Implementing a video decoder using Feldspar

Master's thesis in Computer science and engineering

Daniel Heurlin
Stanisław Zwierzchowski

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Master's thesis 2020

Implementing a video decoder using Feldspar

Daniel Heurlin
Stanisław Zwierzchowski

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Implementing a video decoder using Feldspar
Daniel Heurlin, Stanisław Zwierzchowski

Supervisor: Thomas Hallgren, Department of Computer Science and Engineering
Examiner: K. V. S. Prasad, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A picture of the Statue of Liberty in New York City, heavily compressed using MPEG-1.

Typeset in LaTeX
Gothenburg, Sweden 2020

Implementing a video decoder using Feldspar
Daniel Heurlin, Stanisław Zwierzchowski
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In this project, we attempt to implement a decoder for the MPEG-1 video standard using Feldspar, an embedded domain specific language in Haskell. The idea is to evaluate whether Feldspar is suitable for more advanced DSP applications and whether it can result in simpler code compared to a lower level language like C while maintaining similar performance. We succeed in implementing a functioning, albeit limited and slow decoder which shows that Feldspar is capable of such a task. We discover that some parts of the implementation are simpler compared to a low level language while others are similar, if not more complicated. We also discuss the pros and cons of Feldspar and suggest improvements that could be made to the language.

Keywords: video, video decoding, codec, DSP, functional programming, Haskell, Feldspar, MPEG-1.

Acknowledgements

We would like to thank our supervisor Thomas Hallgren and our examiner K. V. S. Prasad for their guidance and advice throughout this project. We would also like to thank Emil Axelsson, one of the authors of the original Feldspar paper, for helping us to get started by providing some resources related to Feldspar. Lastly, we would like to thank our friends and family for their support and encouragement during this project.

Daniel Heurlin & Stanisław Zwierzchowski, Gothenburg, June 2020

Contents

1

Introduction

Video compression is essential for modern streaming services and local storage. This is because uncompressed video would require not only an enormous amount of storage, but also very high bandwidth for real time playback. Imagine storing a 1920x1080 video with 30 frames per second where each pixel is represented by three 8-bit RGB values (for red, green and blue colors). This would require $8 \cdot 3 \cdot 1920 \cdot 1080 \cdot 30 \approx 1.5$Gb/s or about 190MB/s. A two hour long movie with these specifications would occupy about 1.3TB of disk space. This would most likely occupy a significant portion of a typical computer hard drive, if it would fit at all. Modern optical media such as Blu-ray discs would also be insufficient to store such a movie, with a four layer Blu-ray disc fitting only about 100GB per side [1]. The bandwidth of 1.5Gb/s also greatly exceeds a typical home internet connection which would make streaming of such a movie impossible. Video is therefore always encoded into a compressed format, distributed and decoded during playback.

Because of the high compression ratio required, video encoding and decoding with modern formats is a computationally intensive process. It is therefore often implemented in hardware with the disadvantage of not being modifiable at a later point. A more flexible approach is to use a low-level language like C to implement codecs (video encoders and decoders) in software. This comes with the disadvantage that the complex mathematical computations required are difficult to express because of the imperative low-level programming style which hides the high-level logic. This makes a C implementation time consuming, expensive and prone to bugs. It would be easier using a more declarative language which is closer to the mathematical notation.

Feldspar [2] is an embedded domain specific language (DSL) in Haskell, designed for performing digital signal processing (DSP) computations on a high level while generating C code. We propose to evaluate the extent to which Feldspar is an appropriate tool for the above mentioned task by implementing a video decoder compliant with a video coding standard. By doing so, we want to evaluate whether Feldspar is appropriate for more advanced DSP applications, perhaps serving as a superior alternative to C implementations.

1.1 Purpose and Delimitations

The purpose of this project is to evaluate how well the Feldspar language is suited for nontrivial signal processing applications, in our case video decoding. To do this we will attempt to implement a video decoder compliant to the MPEG-1 standard [3] using Feldspar. Based on our experiences from this process, we will discuss to what extent Feldspar results in simpler code than a C implementation, as well as how it compares in terms of performance.

Since video decoding is is a complex process, and the time frame for this project is limited, we will not attempt to implement every aspect of the standard. For instance, the decoder will be "offline" only, i.e. it will output a series of decoded frames as raw image files rather than outputting a decoded stream in real time. It will also not attempt to handle errors in the input video stream, instead exiting if one were to be encountered. In spite of these delimitations, the decoder will still implement the most essential parts of the standard, which should be more than enough to answer the question at hand.

2

Background

2.1 MPEG-1

MPEG-1 is a video file format standardized in 1993 with the purpose of achieving a high compression ratio while maintaining good picture quality on digital storage media such as compact discs [3]. While the MPEG-1 standard also describes an audio compression algorithm and a "container" file format for containing both audio and video streams, this project only focuses on the video compression algorithm.

The decision to use MPEG-1 as the standard for this project was based on a few criteria: The standard should be complex enough to pose a challenge and to prove that Feldspar can be used for practical applications. At the same time it shouldn't be so complex that it would be unrealistic to implement within our time span. We believe the MPEG-1 standard to be a good choice given these criteria; although nowadays obsolete, it consists of the same basic building blocks as modern standards. It should therefore serve as a good model for modern standards.

2.1.1 Compression methods

The MPEG-1 video compression algorithm is lossy, meaning that the exact original data can't be reconstructed from the compressed data. Rather, a "good enough" picture can be reconstructed, ideally not distinguishable from the original data by the human eye. This enables a higher compression ratio comparing to a lossless compression algorithm as not all data has to be stored.

The following subsections introduce various compression methods used by MPEG-1, some of which make it lossy.

2.1.1.1 Variable length coding

Variable length coding (VLC) is a lossless compression method based on assigning shorter code words to more frequent values and longer code words for less frequent values. MPEG-1 uses VLC for storing a variety of data. Such data can be decoded using predefined tables that map a bit sequence of variable length to one or multiple corresponding values. Such tables can be generated by statistical analysis of the values to be represented, using a method such as Huffman coding [4].

Table 2.1: Part of the VLC table for `dct_coeff_next`

VLC	run	level
10	`end_of_block`	
1 s	0	1
011 s	1	1
0100 s	0	2
0101 s	2	1
00101 s	0	3
00111 s	3	1
00110 s	4	1
000110 s	1	2
⋮	⋮	⋮
0000000000011011 s	31	1

Table 2.1 shows an example VLC table from the MPEG-1 standard where each binary VLC code is mapped to two integer values: `run` and `level`, except for "`10`" which marks the end of a block (and signifies the end of a string of VLC codes). The "`s`"-bit at the end of the VLC codes denotes the sign of the `level` value: `0` means positive and `1` means negative.

2.1.1.2 Run-length encoding

Run-length encoding is a lossless compression method based on describing recurring data values by a single value and the number of its recurrences (the run-length). An example with ASCII characters could be `AAAAABAAAA` encoded as `A5BA4`, where the encoded version, although shorter, can be decoded back to the original string.

2.1.1.3 Chroma subsampling

MPEG-1 stores pixels in the *YCbCr* color space as opposed to the *RGB* (red, green, blue) color space. The *YCbCr* color space consists of the luminance component *Y*, denoting brightness of a pixel and two chrominance components *Cb* and *Cr*, denoting color independently of the brightness [5]. The *Cb* component denotes the difference between the intensity of blue color and the luminance while the *Cr* component denotes the difference between the intensity of red color and the luminance. A visualization of the *YCbCr* color space is shown in Figure 2.1. This method of storing pixels takes the human vision system into account which is more sensitive to details in the luminance component than the chrominance components. Using *YCbCr* therefore allows for chroma subsampling – storing chrominance at a lower resolution than luminance. This technique lowers the amount of data needed to be stored for each pixel with minimal loss in the perceived quality of the image. MPEG-1 uses *4:2:0* chroma subsampling which means that chrominance is stored at half vertical and horizontal resolution compared to the luminance.

