Implementing, Specifying, and Verifying the QOI Format in Dafny: A Case Study

✉ Ștefan Ciobâcă[1][0009−0000−4082−570X] and Diana-Elena Gratie[1][0009−0003−8938−4396]

Alexandru Ioan Cuza University, Iași, Romania
{stefan.ciobaca, diana.gratie}@uaic.ro

Abstract. We present as a case study a verified implementation in Dafny for the Quite OK Image Format, a recently introduced lossless image compression/decompression algorithm that aims to be simple, have a good compression ratio and be fast to execute. We present the choices we make in the implementation and the specification, which enable the verification effort.

Keywords: QOI Format · Dafny · Formal Verification · Image Compression · Satisfiability Modulo Theories · Program Proof · Deductive Verification · Case Study.

1 Introduction

The Quite OK Image (QOI) format [16] was introduced by Dominic Szablewski in 2022 as a high-performance lossless format for image compression that is very simple. The English-language specification of the format consists of a single A4 page document [16] and the reference implementation of the encoder and decoder has around 300 lines of C code [15]. Not only is QOI simpler than other lossless image compression algorithms, but QOI allows for typically 20% better compression than the more mainstream PNG format. Additionally, the QOI compression algorithm is typically 20 to 50 times as fast as PNG, while the decompression algorithm is typically 3 to 4 times as fast according to benchmarks performed by the author of the QOI format (https://qoiformat.org/benchmark/).

QOI is inherently sequential and it combines, in a clever manner, several well-known techniques for compression, such as run-length encoding, dictionary-based compression, and delta encoding. We briefly describe the format in Section 2 and refer to its specification [16] for reference. QOI is not difficult to implement, but the potential for edge cases makes it an interesting and non-trivial target for formal verification. Edge cases could include off-by-one errors, infinite loops when decoding or encoding, out-of-bounds array accesses or other types of crashes, incorrect encoding or decoding. While typical implementations are validated by testing, our verified implementation gives a higher degree of confidence in the lack of such errors.

We implement, specify, and formally verify the functional correctness of QOI in the Dafny [10] language. Our proof ensures there are no runtime errors such as

out-of-bounds array accesses, but also that encoding and decoding match their specification, ensuring that decoding is the inverse of encoding. We chose Dafny because it supports autoactive verification of imperative programs. A similar choice could be Why3 [7], but Isabelle/HOL [13] and the Coq proof assistant [17] could also be used, although these approaches would require a different style of verification involving either a program logic such as Iris [8] or VST [2] or a refinement logic such as proposed by Lammich [9]. In this paper, we present our approach to the entire verification process as a case study in formal verification.

Dafny is an imperative language with some functional and object-oriented features, and it features an auto-active verifier based on translation into Boogie [3], which discharges verification conditions using the Z3 solver [12]. We feature a very brief overview of Dafny in Section 3 and we refer to the Dafny reference manual [18] for more details.

In Section 4 we describe the main formalisation effort. We focus on the architecture of our formalisation, with the encoding and decoding process being split into two phases, which is, to our knowledge, novel. These phases enable an efficient verification process by separating concerns. We discuss related work in Section 5 and we conclude with a brief discussion in Section 6.

2 Preliminaries: The QOI Format

The QOI file format starts with a header: 14 bytes consisting of the `"qoif"` identifier, the width and height of the image, each 4 bytes in length, one byte for the number of channels used (3 for RGB, 4 for RGBA), and the last byte for the colorspace (0 for sRGB with linear alpha; 1 for all channels linear). The header is followed by a number of data `chunks` that encode the pixel data. The file ends with an 8-byte end marker (seven 0 bytes and one byte with a value of 1). There are six possible data chunks of various sizes (all with a bit length multiple of 8), each starting with either a 2-bit or an 8-bit tag. The 8-bit tags have precedence over the 2-bit ones. Further details can be found in the QOI specification [15].

2.1 Image encoding

The images are processed in a single pass, from left to right, top to bottom (row-major order). Each pixel is internally represented using a structure that gives the `(r,g,b,a)` values of the pixel. A pixel can be encoded in one of four ways:

1. As a run of the previous pixel `(QOI_OP_RUN)`
   A variable `run` is used to count the number of consecutive pixels that coincide. If the value of the current pixel `px` is the same as the previous pixel, `pxPrev` (initialized to `r=g=b=0, a=255`), increase the run length by one, up to a value of 62. If `px` $\neq$ `pxPrev`, the current pixel is encoded using one of the other methods.
2. As an index into an array of previously seen pixels `(QOI_OP_INDEX)`

An array `index` is used to remember 64 of the previously seen pixels. For each pixel `px`, a hash value (`0..63`) is computed that represents the index of `px` into the `index` array. Let `pos(px)` denote this hash value. If `index[pos(px)]` and `px` have the same value (the exact same color values are encoded in the `index` array), an index chunk is written to the stream. Otherwise (a different color with the same hash code is encoded in the `index` array), `index[pos(px)]` is updated and the pixel is encoded using one of the remaining methods.

