

# Formally Verified Quite OK Image Format

Mario Bucev

*School of Computer and Communication Sciences*  
EPFL

1015 Lausanne, Switzerland  
mario.bucev@epfl.ch

Viktor Kunčák

*School of Computer and Communication Sciences*  
EPFL

1015 Lausanne, Switzerland  
viktor.kuncak@epfl.ch  
<https://orcid.org/0000-0001-7044-9522>

**Abstract**—Lossless compression and decompression functions are ubiquitous operations that have a clear high-level specification and are thus suitable as verification benchmarks. Such functions are also important. On the one hand, they improve the performance of communication, storage, and computation. On the other hand, errors in them would result in a loss of data. These functions operate on sequences of unbounded length and contain unbounded loops or recursion that update large state space, which makes finite-state methods and symbolic execution difficult to apply.

We present deductive verification of an executable Stainless implementation of compression and decompression for the recently proposed Quite OK Image format (QOI). While fast and easy to implement, QOI is non-trivial and includes a number of widely used techniques such as run-length encoding and dictionary-based compression. We completed formal verification using the Stainless verifier, proving that encoding followed by decoding produces the original image. Stainless transpiler was also able to generate C code that compiles with GCC, is inter-operable with the reference implementation and runs with performance essentially matching the reference C implementation.

**Index Terms**—formal verification, compression, Stainless, SMT solver, mechanized induction

## I. INTRODUCTION

Lossless conversions are ubiquitous. Examples include compression tools such as zip, as well as lossless image formats such as PNG. Unfortunately, common compression formats, especially ones for pictures, are more complex than one would expect a first. As a result of this complexity and the absence of precise specifications, it has proven difficult to reason about implementations of these algorithms. Consequently, the practice in the field is to use software testing, possibly backed by advanced testing algorithms [1], which do not guarantee correctness. As a reaction to the complexity of existing formats, Dominic Szablewski announced the “quite OK image format” [2] on 24 November 2021. The proposal was accompanied by a concise and efficient implementation. It attracted significant attention, with re-implementations quickly emerging in different programming languages (including Verilog) as well as variations such as streaming implementations.

Inspired by these developments, this paper presents an executable and formally verified implementation of the quite OK image encoding and decoding algorithms. We have presented

this formal development and shared the code on GitHub as part of the ASPLOS 2022 tutorial at EPFL in March 2022 [3], but no reviewed record of the work existed until now. The verified case study is now also available at:

<https://github.com/epfl-lara/bolts/tree/master/qoi/>

We are not aware of a formally verified implementation of functional correctness of QOI. Recently, a blog appeared referring to an implementation in Ada/SPARK<sup>1</sup>. Our understanding is that this Ada/SPARK implementation only proves the absence of run-time errors and not full correctness.

In a broader line of work, formal verification was applied either to specific algorithms or domain-specific languages. The Deflate algorithm [4] specification has been formalized, implemented, and verified in [5] in Coq. Researchers also formalized common lemmas in information theory in Coq and apply these to Shannon-Fano codes [6].

Related approaches verify serialization tasks, which do not typically aim to compress data. Examples of such work include [7] formally verified Protocol buffer compiler implementation in Coq, for a commonly used subset of this serialization format. Correct by construction pretty printing in parsing libraries also ensures correctness subject to certain local invertibility conditions [8, Section 6.4], as do invertible lenses [9]. Our case study may thus also provide a starting point for exploring the expressive power of provably invertible domain-specific languages for data transformation.

## II. BACKGROUND

### A. Stainless Verifier and C Transpiler

Stainless [3], [10]–[12] accepts as input source code in a subset of the Scala programming language [13]. Typical Stainless programs can thus be compiled using the existing Scala compilers and run using the Java Virtual Machine.

Stainless supports formal verification of assertions, preconditions, postconditions, and invariants using the Inox solver. Inox in turn relies on unfolding of function definitions and uses SMT solvers, notably Z3, CVC4, and Princess.

Stainless also supports generation of C code (transpilation) for a subset of Scala. This subset targets programs without heap-allocated memory, in the spirit of our previous case study [14]. We wrote our QOI format case study to meet the

This project is supported in part by the EPFL School of Computer and Communication Sciences as well as the Swiss Science Foundation Project 200021\_197288.

