Functional Specification of JPEG Decompression, and an Implementation for Free

Jeroen Fokker
Department of Computer Science, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
jeroen@cs.ruu.nl, http://www.cs.ruu.nl/~jeroen

August 7, 1995

Abstract

A decoder for images compressed by the JPEG algorithm is stated in the pure functional programming language Gofer. The program can be regarded as a mathematical specification of the decompression algorithm; the concise description (which is included in full) is very suitable for learning about the algorithm. At the same time the 'specification' is an executable program, which shows the usefulness of a functional programming language as a prototyping tool for graphics algorithms.

All functions are defined as much as possible at the function level, i.e. as compositions of other functions. A tutorial on the important concept of a 'State Monad', which plays an important role in the program, is included. From a functional programming theoretical point of view, the new technique of currying a state monad, which is introduced, and its application in the program, are interesting.

1 Introduction

JPEG is a standard for compressing images that has become very popular recently. Unlike general purpose compression algorithms, it exploits redundancy resulting from the two-dimensional structure of pictures, and from the continuous nature of photographic color images. Furthermore, it offers the possibility to let the compression lose some information, which is intended to be hardly noticeable by the human viewer. JPEG is named after its designer, the Joint (ISO and CCITT) Photographic Expert Group.

In the JPEG algorithm various techniques are combined: Huffman encoding, run-length encoding, differential encoding, quantization, cosine transform, and data reordering. A general introduction to the algorithm is given by Wallace [Wall91] in a 17 page article. It contains a numeric example which is quite instructive; however the information is not intended to be detailed enough to be able to implement the algorithm. For that, you would need the official (draft) standard [ISO93] (210 pages) and/or the book that explains it [PeMi93] (334 pages). The ISO description in the standard is not so nice as Wallace's article: algorithms are given by unstructured flowcharts, use fuzzy identifiers and lots of indices and pointers, and are laid out poorly. A typical example is:

```
CODE=(SLL CODE 1)+ NEXTBIT
J=VALPTR (I)
J=J+CODE-MINCODE (I)
```

I would therefore not recommend the ISO document for learning about the JPEG algorithm.

In some circles, functional programming has the reputation of being an academic plaything, only useful for toy problems like 'fibonacci' and '8 queens' and maybe some AI applications. This might be true for the earlier functional languages, but certainly not for the modern, polymorphically typed and lazily evaluated languages like Haskell [HuFa92], Gofer [Jone94] and Clean [PlEe95]. We will prove this by giving an implementation of a JPEG decoder in the Gofer language. This article can serve as a:

- *Specification.* The program has the conciseness of a mathematical description, and thus acts as a 'functional specification'. Unlike other specification formalisms, the language has a well defined semantics, and an interpreter that can check type correctness.
- *Teaching text.* The JPEG format can be understood by studying the decoder. Due to the abstraction mechanisms in the language, various aspects of the algorithm can be isolated and understood separately.
- *Implementation.* The program is executable, and has been applied successfully to decode images. The program is very slow (it takes 14 minutes to decode a 384×256 image). Running time could be improved considerably by using a compiler instead of an experimental interpreter, and by optimizing some functions (the cosine transform function in section 4.2 is a good candidate for this). We have not done so, because we consider the specification aspect of the program more important.
- *Functional programming tutorial and case study.* Some interesting programming techniques are used and explained. It shows that a functional language can be used for real life problems in graphics. In particular, it shows that by using a 'state monad', input-consuming functions can be defined, while keeping the benefits of abstraction in a functional language.

This article assumes no knowledge of JPEG or any of its algorithms, nor of specialized functional programming techniques. Basic knowledge of functional programming (recursion, manipulation of lists and the use of types, as described in the first few chapters of e.g. [BiWa88] or [Jone94]) may be helpful. However, it even may not be necessary, because the most important notions and notations are summarized in section 2.

The rest of this article is divided in two parts: sections 2–3 and sections 4–6.

Sections 2–3 describe some general purpose functions, that are needed in the sequel and that happen not to be part of the standard prelude of most languages. In section 2 matrix manipulation, bit lists and binary trees are dealt with. In section 3 the notions of 'state function' and 'monad' are introduced, and some utilities to manipulate them. Experienced functional programmers may want to skip these sections, although they might want to take a look at the end of subsection 3.4, where the new technique of currying state functions is described.

The JPEG decoding algorithm proper is dealt with in sections 4–6. In section 4 the basic algorithms used by JPEG are defined: Huffman coding, the Discrete Cosine Transform (DCT), and quantization. In section 5 functions for parsing the interleaved image data, and the image header are defined. (Subsection 5.1 is a particularly nice example of using types as a guide to design functions). Section 6 contains the main program of the JPEG decoder, which calls the parser, decodes the image, and converts it to another image format. Section 7 reflects on the program and the methodology.

2 A functional library

2.1 Auxiliary functions

In the functions in this article, we will use standard functions on lists, like `map`, `concat`, `zipWith` and `transpose`. These functions are defined in the standard prelude of most functional languages. Six functions of general nature that we need are not defined in the Gofer prelude. They are defined in this section, and may also serve to get used to the Gofer syntax.