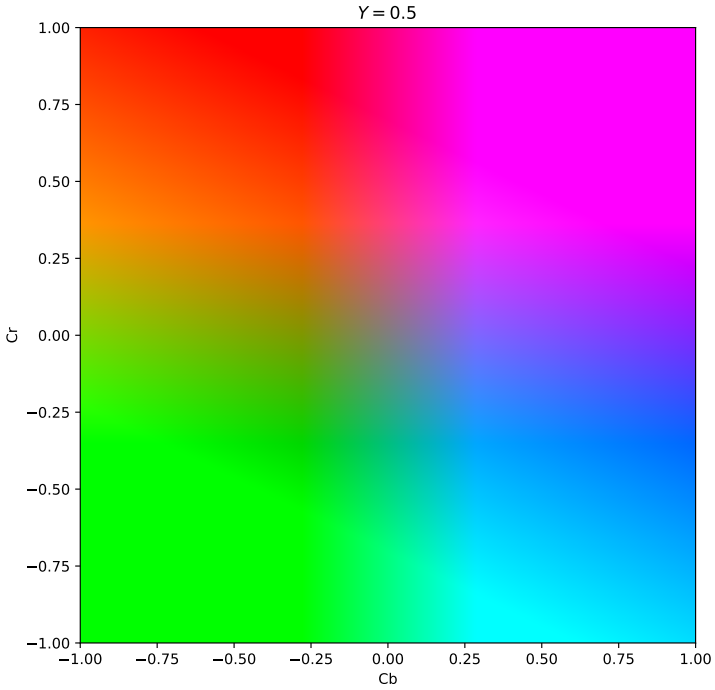

Figure 2.1: An approximate chart of the $YCbCr$ color space with constant $Y = 0.5$.

2.1.1.4 Discrete cosine transform and quantization

The discrete cosine transform (DCT) is a representation of a discrete signal as a sum of cosine functions at different frequencies, representing the frequency components of the signal [6]. Its ability to separate a signal into separate frequency components is heavily utilized in most image, video and audio compression formats, including MPEG-1. This is due to the fact that high frequency components typically have a low impact on perceived video quality and can therefore be discarded – reducing data size.

Discarding or reducing the importance of different frequency components is done using quantization. This involves division of each frequency component by a certain quantization value and rounding the result down to an integer. The frequency components can later be reconstructed by multiplication with the same value but the result will have a loss in precision. High frequency components that have a small impact on the perceived video quality will often be divided by a large number and therefore rounded down to 0.

2.1.1.5 Motion vectors

It is common in video that a frame is similar to the frames preceding and succeeding it. For example, in a video of a person speaking in front of a static background with the camera on a tripod, the background will be identical for each frame with only the person moving. In such a case, it is redundant to store the entire background for each frame. Instead, it would be desirable to store it only once and recycle that

information for every frame that uses it. In a similar video but with the camera slowly panning, a large part of the background would still be identical – only in a different position. In this case, it would also be desirable to recycle this information but with the ability to describe movement of parts of the picture, from one frame to the next.

In MPEG-1, this is accomplished using *motion vectors* which refer to a relative coordinate in a reference frame. During decoding, the pixels at that coordinate can be copied to the current coordinates in the current frame. Additionally, motion vectors may be paired with DCT coefficients describing a so called *prediction error* – the difference between the copied data and the current data to be reconstructed. This is all done to reduce the amount of data needed to be stored for each frame.

2.1.2 Specifications

The standard divides the video stream into the following layers:

- **Sequence layer:** The start of a video sequence. It must contain at least one sequence header, specifying values such as resolution and bit rate.
- **Group of pictures (GOP) layer:** Group of one or more pictures encoded independently from other GOPs.
- **Picture layer:** A picture frame which after decoding results in three matrices: one for luminance (brightness) and two for chrominance (color) values.
- **Slice layer:** A non-overlapping slice of a picture containing one or more macroblocks.
- **Macroblock:** A 16x16 pixel rectangle of a picture. May contain up to six blocks: four luminance and two chrominance blocks.
- **Block:** The lowest layer of a video stream with data needed to reconstruct 8x8 pixels of luminance or chrominance data.

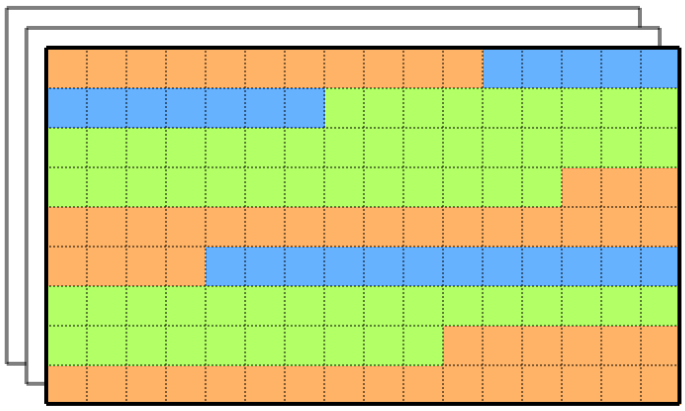

Figure 2.2: A visual representation of the division of a picture. The colors represent slices and each small square is a macroblock.

2.1.2.1 Pictures

As previously mentioned, in order to achieve a high compression radio, MPEG-1 utilizes temporal compression between pictures. This means that parts of pictures may contain references (motion vectors) to previous and/or future pictures and store only the difference between them. The standard defines the following types of pictures, differentiating how they use temporal compression.

- **Intra-coded pictures** (I-pictures) are self-contained and are only compressed spatially within themselves.
- **Predictive-coded pictures** (P-pictures) also contain temporal compression by including motion vectors pointing to previous I- or P-pictures.
- **Bidirectionally predictive-coded pictures** (B-pictures) contain temporal compression by including motion vectors pointing to both future and previous I- or P- pictures. However, no other pictures may contain references to B-pictures.
- **DC-coded pictures** (D-pictures) are, like I-pictures, encoded independently but contain only DC coefficients from the discrete cosine transform. This makes them low quality but fast to decode which enables fast forward playback with preview. These will be omitted in this report because of their rare usage.

2.1.2.2 Slices

Pictures are divided into slices, containing one or more macroblocks. They are mostly used to mitigate the results of data corruption. If a part of a slice gets corrupted or unreadable, it may be possible for the decoder to decode the next slice, only discarding the corrupted slice as opposed to the entire picture. An example of how a picture may be divided into slices is shown in Figure 2.2.

2.1.2.3 Macroblocks

A macroblock represents a 16x16 pixels rectangle of a picture. It may consist of up to six 8x8 pixel blocks: four luminance blocks (making the luminance component full 16x16 resolution) and two chrominance blocks (one for each chrominance component, making them half the resolution of the luminance component).

Depending on the picture type, a macroblock may contain different sets of data. For example, macroblocks in P- and B-picture may have the following features:

- They may contain motion vectors pointing to a past or future frame.
- They may be intra-coded meaning that they are only compressed spatially within themselves.
- They may be skipped entirely in which case they are copied from a previous frame.

- Some of their blocks may be skipped, in which case the skipped blocks are copied from the previous frame.

2.1.2.4 Blocks

A block contains variable length encoded run-length values for quantized DCT coefficients describing 8x8 pixels of the final image. For chrominance blocks, these are stretched to an area of 16x16 pixels during playback due to chroma subsampling. The 8x8 pixel values can be reconstructed from a block by dequantizing and performing an inverse DCT on these coefficients.

Dequantization is performed by multiplying the DCT coefficient matrix with a quantizer matrix element-wise. The quantizer matrix may be specified by the encoder in the sequence header, otherwise a standard quantizer matrix defined by the MPEG-1 standard is used.

The coefficients are stored in a zig-zag pattern beginning at the lowest frequency and moving towards the highest, as seen in Figure 2.3. As previously mentioned, high frequency components (towards the right bottom of Figure 2.3) are often rounded to 0. This means that the sequence can be terminated with a specific VLC code and the rest of the block values are assumed to be 0 – therefore saving space.

If the macroblock in which a block is located specifies a motion vector, the block is first copied from the coordinates pointed at by the motion vector in the reference picture. The 8x8 pixel values reconstructed by the above described process are then added to the copied data.