<sup>1</sup><https://blog.adacore.com/quite-proved-image-format>

expectations of the C code generator; it is the generated C code that we use for the performance comparison (Section IV-C).

### B. QOI Format Overview

To encourage subsequent verification efforts and comparisons, we summarize here the QOI format definition. The format is structured with a header, followed by the actual data, and terminated by a marker (7 zero bytes followed by 01<sub>16</sub>). Table I describes the header format. Images are encoded in a row-major order (left-to-right, top-to-bottom).

QOI encoder is single-pass. It manipulates the following data structures:

- The image to encode pixels. Each pixel is constituted of *chan* bytes.
- The current index *pxPos* within pixels (multiple of *chan*), the current pixel *px*, as well as the previous pixel *pxPrev* (initialized to  $R = G = B = 0$  and  $A = 255$ ).
- The encoded image bytes and the output position *outPos* within bytes.
- *index*, an array of 64 pixels denoting previously-seen pixels. It is zero-initialized.
- *run*, counting the number of equal consecutive pixels (initialized to 0).

In the following, we write *px.r*, *px.g*, *px.b*, *px.a* to refer to the red, green, blue, and alpha channels of a pixel *px*. When a pixel does not have an alpha channel, we default *px.a* to 255.

Each pixel is encoded in one of four different cases, two of which have two subcases. Encoded pixels are written in tagged chunks, uniquely identifying the applied (sub)case. The details of the chunk formats and computations can be found in [2].

**Case A.** If  $px = pxPrev$ , we increment the run counter. Whenever it reaches 62, we write a *run chunk*, reset *run* to 0 and continue with the next pixel.

Otherwise, if  $px \neq pxPrev$  and  $run > 0$ , we write a run chunk as well, reset *run* to 0 and proceed to encode *px* using the remaining three methods.

**Case B.** We compute a hash of the current pixel *px*, denoted by  $colorPos(px)$ . The hash function is set by the QOI standard and yields a non-negative number smaller than 64. Then, if  $index(colorPos(px)) = px$ , we write an *index chunk* using the computed position and proceed with the next pixel. Otherwise, we update  $index(colorPos(px))$  with *px* and encode *px* using the two remaining methods.

**Cases C.i and C.ii.** The idea is to encode a difference between the current and previous pixel, provided the difference

TABLE I  
QOI FILE HEADER STRUCTURE. OFFSET AND SIZE ARE GIVEN IN BYTES.

Name	Offset	Size	Description
Magic	0	4	qoif to indicate a QOI image
w	4	4	Image width in pixels (in big-endian)
h	8	4	Image height in pixels (in big-endian)
chan	12	1	Channels: 3 for RGB; 4 for RGBA
Color space	13	1	0: sRGB with linear alpha, 1: all channels linear (informative)

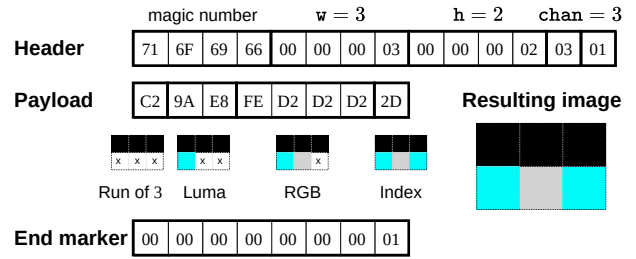


Fig. 1. Example of a Compressed Image in QOI format

is “small enough”. This case comes with two variants: the *diff* subcase (C.i) with a chunk size of 1 byte and the *luma* subcase (C.ii) for larger magnitudes with a chunk size of 2 bytes.

**Cases D.i and D.ii.** Whenever all above cases do not apply, we resort to encoding the full RGB value if  $px.a = pxPrev.a$  (D.i) or the full RGBA value otherwise (D.ii).

Decompression is single-pass as well and maintains the same data structures as the compression counterpart. The decoder iterates over all chunks and applies the reverse transformation.

**Example of decoding an image.** Consider the encoded QOI image depicted in fig. 1. Squares denote bytes in hexadecimal while thick black boxes delimit the chunks. Though this figure actually transcribes the shown  $3 \times 2$  image in the QOI format, knowing the exact details of the computations is unnecessary for this discussion.