The result of integer divisions is truncated. We provide a version which calculates the ceiling $\lceil n/d \rceil$ of a division instead. In the type of the function, an arrow is written not only between the type of the parameters and the result, but also between the two parameters. The type $a \to b \to c$ is to be read as $a \to (b \to c)$, which stresses the fact that a function may also be partially parameterized with its first parameter only. This mechanism is known as 'Currying'.

```
ceilDiv    :: Int -> Int -> Int
ceilDiv n d = (n+d-1)/d
```

Partial parametrization is also useful when defining functions. The function `multi` takes an integer n and a list, and replicates each element of the list, which remains unnamed, n times.

```
multi  :: Int -> [a] -> [a]
multi n = concat . map (copy n)
```

The function is defined as a functional composition (denoted by the standard operator 'dot') of the `map (copy n)` function (which turns every element into a list of length n) and the `concat` function (which concatenates all lists into one big list). The function `multi` could also have been defined by explicitly naming the second parameter: `multi n xs = concat (map (copy n) xs)`. However, this is avoided whenever possible in order to not overwhelm the reader with unnecessary names. Occasionally, we will also need to compose functions of two parameters. As this is not a standard function, we will define it here. The function `o` may be used as an infix operator by writing its name in back quotes.

```
infixr 9 `o`
o :: (c->d) -> (a->b->c) -> (a->b->d)
(g `o` f) x y = g (f x y)
```

In addition, we define an explicit denotation `ap` for functional application, and a variant `ap'` with its parameters reversed:

```
ap     :: (a->b) -> a -> b
ap f x  = f x

ap'    :: a -> (a->b) -> b
ap' x f = f x
```

An unorthodox use of functions is their use as updatable association tables. The function `subst` modifies a given function with respect to one possible parameter. The predicate `Eq a =>` in front of the type means that the function is only defined for types `a` for which equality is defined. In section 6 we will use this function for integer indexed lookup tables, for which we provide the type synonym `Table` here.

```
subst :: Eq a => a -> b -> (a->b) -> (a->b)
subst i e t j  | i==j      =  e
               | otherwise =  t j

type Table a  =  Int -> a
```

2.2 Matrix manipulation

Matrix manipulation is a rewarding area for functional programming, as the definitions of most operations are short and elegant and don't need lots of indices as in many other formalisms. More important, we will need these functions in section 4 for the DCT operation, and in section 6 for color space conversion. A matrix is simply a list of lists, of which we will assume that the rows have equal length. The dimensions of a matrix can be indicated by a pair of two integers.

```
type Dim   = (Int,Int)
type Mat a = [[a]]
```

We provide a function `matmap` which applies a function to all elements of a matrix, a function `matconcat` which collapses a matrix of sub-matrices into one big matrix, and a function `matzip` which transforms a list of matrices into a matrix of lists of corresponding elements.

```
matmap :: (a->b) -> Mat a -> Mat b
matmap  = map . map

matconcat :: Mat (Mat a) -> Mat a
```

```
matconcat  = concat . map (map concat . transpose)

matzip :: [Mat a] -> Mat [a]
matzip  = map transpose . transpose
```

The classic operations from linear algebra (inner product of vectors and linear transformation of a vector by a matrix) presuppose the existence of arithmetical operations on the elements, which is indicated by the `Num a` predicate in front of the type.

```
inprod :: Num a  =>  [a] -> [a] -> a
inprod  = sum `o` zipWith (*)

matapply    :: Num a  =>  Mat a -> [a] -> [a]
matapply m v = map (inprod v) m
```

Inner product is defined as elementwise multiplication followed by summation; matrix application as calculating the inner product of a vector with all rows of the matrix.

2.3 Bit Streams

Of a more mundane nature are some functions that address the individual bits in a byte, and by extension, in a string. In the same vein the function `byte2nibs` splits a byte in two four-bit nibbles. The standard function `rem` is used to calculate the remainder after division.

```
type Bits = [Bool]

byte2bits  :: Int -> Bits
byte2bits x = zipWith (>=) (map (rem x) powers) (tail powers)
      where powers = [256,128,64,32,16,8,4,2,1]

string2bits :: String -> Bits
string2bits  = concat . map (byte2bits.ord)

byte2nibs  :: Int -> (Int,Int)
byte2nibs x = (x/16, x`rem`16)
```

With some effort, the `rem` operation could be avoided by repeated subtraction, but as our goal is a clear specification rather than an efficient implementation, we don't do that here. In other languages shifting and masking operators may be used.

2.4 Binary Trees

Binary trees, which will be used to represent Huffman trees in section 4, are defined by an algebraic type definition. Information is stored in the `Tips` of the tree, there may be `Nil` ends, and in `Bin` branching points only two subtrees are given.

```
data Tree a  =  Nil
             |  Tip a
             |  Bin (Tree a) (Tree a)
```

The function `map` can be overloaded to also operate on trees, by making `Tree` an instance of `Functor`, the class of all types supporting the `map` function.

```
instance Functor Tree where
    map f Nil       =  Nil
    map f (Tip a)   =  Tip (f a)
    map f (Bin x y) =  Bin (map f x) (map f y)
```

3 Modelling of State

3.1 State Functions

Modelling of state has long been a problem when using pure functional languages, which by their nature are side-effect free. However, recently it has been discovered that state can be adequately dealt with using so-called 'monads' [Wadl92, Jone93, Jone95].

A 'state function from s to r', or `StFun s r` for short, is a function that operates on a type s (the 'state') and yields not only a value of type r (the 'result'), but also a value of type s (the 'updated state'). An algebraic type definition, involving an explicit conversion `ST` is used rather than a type synonym definition, as state functions are to be regarded as an abstract data type, to be manipulated only by the functions below.

```
data StFun s r = SF (s -> (r,s))
```

Firstly, state functions are made an instance of `Functor`, where the `map` function applies a given function to the 'result' part of a state function:

```
instance Functor (StFun s) where
    map h (SF f)    = SF g
                where g s = (h x,s')
                       where (x,s') = f s
```

Furthermore, state functions are made an instance of the `Monad` class. For this, a function `result` and a function `bind` need to be defined that fulfil certain laws. In this instance, the `result` function constructs a state function which delivers some result x without changing the state, and the `bind` function composes two state functions in an intricate way:

```
instance Monad (StFun s) where
    result x        = SF g
                where g s = (x,s)
    SF f `bind` sfh = SF g
                where g s = h s'
                        where (x,s') = f s
                              SF h   = sfh x
```