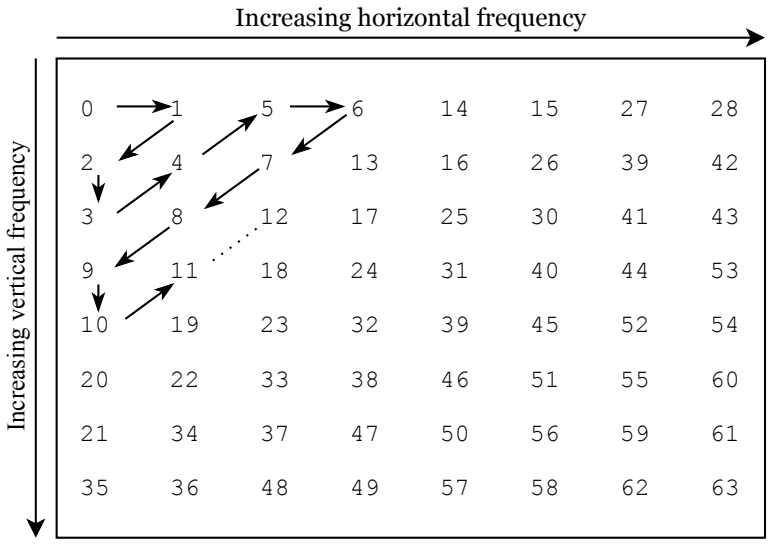

Figure 2.3: Zig-zag scan pattern. The integer values represent the scanning order.

2.2 Feldspar

Feldspar is an embedded domain specific language (EDSL), written in Haskell. Its purpose is to enable high-level functional programming of digital signal processing algorithms while generating C code. There are currently two implementations of Feldspar: the original implementation as described in the Feldspar paper [2] and "raw-feldspar" [7] – a new implementation currently recommended by the developers and therefore used in this project. The new implementation is based around the idea of combining both deep and shallow embeddings of EDSLs [8]. Feldspar therefore consists of a shallow programming interface which is translated into a small, deeply embedded core language, semantically similar to C. This approach gives the programmer an expressive and extensible programming interface which can translate to efficient core language while keeping the core language small.

2.2.1 Introduction

Similarly to Haskell with its `IO` monad [9] which represents a computation that may have side effects [10], Feldspar programs are based around the `Run` monad which provides basic I/O functionality and an interface for interacting directly with C, such as including C-libraries and calling C-functions. For a more thorough explanation of monads and the `IO` monad in particular, we recommend reading [9] and [10].

A "Hello World" program in Feldspar looks like this:

```
main :: Run ()
main = printf "Hello world!\n"
```

This program can be evaluated directly in Haskell by calling `runIO main` or compiled to C and then executed by calling `runCompiled main`. To output the generated C code only, one can call `icompile main` for which the result is shown in Listing 1. All three functions mentioned above are a part of Feldspar.

```
#include <stdio.h>
int main()
{
    fprintf(stdout, "Hello world!\n");
    return 0;
}
```

Listing 1: A "Hello World" example program generated with Feldspar.

2.2.2 Vectors

Feldspar contains its own vector library built on top of its array implementation. Although one can use arrays directly, there are two main advantages of using the vector library instead: code clarity and efficiency.

The vector library works similarly to Haskell's lists in the sense that it contains common list manipulation functions such as `map` and `fold`. By using these functions on vectors (instead of loops and indexing into individual array items), one can achieve higher code clarity.

Additionally, the vector library utilizes fusion [11, 8] which means that vector functions don't result in unnecessary intermediate arrays in memory. Consider the following function taken from Feldspar's examples:

```
sumSq :: Data Word32 -> Data Word32
sumSq n = sum $ map (\x -> x*x) (1...n)
```

The function's argument and result type is a Feldspar data type for representing 32-bit unsigned integers. The function essentially consists of two vectors. The first is constructed using the binary operator `(...)` which works similarly to Haskell's `(..)` operator, in this case constructing a vector with values from `1` to `n`. The second vector is constructed upon using the `map` functions which consumes the first vector and returns a vector of squared values. This vector is however also consumed when applied to the `sum` function.

Let's see what C code Feldspar generates for `sumSq`. Since `sumSq` is a pure function, we must first define a runnable function of type `Run ()` which prints the result. Although we could supply `sumSq` with a constant argument `n`, this would cause Feldspar to pre-compute the entire program which would result in a single `fprintf` statement printing the result, in the generated code. To avoid this, we will read the argument from `stdin`, run the function and then print the result.

```
runSumSq :: Run ()
runSumSq = fget stdin >>= printf "%d" . sumSq
```

By running `icompile runSumSq` we can see the code shown in Listing 2. Although fairly unreadable, we can see that Feldspar didn't generate a single C array.

```c
#include <stdint.h>
#include <stdio.h>
int main() {
    uint32_t v0;
    uint32_t state1;
    uint32_t v2;
    fscanf(stdin, "%u", &v0);
    state1 = 0;
    for (v2 = 0; v2 < (uint32_t) (1 < v0 + 1) * v0; v2++) {
        uint32_t let3;

        let3 = v2 + 1;
        state1 = let3 * let3 + state1;
    }
    fprintf(stdout, "%d", state1)
    return 0;
}
```

Listing 2: The generated C code from the `sumSq` function.

Feldspar provides (among others) the following vector types.

- **Manifest** vectors are equivalent to immutable arrays – a part of the Feldspar core language – and translate directly into C arrays.
- **Pull** vectors have no runtime representation, they are implemented abstractly as a function from an index to an element. They support many of the list operations one would expect to see in functional programming, such as `map` and `fold`.
- **Push** vectors, like Pull vectors, have no runtime representation. They are also represented by a function, but this time the function is from an index and the element at that index to a monadic computation. The fact that the function is monadic means that it can be used to write to an array, i.e. storing the values of these abstract vector types to a physical memory location. One useful feature of Push vectors is that they can be concatenated.

When working with vectors, it is common to start with a Pull vector and perform the necessary operations that it supports. The Pull vector can then be converted to a Push vector when operations such as concatenation or monadic actions are needed. Finally, the Push vector can be written to memory – turning it into a Manifest vector. The fact that all computations are performed on Pull and Push vectors means that no intermediate arrays have to be allocated since these are essentially just functions, meaning the final value that gets written to memory is basically computed by evaluating a function composition.

3

Implementation

The implementation was accomplished by strictly following the MPEG-1 standard which describes each step required to read and decode a video stream. A significant part of this process consists of sequentially reading the different layers of a video stream in chunks of data as small as individual bits. The standard describes how each layer is structured using a so called *bitstream syntax*, describing what data each layer consists of using a C-like pseudocode. An example of such syntax for the slice layer can be seen in Listing 3. Lines marked with "No. of bits" represent variables of the specified number of bits that need to be read. How this example is read using Feldspar is described later in this chapter, after a few necessary mechanisms have been introduced.

```
slice() {                                                  No. of bits
    slice_start_code                                           32
    quantizer_scale                                             5
    while (nextbits() == '1') {
        extra_bit_slice                                         1
        extra_information_slice                                 8
    }
    extra_bit_slice                                             1

    do {
        macroblock()
    } while (nextbits() != "000 0000 0000 0000 0000 0000")

    next_start_code()

}
```

Listing 3: Bitstream syntax for the slice layer from the MPEG-1 standard.

Each variable in the bitstream syntax is also described in a separate section of the standard. Some are simply constants used for marking different parts of the video stream such as a start code for a layer, while others are needed later in the decoding process and thus have to be saved in some global state.

This chapter highlights the most noteworthy parts of the implementation, describing what challenges they posed and how these were solved.

3.1 Maintaining a state

Early in the implementation process, it became clear that maintaining a global state would be necessary. Since Feldspar doesn't provide any mechanism to handle this, we decided to use the `ReaderT` monad transformer from Haskell's `Control.Monad` module. The idea was that the `ReaderT` monad transformer would encapsulate Feldspar's `Run` monad, allowing us to utilize all of Run's functions such as while-loops and reading files.

Using `ReaderT` instead of `StateT` may seem counterintuitive since `ReaderT` only supports an immutable environment, but attempting to use `StateT` resulted in some technical difficulties. Fortunately, we found that `ReaderT` was sufficient due to Feldspar's support for immutable references to mutable variables; instead of mutating the values stored in the state, we could store references to those values in the environment and modify them using the `setRef` and `modifyRef` functions.