3. As a difference to the `r,g,b` values of the previous pixel
   - 3.1 As a small difference (`QOI_OP_DIFF`)
     If the difference between the current and the previous pixel for each of the `r,g,b` channels is small enough (between `-2` and `1` for each color channel), encode this difference on two bits for each color channel, and write a diff chunk to the stream.
   - 3.2 As a significant difference (`QOI_OP_LUMA`)
     If the difference between the current (`px`) and previous pixel (`pxPrev`) for at least one of the `r,g,b` channels is outside of the `[-2,1]` interval, more bits are needed for the encoding. The green channel difference from `pxPrev` is encoded on 6 bits, and indicates the general direction of the change (values between `-32` and `31`). The red and blue channels' differences are computed relative to the green difference, and stored on 4 bits each (values between `-8..7`). The luma 2-byte chunk is written to the stream.
4. As full `r,g,b` or `r,g,b,a` values
   - 4.1 For the same alpha channel value (`QOI_OP_RGB`)
     If the alpha values of `px` and `pxPrev` are the same, encode the `r,g,b` channel values in an RGB chunk and write it to the stream (a leading 8-bit `b11111110` tag is used).
   - 4.2 For different alpha channel values (`QOI_OP_RGBA`)
     If the alpha values of `px` and `pxPrev` differ, encode the `r,g,b,a` channel values in an RGBA chunk and write it to the stream (a leading 8-bit `b11111111` tag is used).

2.2 Image decoding

The decompression of QOI images is also single-pass. The decoder first identifies the image width, height, the number of channels and colorspace from the header bytes, then iterates through the data chunks doing the reverse operations to get the pixel information. We present an example of the decoding process for the QOI format using the compressed image in Figure 1. The thicker boxes denote chunks, while the squares are bytes in hexadecimal.

The first four bytes are dedicated to the magic number (`"qoif"`), the next two groups of 4 bytes give the width and height of the picture (3 pixels for each, in our example). The two trailing bytes of the header give the number of channels (3, for `r,g,b` in our case) and the colorspace (1, for all channels linear, in the example).

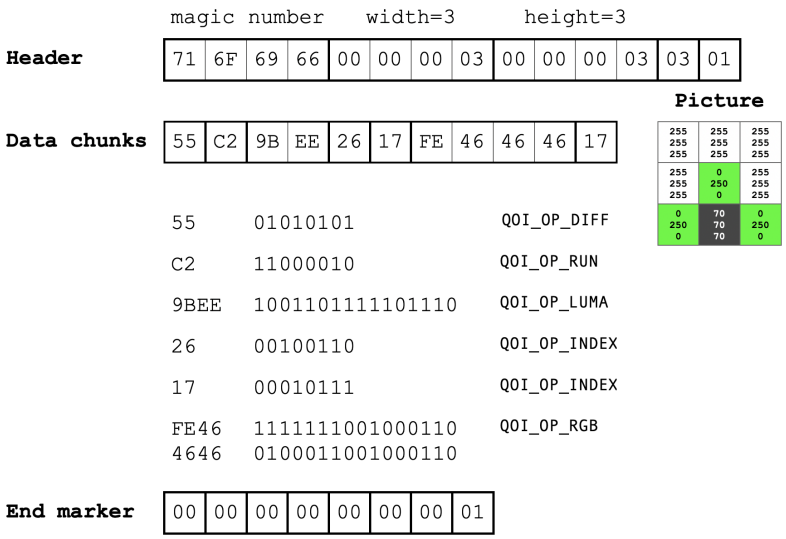

Fig. 1. Example of a compressed image in QOI format. Each color has its `r,g,b` code displayed.

There are seven data chunks encoded. `0x55` is decoded as a `QOI_OP_DIFF`, because an 8-bit valid tag is absent, and the 2-bit tag indicates a small difference (namely of -1 on each channel) between the first pixel of the picture and the default previous pixel, which is considered to be black and opaque. Due to the wraparound operation there is a difference of -1 between black $(0, 0, 0)$ and white $(255, 255, 255)$, so a small difference. The white pixel is stored in the `index` array.

The next byte, `0xC2`, is identified as a `QOI_OP_RUN` of length 3, so 3 more white pixels. The third chunk, consisting of two bytes, `0x9BEE`, is decoded as a `QOI_OP_LUMA`, with a difference of -5 on the `g` channel (250), and differences of 6 between the red/blue channels' differences and the green channel difference, so a green hue pixel (`0,250,0`). It is stored in the `index` array. The next two chunks are `QOI_OP_INDEX` chunks, with the indices of the white and green (respectively) pixels decoded previously.

The `0xFE` tag indicates a `QOI_OP_RGB` chunk, so together with the next 3 bytes it is decoded as a new color pixel, distant to the previously read ones, namely a dark grey (`70,70,70`). It is also added to the `index` array. The last data chunk is an index, decoded as the position of the green color pixel. The compressed image file ends with the 8-byte end marker.