We will not use the `bind` function explicitly in the sequel. Instead we make use of a syntactic sugaring known as 'monad comprehension', provided in the Gofer language [Jone94], which is discussed in subsection 3.3. A state function can be applied to an initial state using the function `st'apply`. This yields the proper result only, and discards the final state.

```
st'apply :: StFun a b -> a -> b
st'apply (SF f) s  =  x
               where (x,_) = f s
```

3.2 Primitive State Functions

In the JPEG decoder, as a state we will basically use a list. We provide three primitive functions that operate on list states, from which the more involved ones can be constructed. The `empty` state function reports whether the list in the state is empty, and leaves the state unmodified. The `item` state function returns the first element of the list in the state (which is assumed to be non-empty), and removes that element from the list. The `peekitem` state function returns the first element without removing it from the list.

```
empty  ::  StFun [a] Bool
empty   =  SF f
    where  f [] = (True,  [])
           f xs = (False, xs)

item   ::  StFun [a] a
```

```
item    =  SF f
    where  f (x:xs) = (x, xs)

peekitem   ::  StFun [a] a
peekitem    =  SF f
        where  f ys@(x:xs) = (x, ys)
```

A fourth primitive function meets a more special purpose. In the JPEG format, a binary data stream is terminated by a two-byte marker consisting of an '`\xFF`' byte and a non-zero byte. If an '`\xFF`' byte occasionally occurs in a data stream, it is padded by an extra zero byte. The state function `entropy` below gets one segment of binary data, taking care of the padding, and leaves behind as final state a list that begins with the terminating marker.

```
entropy :: StFun String String
entropy =  SF f
    where  f ('\xFF':'\x00':xs)  = let (as,bs) = f xs in ('\xFF':as,bs)
           f ys@(    '\xFF':_ )  = ([],ys)
           f (            x:xs)  = let (as,bs) = f xs in (x:as,bs)
```

3.3 Auxiliary State Functions

The state function `item` gets one character from a string state, removing it from the state. The state function `byte` does the same, but yields its result as an integer rather than as a character. It can be defined as `map ord item` (where `ord` is the primitive char-to-int function). Recall that `map` was overloaded in subsection 3.1, so that `map` h f applies a function h to the result part of a state function f. We write the definition however in the form:

```
byte :: StFun String Int
byte  = [ ord c | c <- item ]

nibbles :: StFun String (Int,Int)
nibbles  = [ byte2nibs a | a <- byte ]
```

These look like list comprehensions, but as `map` is overloaded to operate on arbitrary `Functor`s, so is the comprehension notation. From the type in the expressions above it can be inferred that these comprehensions are actually 'string-state function comprehensions'. The comprehension notation is especially useful when more than one generator is being used:

```
word :: StFun String Int
word  = [ a*256+b | a<-byte, b<-byte ]
```

Comprehensions with multiple generators may be used not only for lists, but for arbitrary `Monad`s, and hence in particular for state functions. The semantics of comprehension like this is defined using the `result` and `bind` functions (see [Jone94]), but can be more easily understood intuitively: a 16-bit word can be fetched from a string state by successively fetching two bytes and combining them arithmetically.

3.4 State Function Combinators

The generalized comprehension notation will be used in this subsection to define some transformation utilities ('combinators') for state functions. The function `list` transforms a list of state functions into one state function with a list as result. As it makes use of no particular property of state functions, the function is actually applicable to any monad (for the list monad, the `list` function boils down to `transpose`).

```
-- list    ::             [StFun s r] -> StFun s [r]
list       :: Monad m => [m       a] -> m       [a]
list []     = result []
list (f:fs) = [ x:xs | x<-f, xs<-list fs]
```

The function `exactly` transforms a state function into one that applies the original one a given number of times, and thus results in a list of values. The function `matrix` is parameterized with two integers, and constructs a matrix-valued state function. Just a `list`, the functions `exactly` and `matrix` are defined for any monad, but are used only for state functions in the sequel. You may therefore read `StFun s` instead of the arbitrary monad `m` in the definitions below.

```
exactly          :: Monad m => Int -> m a -> m [a]
exactly 0      f  = result []
exactly (n+1) f  = [ x:xs | x<-f, xs<-exactly n f ]

matrix       :: Monad m => Dim -> m a -> m (Mat a)
matrix (y,x) = exactly y . exactly x
```

A combinator that is specific for state functions that have a list as state is `many`, which applies a state function as many times as possible until the state has become the empty list.

```
many   :: StFun [a] b -> StFun [a] [b]
many f  = [ if b then [] else y:ys
          | b  <- empty
          , y  <- f
          , ys <- many f
          ]
```

The type 'state function' was defined as a function that operates on a state, and returns a result and an (updated) state. Now imagine the situation where in conjunction to applying a state function `StFun a c`, a function `b->b` has to be evaluated. This could be modelled in two ways:

- the type `b` is tupled with the state: `StFun (a,b) c`
- the type `b` is added as an extra parameter, and tupled with the result: `b -> StFun a (b,c)`.

Both approaches are equivalent, in the sense that conversion functions between the two implementations can be written:

```
sf'curry  ::  StFun (a,b) c  ->  (b -> StFun a (b,c))
sf'curry (SF h) = f
            where f b = SF g
                   where g a = ((b',c),a')
                          where (c,(a',b')) = h (a,b)

sf'uncur  ::  (b -> StFun a (b,c))  ->  StFun (a,b) c
sf'uncur f = SF h
       where h (a,b) = (c, (a',b'))
                 where SF g        = f b
                       ((b',c),a')  = g a
```

These transformations are the analogues for state functions of the `curry` and `uncurry` operations on normal functions. Note the nice symmetry in the definitions: the equations in `sf'uncur` are the same as in `sf'curry`, written right to left.