The decoder starts with a black and opaque *pxPrev*. It reads the first data byte (C2<sub>16</sub>) and uniquely identifies a run chunk indicating to repeat the previous pixel *pxPrev* 3 times (case A). The decoder then proceeds with the next chunk.

The following 9A<sub>16</sub> signals this byte and the following one, E8<sub>16</sub>, constitutes a luma chunk (case C.ii). The decoder computes a cyan<sup>2</sup> pixel based on the previous pixel and the differences stored in this chunk. Before moving on, this pixel is stored in *index* at the position given by  $colorPos(\cdot)$ .

Next, FE<sub>16</sub> identifies an RGB chunk (case D.i) with three following repeating bytes D2<sub>16</sub>, producing a light gray pixel. The decoder computes a position for this pixel and stores it in *index* (which happens to not collide with the previous cyan pixel).

Finally, 2D<sub>16</sub> specifies an index chunk (case B) with the position of the cyan pixel decoded previously.

### III. VERIFICATION APPROACH

We proved two classes of properties (memory safety is ensured by the programming language model):

- Runtime safety: for any input, the encoder and decoder do not access arrays out of bound or throw exceptions.
- Correctness: decoding is the inverse of encoding (invertibility).

It is much less work to show only the first property, so we focus our presentation on the second one.

<sup>2</sup>Dark gray in monochromatic.

To prove correctness, we proceed by “running” the encoder on an arbitrary but fixed input and decode the image at the same time as it is encoded. Once we are finished, the decoded image must be the same as the original one.

We establish not only separate invariants for the encoder and decoder’s respective states, but also an invariant that ties them. For example, if the encoder encounters a sequence of repeating pixels (case A), it delays writing down the chunk until the end of this sequence. In such a case, the decoder is expected to lag behind the encoder. On the other hand, for cases B, C and D, both the encoder and decoder are expected to advance at the same pace and are, in some sense, synchronized.

Then, given encoder and decoder states satisfying the invariants, we show that encoding a single pixel and decoding it should give the same pixel while maintaining these invariants. We then generalize this result to the entire image, leveraging induction.

To describe invertibility in Stainless, we write plain Scala code in terms of `encode` and `decode`, and provide the appropriate conditions. Before presenting the inversion theorem, we deem it helpful first to introduce some definitions.

The following snippet contains the declarations of three records (or *case classes* in Scala’s terminology). For conciseness, we abbreviate `a: T`, `b: T`, `c: T` to `a`, `b`, `c: T` below.

```
// Encoding context
case class EncCtx(pixels: Array[Byte], w, h, chan: Long) {
  // invariants on the fields (only one conjunct shown)
  require(pixels.length == w * h * chan)
}
case class EncodedResult(encoded: Array[Byte], length: Long)
case class DecodedResult(pixels: Array[Byte], w, h, chan: Long)
```

`EncCtx` contains the input of the encoder: the image (pixels, an array of RGBA bytes) as well as its dimensions and the number of channels. As these values may not be arbitrary (for instance, we must have `pixels.length == w * h * chan`), we add a `require` clause that specifies an invariant over these fields. Stainless then injects these assumptions into proofs when the values of the type appear in verification conditions.

`EncodedResult`, as its name suggests, holds the result of the encoding process. As `encoded` must be big enough to account for the worst case, the `length` field indicates the effective size of the compressed image.

We can now state the “invertibility theorem” with the `decodeEncodelsIdentityThm` function in the snippet below<sup>3</sup>.

```
def encode(ctx: EncCtx): EncodedResult = ...
def decode(bytes: Array[Byte], /* exclusive end index for decoding: */
  until: Long): Option[DecodedResult] = ...

def decodeEncodelsIdentityThm(ctx: EncCtx): Boolean = {
  val res = encode(ctx)
  decode(res.bytes, res.length) match
  case Some(DecodedResult(decoded, w, h, chan)) =>
    w == ctx.w && h == ctx.h && chan == ctx.chan &&
    // Predicate for comparing arrays within a range
    arraysEq(ctx.pixels, decoded, 0, pixels.length)
  case None() => false // i.e. should be unreachable
}.holds
```

<sup>3</sup>For brevity of presentation, code and specification snippets may slightly differ from the actual case study available on the URL shown in the introduction.