3 Preliminaries: Dafny

Dafny is an imperative language and verifier supporting classes and dynamic allocation. Its methods contain specification constructs (preconditions - intro-

duced by `requires` -, postconditions - introduced by `ensures` -, framing constructs, invariants, and termination metrics). The features of the language allow for modular verification, where the correctness of a program is implied by the verification of each of its parts. The verification is auto-active, which implies an automatic verification part, and an interactive one. The preconditions and postconditions are automatically checked. For more complex code involving loops, the user can define loop invariants, variants, and other helper annotations, this being the interactive part.

A Dafny program is verified by translating it into Boogie [3] (such that the correctness of the translation implies that of the original program), which is then used to generate verification conditions that are passed on to the Z3 solver [12].

Figure 2 contains an example of a Dafny method implementing binary search. The precondition (Line 2) of the method requires that the input array `a` be sorted increasingly. It ensures that the output index `p` is either the index on which value `key` is found in the array (if that is the case), or negative (-1), if `key` is not among the values in `a`. The first while loop invariant (Line 9) specifies that `l` and `r` are indices between `0..a.Length`. The second loop invariant (Line 10) specifies that `key` is surely not found in `a` before position `l` or after position `r`. The variant `decreases r - l` on Line 11 gives a termination metric. More information on Dafny can be found in the user manual [18].

```
1  method BinarySearch(a: array<int>, key: int) returns (p: int)
2    requires forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
3    ensures 0 <= p ==> p < a.Length && a[p] == key
4    ensures p < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key
5  {
6    var l :int := 0;
7    var r :int := a.Length;
8    while (l < r)
9      invariant 0 <= l <= r <= a.Length
10     invariant forall i :: 0 <= i < a.Length && !(l <= i < r) ==> a[i] != key
11     decreases r - l
12   {
13     var mid := (l + r) / 2;
14     if a[mid] < key {
15       l := mid + 1;
16     } else if key < a[mid] {
17       r := mid;
18     } else {
19       return mid;
20     }
21   }
22   return -1;
23 }
```

Fig. 2. An example of verified Dafny code.

4 Formalisation

We now describe the formalisation of the compression and decompression algorithms. The full source code can be found at https://github.com/ciobaca/qoi-dafny/. The high-level architecture of the Dafny development is summarized in Figure 3.

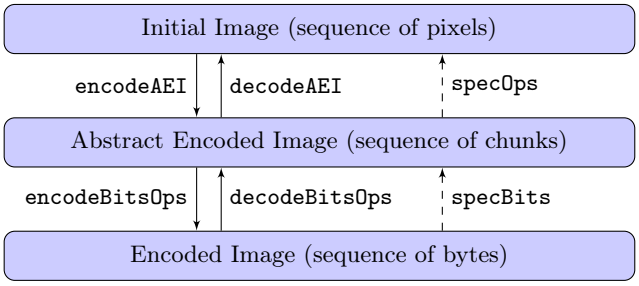

Fig. 3. The architecture of the development.

In order to simplify the compression (encoding) and decompression (decoding) processes, we use an intermediate layer for representing images that sits between the initial image (seen as a sequence of pixels) and the compressed image (seen as a sequence of bytes). This intermediate layer stores the encoded image as a list of chunks, but the chunks are represented at the logical level, not as bytes.

This makes the verification and development effort more clear by separating two orthogonal concerns: • the generation (respectively interpretation) of the logical chunks and • the encoding (respectively decoding) of the chunks into bytes.

Using the three-layer approach described above, we implement compression as the composition of two phases:

1. The method `encodeAEI` (`AEI` stands for Abstract Encoded Image) takes as input an image and computes the chunks.
2. The method `encodeBitsOps` takes as input the chunks and encodes them as a sequence of bytes.

Similarly, we use two phases for decompression:

1. The method `decodeBitsOps` takes as input a sequence of bytes and decodes it into a sequence of chunks.
2. The method `decodeAEI` takes as input the chunks, represented at the logical level, and computes the image.

The end-to-end decompression phase is the composition of the two methods.

We formally verify that the implementation terminates, does not have memory errors such as out-of-bounds accesses, and is functionally correct, in the sense that encoding and decoding meet their specification. The specification is given by the `specOps` function (respectively by the `specBits` function), shown in the diagram in Figure 3 on the right. These specification-level functions specify the decoding process.

As part of the verification process, we show that the decoding methods (`decodeAEI`, `decodeBitsOps`) match these two functions exactly. For the encoding process, we show that applying these two functions on the result yields back the initial input. From this it follows immediately that decoding is the inverse of encoding. We choose to specify the decoding phase (instead of encoding, or both) since it is deterministic, while the encoding can be nondeterministic: there can be several encodings for the same image (for example, for a pixel that is close to the previous pixel, and that is also stored in the index, an encoder could emit either an `OpIndex` or an `OpDiff`).

We now present the core data structures that we use and we explain the implementation choices.

4.1 Representing the Initial Image

We represent one pixel by an algebraic data type: either `RGB` if the image has three color channels, or `RGBA` if there is an additional *alpha* channel.

```
datatype RGB = RGB(r : byte, g : byte, b : byte)
datatype RGBA = RGBA(r : byte, g : byte, b : byte, a : byte)
```

The identifier `RGB` (respectively `RGBA`) doubles as both the name of the data type and the name of the only constructor for the data type. The names `r`, `g`, `b` (and respectively `a`) act as destructors.

