expressions (and for short-cut “and” and “or”, which desugar into conditionals).

3.5 Compilation Example

As a relatively simple example of compilation, Figure 2 shows the result of inlining definitions in the following expression, which generates the image of Figure 1. (For brevity and comprehensibility, CSE has also been performed.)

```
swirl 100 (checkerBoard 10 black white)
```

Simplification involves application of a few dozen rewrite rules, together with constant folding, "if-floating", and code hoisting. The result for our example is shown in Figure 3 (again, after CSE). These three transformations work roughly as follows.

Constant folding. Inlining often results in functions applied to constant arguments. For example, the definition of L45 in Figure 2 is simplified from "`2.0 * 3.14159 / 100.0`" to `0.0628318`, through two applications of constant folding. Constant folding is implemented as part of the rewriting phase.

If-floating. Conditionals are floated upwards to create more opportunities for constant folding and other simplifications. As an example, consider the definition of `L28` in Figure 2. One application of if-floating results in

```
f2i (if (floor L41 + floor L40) ‘mod‘ 2 == 0 then
        (0.0 * 255.0)
      else (1.0 * 255.0))
```

Another if-floating application yields

```
if (floor L41 + floor L40) ‘mod‘ 2 == 0 then
   f2i (0.0 * 255.0))
else (f2i (1.0 * 255.0))
```

Constant folding then produces

```
if (floor L41 + floor L40) ‘mod‘ 2 == 0 then
   0
else 255
```

```
loop height (\ v1 ->
  loop width (\ v2 ->
    let
      L45 = 2.0 * 3.14159 / 100.0
      L64 = negate (height ‘div‘ 2)
      L58 = zoom * i2f (v1 + L64) + panY
      L46 = L58 * L58
      L66 = negate (width ‘div‘ 2)
      L62 = zoom * i2f (v2 + L66) + panX
      L44 = sqrt (L62 * L62 + L46) * L45
      L39 = sin L44
      L29 = cos L44
      L35 = 1.0 / 10.0
      L40 = L35 * (L58 * L29 + L62 * L39)
      L41 = L35 * (L62 * L29 - L58 * L39)
      L28 = f2i ((if (floor L41 + floor L40) ‘mod‘ 2 == 0 then
                    0.0
                  else 1.0)
                 *
                 255.0)
      L18 = 0 <<< 8
      L9  = output + 4 * v1 * width
    in
      setInt (L9 + 4 * v2) (((L18 .|. L28) <<< 8 .|. L28) <<< 8 .|. L28)))
```

Figure 2. Unoptimized `swirl 100 (checkerBoard 10 black white)`

```
let
  L49 = negate (height `div` 2)
  L47 = negate (width `div` 2)
in
  loop height (\ v1 ->
    let
      L44 = zoom * i2f (v1 + L49) + panY
      L32 = L44 * L44
      L4  = v1 * width
    in
      loop width (\ v2 ->
        let
          L51 = zoom * i2f (v2 + L47) + panX
          L31 = 0.0314159 * sqrt (L51 * L51 + L32)
          L30 = sin L31
          L20 = cos L31
          L26 = 0.1 * (L44 * L20 + L51 * L30)
          L27 = 0.1 * (L51 * L20 + L44 * negate L30)
        in
          setInt (output + 4 * (L4 + v2))
                 (if (floor L27 + floor L26) `mod` 2 == 0 then
                    0
                  else 16777215)))
```

Figure 3. Optimized `swirl 100 (checkerBoard 10 black white)`

Code hoisting. Finally, an expression `e` inside a loop may be lifted outside of the loop if `e` does not make use of the loop index variable, and thus may be computed once and reused. Notice that in Figure 2 all computation is done inside of the inner loop. In contrast, the optimized version in Figure 3 has three phases of computation: once per frame (`L49`, `L47`), once per row of pixels (`L44`, `L32`, `L4`), and once per pixel (`L51`, `L31`, etc). It is well known that code hoisting and common subexpression elimination can be viewed as instances of a more general code motion transformation, and indeed we implement them in the same pass. The efficiency of this pass greatly benefits by the representation of terms as acyclic graphs.