The `.holds` construct in `decodeEncodelsIdentityThm` asks Stainless to prove the following. Given a valid `EncCtx` – representing the encoder input – satisfying its stated invariant, if we feed the result `res` of the encoder to the decoder, it always succeeds (by having `case None()` returning `false`). Additionally, the decoded dimensions and number of channels correspond to the original input. Furthermore, the original and decoded images are equal.

To help Stainless prove this theorem, we must establish contracts for several functions, provide sufficient proof annotations to guide the solver, and write lemmas – which are just (possibly recursive) functions stating a property. However, `decodeEncodelsIdentityThm` does not contain any proof annotation, as everything needed to derive the conclusion is contained in the definitions of `encode` and `decode`.

In fact, `encode` and `decode` contain few annotations. They delegate the work (alongside the proofs) to `encodeLoop` and `decodeLoop`. In particular, `encodeLoop` iterates (through recursion) over the pixels and invokes `encodeSingleStep` for the actual work. By stating a sufficiently strong induction hypothesis (IH) on `encodeLoop` and combining the IH with the properties of `encodeSingleStep`, we obtain proof of invertibility.

As `encodeLoop` is “just” gluing the pieces together, we instead present `encodeSingleStep`:

```
// Pixel read from the pixels array, updated output
// position within the bytes array and updated run.
case class EncodingIteration(px: Int, outPos, run: Long)

// Contains the state of the decoder, that is mutated
// in encodeSingleStep ('var' marks a field as mutable).
case class GhostDecoded(var index: Array[Int],
  var pixels: Array[Byte], var inPos, var pxPos: Long)

def encodeSingleStep(index: Array[Int], bytes: Array[Byte],
  pxPrev: Int, run0, outPos0, pxPos: Long, ctx: EncCtx,
  @ghost decoded: GhostDecoded): EncodingIteration = // ...
```

`encodeSingleStep` returns `EncodingIteration` that gives the last read pixel (`px`) and one-past-the-end position of the last written byte (`outPos`). For a sequence of repeating pixels, the `run` field of the returned record is incremented. Otherwise, the encoded pixels are written (in-place) in bytes and `outPos` is updated accordingly.

Notably, `encodeSingleStep` takes a ghost parameter, `decoded`, which models the decoder state that would arise during possible future decoding runs. Ghost variables are subject to ghost elimination, which we discuss in more detail in IV-C. Intuitively, ghost variables allow tracking some extra information that may only be used for contracts and proof annotations: in particular, they cannot influence the execution of the algorithm [15].

The precondition of `encodeSingleStep` requires that the decoder state is consistent: for instance, the currently decoded pixels correspond to the original ones. At the end of the function, before returning, we “run” the decoder on `decoded` by calling `decodeLoop` with the updated index and bytes arrays.

Then, we can express local invertibility as follows. If we run the decoder from the *old* `decoded` state (i.e. before entering `encodeSingleStep`) on the bytes we wrote when executing

encodeSingleStep, then the decoded pixels must correspond to the pixels that have been encoded.

To prove this key property, we proceed in two phases, akin to how the encoder proceeds. The snippet below shows an excerpt of the encodeSingleStep, highlighting these two phases.

```
// Record returned by updateRun
case class RunUpdate(reset: Boolean, run, outPos: Long)

def encodeSingleStep(...) = {
  // ... Some preconditions
  // A copy of the "original" index, will be erased by ghost elimination:
  @ghost val oldIndex = freshCopy(index)
  // Phase 1: Run-length processing (case A)
  val runUpd = updateRun(bytes, run0, outPos0)
  val run1 = runUpd.run
  val outPos1 = runUpd.outPos
  // The premise holds when flushing (writing down the run chunk)
  assert(runUpd.reset ==>
    updateRunProp(pxPrev, px, bytes, run0, outPos0, outPos1))
  // ... other assertions
  // Phase 2: Encode pixel individually (cases B, C, D)
  val outPos2 = if px != pxPrev then
    val outPos2 = encodeNoRun(index, bytes, outPos1)
    // ... some assertions and lemmas to support this claim
    assert(encodeNoRunProp(pxPrev, px, oldIndex, index, bytes,
      outPos1, outPos2))
  outPos2
else
  // ... assertions stating invariants are preserved
  outPos1
  // ... assertions to glue everything together
  EncodingIteration(px, outPos2, run1)
} ensuring /* postconditions stating distilled properties */
```