The number of channels is represented as a refinement type which will be extracted to a byte.

```
newtype {:nativeType "byte"} Channels = x : int | 3 <= x <= 4 witness 3
```

The color space is represented as an enumeration and it is used only for storing in the file header – it does not impact the compression or decompression process in any way.

```
datatype ColorSpace = SRGB | Linear
```

Images are represented by metadata (the data type `Desc`) and the sequence of bytes representing the pixel data.

```
datatype Desc = Desc(width : uint32,
                     height : uint32,
                     channels : Channels,
                     colorSpace : ColorSpace)
datatype Image = Image(desc : Desc, data : seq<byte>)
```

We define `byte`s (respectively `uint32`s) to represent 8-bit unsigned integers (respectively 32-bit unsigned integers).

```
newtype {:nativeType "byte"} byte = x:int | 0 <= x < 256
newtype {:nativeType "uint"} uint32 = x:int | 0 <= x < 4294967296
```

As is usual in Dafny, we define a validity predicate that is responsible for identifying valid `Image`s. In our case the validity predicate simply checks that the image data contains enough bytes to represent the entire image.

```
predicate validImage(image : Image)
{
  |image.data| == image.desc.width as int *
    image.desc.height as int *
    image.desc.channels as int
}
```

4.2 Representing the Chunks at the Logical Level

We represent chunks by using an algebraic data type. Each type of chunk has one constructor.

```
datatype Op = OpRun(size : Size) // chunk represents a segment of the same color
  | OpIndex(index : Index64)     // index into previously seen colors
  | OpDiff(diff : RGBDiff)       // delta encoding (first type)
  | OpLuma(luma : RGBLuma)       // delta encoding (second type)
  | OpRGB(rgb : RGB)             // pixel value (3 channels)
  | OpRGBA(rgba : RGBA)          // pixel value (4 channels)
```

The arguments of each constructor are defined using refinement types so that only valid values can be represented.

```
newtype {:nativeType "byte"} Size = x : int | 1 <= x <= 62 witness 1
newtype {:nativeType "byte"} Index64 = x : int | 0 <= x <= 63
newtype {:nativeType "short"} Diff64 = x : int | -32 <= x <= 31
newtype {:nativeType "short"} Diff16 = x : int | -8 <= x <= 7
newtype {:nativeType "short"} Diff = x : int | -2 <= x <= 1
datatype RGBDiff = RGBDiff(dr : Diff, dg : Diff, db : Diff)
datatype RGBLuma = RGBLuma(dr : Diff16, dg : Diff64, db : Diff16)
```

The type `Size` denotes integers between 1 and 62, which represent valid lengths for the run-length encoding operator `OpRun`. The type `Index64` represents integers between 0 and 63, which are valid indices into the dictionary for the compression operator `OpIndex`. The difference-based operators `OpDiff` and `OpLuma` use the `Diff64`, `Diff16`, and `Diff` types to store the color differences from the previous pixel.

Finally, to represent an encoded image at the abstract level of chunks, we use the type

```
datatype AEI = AEI(width : uint32, height : uint32, ops : seq<Op>)
```

The `AEI` datatype does not recall the header information other than `width` and `height`. The additional information will be stored elsewhere to save it into the final byte stream.

4.3 Specification

To prove that the encoding and decoding processes are correct, we need a specification. We specify the meaning of one chunk using the function `specDecodeOp`:

```
function specDecodeOp(state : State, op : Op) : seq<RGBA>
  requires validState(state)
{
  match op {
  case OpRGB(RGB(r, g, b)) => [ RGBA(r, g, b, state.prev.a) ]
  case OpRun(size) => seq(size, i => state.prev)
  case OpIndex(index) => [ state.index[index] ]
  case OpDiff(RGBDiff(dr, dg, db)) =>
    [ RGBA(add_byte(state.prev.r, byte_from(dr as int)),
        add_byte(state.prev.g, byte_from(dg as int)),
        add_byte(state.prev.b, byte_from(db as int)),
        state.prev.a ) ]
  case OpLuma(RGBLuma(dr, dg, db)) =>
    [ RGBA(add_byte(add_byte(state.prev.r, byte_from(dg as int)), byte_from(dr as int)),
        add_byte(state.prev.g, byte_from(dg as int)),
        add_byte(add_byte(state.prev.b, byte_from(dg as int)), byte_from(db as int)),
        state.prev.a ) ]
  case OpRGBA(rgba) => [ rgba ]
  }
}
```

This function returns a sequence of pixels represented by one chunk. Sequences, represented by `seq`, are immutable data structures. Because this function is only used at the specification level, the runtime overhead induced by immutability is not relevant. We do not show the functions `add_byte`, `byte_from` to save space. In order to remove duplication, we define the function to return `RGBA`s, independently of the number of channels of the image. For 3-channel images, the *alpha* components will be 255. The decoding function depends crucially on the current *state* of the encoder (the first argument), as defined in the QOI format specification. We represent the state at the specification level as a pair consisting of the previous pixel and a sequence of 64 previously-seen pixels (as defined in the QOI format specification):

```
datatype State = State(prev : RGBA, index : seq<RGBA>)
ghost predicate validState(state : State)
{
  |state.index| == 64
}
```