In order for the `ReaderT` monad transformer to be used with the `Run` monad, we needed to implement an instance of Feldspar's `MonadComp` [12] for `ReaderT`. `MonadComp` is a monad which supports mutable data structures and control flow functions. This involved implementing the functions `liftComp`, `iff`, `for` and `while`, describing how the environment should be handled in those functions.

```
instance (MonadComp m) ⇒ MonadComp (ReaderT r m) where
  liftComp = lift . liftComp
  iff c t f = do
    env ← ask
    lift $ iff c (runReaderT t env) (runReaderT f env)
  for rng body = do
    env ← ask
    lift $ for rng $ \n → runReaderT (body n) env
  while c body = do
    env ← ask
    lift $ while (runReaderT c env) (runReaderT body env)
```

Most of the functions in our decoder implementation are of type `Decoder`, a `ReaderT` monad transformer with an environment record `DecoderEnv` and Feldspar's `Run` monad:

```
type Decoder = ReaderT DecoderEnv Run
```

3.2 Reading binary files

Although Feldspar has built in functions to open and read files (`fopen`, `fget`, ...), `fget` assumes that the file opened by `fopen` is a text file and returns a character rather than an individual byte. For this reason, we had to define our own function `readData` which reads a given number of bytes from a file by calling the C-function `fread`:

```
readData :: (PrimType' a, Type a)
         => FileHandle
         -> (Data Int8)
         -> Run (Data a)
readData (FH handle b w) typesize = do
  ref :: Ref (Data a) <- newNamedRef "read_target"
  callProc "fread"
    [ refArg ref
    , valArg typesize
    , valArg (1 :: Data Length)
    , objArg handle
    ]
  setRef b 0
  getRef ref
```

The line `setRef b 0` is used in the mechanism we use to read individual bits from the file, which we describe later in this section.

Using `readData`, we could write helper functions, such as for reading a 32-bit word:

```
readWord32 :: Decoder (Data Word32)
readWord32 = do
  fh@(FH _ b w) <- asks fileHandle
  word <- liftRun $ readData fh 4 >>= bswap
  setRef w word
  setRef b 0
  pure word
```

Note that the results from `readData` are piped to the `bswap` function. This is because MPEG-1 uses big-endian byte order while x86-processors (which this project is developed for) use little-endian byte order. By swapping the byte order upon reading from a file, we ensure that we can follow the MPEG-1 standard directly since it's also written using big-endian byte order.

While swapping the byte order could be achieved with a pure Feldspar function, we opted for the `bswap` C-function which can compile to a single instruction on x86-processors. The following code snippet shows how `bswap` can be called in Feldspar:

```
bswap :: Data Word32 ->  Run (Data Word32)
bswap w = do
  addInclude "<byteswap.h>"
  callFun "bswap_32" [ valArg w ]
```

3.2.1 Reading individual bits

The MPEG-1 standard requires us to read individual bits. Since the smallest unit that can be directly read using `fread` is a byte, we implemented functions allowing us to read single bits.

Our state contains of a `FileHandle`, consisting of a reference to a C file handle, the index of the current bit to be read and the last 32-bit word that has been read.

```
data FileHandle = FH
{
    fhl         :: Object             -- The file handle (FILE* in C)
  , currentBit  :: Ref (Data Int32)   -- Reference to the current bit
  , currentWord :: Ref (Data Word32)  -- Reference to the last read word
}
```

With the `FileHandle` data type in our state, we could write the following function for reading a single bit. It works by keeping track of the last word that was read using `readWord32` (`currentWord`) and the last bit that was extracted from that word (`currentBit`). When the next bit is requested, it increments the `currentBit` counter and extracts that bit from `currentWord`, and then increments the `currentBit` counter. If `currentBit` exceeds 31, meaning every bit of `currentWord` has been read, it reads a new word and resets the bit counter to zero.

```
nextBit :: Decoder (Data Word32)
nextBit = do
  fh@(FH _ rbit rword) <- asks fileHandle
  bit <- getRef rbit
  if1 (bit > 31) (void readWord32)
  word <- getRef rword
  modifyRef rbit (+1)
  pure $ testBit word (31 - bit)
```

3.3 Decoding variable length codes

Some of the values specified in the MPEG-1 bitstream syntax are variable length codes (VLC), as described in section 2.1.1.1. In order to decode VLC values, the following points were taken into account:

- **The ease of storing the VLC tables:** Ideally, we wanted to store the VLC in the same format as they are presented in the MPEG-1 standard, with binary codes and their corresponding integer values written in plain text.
- **The method of decoding the VLC values:** This includes not only the computational complexity of looking up which binary code corresponds to which values but also the fact that our file reading library allows us to read a single bit at a time. Ideally, we wanted to only read as many bits as necessary to decode a certain VLC.

With this in mind, the following method was implemented. The VLC tables are stored in plain text files with one row per VLC. Each row begins with the binary code followed by the corresponding integer values, separated by a whitespace character. These files are parsed by a Haskell program that generates a Haskell source file, containing a binary VLC tree. The generated VLC trees are of the following data type:

```haskell
data VLC a = Node (VLC a) (VLC a)
           | Leaf a
           | End
```

Each node consists of two subtrees, the first being evaluated if the next read bit is `0` and the second if it's `1`. Each subtree may be another node, a leaf containing the value which the binary code corresponds to or an `End` node which signifies that no value exists for the binary code. Reaching an `End` node during decoding means that the binary code is invalid.

This approach allowed us to implement the following recursive decoding function, taking a VLC tree as an argument and returning the decoded value. Upon encountering a node, a single bit is read using the `nextBit` function and the corresponding subtree is traversed.

```haskell
getVLC :: (Syntax a) => VLC a -> Decoder a
getVLC End        = exit "getVLC reached `End`"
getVLC (Leaf v)   = return v
getVLC (Node i o) = do
  rbit <- nextBit
  ifE (rbit == 0) (getVLC o) (getVLC i)
```

Listing 4: Recursive function for decoding VLC values.

3.4 Reading the bitstream syntax

With the above mechanisms in place, it is possible to read every part of the video stream as described by the bitstream syntax. The slice layer, whose bitstream syntax can be seen in Listing 3 can be read as shown in Listing 5. The `loop` function is an infinite loop which repeats until the `break` function is reached, equivalent to `while (pure true)`.

```
readSlice :: Decoder ()
readSlice = do
    slice_start_code <- nextBits 32
    quantizer_scale  <- nextBits 5

    while ((==1) <$> (peekBits 1)) $
        void $ nextBits 9

    extra_bit_slice <- nextBit

    loop do
        readMBlock
        peek <- peekBits 23
        if1 (peek == 0) (break)

    nextStartCode
```

Listing 5: Simplified version of our slice parsing function.

3.5 Inverse Discrete Cosine Transform

As described in 2.1, blocks contain DCT (discrete cosine transform) coefficients. Once these have been decoded, an *inverse discrete cosine transform* (IDCT) must be performed in order to transform these into 8x8 pixel values.

The MPEG-1 standard [3] dictates that this IDCT conform to IEEE Draft Standard, P1180/D2, July 18, 1990 [13], which gives the following definition of the IDCT:

$$x(i,j) = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)X(u,v)\cos\left(\frac{(2i+1)u\pi}{16}\right)\cos\left(\frac{(2j+1)v\pi}{16}\right) \tag{3.1}$$

where

$$x(i,j),\ i,j \in \{0,...,7\}$$

is the pixel value at coordinate i,j,

$$X(u,v),\ u,v \in \{0,...,7\}$$

is the transformed coefficient at coordinate u, v within the block, and

$$C(0) = \frac{1}{\sqrt{2}}, \text{ and } C(x) = 1,\ x \in \{1, ..., 7\}.$$

Equation 3.1 can be rewritten as a matrix equation as $\mathbf{S} = \mathbf{AFA}^T$, where the matrix $\mathbf{S}_{ij} = x(i, j)$, $\mathbf{A}_{ux} = \eta(u)\cos((2x+1)\pi u/16)$, $\mathbf{F}_{uv} = X(u, v)$ and $\eta(u) = C(u)/2$ [14]. A direct translation of this definition can be implemented in Feldspar as shown in Listing 6.

```
idct :: DPull2 Int32 → DPull2 Int32
idct v = map2 round $ a `matMul` f `matMul` (transpose a)
  where
    f   = map2 i2n v :: DPull2 Double
    η u = cond (u == 0) (1 / sqrt(8)) (1/2) :: Data Double
    a   = Pull2 8 8 $ \x u → η u * cos ((i2n $ (2*x+1) * u) * π/16)
```

Listing 6: Inverse discrete cosine transform in Feldspar.