First, the encoder handles the run-length part of the algorithm, corresponding to case A as described in II-B. The work is delegated to updateRun and returns a record telling (through the reset field) whether a run chunk was written to bytes. If not, then invertibility is of course preserved as the encoded pixels are left untouched. Otherwise, updateRun guarantees that reading the written chunk gives us a run chunk whose value is the run counter we have just written – expressed with updateRunProp, presented afterward.

Second, in the case where the previous and current pixels are different, the encoder picks methods B, C or D to encode the current pixel. The task is handed over to encodeNoRun and states with encodeNoRunProp that reading the written chunk yields back the pixel.

updateRunProp and encodeNoRunProp both use doDecodeNext to decode the written chunk. The latter returns an ADT with two variants describing the decoded chunk. Run( $r$ ) indicates a run chunk with  $r + 1$  repeating pixels. The  $+1$  is a result of the run counter being shifted by one when encoded. DiffOrIndexOrColor( $px$ ) denotes a pixel encoded by method B, C or D. Due to the variable length nature of chunks, doDecodeNext also returns the position of the next chunk to be decoded (if any).

```
enum DecodedNext:
  case Run(run: Long)
  case DiffOrIndexOrColor(px: Int)

def doDecodeNext(bytes, index: Array[Int],
  pxPrev: Int, inPos0: Long): (DecodedNext, Long) = ...
```

Expressing the desired properties is then a matter of pattern-matching over the result of doDecodeNext and tying it with appropriate equalities.

```
def updateRunProp(pxPrev, px: Int, bytes: Array[Byte],
  run0, outPos0, outPos1: Long): Boolean =
  // ... preconditions including e.g. ordering on outPos0, outPos1
  // If px == pxPrev, the current run counter run0 is incremented
  // (reflected by the conditional +1).
  val run = run0 + bool2int(px == pxPrev)
  // The index does not matter for this case, we give an arbitrary array
  val dummyIndex = Array.fill(64)(0)
  doDecodeNext(bytes, dummyIndex, pxPrev, outPos0) match
    case (Run( $r$ ), inPos) =>  $r + 1 == run \ \&\& \ inPos == outPos1$ 
    case _ => false

// oldIndex refers to the index at the beginning of encodeSingleStep
def encodeNoRunProp(pxPrev, px: Int, oldIndex, index: Array[Int],
  bytes: Array[Byte], outPos1, outPos2: Long): Boolean =
  // ... preconditions including e.g. ordering on outPos1, outPos2
  doDecodeNext(bytes, oldIndex, pxPrev, outPos1) match
    case (DiffOrIndexOrColor(decodedPx), inPosRes) =>
      decodedPx == px \ \&\& \ inPosRes == outPos2 \ \&\& \
      oldIndex.updated(colorPos(px), px) == index
    case _ => false
```

We rely on Inox (Stainless’ underlying solver) to unfold function definitions to prove that the calls to updateRunProp and encodeNoRunProp in encodeSingleStep hold. To help with the proof, we also provide assertions whose content is similar to the properties stated by updateRunProp and encodeNoRunProp.

Now that we have these two invertibility properties, we show that the composition of these two phases preserves invertibility by tying all facts together (see the end of the body of encodeSingleStep in the source code of encoder.scala).

## IV. RESULTS

We first present some statistics and remarks about the verification before considering the performance of the generated C code with respect to the reference implementation.

For all experiments, we used a server with  $2 \times$  Intel® Xeon® CPU E5-2680 v2 at 2.80GHz (release date Q3’13, for a total of 20 physical cores) running on Ubuntu 20.04.3 LTS.