The hoisting transformation is only valid if the loop body is executed at least once – this can be guaranteed by enclosing the generated code in a conditional that verifies that both loop bounds are greater than zero.

As the above example shows, our present implementation does not take advantage of subexpressions that only depend on the inner loop index variable, such as `L51` in our example. If the evaluation of such an expression is sufficiently expensive, it is worthwhile to store its values in an intermediate array. We have not yet implemented that optimization.

4 Related Work

The idea to compile code by specializing an interpreter has been pervasive in the partial evaluation literature, starting with the pioneering work of Futamura [14]. Scott Draves explored the use of specialization to perform run-time code generation for multi-media applications [7][8][9]. His approach is to use a general-purpose partial evaluator for a language that includes recursion. The only algebraic optimizations that his specializer can apply are the laws of linear algebra. This contrast with the present paper, where we have chosen to work with a very restricted language, and instead of a partial evaluator, we use domain-specific rewrite rules. Draves reports a number of problems with his implementation. First, it frequently fails to terminate, primarily because of code explosion. Second, code generation is too slow. By focusing on a more restricted language, and by using a DAG representation of program terms, we have been able to avoid the first problem. Our current implementation in Hugs has acceptable speed for experimentation, but we are not yet certain whether we can attain the speed necessary in the envisaged integration with a user interface.

Our implementation method is very similar to that suggested by Kamin [18]: first design the semantic domains, and the associated combinatory functions. The combinatory functions are then written as program generators in a modern functional programming language, in Kamin's case ML, and in our case Haskell. Evaluating these program generators corresponds to inlining of non-primitive definitions. Our approach differs from Kamin's in that we generate term structures, not strings. Kamin generates strings instead of tree structures because he wishes to embed his domain-specific programs in a larger programming language, for instance C++. Generating terms would require the programmer to define an abstract syntax for all C++ constructions used, a significant effort. For us, such embedding is not an issue. Consequently we can build our terms with ‘smart constructors’ that apply optimizing rewrite rules before returning a result. It also enables us to reduce code bloat by application of common subexpression elimination.

There are a rather large number of papers in the functional programming literature that treat a similar problem domain, but rather than designing combinators that generate programs in a target language, the semantic domains are directly embedded in a host language. One of the early examples of this approach is Henderson’s ‘functional geometry’, where pictures are modeled as functions, and pictures are combined with higher-order operations [15]. More recently, these ideas were extended to animations [10], and more recently *Fran*, a modeling language for interactive multimedia, embedded in Haskell [12][10]. Another example of the embedded approach is the ‘geometric region server’ of [16], which represents regions as what we call “Boolean images”. The advantages of such semantic embedding are manifold [17]: one obtains all the power of the host language in extending an existing library of primitives, in particular type checking. The approach is not suitable for the purpose of the interactive authoring environment sketched here, however, because the resulting domain-specific programs lack efficiency. It is quite difficult to overcome that problem, because it would involve tailoring a compiler for a general-purpose programming language with domain-specific optimizations. Indeed, the latest version of the Glasgow Haskell Compiler (GHC) has an experimental feature that allows the programmer to annotate his modules with domain-specific rewrite rules [21]. One of us (ODM) attempted to implement the language of this paper and its optimizations using that new feature of GHC, but the attempt failed, due to the complicated interaction between rewriting and inlining.

5 Future Work

5.1 Program generator toolkit

Our original implementation of the ideas in this paper made use of a small program transformation system called MAG [6]. That system implements a rather general engine for higher-order rewriting. We found that the compilation process quickly became painfully slow, mostly because repeated subterms were rewritten at each occurrence separately. In our current implementation, that problem is circumvented by implementing the rewrite rules as functions in Haskell: the Haskell interpreter will ensure that repeated arguments are only evaluated once. A disadvantage of that technique is that it is rather brittle: applying a substitution after the rewriting process would destroy the sharing, unless memoized. So while we gained efficiency, we lost much of the flexibility of our original MAG based implementation.