The type `DPull2 Int32` represents a matrix of 32-bit integers using the `Pull` vector type described in Section 2.2.2. The argument `v` of the function is a matrix containing each $X(u, v)$ as described above, and the returned value is a matrix containing each $x(i, j)$ as described above. The use of `round` and `i2n` is to convert back and forth between floating and integer types.

Also note how we perform rounding after computing the result. This is not a part of [13] but is demanded by the MPEG-1 standard [3].

3.6 Testing

Once the decoder was able to produce an image output, we first inspected it visually to confirm that no major graphical distortions were visible compared to the original file. We also had access to the MPEG-1 reference video decoder [15] which by definition produces a correctly decoded file. The reference decoder was later used to programatically compare the output with our decoder. Both methods of comparison are suggested by the MPEG-1 compliance testing document [16].

3.6.1 Creating test files

A typical MPEG-1 file is a "container" file containing a video and audio stream. The video stream may contain any of the supported picture types described in Section 2.1.2.1. As input, our decoder needs a raw MPEG-1 video stream file containing only I and P-pictures. Such test files were created using ffmpeg [17] with the following command:

```
$ ffmpeg -i INPUT -f mpeg1video -an OUTPUT
```

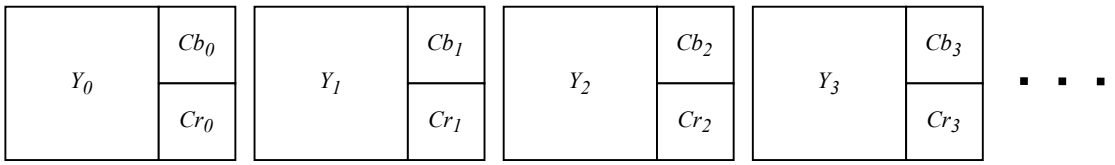

Figure 3.1: A visual representation of the YV12 video format where each group of blocks represents a frame.

The input file `INPUT` can be any video file supported by ffmpeg. The `-f mpeg1video` argument signifies that the output should be a raw MPEG-1 video stream. `-an` means that no audio track should be present in the output. The resolution and frame rate of the output will be the same as the input (unless otherwise specified) and the output bitrate is automatically selected by ffmpeg. By default, ffmpeg only outputs I- and P-pictures which are also the only picture types supported by our decoder. During initial testing, when the decoder only supported I-pictures, the `-force_key_frames expr:1` arguments was used, forcing ffmpeg to only generate I-pictures.

3.6.2 Output file format

When decoding a video file, our decoder decodes the luminance and chrominance blocks as described in Section 2.1.2.4. The luminance (Y) and chrominance (Cb and Cr) blocks are each stored in a corresponding 2D array at the appropriate location. The three arrays are later written to a file in the YV12 format [18]. This format was chosen because of its simplicity and since it seems to be a common way of storing raw $YCbCr$ data. This is also a format that the reference MPEG-1 video decoder can ouput. A YV12 file simply consists of the luminance component followed by the two chrominance components stored consecutively. Since MPEG-1 uses *4:2:0* chroma subsampling, the two chrominance components are stored at half the resolution compared to the luminance component. This format allows storing multiple frames in a single file by repeating the above pattern for each frame in sequence as shown in Figure 3.1.

3.6.3 Displaying the output files

Since the YV12 format doesn't contain any header information about its contents, one needs to specify the resolution and frame rate (if the file contains multiple frames) for correct playback. One tool we used to display our output files is ffplay (a part of FFmpeg [17]), which we used in the following way:

```
$ ffplay -r FRAME_RATE -s WIDTHxHEIGHT INPUT.yuv
```

This method can also be used for displaying the output of the MPEG-1 reference decoder.

3.6.4 Comparing output files programatically

Since our decoder generates output in the same file format as the reference decoder, we expect a decoded frame to be identical byte-for-byte between the two. In order to programatically compare the output from our decoder with the output from the reference decoder, the `cmp` Unix program was used as follows:

```
$ cmp -l ref.yuv comp.yuv
```

Where `ref.yuv` and `comp.yuv` is the output from the reference decoder and our decoder respectively, and the `-l` argument sets the program to output the address of each byte that differs along with the the two different values.

3.6.5 Performance measurement

The performance of our decoder was compared to the reference decoder by measuring the time it took for both decoders to decode the same MPEG-1 video file, generated according to Section 3.6.1.

4

Results

This project has resulted in a functioning, albeit limited, MPEG-1 video decoder written in Feldspar. The source code is freely available at `https://github.com/mightynerd/feldspar-mpeg1-decoder`. The decoder should be seen as a proof of concept rather than a usable decoder. The main reasons are the following:

- The decoder only supports decoding of I-frames and P-frames. This means that an encoder used to produce the test files needs to only encode I- and P-frames and therefore doesn't utilize the full compression potential of the format.
- The decoder works offline only, meaning that it does not support real time decoding. This means that it can only be used for outputting decoded files and not for playing back the video in real time while decoding.
- Although the output looks visually identical, the programmatic comparison between the output of our and the reference decoder (described in Section 3.6) showed that our decoder does not produce identical output compared to the reference decoder.
- Our decoder is severely lacking in performance and is many times slower than the reference C-implementation [15]. This is described in more detail in Section 4.1.

During the course of developing the decoder, we have come to the following conclusions regarding the Feldspar language:

- Based on our decoder implementation, we believe we have shown that Feldspar is capable of more advanced DSP applications. The fact that our decoder is incomplete is not due to limitations of the language but rather our time constraint.
- We can however not confirm our hypothesis that implementing a video decoder in Feldspar would be easier compared to a low level language. We found that while some parts were in fact easier to implement in Feldspar, most parts were comparable, if not more complex than using a low level language. Some of these issues could be eliminated by extending the Feldspar language, as discussed in section 5.1.

- The areas where Feldspar turned out to be easier were those that could be expressed as pure functions. This turned out to be more rare than anticipated, as much of the code had to be written in a more imperative style. Components that could be expressed in a functional manner include the IDCT implementation as well as traversing a binary tree for decoding VLCs.
- Another advantage of Feldspar is that being an EDSL, it can be extended by using the rich functionality of its host language. This could be utilized to define custom control structures, which would not be possible in a standalone DSL or an existing imperative language.

4.1 Performance

The performance of our decoder is quite poor and not comparable with the reference C-implementation [15]. Decoding the same 125-frame video at 640x360 resolution takes our decoder around nine times longer than the reference decoder, as shown in Table 4.1.

Since we suspected the IDCT might be a bottleneck, we tried replacing our IDCT function with one that returns a constant value of zero. Doing so results in a speedup by a factor of about 50, confirming that the IDCT is indeed the most time consuming part of the code.

Reference Decoder	10.3s
Feldspar Decoder	88.7s

Table 4.1: The time required to decode a 125 frame 640x360 video by the reference decoder and our decoder respectively.

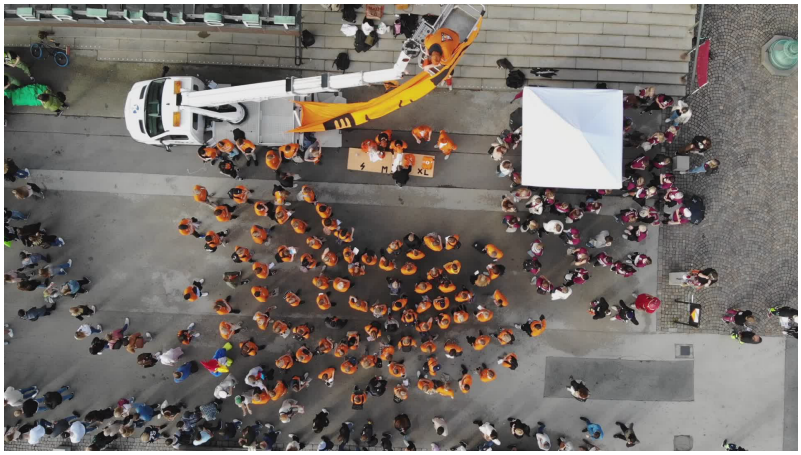

Figure 4.1: An example frame decoded by our decoder.