### A. Verification Statistics

Our QOI implementation in Scala without annotations consists of 313 lines of code (LOC)<sup>4</sup>. The annotated version has 2789 LOC, of which 1405 are for lemmas and helpers. This yields a ratio of 8.9 lines of specifications per executable line. The specification lines include 42 lemmas, 19 of which are general purpose and could become part of the standard library.

Table II shows for each category of verification condition (VC) their respective numbers and their cumulative times. It took roughly 1h30min to verify all VCs. The lower quartile, the median, and the upper quartile are 0.5s, 1.8s, and 5.7s respectively. Around 9.5% of VCs took more than 30s to verify, the highest being 3min.

For each function call, Stainless generates VCs corresponding to the function preconditions. Assertions annotations and postconditions of functions are translated into VCs as well.

<sup>4</sup>Counted with `cloc v1.82`

TABLE II  
SUMMARY OF THE VERIFICATION CONDITIONS.

Verification Condition	#	Total time [min]
Preconditions	2387	370.9
Body assertions	787	203.3
Postconditions	145	31.2
Array index within bounds	126	4.9
Remainder by zero	87	10.6
Non-negative measure	23	2.1
Class invariant	21	1.5
Cast correctness	6	0.1
Match exhaustiveness	5	0.4
Measure decreases	4	4.4
<b>Total</b>	<b>3591</b>	<b>629.4</b>

Stainless furthermore generates other runtime safety verification conditions, such as array bounds checks and remainder by zero checks. It is sometimes necessary to provide sufficient annotations (e.g., assertions and invariants) to help Stainless prove these VCs.

### B. Verification Effort

The case study was implemented and formally verified by the first author (who had a few months of experience with Stainless) over the period of approximately 4 to 5 weeks.

We have first implemented a version closely following the C reference version. Though we could prove runtime safety, describing deeper properties turned out to be difficult. For example, we could not refer to the result of decoding a range, but only the end-to-end decompression result of the entire image.

We have thus rewritten the implementation multiple times making both small and larger changes. Since the encoder and decoder are succinct, the rewrites took a relatively small amount of time compared to the remaining verification effort.

During repeated verification runs, the VC cache and the ability to selectively verify only provided functions greatly speed up the interactive experience. For example, making a few changes to a previously verified version requires less than two minutes to check all VCs, compared to the 1h30min for a clean-state re-run.

### C. Generated C Code and Its Efficiency

We compare the encoding and decoding throughput of the transpiled C code with the reference implementation. Though the primary goal of the reference is simplicity, its decoding and encoding throughput are respectively 3.4x and 29x higher than `libpng` while achieving a similar compression ratio<sup>5</sup>.

As briefly mentioned in IV-B, we make use of ghost states for proving invertibility. Stainless first checks for correct usage of ghost variables before eliminating them in a phase of the C transpiler. Assertions and functions contracts are removed as well<sup>6</sup>. In summary, “proof infrastructure” is erased and incurs no cost at runtime.

<sup>5</sup>Derived from the section “Grand total for images (AVG)” at <https://qoiformat.org/benchmark/> (consulted the 11.08.2022).

<sup>6</sup>To ensure removal, developers should import the `StaticChecks` library.

The generated C code is 661 LOC long, against 311 for the reference implementation. For the purpose of evaluation, we also wrote unverified glue C code that performs I/O. We do not make any correctness claims about this code, only about the part that converts arrays of bytes between uncompressed and compressed form. We evaluated the throughput of the generated C code (`genc-qoi`) against the reference implementation (`qoi`) using a modified version of the benchmark utility shipped with `qoi`. We run the benchmark with 3 runs over 7 images ranging from 3 to 13.8 megapixels, and report the result in table III.

We compiled all involved C sources using GCC 11.1.0 with `-O3`. As our implementation uses tail recursion, so does the generated C code<sup>7</sup>. It is necessary to pass an optimization level of at least `-O2` or explicitly pass the `-foptimize-sibling-calls` to GCC in order to have the tail calls eliminated.

To our surprise, the transpiled version is on-par with the reference implementation: it is approximately 7% faster in decoding and 2% slower in encoding. Disassembling the decoding functions reveals that both were compiled similarly. Nevertheless, the `genc-qoi` version uses more instructions for all cases but index decoding (case B). These extra instructions are of an arithmetic and logical nature and do not involve memory operations. For case B, GCC produced one 4-bytes memory load operation for `genc-qoi`, while it emitted four 1-byte memory load operations for `qoi`. We conjecture that the reported difference may be explained by these three extra memory loads.