We specify how the state changes after visiting one pixel and what the initial state is by following the official specification:

```
function updateState(previous : State, pixel : RGBA) : State
  requires validState(previous)
  ensures validState(updateState(previous, pixel))
{
  State(prev := pixel, index := previous.index[hashRGBA(pixel) := pixel])
}
function initState() : State
{
  State(prev := RGBA(0, 0, 0, 255), index :=
    seq(64, i => RGBA(r := 0, g := 0, b := 0, a := 255)))
}
```

We specify the meaning of an `AEI` as a sequence of pixels by concatenating the information given by subsequent calls to `specDecodeOp`:

```
function spec(aei : AEI) : seq<RGBA>
  requires validAEI(aei)
{
  specOps(aei.ops)
}
function specOps(ops : seq<Op>) : seq<RGBA>
{
  specOpsAux(ops, initState())
}
function specOpsAux(ops : seq<Op>, state : State) : seq<RGBA>
  requires validState(state)
{
  if |ops| == 0 then []
  else var pixels : seq<RGBA> := specDecodeOp(state, ops[0]);
    pixels + specOpsAux(ops[1..], updateStateStar(state, pixels))
}
```

For brevity, we skip the function `updateStateStar`, which uses `updateState` repeatedly to specify how the state should be updated after visiting *several* pixels. We also specify the meaning of a sequence of bytes as a chunk at the logical level:

```
function decodeBits(bits : seq<byte>) : Op
  requires validBits(bits)
{
  match opTypeOfBits(bits[0]) {
  case TypeRun => OpRun((bits[0] - 128 - 64 + 1) as Size)
  case TypeLuma => OpLuma( RGBLuma( ((bits[1] / 16) as int - 8) as Diff16,
    ((bits[0] - 128) as int - 32) as Diff64, ((bits[1] % 16) as int - 8) as Diff16))
  case TypeDiff => OpDiff( RGBDiff( (((bits[0] - 64) / 16) as int - 2) as Diff,
    (((bits[0] / 4) % 4) as int - 2) as Diff, ((bits[0] % 4) as int - 2) as Diff))
  case TypeIndex => OpIndex(bits[0] as Index64)
  case TypeRGBA => OpRGBA(RGBA(bits[1], bits[2], bits[3], bits[4]))
  case TypeRGB => OpRGB(RGB(bits[1], bits[2], bits[3]))
  }
}
```

For brevity, we do not show the `validBits` predicate, nor the `opTypeOfBits` function. We concatenate the results of the `decodeBits` function to obtain the sequence of chunks represented by a sequence of bytes.

```
function specBits(bits : seq<byte>) : seq<Op>
  requires validBitSeq(bits)
{
  if |bits| == 0 then []
  else ( var len := sizeBitEncoding(opTypeOfBits(bits[0]));
    [ decodeBits(bits[0..len]) ] + specBits(bits[len..]) )
}
```

We would like to emphasize the recursion pattern in `specBits` above, which has a significant influence on proofs later in the paper. Note that `bits` is a concatenation of potentially variably-sized chunks. There, we cannot know beforehand at which point one chunk ends and another starts. In order to find this information, we need to inspect the first byte (`bits[0]`), which contains enough information to determine the size of the first chunk (saved in the variable `len`). The function `opTypeOfBits` computes the type of the chunk depending on the first byte and `sizeBitEncoding` computes the length, in bytes, of such a chunk. These functions are not shown to save space. Once we know where the first chunk ends, we can perform the recursive call. For brevity, we do not show the `validBitSeq` function, which uses a similar recursion pattern to check whether the input data is a valid concatenation of chunks.

```
method encodeAEI(image : seq<RGBA>) returns (r : array<Op>, len : int)
  ensures 0 <= len <= r.Length
  ensures specOps(r[..len]) == image
{
  r := new Op [|image|];
  var prev : RGBA := RGBA(r := 0, g := 0, b := 0, a := 255);
  var index : array<RGBA> := new RGBA[64](i => RGBA(r := 0, g := 0, b := 0, a := 255));
  var i : int := 0;
  var wh : int := |image|;
  len := 0;
  ghost var state := initState();
  while (i < wh)
    invariant 0 <= len <= i <= wh
    decreases wh - i
    invariant state == updateStateStar(initState(), image[..i])
    invariant state.prev == prev
    invariant state.index == index[..]
    invariant specOps(r[..len]) == image[..i]
  {
    var curr := image[i];
    len := encodePixelAEI(curr, prev, index, state, r, len);
    state := updateState(state, curr);
    prev := curr;
    i := i + 1;
 }
}
```

```
method encodePixelAEI(curr : RGBA, prev : RGBA, index : array<RGBA>,
  ghost state : State, encoding : array<Op>, len : int) returns (newlen : int)
  ensures specOps(old(encoding[..len])) + [ curr ] == specOps(encoding[..newlen])
{
  newlen := len + 1;
  if (curr == prev) {
    if (len > 0 && encoding[len - 1].OpRun? && encoding[len - 1].size < 62) {
      encoding[len - 1] := OpRun(encoding[len - 1].size + 1);
      newlen := len;
    } else {
      encoding[len] := OpRun(1);
    }
  } else if (index[hashRGBA(curr)] == curr) {
    encoding[len] := OpIndex(hashRGBA(curr) as Index64);
  } else if (canDiff(curr, prev) != None) {
    var result := canDiff(curr, prev).some;
    encoding[len] := OpDiff(result);
  } else if (canLuma(curr, prev) != None) {
    var result := canLuma(curr, prev).some;
    encoding[len] := OpLuma(result);
  } else if (curr.a == prev.a) {
    encoding[len] := OpRGB(RGB(curr.r, curr.g, curr.b));
  } else {
    encoding[len] := OpRGBA(RGBA(curr.r, curr.g, curr.b, curr.a));
  }
  var h := hashRGBA(curr);
  index[h] := curr;
}
```

