Dependency Assessment: Binary

Prepared by FP Complete

For The Cardano Foundation

CARDANO FOUNDATION

July 2018

Legend

The following report uses a grading system that provides a subjective rating of the impact, significance and execution of the audited aspects. This is intended to assist the audience's understanding.

Flag Severity

Green Well executed in the project

Yellow Potential flaw, or process limitation which may lead to a flaw

Red Confirmed flaw

Question Scoring

Good Meets or exceeds industry standards.

Fair Partial implementation does not meet industry standards.

Poor No implementation.

This report pertains to the following commit:

`github.com/kolmodin/binary: 333f8366f86a5af86062e94897eec8ef9aee4d1d`

Summary

Questions

Title	Links
General	
Is the library mature?	Details
Is the library the standard solution for the problem it solves?	Details
Does the library have a reputation?	Details
Is the library well-maintained?	Details
Quality	
Does it have CI?	Details
Does it have 90-100% code coverage?	Details
A test suite with property-based testing?	Details
Has a test suite that is easy to run?	Details
Aggressive test suite?	Details
Does it have benchmarks?	Details

Flags

Title	Links	Status
Bugs		
Known bug: Roundtrip property does not hold for NaN	Details	
Vulnerabilities		
Deserialisation of nullary types makes the decoder vulnerable to a DoS attack	Details	
The Map/IntMap/IntSet instances do not check preconditions.	Details	
Performance Issues		
Expensive bytestring copying for backtracking	Details	
Expensive strictness for lazy `ByteString` in `pushChunks`	Details	
Maintenance		
Undocumented functions	Details	

Title	Links	Status
Claim about flushing the current buffer	Details	
Test suite use modules from `src` directly	Details	
No tests for `Binary` instances for `Generic` type class	Details	
Code		
Partial instance for NonEmpty	Details	

Flags

Detailed flags: Bugs

Known bug: Roundtrip property does not hold for NaN

See https://github.com/kolmodin/binary/issues/64

```
> (decode (encode (0 / 0 :: Double)) :: Double)
-Infinity
it :: Double
(0.05 secs, 609,616 bytes)
> 0 / 0 :: Double
NaN
it :: Double
(0.00 secs, 548,944 bytes)
```

First occurrence: Monday 2 July 2018

Detailed flags: Vulnerabilities

Deserialisation of nullary types makes the decoder vulnerable to a DoS attack

Instances that work with arrays, lists, sequences are all prefixed with a length indicator. For a list of a unary sum type like (), you can specify any arbitrary length you like, making the decoder vulnerable to denial of service attacks.

E.g. with just 8 bytes we could produce decoded list with size of 1000000 elements:

```
λ> import Data.Binary
λ> import qualified Data.ByteString.Lazy as BL
λ> bs = encode (1000000::Int)
λ> BL.length bs
8
λ> length $ (decode bs :: [()])
1000000
```

We recommend to fix serialisation of nullary types or advise explicitly against their use with `binary`

First occurrence: Monday 2 July 2018

The Map/IntMap/IntSet instances do not check preconditions.

The `Map/IntMap/IntSet` instances do not check preconditions for constructing the tree, they use e.g. `fromDistinctAscList` which is more performant but can be manipulated to produce an inconsistent data structure.

An example of this could be seen in the following snippet:

```
λ> import qualified Data.Map as M
λ> import Data.Binary
```

```
λ> m = (decode (encode [(2::Int,0::Int),(2,7),(1,1),(0,2)]) :: M.Map In
t Int)
λ> M.size m
4
λ> M.lookup 1 m
Nothing
λ> M.lookup 2 m
Just 7
λ> M.lookup 0 m
Nothing
λ> M.keys m
[2,2,1,0]
λ> M.split 2 m
(fromList [(2,0)],fromList [(1,1),(0,2)])
```

First occurrence: Monday 2 July 2018

Detailed flags: Performance Issues

Expensive bytestring copying for backtracking

Backtracking functions (<|>), `lookAhead` and `isolate` doing it internally do expensive bytestring operations (with O(n) complexity and new bytestring allocations) - `Data.ByteString.concat` and `Data.ByteString.append`.

It is a known problem and for example the `attoparsec` library use special data type `Buffer` which provides asymptotically fast appends.

First occurrence: Monday 2 July 2018

Expensive strictness for lazy `ByteString` in `pushChunks`

`pushChunks` causes the rest of the lazy bytestring to be forced strictly, because as a leftover it constructs a strict bytestring from the lazy input string. If you parse one byte from a 1GB file, it will allocate 1GB just to produce the single byte.

This can be show using the following test program:

```
-- | pushChunks bug test case.

module Main where

import qualified Data.Binary.Get as Binary
import qualified Data.ByteString.Lazy as L

main :: IO ()
main = do
  lazyBytes <- L.readFile "file.txt"
  case Binary.pushChunks
         (Binary.runGetIncremental Binary.getWord8)
         lazyBytes of
    Binary.Done _rest _offset byte -> putStrLn ("Byte: " ++ show byte)
    Binary.Fail {} -> putStrLn "Fail"
```