All functions defined thus far (except `entropy`) are quite abstract, and should really be part of a monad or state function library. They have been treated here to make this article self-contained. The implementation of the proper JPEG algorithm starts in the next section.

4 JPEG Fundamental Algorithms

4.1 Huffman Trees

A Huffman coding translates values with a higher probability of occurrence into codes of shorter length, thus reducing the overall length of a message. Huffman codes can be decoded if all possible values are stored in a binary tree. The bits in a code are used as navigating instructions in the tree; on arriving in a tip, the value found there is the value corresponding to the bits consumed. As the number of bits that make up one code is variable, the decoding function is best modelled as a state function, which consumes as many bits as necessary from a `[Bool]` (or `Bits`) state.

```
lookup              :: Tree a -> StFun Bits a
lookup (Tip x)       = result x
lookup (Bin lef rit) = [ x
                       | b <- item
                       , x <- lookup (if b then rit else lef)
                       ]
```

In the JPEG algorithm, four different Huffman trees are used. Each of them is specified in the header of the file as a list of lists. These lists contain the values that have codes of equal length. The construction of the corresponding Huffman tree can be elegantly described by an inductively defined state function. The state type used is `[(a,Int)]`, a list of values with associated code length. The initial state can be easily computed from the `[[a]]` representation:

```
huffmanTree ::  [[a]] -> Tree a
huffmanTree  =  st'apply (build 0) . concat . zipWith f [1..16]
         where  f s = map (\v->(v,s))

build  :: Int -> StFun [(a,Int)] (Tree a)
build n = [ if b then Nil else t
          | b     <- empty
          , (_,s) <- peekitem
          , t     <- if   n==s
                     then [Tip v   | (v,_) <- item]
                     else [Bin x y | x <- build (n+1), y <- build (n+1)]
          ]
```

In the JPEG algorithm, Huffman coding is not used to code values directly, but instead to code the *number of bits* that is used to store values. Given its number of bits, a value can be fetched from a bit-stream by the function `receive`.

```
receive      :: Int -> StFun Bits Int
receive 0     = result 0
receive (k+1) = [ 2*n + (if b then 1 else 0)
                | n <- receive k
                , b <- item
                ]
```

The decoding of a value is thus a two-stage process: first some bits are consumed to find a number of bits in the Huffman tree, and next that number of bits is received from the state. Before yielding the value, the function `extend` is applied to it. This function cares about negative numbers. For example with $s = 5$, the numbers 16 to 31 stand for themselves, but numbers 0 to 15 stand for -32 to -16. The numbers 0 to 15 could have been described in less than 5 bits, after all.

```
dcdecode  :: Tree Int -> StFun Bits Int
dcdecode t = [ extend v s
             | s <- lookup t
             , v <- receive s
             ]

extend v s | s==0      =  0
           | v>=half   =  v
           | otherwise =  v + 1 - 2*half
                 where  half = 2^(s-1)
```

The function `dcdecode` could be repeatedly used to fetch multiple numbers. However, in practice long sequences of zeroes are to be expected in the data, and to code them even with a 1-bit Huffman code would be a waste of space. Instead, the mechanism described above is combined with run-length encoding of sequences of zeroes. For this situation, a different Huffman tree is used, which stores not only a code size s but also a zero-run length r. Decoding now first involves consuming enough bits from the state to find an (r, s) pair in the Huffman tree. Next, a value x is `receive`d and `extend`ed, but it is prepended with r copies of zero in the result. Finally, the function is called recursively to get even more values. An extra parameter k is used to stop the recursion when 63 values have been accumulated. A special case is when both r and s are zero, which is a signal that only zeroes should be appended to complete a batch of 63 values.

```
acdecode :: Tree (Int,Int) -> Int -> StFun Bits [Int]
acdecode t k
  = [ x
    | (r,s) <- lookup t
    , x  <- let  k' =  k + r + 1
            in   if   r==0 && s==0
                 then [ copy (64-k) 0 ]
                 else [ copy r 0 ++ (extend x s : xs)
                      | x  <- receive s
                      , xs <- if k'>=64 then [[]] else acdecode t k'
                      ]
    ]
```

Both the function `dcdecode` and the function `acdecode` will be used in subsection 5.1. The names 'dc' and 'ac' are used frequently in the JPEG literature, as the interpretation of the values will turn out to be vaguely reminiscent of 'direct current' and 'alternating current' in electricity.

4.2 Discrete Cosine Transform

The key to the high compression ratios of JPEG compression is that the image data is not compressed directly, but first transformed by a mathematical operator known as the 'Discrete Cosine Transform' (DCT). The cosine transform amounts to expressing a continuous function as a sum of (infinitely many) cosine functions with increasing frequencies. In the discrete version, the function is sampled at a number of points (JPEG uses 8 sampling points), and only that number of cosine functions are needed. The correspondence between samples s and their transformations t is:

$$\begin{aligned} t(u) &= \textstyle\sum_{x=0}^{7} \frac{C(u)}{2} s(x) \cos\left((2x+1)\ u\ \pi/16\right) \\ s(x) &= \textstyle\sum_{u=0}^{7} \frac{C(u)}{2} t(u) \cos\left((2x+1)\ u\ \pi/16\right) \end{aligned}$$

where $C(0) = 1/\sqrt{2}$ and $C(u) = 1$ for $u \neq 0$. For the inverse transformation the second relation can be directly coded into Gofer

```
idct1 :: [Float] -> [Float]
idct1  = matapply cosinuses

cosinuses :: Mat Float
cosinuses  = map f [1,3..15]
       where f x = map g [0..7]
               where g 0 = 0.5 / sqrt 2.0
                     g u = 0.5 * cos(fromInteger(x*u)*(pi/16.0))
```

In JPEG a two-dimensional version of the DCT is used, which transforms an 8×8 block of data into 8×8 DCT coefficients. A two-dimensional DCT can be performed by first transforming each row, and then transforming each column of the resulting 8 rows of coefficients:

```
idct2 :: Mat Float -> Mat Float
idct2  = transpose . map idct1 . transpose . map idct1
```

In the resulting block of coefficients, the upper left value can be interpreted as the average of the original data (if you know enough of electricity this may justify the name 'd(irect) c(urrent) coefficient'). The other 63 values ('ac coefficients') represent higher and higher harmonics as the bottom right is approached.