It would be very convenient to have the option of matching modulo associativity and commutativity. For example, we would like to apply the rule

```
  a + (-a) = 0
```

To expressions such as

```
  a + (b – a)
```

At present, such optimizations are missed out. In the same vein, it would be convenient if the code motion phase took account of associativity and commutativity: this would result in far more effective hoisting of loop invariant code.

We therefore propose to conduct our next implementation using a program generator toolkit. To keep the advantages of Kamin’s approach to software generation (where the domain-specific combinators are embedded in a typed higher-order language), we need similar functionality to rewriting engines such as Elan [3] or Maude [4], but with a programming interface in ML or Haskell. Both Elan and Maude provide matching modulo associativity and commutativity, as well as sophisticated ways of controlling the rewrite strategy. That freedom of rewrite strategy (not restricted to bottom-up rewriting via smart constructors) necessitates a “hashing cons” implementation of terms to preserve sharing of repeated subterms. Ideally, we would be able to smoothly interface to the implementations of Elan or Maude from Haskell. Given the current state-of-the-art in exchanging data with Elan [22], however, such an interface would probably require a lot of work, and the repeated conversions would significantly slow down the compilation process.

A separate issue concerns incremental compilation. Recall that we plan to use the language and compilation scheme introduced here in conjunction with a graphical user interface for generating the syntax. Typically the user will only make small changes – after each of these the picture has to be re-compiled and displayed. It would be very advantageous, therefore, if compilation could be done incrementally, even if that results in slightly worse code than complete recompilation. We hope to exploit the vast body of results on incremental evaluation of attribute grammars [23] (and its relation to rewriting [24]) to achieve this effect.

5.2 Spatial partitioning

The `reconstruct` function from Section 2.4.2 uses the `inBounds` function to test whether any given sample point is in the bounding region of a bitmap. Consider this test as a partitioning of 2D space into three regions. In one region the condition is false everywhere, in one it is true, and in the last it may be mixed. The compiler could make simplified versions for the false and true regions and leave the general version for the mixed. (Two cases, false and mixed, might suffice in relatively simple cases.) Compositions involving more than one conditional will cause further specialization (potentially exponentially many).

How can we come up with these partitions and verify constancy of conditions? One direct means is to synthesize them from the condition itself. Another means of partitioning would be to subdivide image space recursively into axis-aligned rectangular regions, and then try to prove condition constancy within each partition. Subdivision stops when either the expression contains no more conditionals or the region is small enough. This technique is closely related to Warnock's algorithm for hidden surface/line elimination [25][13].

It is not clear how to choose between these two partitioning techniques. The geometric approach may yield fewer regions and eliminate the mixed true/false regions altogether, but it can yield complex regions that are not convex and even have holes in them. Iterating over such regions correctly would be time-consuming, at least partially defeating the gains of the optimization.

5.3 Incremental computation

We have already indicated ways in which the hoisting of loop-invariant computations could be improved. Furthermore, it would pay to exploit time-invariance in the same way.

5.4 Video, 3D, sound

References

[1] K. Arya. A functional animation starter kit. Journal of Functional Programming, 4(1):1-18, 1994.

[2] P. Borovansky, S. Jamoussi, P.-E. Moreau, and C.Ringeissen: Handling ELAN Rewrite Programs via an Exchange Format. *In C. and H. Kirchner, editors. Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications, Pont-A-Mousson, France, September 1998.*

[3] P. Borovansky, C. Kirchner, H. Kirchner, P.E. Moreau and C. Ringeissen: An Overview of ELAN. In C. and H. Kirchner, editors. Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications, Pont-A-Mousson, France, September 1998.

[4] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In Proc. 1st Intl. Workshop on Rewriting Logic and its Applications, Electronic Notes in Theoretical Computer Science, Elsevier Sciences, 1996.