Which shows output similar to the following:

```
bash-3.2$ stack ghc -- audit/push-chunks-test.hs -O -o push-chunks-test -rt
[1 of 1] Compiling Main             ( audit/push-chunks-test.hs, audit/push
Linking push-chunks-test ...
bash-3.2$ mkfile -n 1g file.txt
bash-3.2$ ./push-chunks-test file.txt +RTS -s
Byte: 0
   2,180,932,296 bytes allocated in the heap
       8,345,272 bytes copied during GC
   1,076,382,128 bytes maximum residency (11 sample(s))
     133,798,480 bytes maximum slop
            2240 MB total memory in use (40 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pau
  Gen  0      2040 colls,     0 par    0.009s   0.011s     0.0000s    0.000
  Gen  1        11 colls,     0 par    0.013s   0.277s     0.0252s    0.265

  INIT    time    0.000s  (  0.002s elapsed)
  MUT     time    0.276s  (  0.927s elapsed)
  GC      time    0.022s  (  0.288s elapsed)
  EXIT    time    0.003s  (  0.265s elapsed)
  Total   time    0.303s  (  1.483s elapsed)

  %GC     time       7.3%  (19.4% elapsed)

  Alloc rate    7,894,406,040 bytes per MUT second

  Productivity  92.7% of total user, 80.4% of total elapsed

bash-3.2$
```

First occurrence: Monday 2 July 2018

Detailed flags: Maintenance

Undocumented functions

The functions `withInputChunks`, `failOnEOF`, `putBuilder` are exported from the package they have not documentation.

First occurrence: Monday 2 July 2018

Claim about flushing the current buffer

The `putByteString` function claims to flush the current buffer, but nothing in the bytestring `Builder` docs seems to support this.

We recommend to fix the function haddock - either to remove words about flushing or rephrase it in a way so it will be explicit what this flushing is.

First occurrence: Monday 2 July 2018

Test suite use modules from `src` directly

The cabal file of the project uses library modules directly and not library as a dependency.

This prevents `hpc` from producing proper coverage report and to create it changes in cabal file are required.

First occurrence: Monday 2 July 2018

No tests for `Binary` instances for `Generic` type class

The library has a special module implementing `Binary` instances for types having instance of `Generic` type class.

There are benchmark testing this code but at the same time no tests checking it were found.

First occurrence: Monday 2 July 2018

Detailed flags: Code

Partial instance for NonEmpty

The type class instance for `Binary` of `NonEmpty` is partial because of using partial function `Data.List.NonEmpty.fromList`.

@src/Data/Binary/Class.hs:832

```
-- | /Since: 0.8.4.0/
instance Binary a => Binary (NE.NonEmpty a) where
  get = fmap NE.fromList get
  put = put . NE.toList
```

This could result in an error which won't get raised and caught in context of the `MonadFail` type class instance of `Get` :

```
λ> import Data.List.NonEmpty
λ> import Data.Binary
λ> import Data.Binary.Get
λ> runGetOrFail (get :: Get (NonEmpty Word)) (encode ([]:: [Word]))
Right ("",8,*** Exception: NonEmpty.fromList: empty list
```

First occurrence: Monday 2 July 2018

Questions

Questions in Detail

General

Is the library mature?

Answer:

Yes, it was first published in 2007 and has been maintained since.

Score: Good

Source: source-code

Is the library the standard solution for the problem it solves?

Answer:

YES, many projects have used this package since 2007.

Score: Good

Source: source-code

Does the library have a reputation?

Answer:

YES. Subjectively, the audit team observes that the general view of the `binary` package is stable and usable. However, it is also generally reputed to not be the fastest serialization library in Haskell.

Score: Good

Source: source-code

Is the library well-maintained?

Answer:

YES and NO, it has been well-maintained for years, but recently has not been touched since 2017 Nov, and there are some issues and PRs open that are not being tended to. Much discussion involves the “upcoming” cbor package.

Score: Good

Source: source-code

Quality

Does it have CI?

Answer:

YES, but it is NOT PASSING. See https://travis-ci.org/kolmodin/binary/builds/352415458.

Score: Fair

Source: source-code

Does it have 90-100% code coverage?

Answer:

The coverage is not quite close to 100% but appears to be not bad:

```
72% expressions used (2026/2786)
52% boolean coverage (13/25)
     50% guards (11/22), 9 always True, 1 always False, 1 unevaluated
     66% 'if' conditions (2/3), 1 unevaluated
    100% qualifiers (0/0)
63% alternatives used (133/209)
85% local declarations used (34/40)
73% top-level declarations used (261/356)
```

Score: Good

Source: source-code

A test suite with property-based testing?

Answer:

Yes. In `tests/QC.hs`, there are many property tests. These tests mostly check one of the two possible round-trip properties: (pseudo code)

```
forAll arbitrary $ \a -> decode (encode a) == a
```

The following round-trip property in the opposite direction is not tested: (pseudo code)

```
forAll arbitrary $ \a -> (encode (decode bs)) == bs
```

Score: Good

Source: source-code

Has a test suite that is easy to run?

Answer:

Yes. All it takes is to run `stack init && stack test`.

Score: Good

Source: source-code

Aggressive test suite?

Answer:

Yes.

- In `tests/QC.hs`, there is the `mustThrowError` combinator.
- In `tests/ACtion.hs`, there are the `prop_fail` and `prop_label` properties.

Score: Good

Source: source-code

Does it have benchmarks?

Answer:

Yes. In the `benchmarks` directory, there are very thorough benchmarks. In fact, there are as many lines of benchmarks as there is code in the `binary` repository.

Score: Good

Source: source-code

Report Statistics

	Public	Private	Total
Red flags	6	0	6
Yellow flags	4	0	4
Green flags	0	0	0
Total Flags	10	0	10

Question Score	Total
Good	9
Good	1
Poor	0
Un-Scored	0
Total	10