The function `idct1` performs 64 multiplications, and hence `idct2` requires $(8 + 8) \times 64 = 1024$ multiplications. Various clever schemes, which exploit the symmetry in the cosine matrix to bring down the number of multiplications, are summarized by Pennebaker [PeMi93]. Hartel and Vree [HaVr92] discuss how fast algorithms for the related Fourier transform (FFT) can be implemented and optimized in a functional language.

4.3 Quantization and Downsampling

The DCT as such does not bring in any data reduction, as an 8×8 block of image data is transformed into an 8×8 block of DCT coefficients. However, in continuous-tone images, such as scanned photographs, higher harmonics tend to be absent. Sequences of zeroes in the DCT coefficients are coded very efficiently by the run-length encoding described in subsection 4.1.

To encourage the presence of zeroes, the DCT coefficients are *quantized* during encoding, i.e. mapped to a smaller interval, by dividing by a constant and rounding to an integer. Small coefficients will vanish, and larger coefficients lose unnecessary precision. The quantization factor can be specified for each coefficient separately. Thus the unimportant higher harmonics can be quantized more than the lower harmonics. The quantization factors are determined during encoding (typically by a user selecting a 'quality'), and stored in the image header. During decoding, which we deal with here, the coefficients are multiplied again by the quantization factors.

The coefficients in the 2-dimensional matrix of coefficients are not stored in row or column order, but in a zigzag order, which again promotes long sequences of zeroes, especially at the end if the list (bottom-right of the matrix) of coefficients. The function `dequant` below takes a list of 64 quantization factors and a list of 64 quantized coefficients, multiplies them together, constructs a matrix in zigzag order, and then performs a 2-dimensional inverse DCT.

```
type QuaTab = [Int]

dequant ::QuaTab -> [Int] -> Mat Int
dequant = matmap truncate `o` idct2 `o` zigzag `o` map fromInteger `o` zipWith (*)
```

The function `zigzag` puts the 64 elements of a list in the desired order in a matrix:

```
zigzag xs = matmap (xs!!) [[ 0, 1, 5, 6,14,15,27,28]
                          ,[ 2, 4, 7,13,16,26,29,42]
                          ,[ 3, 8,12,17,25,30,41,43]
                          ,[ 9,11,18,24,31,40,44,53]
                          ,[10,19,23,32,39,45,52,54]
                          ,[20,22,33,38,46,51,55,60]
                          ,[21,34,37,47,50,56,59,61]
                          ,[35,36,48,49,57,58,62,63]
                          ]
```

During compression, an image can optionally be 'downsampled' by averaging over an (e.g. 2×2) block of pixels, and treating them as one pixel. It is common practice to decompose an image in a grey-value ('luminance') component and two components describing color information ('chrominance'), and downsample the chrominance information to half its resolution. This immediately alters the size of an image by a factor of $\frac{2}{3}$ (two out of three components) $\times\frac{3}{4}$ (three out of four pixels are discarded) $= \frac{1}{2}$. During decoding the image is 'upsampled' again by:

```
upsamp       :: Dim -> Mat a -> Mat a
upsamp (1,1) = id
upsamp (x,y) = multi y . map (multi x)
```

5 JPEG Data Organization

5.1 Decoding Units

The DCT transformation operates on an 8×8 block of integers. However, a picture is usually bigger, and consists of more components (e.g., red/green/blue or luminance/twice chrominance). In this subsection we will compose the blocks together.

First, let's formalize how one 8×8 block is processed. The function `dataunit` is basically a state function, which consumes bits from the state and returns a `DataUnit` (a matrix of integers). The function is parameterized by upsampling factors, a quantization table, and two Huffman trees, which are conveniently grouped together in a four-tuple called `DataSpec`. The function uses `dcdecode` to fetch a 'dc' coefficient from the bits in the state. As a final optimization this is not the value of the dc coefficient itself, but the difference from the dc coefficient of the previous block. Therefore, the function has an additional integer parameter which specifies the previous dc coefficient, and returns the new coefficient as part of the result for use in the next block. After the dc coefficient, the ac coefficients are fetched form the bits state. Together, the coefficients are dequantized, DCTransformed and upsampled by the functions defined in subsection 4.3.

```
type DataUnit =  Mat Int
type DataSpec =  (Dim, QuaTab, Tree Int, Tree (Int,Int))

dataunit ::  DataSpec -> Int -> StFun Bits (Int,DataUnit)
dataunit (u,q,dc,ac) x = [ let y=x+dx
                            in  (y, upsamp u (dequant q (y:xs)))
                          | dx <- dcdecode dc
                          , xs <- acdecode ac 1
                          ]
```

To process more than one dataunit, we need a way to communicate the dc coefficient that was part of the result of the first block to the next block. It would be convenient if it was part of the state, so that it could be 'modified' after processing each dataunit. Careful inspection of the type of `dataunit` reveals that this is possible: the extra `Int` parameter to the state function and the `Int` part of its result can be attached to the state by the `st'uncur` combinator from section 3.4. Another combinator, `matrix`, can be used to transform the `StFun (Bits,Int) DataUnit` state function into one that returns a *matrix* of `DataUnits` instead of just one: `StFun (Bits,Int) (Mat DataUnit)`. However, we immediately let this matrix collapse to one big `DataUnit` by applying `matconcat` to the result of the state function (using `map` in its state function result transforming role). All together, we have a state function that fetches a block (of which the dimensions are given by `dim`) of 8×8 blocks:

```
units    :: Dim -> DataSpec -> StFun (Bits,Int) DataUnit
units dim = map matconcat . matrix dim . sf'uncur . dataunit
```

