The text package: finally with UTF-8

Andrew Lelechenko
andrew.lelechenko@gmail.com

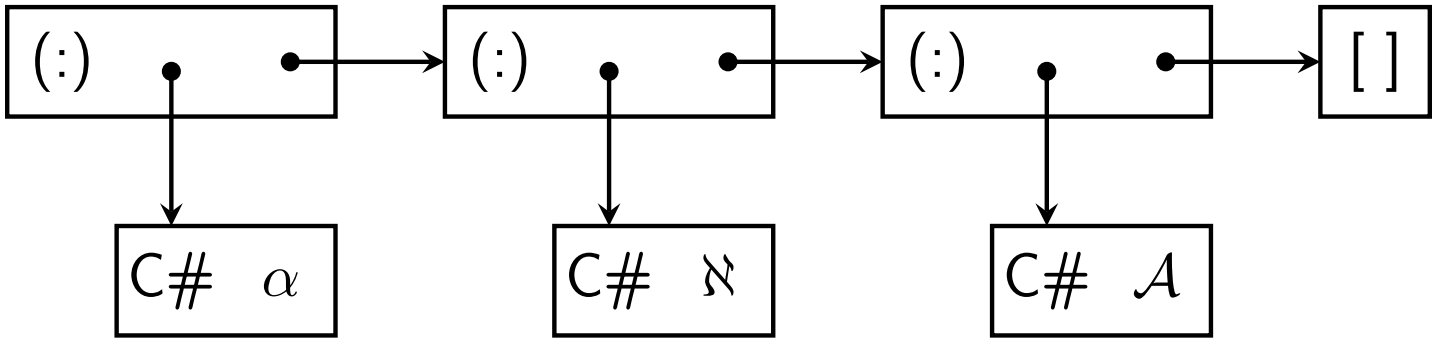
Barclays, London

ZuriHac, 12.06.2022

String

```
type String = [Char]
```

- **Pros:** supports full Unicode range, e. g., "$\alpha\aleph\mathcal{A}$".
- **Cons:** allocates two heap objects and 40 bytes per character.



ByteString

```
data ByteString = BS !(ForeignPtr Word8) -- payload
                     !Int                -- length

data ShortByteString = SBS ByteArray#
```

- **Pros:** allocates a single heap object.
- **Pros:** spends 1 byte per character.
- **Cons:** does not support Unicode, only ASCII subset.

- ASCII assigns 128 code points, each encoded as 1 byte: newline, space, punctuation, digits, Latin letters.
- Other 8-bit encodings extend ASCII up to 256 code points.

Encodings

- ASCII assigns 128 code points, each encoded as 1 byte: newline, space, punctuation, digits, Latin letters.
- Other 8-bit encodings extend ASCII up to 256 code points.
- Unicode unifies multiple regional and national encodings into a single character set.
- 256 code points are not enough to fit the entire Unicode.
- Well, surely, 64K ought to be enough for anybody!
- Early Unicode embraced UCS-2 encoding, assigning up to 64K code points, each encoded as 2 bytes.
- Guess what happened next?

Unicode history

Ver.	Date	#chars	Ver.	Date	#chars
1.0.0	Oct 1991	7 129	6.0	Oct 2010	109 384
1.0.1	Jun 1992	28 327	6.1	Jan 2012	110 116
1.1	Jun 1993	34 168	6.2	Sep 2012	110 117
2.0	Jul 1996	38 885	6.3	Sep 2013	110 122
2.1	May 1998	38 887	7.0	Jun 2014	112 956
3.0	Sep 1999	49 194	8.0	Jun 2015	120 672
3.1	Mar 2001	94 140	9.0	Jun 2016	128 172
3.2	Mar 2002	95 156	10.0	Jun 2017	136 690
4.0	Apr 2003	96 382	11.0	Jun 2018	137 374
4.1	Mar 2005	97 655	12.0	Mar 2019	137 928
5.0	Jul 2006	99 024	12.1	May 2019	137 929
5.1	Apr 2008	100 648	13.0	Mar 2020	143 859
5.2	Oct 2009	107 296	14.0	Sep 2021	144 697

UTF-32, UTF-16, UTF-8

- UTF-32 aka UCS-4: each code point is encoded as 4 bytes:
  - Fixed width encoding = trivial decoding.
  - Incompatible both with ASCII and UCS-2.
- UTF-16: the majority of code points are encoded as 2 bytes, but some as 4 bytes:
  - Almost fixed width encoding = simple decoding.
  - Compatible with UCS-2.
  - Incompatible with ASCII.
- UTF-8: ASCII code points are encoded as 1 byte, and others as 2, 3 or 4 bytes:
  - Variable width encoding = complicated decoding.
  - Compatible with ASCII.

The text package

Developed by Thomas Harper (Fusion on Haskell Unicode Strings, MSc. Thesis, Oxford, 2008). The main goal was to "create a compact representation for Unicode strings that can take advantage of stream fusion".