5

Discussion

5.1 Feldspar

As described above, we have succeeded in implementing a partially functioning MPEG-1 decoder using Feldspar. We believe this confirms that the language is indeed capable of more advanced DSP applications, at least in principle. However, we cannot confirm the hypothesis that using Feldspar would result in simpler code compared to lower level languages. Below, we discuss some of the issues we have encountered and possible solutions to them. If we were able to resolve these issues, we believe we might come closer to confirming the aforementioned hypothesis.

In Chapter 3, we showed how the bitstream syntax for the slice layer (Listing 3) is read using Feldpar (Listing 5). As clearly visible, the Feldpar function is fairly similar in structure to the bitstream syntax (albeit with a different syntax), since we are unable to express this on a higher level in Feldspar. For example, the Feldspar function uses the C-like `while`-loop (see line 6) as in the bitstream syntax. This pattern of writing code which is essentially a facsimile of imperative code occurs frequently in our implementation. There are even cases where the Feldspar implementation could be considered less clear, such as the boolean condition in the aforementioned `while` loop. Since the `peekBits` function is monadic, we have to use the somewhat cluttered functor syntax instead of the more straightforward expression seen in the C-like code.

We believe that the issue above could be partially resolved by splitting up the code into two parts: one for reading the bitstream syntax into data structures corresponding to the different layers of the video stream, and another for decoding those data structures. The reading part would place all values it read into the data structure instead of writing them to the global state to be used deeper down in the layer hierarchy, as is currently done. For instance, the slice slice layer could be read according to the pseudo code in Listing 7.

This looks promising at a first glance, but this method would introduce a number of difficulties: Firstly, the bitstream syntax is written in an imperative, C-like style, where decoding is done "on the fly" while reading the video stream. The decoding operations are also expressed in this imperative style. Converting this style to the proposed one would therefore require a quite profound understanding of the standard

```haskell
data Slice = Slice { vPos    :: Data Word32
                   , qScale  :: Data Word32
                   , mBlocks :: DPull Word32
                   }

readSlice :: Decoder Slice
readSlice = Slice <$> (getVPos <$> nextBits 32)
                  <*> (nextBits 5)
                  <*> (skipExtra >> readMBlocks)
    where
        getVPos = splitBits [(0,8)]
        skipExtra = ...
```

Listing 7: How reading the slice layer might look in a more functional style.

which we lacked when we started implementing the code, and probably still do. Following the standard directly was therefore the only realistic option. Had the standard expressed the decoding operations in a mathematical notation rather than in pseudo C-code it would likely have been easier to express them functionally in Feldspar. Secondly, there are many cases where certain values have to be decoded or evaluated to determine how to proceed reading. This would require the use of `iff` and `while` similarly to the current implementation, thus most likely not making much difference. Finally, since certain layers such as blocks and macroblocks come in multiple variants, it would probably be necessary to represent these using Haskell sum types, e.g.

```haskell
data Macroblock = IntraMBlock [Block] ...
                | ForwardMBlock MotionVector [Block] ...
                | ...
```

As will be discussed below, most Feldspar operations do not support sum types, and functions defined by pattern matching are not supported by Feldspar. This means that even if it were possible to read the video stream into data structures, it would be impractical, if not impossible, to work with these during the decoding stage.

It is important to remember that this is an issue regarding reading binary files, MPEG-1 in particular. This might not be an issue with other binary formats and certainly doesn't affect Feldspar's DSP capability.

There are also other issues related directly to Feldspar. Although their main cause might be the fact that Feldspar is still under development and its documentation clearly states that some features are missing, they are still worth mentioning since they were a significant part of this project. One such issue is the lack of essential functions for reading files (described in Section 3.2) and maintaining a global state (described in Section 3.1). We also encountered a lack of functions related to two-

dimensional arrays and vectors which we had to implement ourselves. This was time consuming, largely because of the lack of more advanced tutorials and the rather sparse documentation.

There are however some limitations of Feldspar itself that can not be resolved without modifying the language. One such limitation is the lack of support for Haskell sum types, i.e. types of the form `data D = A | B | C | ...`. We found that this forced us to use a more imperative programming style rather than a idiomatic Haskell style. For example, decoding different block types requires slightly different logic. We would ideally want to separate this logic in the following way:

```haskell
data BlockType = YFstBlock | YBlock | CbBlock | CrBlock

decodeBlock :: BlockType -> Decoder ()
decodeBlock bt = case bt of
    YFstBlock -> ...
    YBlock    -> ...
    CbBlock   -> ...
    CrBlock   -> ...
```

However, since it is not possible to use Haskell data types in this way, we were forced to integers to differentiate the block types instead:

```haskell
type BlockType = Data Word32

decodeBlock :: BlockType -> Decoder ()
decodeBlock bt = do
  if1 (bt == 0) $ do
    ...
  if1 (bt > 0 && bt < 4) $ do
    ...
  if1 (bt == 4) $ do
    ...
```

This approach is essentially imitating C which in this case defeats the purpose of using a functional language. Feldpar supports Haskell product types (records) by translating every field of the record to a separate variable in C. We believe it should in principle be possible to also support sum types by assigning a number to every value the type supports and using those in the generated C code. For instance, in the datatype `data D = A | B | C | ...`, `A` would be represented by the integer `1` in C, `B` by `2` and so on.

Another indication that Feldspar isn't complete is the fact that it lacks support of generating more than one C function. This means that an entire Feldspar program will be compiled into a single (possibly huge) `main`-function in C. This leads to

code repetition such as if we were to call our IDCT function twice, it would lead to generation of the same code twice. This also makes generation of recursive functions impossible – further forcing the programmer towards a more imperative style with C-like loops. One exception to this are recursive functions for which the data the function operates on is known at compile time, such as our VLC decoding function seen in Listing 4. In this case, Feldspar is able to generate a series of nested `if`-statements.

5.2 Advantages of Feldspar

The general advantage of Feldspar is that parts that can be written in a functional manner are typically more expressive than in C. One major example is our IDCT function seen in Listing 6. In our opinion, the Feldspar implementation is much closer to the mathematical definition, consisting of operations on entire matrices. The IDCT function in the reference C implementation [15] on the other hand expresses these operations as a series of nested loops and array indexing which we think hides the underlying meaning.

Another advantage is the strict typing that Feldspar inherits from Haskell. C is a weakly typed language meaning that values can be cast into different types even when doing so will break the program. Haskell and therefore Feldspar uses a strong type system where this is not possible. This gives the programmer a stronger guarantee that their program will function correctly, given that it type checks. Although understanding Feldspar's types was quite difficult in the beginning, it later felt like it prevented us from writing bad programs.

A third advantage of Feldspar is the ability to define custom control structures for monadic actions, due to it being DSL embedded in Haskell and thus inheriting this ability from the host language. This enabled us not only to implement functionality that is present in most languages but missing from Feldspar, but also to create novel functionality that many languages lack. An example of the former is the function `if1`, an *if* statement without an *else* clause. Most imperative languages have such a construct, but Feldspar by default only has the `iff` function which requires both an *if* and an *else* clause. The construct can however easily be emulated in Feldspar as shown in Listing 8.

```
if1 :: MonadComp m => Data Bool -> m () -> m ()
if1 cond a = iff cond a (pure ())
```

Listing 8: An *if* statement without an *else* clause implemented in Feldspar.

An example of a novel control structure is the `for2` loop. Instead of having to do a nested for loop, this construct allows us to iterate over two indices on a single line, as such:

```
for2 (0, 1, Excl 8) (7, -1, Incl 0) $ \i j ->
  printf "i + j = %d\n" (i + j)
```

This feature proved useful as we frequently traverse matrices during the decoding process. Again, the implementation is quite simple, as shown in Listing 9.

```
for2 :: (Integral n, PrimType n, MonadComp m)
     => IxRange (Data n)
     -> IxRange (Data n)
     -> (Data n -> Data n -> m ())
     -> m ()
for2 r1 r2 b = for r1 (\i1 -> for r2 (b i1))
```

Listing 9: A single statement for generating a nested *for* loop, implemented in Feldspar. An `IxRange` is a triple describing the starting value, increment, and end value of the loop index.