TABLE III  
BENCHMARK RESULTS OF QOI AND GENC-QOI

	Decoding throughput [megapixels/s]	Encoding throughput [megapixels/s]
qoi (unverified)	90.92	86.24
genc-qoi (verified)	97.65	84.45

## V. CONCLUSIONS

We have presented a QOI implementation in Scala and verified with Stainless that decoding is the inverse of encoding. We have also seen that the transpiled C version matches the performance of the reference implementation. Going forward, we expect that other verified implementations will emerge and that QOI will become a useful benchmark for testing verification approaches and tools.

## ACKNOWLEDGMENT

We thank FMCAD 2022 reviewers for helpful comments. We thank Georg S. Schmid for useful discussions and Jad Hamza for developing the C code generator in Stainless. We thank the organizers of ASPLOS 2022 conference for the opportunity to present a summary of the case study as one part of the tutorial.

<sup>7</sup>We thank GCC! Our C code generator does not (yet) eliminate tail calls.

## REFERENCES

- [1] A. Kanade, R. Alur, S. Rajamani, and G. Ramanlingam, “Representation dependence testing using program inversion,” in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 277–286. [Online]. Available: <https://doi.org/10.1145/1882291.1882332>
- [2] “The Quite OK Image format for fast, lossless compression.” [Online]. Available: <https://qoifformat.org/>
- [3] “Verifying programs with Stainless (ASPLOS 2022 tutorial on Stainless).” [Online]. Available: <https://epfl-lara.github.io/asplos2022tutorial/>
- [4] L. P. Deutsch, “DEFLATE Compressed Data Format Specification version 1.3,” Internet Engineering Task Force, Request for Comments RFC 1951, May 1996, num Pages: 17. [Online]. Available: <https://datatracker.ietf.org/doc/rfc1951>
- [5] C.-S. Senjak and M. Hofmann, “An implementation of deflate in coq,” 2016. [Online]. Available: <https://arxiv.org/abs/1609.01220>
- [6] R. Affeldt, J. Garrigue, and T. Saikawa, “Examples of Formal Proofs about Data Compression,” in *2018 International Symposium on Information Theory and Its Applications (ISITA)*. Singapore: IEEE, Oct. 2018, pp. 633–637. [Online]. Available: <https://ieeexplore.ieee.org/document/8664276/>
- [7] Q. Ye and B. Delaware, “A verified protocol buffer compiler,” in *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 222–233. [Online]. Available: <https://doi.org/10.1145/3293880.3294105>
- [8] R. Edelmann, “Efficient parsing with derivatives and zippers,” Ph.D. dissertation, EPFL, Lausanne, 2021. [Online]. Available: <http://infoscience.epfl.ch/record/287059>
- [9] M. Hofmann, B. Pierce, and D. Wagner, “Symmetric lenses,” in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 371–384.
- [10] J. Hamza, N. Voirol, and V. Kunčák, “System FR: Formalized foundations for the Stainless verifier,” *Proc. ACM Program. Lang*, no. OOPSLA, November 2019.
- [11] V. Kuncak and J. Hamza, “Stainless verification system tutorial,” in *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*. IEEE, 2021, pp. 2–7.
- [12] “Stainless,” 2022. [Online]. Available: <https://github.com/epfl-lara/stainless/>
- [13] M. Odersky, L. Spoon, B. Venners, and F. Sommers, *Programming in Scala (Fifth Edition, Updated for Scala 3.0)*. Artima Press, 2021.
- [14] J. Hamza, S. Felix, V. Kunčák, I. Nussbaumer, and F. Schramka, “From verified Scala to STIX file system embedded code using Stainless,” in *NASA Formal Methods (NFM)*, 2022, p. 18. [Online]. Available: <http://infoscience.epfl.ch/record/292424>
- [15] M. Abadi and L. Lamport, “The existence of refinement mappings,” in *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*, July 1988, pp. 165–175, IICS 1988 Test of Time Award. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/the-existence-of-refinement-mappings/>