Fig. 4. The functions that compute the chunks representing an image.

4.4 Encoding

The encoding function that takes images into abstract encoded images, shown in Figure 4 follows typical encoder logic.

At each step, we show that the state of the algorithm (stored in the variables `prev` and `index`) matches the specification (Line 16) and that what was encoded so far matches a corresponding prefix of the image data (Line 18). The main postcondition (Line 3) establishes that decoding the result yields back the original image data. The method `encodePixelAEI(curr, prev, index, state, r, len)` does the work for a single pixel (several verification conditions removed for brevity).

Encoding an abstract encoded image as a sequence of bytes (Figure 5) is straightforward (some verification conditions removed to save space).

```
1  method encodeBitsOps(ops : array<Op>, opsLen : int, result : array<byte>,
2    pos : int) returns (newpos : int)
3    requires 0 <= opsLen <= ops.Length
4    ensures result[pos..newpos] == encodeOpsSpec(ops[..opsLen])
5  {
6    var i := 0;
7    newpos := pos;
8    while (i < opsLen)
9      invariant 0 <= i <= opsLen
10     invariant pos <= newpos
11     invariant newpos + 5 * (opsLen - i) + 8 <= result.Length
12     invariant result[..pos] == old(result[..pos])
13     invariant result[pos..newpos] == encodeOpsSpec(ops[..i])
14   {
15     ghost var oldpos := newpos;
16     newpos := writeBitsOp(ops[i], result, newpos);
17     i := i + 1;
18   }
19 }
```

Fig. 5. The method for encoding an abstract encoded image as a sequence of bytes. The helper method `writeBitsOp` (not shown to save space) encodes a single chunk.

4.5 Decoding

Decoding an abstract encoded image (Figure 6, top) into an image is relatively straightforward (some helper assertions are removed from the code for brevity). Decoding one chunk is done by `decodeOp` (not shown to save space).

Decoding a byte stream into a sequence of chunks (Figure 6, bottom) is significantly more difficult to prove, because each chunk is represented using a potentially variable number of bytes. This means that not all prefixes of the byte stream are valid representations of chunks. We walk the byte stream and check at each step the type of the first potential chunk and its length, `len` (Line 16). If the byte stream contains at least `len` bytes, and they represent a valid chunk (Line 23), we store the decoding of the `len` bytes into the `result` and we advance the index `i` by `len` positions (Line 28). Crucially, we use as an invariant (Line 14) that what was stored so far is the decoding of everything up to the current position `i`. But we never know until the very end if the remaining stream (after position `i`) is valid or not. In order to keep track of this, we use an invariant (Line

```
method decodeAEI(ops : array<Op>, size : int) returns (r : array<RGBA>, ok : bool)
  ensures ok ==> r[..size] == specOps(ops[..])
{
  var i := 0;
  var state := initState();
  var len := 0;
  r := new RGBA [size];
  ok := true;
  while (i < ops.Length)
    invariant specOps(ops[..i]) == r[..len]
  {
    var op := ops[i];
    var ok' : bool;
    len, state, ok' := decodeOp(state, op, len, r);
    if (!ok')
    {
      ok := false;
      return;
    }
    i := i + 1;
  }
  if (ok) {
    if (len != size)
    {
      ok := false;
      return;
    }
  }
}
```

```
method decodeBitsOps(bits : array<byte>, start : int, end : int)
  returns (r : Option<array<Op>>, lenr : int)
  ensures !validBitSeq(bits[start..end]) ==> r.None?
  ensures validBitSeq(bits[start..end]) ==>
    r.Some? && r.some[..lenr] == specBits(bits[start..end])
{
  var i := 0;
  var j := 0;
  var result : array<Op> := new Op [end - start];
  while (i < end - start)
    invariant 0 <= j <= i <= end - start
    invariant validBitSeq(bits[start + i..end]) == validBitSeq(bits[start..end])
    invariant validBitSeq(bits[start..start + i])
    invariant result[..j] == specBits(bits[start..start + i])
  {
    var len := sizeBitEncoding(opTypeOfBits(bits[start + i]));
    if (i + len > end - start)
    {
      return None, j;
    }
    else
    {
      var b := areValidBits(bits, start + i, len);
      if (b)
      {
        result[j] := decodeBits(bits[start + i..start + i + len]);
        j := j + 1;
        i := i + len;
      }
      else
      {
        return None, j;
      }
    }
  }
  return Some(result), j;
}
```

Fig. 6. The methods that decodes chunk into a image and a stream of bytes into a stream of chunks. Some verification conditions are not shown to save space.