Ch. 4.1.4. Encoding shootout

"...UTF-8, while more compact in many cases, pays for its complexity. ...Even though UTF-32 is faster than UTF-16, sometimes much faster, its large footprint significantly reduces its usability. In many cases, the distance between UTF-32 and UTF-16 is still far smaller than the gap between UTF-16 and UTF-8, making it harder to justify doubling the space needed for a string".

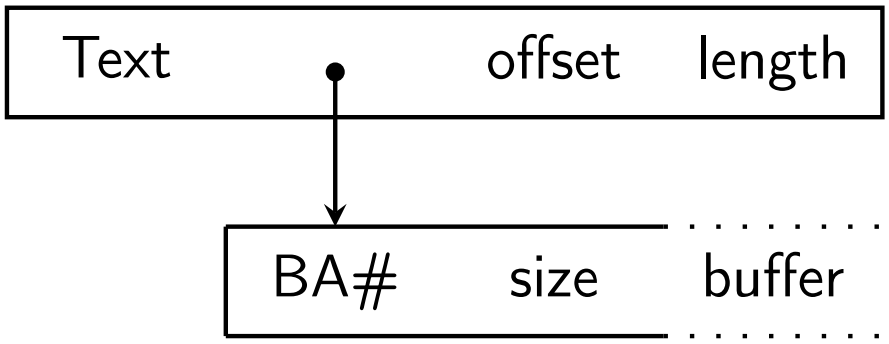
The slowdown is correctly attributed to more complex branching and worse branch prediction for UTF-8 comparing to UTF-16.

Why migrate from UTF-16 to UTF-8?

- In 2011 Jasper Van der Jeugt submitted a GSoC proposal "Convert the `text` package to use UTF–8 internally".
- The majority of the real world data is stored and transferred as UTF-8, because of the graceful degradation to ASCII.
- Harper's benchmarks measured performance of `text` transformations only *after* `Text` was read and decoded from a file and *before* it was written back.
- For mostly-ASCII data UTF-8 takes almost 2x less memory than UTF-16.
- The results, summarized by Jasper in "Text/UTF-8: Aftermath", were inconclusive. The branch got abandoned and by 2021 was $\sim$ 800 commits behind `HEAD`.

UTF-8 challenges

- Not every byte sequence is a valid UTF-8. In 2011 validation of UTF-8 used to be almost as expensive as conversion to UTF-16.
- The fusion framework mandates that any materialized `Text` is decoded into `Stream Char` and encoded back, which is more expensive for UTF-8.
- Memory savings are less than 50%, especially for short data:



Five years later, 2016

- A renewed effort to migrate from UTF-16 to UTF-8 started in 2016, led primarily by Kubo Kováč, Alexander Biehl and Herbert Valerio Riedel.
- The authors rebased parts of Jasper's work in a bulk commit and continued from that point onward, accumulating $\sim$ 100 commits.
- The work came to a halt in 2018 and by 2021 was $\sim$ 200 commits behind `HEAD`.
- Benchmarks show severe regressions in certain disciplines.

Emoji-driven development, 2021

- In March 2021 CLC revamped the team of `text` maintainers.
- Emily Pillmore and I submitted an HF proposal to migrate `text` from UTF-16 to UTF-8, which was approved by the board in May 2021.
- I started the implementation from the scratch and completed it by the end of August.
- The pull request was reviewed and merged in September.
- I prepared a series of release candidates in November, culminating with a release of `text-2.0` on Christmas night.

- What made the attempt #3 more lucky?

The most important principle

Invest in feedback loop!

Unholy trick to avoid circular dependencies

```
name: text

library
  hs-source-dirs:  src
  exposed-modules: Data.Text

test-suite text-tests
  main-is:         Tests.hs
  hs-source-dirs: src tests
  build-depends:  base, test-framework
  -- but not text, because test-framework depends on it
  other-modules:  Data.Text
```

Invest in feedback loop: tests

- `test-framework` is used to support GHC 7.0.
- But it has plenty of dependencies and is not actively developed.
- The closest replacement is `tasty`.
- So I worked on `tasty`
  - to extend compatibility range back to GHC 7.0;
  - to purge as many dependencies as possible.
- This succeeded, and now `tasty` has the lowest dependency footprint among Haskell testing frameworks.

Invest in feedback loop: benchmarks

- Benchmarks are even worse: `criterion` depends on `text` transitively via multiple packages.
- `gauge` does not depend on `text`, but often lags behind the most recent GHC.
- We already have `tasty`, which does not depend on `text`. Can we upgrade it to become a benchmarking framework?
- It appears `tasty` has a modular architecture, and benchmarking can be implemented as a plugin.
- I developed `tasty-bench`:
  - extremely lightweight and flexible,
  - a drop-in replacement for `criterion` and `gauge`,
  - offers built-in comparison between runs and between benchmarks, invaluable for quick feedback.