A group of such big blocks, one for each image component, is called a *minimum coding unit* (MCU) in JPEG terminology. Imagine the case that luminance information is not downsampled, and chrominance components are downsampled by factors 2×2 and 4×4, respectively. Then we need 16 luminance units, and 4 and 1 chrominance units to make up one MCU. Indeed, the units of the various components are interleaved in this order during encoding. The development of the state function `mcu`, which fetches an entire MCU from a bit stream, is driven mainly by inspecting the types involved.

First, we construct a version of `units` in which the `Int` is detached from the state again, and in which the `Dim` and `DataSpec` parameter are tupled together:

```
units'  :: (Dim,DataSpec) -> Int -> StFun Bits (Int,DataUnit)
units'   =  sf'curry . uncurry units
```

Our first approximation of the `mcu` function is just applying `units'` to a `(Dim,DataSpec)` combination for each component:

```
type MCUSpec  =  [(Dim, DataSpec)]
mcu     :: MCUSpec -> [ Int -> StFun Bits (Int,DataUnit) ]
mcu      = map units'
```

The list of functions that is the result of `mcu` could be applied elementwise to a list of integers:

```
mcu'    :: MCUSpec -> [Int] -> [ StFun Bits (Int,DataUnit) ]
mcu'     = zipWith ap . mcu
```

A list of state functions can be transformed into a state function for a list by the `list` combinator from section 3.4. Then we have a state function of type `StFun Bits [(Int,DataUnit)]`. The result part of this can be `unzip`ped. For the functional composition, we use `'o'`, because `mcu'` has two additional parameters:

```
mcu''   :: MCUSpec -> [Int] -> StFun Bits ([Int],[DataUnit])
mcu''    = map unzip `o` list `o` mcu'
```

Now we are almost done. The `[Int]` which appears both as a parameter and as part of the result is attached to the state again, and the list of matrices in the result is `matzip`ped to a matrix of lists:

```
type Picture  =  Mat [Int]
mcu'''  :: MCUSpec -> StFun (Bits,[Int]) Picture
mcu'''   = map matzip . sf'uncur . mcu''
```

The function is now in its ideal form. The state consists of bits that contain the compressed image, and a list of integers that contains the last dc coefficient seen (one for each component). The result is a picture, which is a matrix with for each pixel information about all the components.

What remains to be done is repeatedly fetch MCU's in order to make a complete picture. Note that, as it is part of the state, the list of 'last dc seen' is passed silently from one MCU to the next.

```
picture :: Dim -> MCUSpec -> StFun (Bits,[Int]) Picture
picture dim  = map matconcat . matrix dim . mcu'''
```

All these auxiliary functions can be summarized by the following two definitions, which capture the entire JPEG interleaving scheme in a few lines:

```
units dim =  map matconcat  .  matrix dim  .  sf'uncur   . dataunit
pict dim  =  map matconcat  .  matrix dim  .  map matzip . sf'uncur
          . map unzip     `o` list       `o` zipWith ap
          . map (sf'curry . uncurry units)
```

5.2 JPEG Header structure

What remains to be dealt with, is parsing the JPEG image header in order to collect the various Huffman tables, quantization factors and other parameters. Again, state functions facilitate things considerably.

A JPEG file is partitioned in 'segments'. Each segment starts with a 'marker' (that indicates the type of the segment), followed by additional information. For most pictures, only four segment types are of importance: Start Of Frame (SOF), Define Huffman Table (DHT), Define Quantization Table (DQT), and Start Of Scan (SOS (quite appropriate)). Here are some type definitions that describe the relevant information for each segment type. Some segments contain repeated information, for which we define separate types. The type `XXX` is for all remaining segment types, which we leave uninterpreted.

```
type SOF = (Dim,[FrameCompo])
type DHT = (Int,Int,Tree Int)
type DQT = [QtabCompo]
type SOS = ([ScanCompo],Bits)
type XXX = (Char,String)
type FrameCompo = (Int,Dim,Int)
type ScanCompo  = (Int,Int,Int)
type QtabCompo  = (Int,[Int])
```

For each type we write a state function that is able to fetch the relevant data from a string.

```
frameCompo = [ (c,dim,tq)
             | c <- byte, dim <- nibbles, tq <- byte
             ]
qtabCompo  = [ (id,qt)
             | (p,id) <- nibbles
             , qt <- exactly 64 (if p==0 then byte else word)
             ]
```

```
scanCompo  = [ (cs,td,ta)
             | cs <- byte, (td,ta) <- nibbles
             ]
```

The segments that contain repeated information first contain a count for the replicated information, which can conveniently be passed to the `exactly` function. Note that in monad comprehensions, as in list comprehensions, later generators may make use of values declared by earlier generators. For example, in `sofSeg`, the generator `fcs <- exactly n frameCompo` makes use of the value `n` that was fetched before by `n <- byte`. The bulk of the data is of course consumed in the `entropy` part of an SOS segment.

```
sofSeg = [ ((y,x), fcs)
         | _ <- word, _ <- byte, y <- word, x <- word
         , n <- byte, fcs <- exactly n frameCompo
         ]
dhtSeg = [ (tc, th, huffmanTree v)
         | _ <- word, (tc,th) <- nibbles, ns <- exactly 16 byte
         , v <- list (map (flip exactly byte) ns)
         ]
dqtSeg = [ qts
         | len <- word
         , qts <- exactly ((len-2)`rem`64) qtabCompo
         ]
sosSeg = [ (scs, string2bits ent)
         | _ <- word, n <- byte, scs <- exactly n scanCompo
         , _ <- byte, _ <- byte, _   <- nibbles, ent <- entropy
         ]
```