5.2.1 Error recovery

A feature that we chose not to implement in our decoder is the ability to recover from errors if a video stream contains invalid data. Had we chosen to do so, however, Feldspar contains a feature that we believe would have been useful for this purpose: the `OptionT` monad transformer [19]. To handle errors in a language like C, it is common for a function to return different status codes depending on whether it succeeded or ran into an error. It is then the programmer's job to make sure this return code is propagated to the appropriate place where a possible error should be handled. The `OptionT` monad transformer automates this process by the use of two functions: `some`, which takes a return value as an argument, and `none`, which takes an error message as an argument. Returning `some` will return the provided return value as normal. However, if at any point in the computation `none` is returned, this will override any subsequent return values and `none` will be the final result of the entire computation. An `OptionT` computation can consist of arbitrarily deeply nested functions and control structures, but the `none` will always propagate to the top.

For example, consider the following scenario: As explained in Section 2.1.2, a picture consists of a sequence of slices. If one of those slices were corrupt, we might want to ignore it and skip ahead to the next slice. If we fail to process even a single slice, we might opt to skip the entire picture and proceed to the next one. An outline of what this process might look like using `OptionT` is shown in Listing 10. The function `caseOptionT` chooses between different actions depending on whether the provided

`OptionT` was successful or failed. If an error occurred anywhere in `getSlices`, a `none` value would be produced and propagate all the way up to `decodeFile`, where the error would be caught and handled. The same is true for `decodeSlice`, whose error would be handled in `decodePic`.

```
decodeFile :: Run ()
decodeFile =
  loop do
    let pic = decodePic
    caseOptionT pic (\_ -> skipPic)
                    (\_ -> writePic)
    where
      writePic :: Run ()
      skipPic  :: Run ()

decodePic :: OptionT Run ()
decodePic = do
  slices <- exposeRows <$> getSlices

  for (0, 1, Excl numSlices) \i -> do
    let decoded = decodeSlice (slices ! i)
    lift $ caseOptionT decoded (\_ -> skipSlice)
                               writeSlice
  where
    getSlices   :: OptionT Run (DPull2 Word8)
    skipSlice   :: Run ()
    decodeSlice :: DPull Word8 -> OptionT Run (DPull Word8)
    writeSlice  :: DPull Word8 -> Run ()
```

Listing 10: An outline of how error handling might have been implemented using the `OptionT` monad transformer.

5.3 The decoder

As previously mentioned, although the output of our decoder looks visually identical to the output of the reference decoder, it is not identical byte-for-byte. Upon further inspection using a hex editor, we discovered that the values that differed did so only by a value of 1 or 2. Since bytes in the output represent luminance and chrominance values in the range [0...255], this difference is too small to be visible. We believe this might be the result of different orders of rounding in the different implementations and not necessarily a deviation from the standard on our part.

We also believe that this issue occurs in the IDCT since this is the only operation which involves floating point numbers as opposed to integers. One way to prove that out IDCT implementation is still correct would be to use the accuracy measurement test proposed in IEEE standard for the IDCT [13].

Finally, the reason for why the decoder only supports I- and P-pictures (and not B- and D-pictures) is because of the project's time constraint. Given more time, support for the remaining picture types could most likely be implemented with no significant difficulties. Doing so would however not impact the project's conclusion about Feldspar since the logic behind decoding the remaining picture types is similar to the supported ones.

5.4 Performance

Part of the aim of this project was to see if a high-level language implementation of a video decoder could achieve similar performance to a C implementation. As shown in Section 4.1, we cannot confirm this to be the case based on our results. Our decoder is almost an order of magnitude slower than the C reference decoder [15], which is already quite slow; decoding video at around 12.5 frames per second, it is not even fast enough to play back video in real time (typically between 24 and 60 frames per second).

As previously mentioned, the IDCT seems to be the main bottleneck in our decoder. This is likely because our implementation is directly based on the definition of the IDCT, which is rarely used in practice due to its inefficiency. A number of fast algorithms for computing the IDCT have been developed over the years to enable real time video decoding in software, such as [14]. It would be interesting to implement such an algorithm in Feldspar and evaluate the effect it has on the decoder's performance, as well as whether it could be expressed in a functional way. The C reference decoder [15] does use a faster IDCT algorithm, which we believe explains why it is so much faster, but looking at the source code (which is under copyright protection) it is hard to see how it could be expressed functionally.

The discovery that the IDCT seems to be the main bottleneck in our decoder was based on solely a guess. To discover this in a more systematic way would be difficult, if not impossible, since Feldspar only generates a single `main` function which in our case contains over $80,000$ lines of code. This eliminates two possible ways of discovering bottlenecks. Firstly, the use of profiling tools becomes useless since these typically report the time spent in each function. Secondly, manually inspecting the generated code is infeasible since it is impossible to locate any particular part of the program and see how many times it is run. This problem could be solved if Feldspar was able to generate more than one C function.

Another possibility to speed up performance would be to decode multiple parts of an image in parallel. Since there is no sequential dependency between the macroblocks within a picture it should in theory be possible to decode each one on a separate thread without any data races. Feldspar has a concurrency library which allows for task parallelism (but not data parallelism) which we did not explore during this project due to time constraints, but could likely be used for this purpose. However, decoding macroblocks in parallel would require some significant restructuring of our code so that the whole frame is read into memory before any decoding is done,

so that later parts of the image can be decoded in parallel with earlier ones. Our current code does not work like this; rather, it decodes each macroblock right after reading it from the file.

5.5 Related work

There are several projects trying to solve a similar problem to Feldspar – that is to enable a high-level functional implementation of software that normally has to be done using low-level imperative programming – but within a different domain. Such examples include Futhark [20] – a language focused on data parallel computations on GPUs, Obsidian [21] – an embedded DSL enabling high-level CUDA programming in Haskell, and FAUST [22] – a purely functional audio processing language which compiles to C++ (among others).

A somewhat related project is an encoder and decoder implementation of the JPEG image format, written in pure Haskell [23]. It shows not only the possibility of image signal processing in Haskell, but also demonstrates the simplicity of such processing in a functional language, without the language limitations in our case imposed by Feldspar.

5.6 Conclusion

- It is definitely possible to implement more advanced DSP applications in Feldspar.
- We were not able to confirm our hypothesis that writing a video decoder in Feldspar would be easier compared to a low level language. For some parts, it did result in cleaner and more elegant code, specifically parts where computations could be expressed as pure functions. Feldpar's `OptionT` monad transformer can also make error recovery more elegant than in C. For other parts however, the code was similar of not less elegant compared to C.
- In order to confirm our aforementioned hypothesis, we believe it would be necessary to extend Feldspar in the following ways:
  - An extension for parsing binary data.
  - The possibility to generate more than one C function, possibly including recursive functions.
  - The library should be extended to include operations for multidimensional arrays and vectors.
  - Support for Haskell sum types to be used in Feldspar code.
  - More advanced tutorials and more complete documentation should be written.

Bibliography

[1] Blu-ray Disc Association, "White Paper Blu-ray Disc™ Format," http://blu-raydisc.com/Assets/Downloadablefile/White_Paper_General_5th_20180216.pdf, 2018.

[2] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax, "Feldspar: A domain specific language for digital signal processing algorithms," in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*. IEEE, 2010, pp. 169–178.

[3] "Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s — Part 2: Video," https://www.sis.se/produkter/informationsteknik-kontorsutrustning/kodning-av-information/ssiso111722/, International Organization for Standardization, Geneva, CH, Standard, Aug. 1993.

[4] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[5] E. Dumic, M. Mustra, S. Grgic, and G. Gvozden, "Image quality of 4:2:2 and 4:2:0 chroma subsampling formats," in *2009 International Symposium ELMAR*. IEEE, 2009, pp. 19–24.

[6] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete cosine transform," *IEEE transactions on Computers*, vol. 100, no. 1, pp. 90–93, 1974.