12) stating that the full byte stream is valid iff what is left is also valid. This invariant crucially allows us to prove the postcondition when the byte stream given as input is not valid.

4.6 Statistics

The core development is structured into 4 files:

Name	Line Count	Purpose
`helper.dfy`	69	Bytes, Helper Functions
`spec.dfy`	329	AEI, Specification
`specbit.dfy`	540	Chunks as bytes, Specification
`qoi.dfy`	792	Encoding and Decoding

The Dafny lines of code quoted above include specification, lemmas, helper assertions, comments, white lines. The development has 131 methods, lemmas, and functions to verify. Together, they take around 42 seconds to verify on a modern computer (2,4 GHz 8-Core Intel Core i9, 16 GB 2667 MHz DDR4) in single threaded mode. We estimate that the entire Dafny development took approximately a month of part-time work, stretched over a longer period.

The largest verification time (a bit more than 10 seconds) is required by the `canLuma` function:

```
function canLuma(curr : RGBA, prev : RGBA) : Option<RGBLuma>
  ensures forall luma ::  canLuma(curr, prev) == Some(luma) ==>
  curr.r == add_byte(add_byte(prev.r, byte_from(luma.dg as int)),
    byte_from(luma.dr as int)) &&
  curr.g == add_byte(prev.g, byte_from(luma.dg as int)) &&
  curr.b == add_byte(add_byte(prev.b, byte_from(luma.dg as int)),
    byte_from(luma.db as int))  &&
  curr.a == prev.a
{
  var dr : int := curr.r as int - prev.r as int;
  var dg : int := curr.g as int - prev.g as int;
  var db : int := curr.b as int - prev.b as int;
  var da : int := curr.a as int - prev.a as int;
  if (-32 <= dg <= 31 && -8 <= (dr - dg) <= 7 &&
      -8 <= (db - dg) <= 7 && da == 0) then
    // ... (some assertions removed for brevity)
    Some(RGBLuma((dr-dg) as Diff16, dg as Diff64, (db-dg) as Diff16))
  else
    None
}
```

This checks whether a pixel is close enough to the previous one to be represented by a `QOI_OP_LUMA` chunk and the large verification time is likely due to the SMT solver performing bit-blasting. Most of the other functions and methods are much quicker to verify, with most verifying instantly and just 7 taking more than a second.

Here is a summary of the entire development:

Item	Count
Number of lemmas	25
Number of methods	16
Number of preconditions	81
Number of postconditions	99
Number of invariants	35
Number of helper assertions	161
Number of calculational [11] proofs	5
Number of comments	94
Number of white lines	129

To check interoperability, we extract the above code, which was formally verified in Dafny, into `C#`. We choose `C#` because this is the backend that Dafny supports best. We link it with some unverified `C#/Dafny` code (files `file_input.cs`, `file_input.dfy`, `entry.dfy`) that performs file IO and we validate that the resulting executable interoperates with other implementations of the QOI format. In preliminary testing, we time our implementation at 1.9 mega pixels per second for encoding and 0.17 mega pixels per second for decoding, while the fast reference C implementation delivers a bit more than 100 megapixels per second. In future work, we plan to optimize the implementation further for runtime efficiency and benchmark it more thoroughly against the reference C implementation.

5 Related Work

To our knowledge, no other attempts at implementing, specifying and verifying the QOI format encoding and decoding algorithms using Dafny have been published so far. A presentation of a preliminary version of our implementation using Dafny was given by the first author of this paper at the Tenth Congress of Romanian Mathematicians in 2023. There was no formal publication and the Dafny development has since changed significantly.

A similar line of work on the QOI format encoding and decoding algorithms is by Bucev and Kunčak [4] using the Stainless verifier. It is worth noting that the code extracted from this development is very fast, but the development itself takes much more to verify. To our understanding, the authors manage to prove that decoding is the inverse of encoding without relying on a separate specification. However, the code is written in a functional style, which makes the proof simpler. In future work, it is worth investigating in more depth this asymmetric difference in performance. On the one hand, we suspect that the architecture of our development, using an intermediate encoding at the abstract level, makes the verification process much more feasible. On the other hand, this could impact performance in a non-negligible way.

A formally verified Ada/SPARK implementation was announced in a blog post in [6], proving the absence of runtime errors, but without a full functional correctness proof.

Loosely related to our work, we mention other instances where specific algorithms were formally verified. For example the Deflate [5] compressed data format algorithm was formalized, implemented and verified using Coq, see [14]. A couple of standard lemmas in data compression were formally proven in Coq, with application to the Shannon-Fano codes in [1].

Other approaches aim to verify other tasks that do not imply compression, for example serialization tasks. A compiler for the Protocol Buffer serialization format implemented in Coq was formally verified in [19], proving the correctness of every generated serializer/deserializer.