Next, a parser for 'a segment of any type' is defined. It is parameterized with five functions that cast the five segment types into some type `a`.

```
segment :: (SOF->a, DHT->a, DQT->a, SOS->a, XXX->a) -> StFun String a
segment (sof,dht,dqt,sos,xxx) =
  [ s
  | _ <- item
  , c <- item
  , s <- case c of
         '\xC0' -> map sof sofSeg
         '\xC4' -> map dht dhtSeg
         '\xDB' -> map dqt dqtSeg
         '\xDA' -> map sos sosSeg
         '\xD8' -> [ xxx (c,[]) ]
         '\xD9' -> [ xxx (c,[]) ]
         _      -> [ xxx (c,xs) | n <- word, xs <- exactly (n-2) item ]
  ]
```

6 Main program

6.1 Interpreting JPEG Segments

In the preceding section we have left unspecified what should happen with the data found in various segments. We will now develop a function that lets segments communicate their data to others.

As segments are processed, some data needs to be stored for use in later segments. We define some new types, that contain the relevant data in a more accessible form than they were parsed.

```
type Sof   =  (Dim, Table(Dim,QuaTab))
type Huf   =  (Table(Tree Int), Table(Tree (Int,Int)))
type Qua   =  Table QuaTab
```

```
type State =  (Sof,Huf,Qua,Picture)
```

Now we call the `segment` function `many` times, providing five functions that yield a `State->State`.

```
segments :: StFun String [State->State]
segments = many (segment (sof,dht,dqt,sos,xxx))
     where sof x s@(a,b,c,d) = (evalSOF x s, b, c, d)
           dht x s@(a,b,c,d) = (a, evalDHT x s, c, d)
           dqt x s@(a,b,c,d) = (a, b, evalDQT x s, d)
           sos x s@(a,b,c,d) = (a, b, c, evalSOS x s)
           xxx _ s           = s
```

The main decoder function applies `segments` as a state function to the JPEG compressed data. This yields a list of `State` transformations, which are applied one after another starting with an undefined state `errState`. After all segments are processed, only the `Picture` is retained. (If you would like to inspect, say, the Huffman tables, you could use `pi2` instead of `pi4`).

```
jpegDecode :: String -> Picture
jpegDecode  = pi4 . foldl ap' errState . st'apply segments
        where pi4 (_,_,_,x) = x
              errState      = (error"SOF", error"DHT", error"DQT", error"SOS")
```

What remains to be defined are the four `eval`... functions that interpret the segment data. They are quite boring, as they only dispatch information. The function `evalSOF` sets up an association table in which all frame component information is stored. The function `evalDHT` sorts out Huffman trees for dc and ac coefficients. The ac trees are initialized as trees of pairs instead of integers. The function `evalDQT` again simply sets up an association table, from which quantization tables can be fetched by identifying number.

```
evalSOF :: SOF -> State -> Sof
evalSOF (dim,xs) (~(_,sof),_,qua,_)  =  (dim, foldr f sof xs)
                                 where  f (i,d,q) = subst i (d,qua q)

evalDHT :: DHT -> State -> Huf
evalDHT (0,i,tree) (_,~(hdc,hac),_,_) = (subst i tree hdc, hac)
evalDHT (1,i,tree) (_,~(hdc,hac),_,_) = (hdc, subst i (map byte2nibs tree) hac)

evalDQT :: DQT -> State -> Qua
evalDQT xs (_,_,qua,_) =  foldr f qua xs
                   where  f (i,q) = subst i q
```

The function `evalSOS` sets up the `MCUSpec` that is needed to parameterize the function `picture` from section 5.1. Most important of all, the state function yielded by `picture` is called, using `(xs,[0,0,0])` as initial state: `xs` is the entropy encoded data, and `[0,0,0]` is an initialization of the 'previous dc values'.

```
evalSOS :: SOS -> State -> Picture
evalSOS (cs,xs) (((y,x),sof),(h0,h1),_,_)
                                 =  st'apply thePicture (xs,[0,0,0])
            where thePicture     =  picture mcuCount mcuSpec
                  mcuSpec        =  map f cs
                  f (id,dc,ac)   =  (d, (upsCount d, qt, h0 dc, h1 ac))
                            where   (d,qt) = sof id
                  mcuCount       =  ( ceilDiv y (8*maxy), ceilDiv x (8*maxx) )
                  upsCount (h,w) =  ( maxy/h, maxx/w )
                  maxy           =  maximum ( map (fst.fst) mcuSpec )
                  maxx           =  maximum ( map (snd.fst) mcuSpec )
```

6.2 Using a Picture

The `Picture` that is the result of `jpegDecode` is a matrix of pixels, where each pixel consists of a list of integer components with values in the range -128 to 127. The interpretation of the components is left unspecified by JPEG, but most often they represent luminance, chrominance blue, and chrominance red. These values can be converted to and from the more widely used RGB coordinates by the following linear transformations:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 0 & \frac{8}{5} \\ 1 & -\frac{1}{3} & -\frac{4}{5} \\ 1 & 2 & 0 \end{pmatrix} \begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} \qquad \begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} \frac{3}{10} & \frac{6}{10} & \frac{1}{10} \\ -\frac{3}{20} & -\frac{6}{20} & \frac{9}{20} \\ \frac{7}{16} & -\frac{6}{16} & -\frac{1}{16} \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