[7] A. Persson, A. Ekblad, E. Axelsson, K. Claessen, M. Aronsson, and M. Karácsony, "Resource-AWare feldspar," https://github.com/Feldspar/raw-feldspar.

[8] J. Svenningsson and E. Axelsson, "Combining deep and shallow embedding of domain-specific languages," *Computer Languages, Systems & Structures*, vol. 44, pp. 143–165, 2015.

[9] P. Wadler, "The essence of functional programming," in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1992, pp. 1–14.

[10] S. L. P. J. Philip Wadler, "Imperative functional programming," in *Proceed-*

ing of the 20th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1993, pp. 71–84.

[11] A. Gill, J. Launchbury, and S. L. Peyton Jones, "A short cut to deforestation," in *Proceedings of the conference on Functional programming languages and computer architecture*, 1993, pp. 223–232.

[12] E. Axelsson, "raw-feldspar documentation: MonadComp," https://hackage.haskell.org/package/raw-feldspar-0.3/docs/Feldspar-Frontend.html#t:MonadComp, 2019.

[13] IEEE, "IEEE standard specifications for the implementations of 8x8 inverse discrete cosine transform," *IEEE Std 1180-1990*, pp. 1–12, 1991.

[14] A. C. Hung and T. H.-Y. Meng, "Statistical inverse discrete cosine transforms for image compression," in *Digital Video Compression on Personal Computers: Algorithms and Technologies*, vol. 2187. International Society for Optics and Photonics, 1994, pp. 196–207.

[15] "Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s — Part 5: Software simulation," Document: https://www.sis.se/produkter/informationsteknik-kontorsutrustning/kodning-av-information/kodning-av-information-for-ljud-bild-multimedia-och-hypermedia/isoiectr1117251998/, Software download: https://standards.iso.org/ittf/PubliclyAvailableStandards/c025029_ISO_IEC_TR_11172-5_1998(E)_Software_Simulation.zip, International Organization for Standardization, Geneva, CH, Standard, Oct. 1998.

[16] "Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s — Part 4: Compliance testing," https://www.sis.se/produkter/informationsteknik-kontorsutrustning/kodning-av-information/kodning-av-information-for-ljud-bild-multimedia-och-hypermedia/isoiec111724/, International Organization for Standardization, Geneva, CH, Standard, Mar. 1998.

[17] FFmpeg team, "FFmpeg, a suite of libraries and programs for encoding, decoding and processing multimedia streams." http://ffmpeg.org/.

[18] G. Sullivan and S. Estrop, "Recommended 8-Bit YUV Formats for Video Rendering," https://docs.microsoft.com/en-us/windows/win32/medfound/recommended-8-bit-yuv-formats-for-video-rendering, 2018.

[19] E. Axelsson, "raw-feldspar documentation: OptionT," https://hackage.haskell.org/package/raw-feldspar-0.3/docs/Feldspar-Data-Option.html, 2019.

[20] T. Henriksen, N. G. Serup, M. Elsman, F. Henglein, and C. E. Oancea, "Futhark: purely functional GPU-programming with nested parallelism and in-place array updates," in *ACM SIGPLAN Notices*, vol. 52, no. 6. ACM,

2017, pp. 556–571.

[21] J. Svensson, “Obsidian: GPU kernel programming in haskell,” Ph.D. dissertation, Chalmers University of Technology, 2011.

[22] Y. Orlarey, D. Fober, and S. Letz, “FAUST: an efficient functional approach to DSP programming,” https://hal.archives-ouvertes.fr/hal-02159014/, 2009.

[23] J. Fokker, “jpeg: A library for decoding jpeg files written in pure haskell,” https://hackage.haskell.org/package/jpeg.

Bibliographic notes

Blu-ray Disc Association. ***White Paper Blu-ray Disc™ Format*** [1] A specification for the Blu-ray Disc format which specifies the maximum capacity of Blu-ray discs.

E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson and A. Vajdax. ***Feldspar: A domain specific language for digital signal processing algorithms*** [2] This is the original Feldspar paper and therefore the foundation of this project. It describes the problem of implementing DSP applications in low level languages and proposes Feldspar, a functional language for DSP which compiles to C as a solution.

ISO. ***Coding of moving pictures and associated audio for digital storage media at up to about 1,5Mbit/s – Part 2: Video*** [3] This is an ISO standard for the MPEG-1 video format. It is fundamental to the project as following this specification is how we will produce a program which is compliant with the MPEG-1 standard.

D. A. Huffman. ***A method for the construction of minimum-redundancy codes*** [4] This paper introduces Huffman coding which can be used to construct VLCs.

E. Dumic, M. Mustra, S. Grgic and G. Gvozden. ***Image quality of 4:2:2 and 4:2:0 chroma subsampling formats*** [5] This paper explains chroma subsampling and compares the quality of different subsampling formats.

N. Ahmed, T. Natarajan and K. R. Rao. ***Discrete cosine transform*** [6] This paper introduces the discrete cosine transform and explains some of its applications.

A. Persson, A. Ekblad, E. Axelsson, K. Claessen, M. Aronsson and M. Karác-sony. ***Resource-AWare feldspar*** [7] This is the GitHub page for the Feldspar implementation used in this project.

J. Svenningsson and E. Axelsson. ***Combining deep and shallow embedding of domain-specific languages*** [8] This paper explores the advantages of combining shallow and deeply embedded domains specific languages. Feldspar is based on the described principles to a great extent.

P. Wadler. ***The essence of functional programming*** [9] This paper explores the use of monads in functional programming.

S. L. Peyton Jones and Philip Wadler. ***Imperative functional programming*** **[10]** This paper describes different methods of handling side effects in Haskell an proposes the IO monad as a solution to this problem.

A. Gill, J. Launchbury and S. L. Peyton Jones. ***A short cut to deforestation*** **[11]** This paper introduces fusion, meant for readers interested in the topic.

E. Axelsson. ***raw-feldspar documentation: MonadComp*** **[12]** Documentation of the MonadComp type class in the raw-feldspar library.

IEEE. ***IEEE standard specifications for the implementations of 8x8 inverse discrete cosine transform*** **[13]** A standard describing the 8x8 inverse discrete cosine transform that is used by the MPEG-1 standard.

A. C. Hung and T. H.-Y. Meng. ***Statistical inverse discrete cosine transforms for image compression*** **[14]** Describes a fast algorithm for the inverse discrete cosine transform.

ISO. ***Coding of moving pictures and associated audio for digital storage media at up to about 1,5Mbit/s – Part 5: Software simulation*** **[15]** This is reference implementation of MPEG-1. This is used to programatically verify the correctness of our decoder, by comparing the output of the former to the latter.

ISO. ***Coding of moving pictures and associated audio for digital storage media at up to about 1,5Mbit/s – Part 4: Compliance testing*** **[16]** This document describes procedures for testing whether an MPEG-1 implementation is compliant with the standard.

FFmpeg team. ***FFmpeg, a suite of libaries and programs for encoding, decoding and processing multimedia streams*** **[17]** FFmpeg is a comprehensive open-source multimedia framework. Among other things, it contains an MPEG-1 decoder written in C to which our decoder can be compared. It also includes the `ffplay` tool used to display raw video files.

G. Sullivan and S. Estrop. ***Recommended 8-Bit YUV Formats for Video Rendering*** **[18]** An online document describing, among others, the YV12 raw video format which our decoder outputs.

E. Axelsson. ***raw-feldspar documentation: OptionT*** **[19]** Documentation of the OptionT monad transformer from the raw-feldspar library.

T. Henriksen, N. G. Serup, M. Elsman, F. Henglein and C. E. Oancea. ***Futhark: purely functional GPU-programming with nested parallelism and in-place array updates*** **[20]** Futhark attempts to solve a similar problem to Feldspar, that is to enable highly performant functional programming.

J. Svensson. ***Obsidian: GPU Kernel Programming in Haskell*** **[21]** Obsidian is a functional language for CUDA programming. It is another example of high-level functional languages used for tasks usually programmed at a low level.

Y. Orlarey, D. Fober and S. Letz. ***FAUST: an efficient functional approach to DSP programming*** **[22]** This is another example of a functional language for DSP applications.

Jeroen Fokker. ***jpeg: A library for decoding jpeg files written in pure Haskell*** **[23]** This is an example of image processing in pure Haskell.