6 Discussion

We have implemented, specified, and verified the algorithms for encoding and decoding images into the QOI format. The main novelty of our work is the modular architecture, where we use an intermediate level for representing a compressed image at the abstract level. This allows the verification time to stay reasonable and separates concerns nicely between the bitwise representation of the encoded image and its logical counterpart. Subjectively, this separation of concerns enables the entire verification effort. It might be possible to preserve this modularity at the specification-level only and merge the two phases into one at the implementation level. In future work, we will test whether this approach works and if it improves runtime efficiency.

There are several directions for future work. First of all, there are still many known places in the decoding and encoding implementation where performance can be optimised for runtime efficiency. For example, we rely on `int` in several places in the code, which is extracted as a `BigInteger`, slowing down the encoding/decoding process. We should move towards machine integers. Moreover, especially in the decoding phase, we rely on some immutable data structures, which are slower because they involve copying memory. Furthermore, currently Dafny does not fully support extraction to `C++`. Instead, we use extraction to `C#`, which can be slower as a language than `C++`. Improving support for extracting to `C++` in Dafny would improve the running time. Another approach to obtain a verified efficient implementation would be to use a refinement calculus in the style of Lammich [9].

An open problem which would be interesting to attack is to prove that the encoding and decoding methods (`encodeAEI` and `decodeAEI`, respectively `encodeBitsOps` and `decodeBitsOps`) are reverses one of the other, but without relying on the intermediate specification-level functions (`specOps`, respectively `specBits`). This could prove challenging given that these are methods (imperative code), but it would remove some of the duplication. Additionally, it seems that many of our helper lemmas could be generalized and potentially reused in other types of encoding/decoding processes. For example, the technique we use to verify the decoding of the byte stream into abstract chunks could be useful in other contexts where a variable-length encoding is used. Finally, there is a requirement in the QOI format specification which we do not formalise and leave

for future work instead: a valid encoder is required not to issue 2 or more consecutive `QOI_OP_INDEX` chunks to the same index (instead, `QOI_OP_RUN` should be used). While our encoder does satisfy this requirement, we do not currently formalise, nor prove, this. In contrast, the specification has nothing more to say about optimality: an encoder could simply issue a `QOI_OP_RGB(A)` for each pixel. Some form of minimality for the encoder would be an interesting additional requirement to formalize and verify in future work.

Acknowledgements We would like to thank the reviewers for their thoughtful comments and suggestions and for the interesting questions, which helped improve the paper.

References

1. Affeldt, R., Garrigue, J., Saikawa, T.: Examples of formal proofs about data compression. In: International Symposium on Information Theory and Its Applications, ISITA 2018, Singapore, October 28-31, 2018. pp. 633–637. IEEE (2018). https://doi.org/10.23919/ISITA.2018.8664276
2. Appel, A.W.: Verified Software Toolchain - (invited talk). In: Barthe, G. (ed.) Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6602, pp. 1–17. Springer (2011). https://doi.org/10.1007/978-3-642-19718-5_1
3. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17
4. Bucev, M., Kunčak, V.: Formally verified Quite OK Image Format. In: Griggio, A., Rungta, N. (eds.) 22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022. pp. 343–348. IEEE (2022). https://doi.org/10.34727/2022/ISBN.978-3-85448-053-2_41
5. Deutsch, L.P.: DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (May 1996). https://doi.org/10.17487/RFC1951, https://www.rfc-editor.org/info/rfc1951
6. Fabien Chouteau, J.H.: Quite proved image format, https://blog.adacore.com/quite-proved-image-format
7. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
8. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming **28**, e20 (2018). https://doi.org/10.1017/S0956796818000151

9. Lammich, P.: Refinement to Imperative HOL. Journal of Automated Reasoning **62**(4), 481–503 (2019). https://doi.org/10.1007/S10817-017-9437-1
10. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
11. Leino, K.R.M., Polikarpova, N.: Verified calculations. In: Cohen, E., Rybalchenko, A. (eds.) Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8164, pp. 170–190. Springer (2013). https://doi.org/10.1007/978-3-642-54108-7_9
12. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008, Budapest, Hungary, 29 March - 6 April 2008. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
13. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002). https://doi.org/10.1007/3-540-45949-9
14. Senjak, C., Hofmann, M.: An implementation of Deflate in Coq. CoRR **abs/1609.01220** (2016), http://arxiv.org/abs/1609.01220
15. Szablewski, D.: QOI - the "Quite OK Image Format" for fast, lossless image compression, https://github.com/phoboslab/qoi
16. Szablewski, D.: The Quite OK Image Format specification, https://qoiformat.org/qoi-specification.pdf
17. The Coq Development Team: The Coq reference manual – release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman (2024)
18. The dafny-lang community: Dafny reference manual (2024), https://dafny.org/dafny/DafnyRef/DafnyRef
19. Ye, Q., Delaware, B.: A verified protocol buffer compiler. In: Mahboubi, A., Myreen, M.O. (eds.) Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019. pp. 222–233. ACM (2019). https://doi.org/10.1145/3293880.3294105