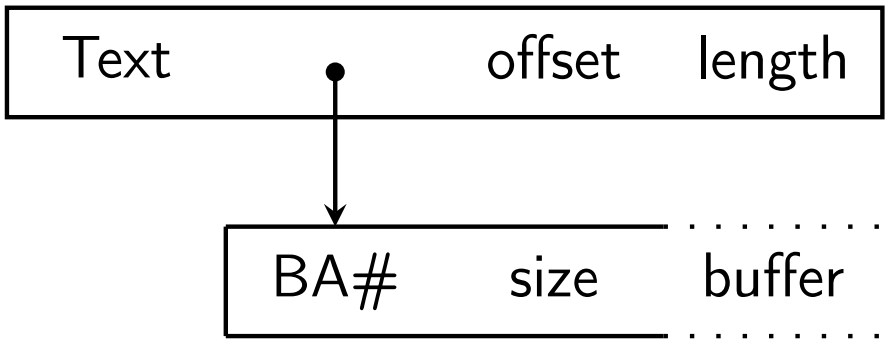
Invest in CI heavily

`text` accumulated CI jobs for:

- each supported version of GHC,
- each build flag enabled and disabled,
- Ubuntu, Windows, MacOS, FreeBSD, CentOS,
- old Ubuntu releases,
- i386, amd64, arm64, ppc64le, s390x,
- IBM mainframes.

UTF-8 challenges

- Not every byte sequence is a valid UTF-8. In 2011 validation of UTF-8 used to be almost as expensive as conversion to UTF-16.
- The fusion framework mandates that any materialized `Text` is decoded into `Stream Char` and encoded back, which is more expensive for UTF-8.
- Memory savings are less than 50%, especially for short data:



Three engines for UTF-8 validation

- `simdutf` library.
  - Daniel Lemire, Wojciech Muła, Transcoding Billions of Unicode Characters per Second with SIMD instructions, Software: Practice and Experience, Sep 2021.
  - Implemented in C++, hard to link from Haskell.
  - Returns a boolean flag.
- `isValidUtf8` function.
  - Developed by Koz Ross, in-kind donation from MLabs to HF.
  - Implemented in C, a part of `bytestring-0.11.2.0`.
  - Returns a boolean flag.
- Native state machine.
  - Adapted from earlier UTF-8 decoding routine in C.
  - Translated to Haskell.
  - Returns a state of decoder.
  - Suitable for malformed and partial data.

Fusion of Text

- Text has two equivalent representations: as a materialized buffer `Text` and as a `Stream` of `Chars`.
- Some functions have two versions: one for `Text` and one for `Stream Char`.
- But most were implemented as wrappers:

```
foo :: Text → Text
foo = unstream . fooS . stream

stream :: Text → Stream Char
unstream :: Stream Char → Text
fooS :: Stream Char → Stream Char
```

- Each fused transformation pipeline needed to decode and encode UTF-8 at least once.

Tale of two tails

`tail` takes $O(1)$ time, but `tail . tail` takes $O(n)$ time.

```
tail :: Text → Text
tail (Text arr off len) =
  Text arr (off + delta) (len - delta)
    where
      delta = iter_ t 0

{-# RULES
  "fuse tail" [~1] tail = unstream . S.tail . stream
  "unfuse tail" [1] unstream . S.tail . stream = tail
  "fusion" stream . unstream = id
  #-}
```

So `tail` remains `tail` after optimizations, but `tail . tail` becomes `unstream . S.tail . S.tail . stream`, decoding everything in full.

No implicit fusion

- Long pipelines of `Text` transformations are rare and unlikely to bottleneck because of lack of fusion alone.
- Instead we need to focus on slicing and concatenation, which are precisely where uncontrolled implicit fusion is harmful.
- Let's disable implicit fusion and rewrite every function to operate on buffers directly.
- I spend
  - one night implementing a full switch from UTF-16 to UTF-8,
  - and three months rewriting routines to avoid streaming.

Performance after transition to UTF-8

- `decodeUtf8` is up to 10x faster for non-ASCII texts.
- `encodeUtf8` is 1.5–4x faster for strict, 10x faster for lazy Text.
- `take` / `drop` / `length` are up to 20x faster.
- `toUpper` / `toLower` are 10-30% faster.
- `Ord` instance is typically 30%+ faster.
- `isInfixOf` and search routines are up to 10x faster.
- `replicate` of `Char` is up to 20x faster.

Geometric mean of relative benchmark times is 0.33
= an average benchmark is 3x faster.

Migration efforts

Multiple libraries rely on the internal representation of `Text`:

- `attoparsec` reuses parts of `Text` to store auxiliary data.
- Streaming libraries often contain C code for UTF-8 decoding.
- `aeson` uses C code to unescape JSON UTF-8 strings and store them as UTF-16 in one go.
- LSP positions assume UTF-16, impacting HLS.
- `text-icu` communicates to `libicu`, which uses UTF-16.

Huge thanks to all maintainers involved in the migration!

- All Stackage Nightly packages are compatible with `text-2.0`.
- `text-2.0` will be shipped with GHC 9.4.

- `text-2.0` is ready for the prime time, give it a try today!
- We are looking for new contributors, feel free to get in touch.
- A brand new `text-builder-linear` is out, up to 20x faster.

Thank you!

@ andrew.lelechenko@gmail.com

Bodigrim

github.com/haskell/text

github.com/Bodigrim/my-talks