[5] J. Cocke and K. Kennedy. An algorithm for reduction of operator strength. Communications of the ACM 20(11), 1977.

[6] O. de Moor and G. Sittampalam. Generic Program Transformation. In: Proceedings of the third International Summer School on Advanced Functional Programming. Springer Lecture Notes in Computer Science, 1999. `http://users.comlab.ox.ac.uk/oege.demoor/papers/braga.ps.gz`.

[7] S. Draves. Automatic Program Specialization for Interactive Media. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, 1997.

[8] S. Draves. Compiler Generation for Interactive Graphics using Intermediate Code. In: O. Danvy, R. Glűck and P. Thiemann (editors), Partial Evaluation. Dagstuhl Castle, Germany, February 1996, Springer Lecture Notes in Computer Science Vol 1110, pages 95-114, 1996.

[9] S. Draves. Implementing Bit-addressing with Specialization. In Proceedings Of the 1997 ACM SIGPLAN International Conference on Functional Programming, Amsterdam, pages 239-250, 1997.

[10] C. Elliott. An embedded modeling language approach to interactive 3D and multimedia animation. IEEE Transactions on Software Engineering, 25(3), May 1999, pages 291-208.

[11] C. Elliott. A Pan Image Gallery, `http://research.microsoft.com/~conal/Pan/Gallery/-default.htm`.

[12] C. Elliott and P. Hudak. Functional Reactive Animation, in *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, `http://research.microsoft.com/~conal/papers/-icfp97.ps`

[13] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes, Computer Graphics – Principles and Practice (Second Edition), Addison-Wesley, 1990.

[14] Y. Futamura. Partial Evaluation of Computation Process – an approach to a compiler-compiler. Systems, Computers and Controls, 2:45-50, 1971.

[15] P. Henderson. Functional Geometry. In Conference Record of the 1982 Symposium on LISP and Functional Programming, Pittsburgh, PA, New York, NY, 1982. ACM.

[16] P. Hudak and M. P. Jones. Haskell vs. Ada vs. C++ vs Awk vs. … An Experiment in Software Prototyping Productivity. July 1994. Technical report, Yale University. `ftp://nebula.cs.yale.edu/pub/yale-fp/papers/-NSWC/jfp.ps`.

[17] P. Hudak. Modular Domain Specific Languages and Tools. Proceedings of the 5th International Conference on Software Reuse, ICSR 1998. IEEE Press.

[18] S. Kamin. Standard ML as a meta-programming language. Technical report, University of Illinois at Urbana-Champaign. `http://www-sal.cs.uiuc.edu/~kamin/pubs/index.html`.

[19] Y. A. Liu and S. D. Stoller. Loop optimization for aggregate array computations, In Proceedings of the 1998 IEEE International Conference on Computer Languages, pages 262-271, Chicago, Illinois, May 1998.

[20] S. Peyton Jones, John Hughes (editors), Haskell 98: A Non-strict, Purely Functional Language, February 1999, `http://www.haskell.org/definition`.

[21] S. Peyton Jones. Domain-specific rewrite rules in GHC. Personal communication. GHC can be downloaded from `http://www.haskell.org/ghc`.

[22] A.R. Smith, Image Compositing Fundamentals, Technical Memo #4, Memo 4, Microsoft, July 1995, `http://www.alvyray.com/Memos/default.htm`.

[23] T. Teitelbaum and T. Reps, The Synthesizer Generator: A System for Constructing Language-Based Editors, 315 pages, Springer-Verlag, NY, 1988

[24] E. van der Meulen. Incremental rewriting. PhD thesis, University of Amsterdam, 1994. `http://www.cwi.nl/-~gipe/xypages/emma.van.der.meulen.html`.

[25] J. Warnock. A Hidden-Surface Algorithm for Computer Generated Half-Tone Pictures, Technical Report, University of Utah, Computer Science Department, Number TR 4-15, NTIS AD-733 671, 1969.