A function that converts JPEG data to an RGB picture is:

```
yCbCr2rgb  =  matmap f
       where  f =  map ((+128).(/15)) . matapply [ [15,  0, 24]
                                                 , [15, -5,-12]
                                                 , [15, 30,  0]
                                                 ]
```

To obtain a black-and-white picture one could set the second and third column of the matrix to zero. The resulting matrices of RGB value can be easily converted to other picture formats. For example, a conversion to 'raw portable pixmap format' is done by the function `ppm`:

```
ppm xss  =  "P6\n# Creator: Gofer JPEG decoder\n"
            ++ w ++ " " ++ h ++ "\n255\n"
            ++ (map (chr.sane) . concat . concat) xss
     where  w = show (length (head xss))
            h = show (length xss)
            sane x = (0 `max` x) `min` 255
```

A main function can be written which reads a source JPEG file and writes a destination PPM file:

```
dst << src  =  readFile src abort
               (\input -> let output = (ppm . yCbCr2rgb . jpegDecode) input
                          in  writeFile dst output abort done
               )
```

This function can be applied to example filenames as follows:

```
main  =  "example.ppm" << "example.jpg"
```

7 Conclusion and Future Work

We have shown that functional programming can be successfully used to implement a complicated algorithm in graphics. We hope that the reader agrees with us that the algorithms needed can be formulated quite concisely and elegantly, and that the borderline between 'specification' and 'implementation' is fading: the correctness of the specification can be demonstrated informally by executing it, and the specification can be used as a prototype implementation of which component functions can be optimized.

The technique of using 'state functions' has been used in the program on three occasions: fetching segments from a character string (section 5.2), fetching Huffman codes and other compressed information from a bit list (section 5.1) and as an auxiliary function for constructing a Huffman tree (section 4.1). The last one shows that 'state' need not be confined to a 'global program state', but that the state function technique can also be used locally.

In the program we have implemented the so-called 'baseline' variant of the JPEG standard, and even from that left out a few less commonly used features, such as 'restart markers'. The program could be easily extended to include these and non-baseline features such as:

- a progressive mode, in which multiple scans per picture frame are allowed, e.g. with increasing resolution or decreasing quantization factors;
- allowing for multiple frames;
- an implementation of 'arithmetical coding' instead of Huffman/DCT coding.

Another interesting project is to write a JPEG *en*coder in the same style. This has been done by my students Arjan van IJzendoorn and Dennis Gruijs. They had as input only a draft of this article, which they inverted function by function. Their program could be extended constructing Huffman tables optimized for a particular picture. Construction of Huffman tables is actually a functional programming classic (see [BiWa88]). The reader may well imagine that collecting the necessary frequency information from the image is just a matter of a few `maps` and `concats`.

Finally, it would be interesting to provide the program of an interactive interface, for which functional programming techniques have become available recently (e.g. in the Clean language [PlEe95]). The very efficient code produced by the Clean compiler could speed up the program dramatically, especially if the `zigzag` and `dct` functions are implemented using arrays instead of lists. We have not done so yet, because the Clean language lacks monad comprehensions, and thus would make the functions less understandable to the human reader.

References

[BiWa88] Richard Bird and Philip Wadler. *Introduction to functional programming.* New York: Prentice Hall, 1988.

[HaVr92] Pieter Hartel and Willem Vree. 'Arrays in a lazy functional language – a case study: the Fast Fourier Transform'. In: *2nd Arrays, functional languages, and parallel systems* (G. Hains, ed.). Montréal: Dept. d'informatique, Université de Montréal (publication 841), 1992, pp. 52–66. also: ftp.fwi.uva.nl/pub/computer-systems/functional/reports/ATABLE92_fft.ps.Z.

[HuFa92] Hudak and Fasel. 'A gentle introduction to Haskell'. *ACM Sigplan Notices* **27**, 5 (may 1992), pp. T1–T53. also: haskell.cs.yale.edu/pub/haskell/tutorial.

[Jone93] Mark P. Jones. 'A system of constructor classes: overloading and implicit higher-order polymorphism'. In: *FPCA '93: Conference on Functional Programming and Computer Architecture.* New York: ACM Press, 1993. also: www.cs.nott.ac.uk/Department/Staff/mpj/fpca93.html.

[Jone94] Mark P. Jones. *Gofer 2.30 release notes.* ftp.cs.nott.ac.uk/nott-fp/languages/gofer.

[Jone95] Mark P. Jones. 'Functional programming with overloading and higher-order polymorphism'. In: *Lecture Notes of the 1st International Spring School on Advanced Functional Programming Techniques* (Johan Jeuring and Erik Meijer, eds). Berlin: Springer, 1995 (LNCS 925).

[ISO93] International Standards Organization. *Digital compression and coding of continuous-tone still images.* Draft International Standard DIS 10918-1. (reprinted in [PeMi93]).

[PeMi93] William Pennebaker and Joan Mitchell. *JPEG still image data compression standard.* New York: Van Nostrand Reinhold, 1993.

[PlEe95] Rinus Plasmeijer and Marko van Eekelen. *Concurrent Clean Language Report.* Nijmegen: Dept. of Computer Science, University of Nijmegen, the Netherlands, 1995 (to appear). ftp.cs.kun.nl/pub/Clean.

[Wadl92] Philip Wadler. 'The Essence of Functional Programming'. In: *19th Annual ACM SIGPLAN Symposium on Principles of Programming Languages.* Santa Fe, 1992 (pp. 1–14). also: ftp.dcs.glasgow.ac.uk/pub/glasgow-fp/papers/essence-of-fp.ps.Z.

[Wall91] Gregory Wallace. 'The JPEG still picture compression standard'. *Communications of the ACM* **34**,4, 1991 (pp. 30–44). also: ftp.uu.net/graphics/jpeg/wallace.ps.gz.