Linear Haskell for string builders

Andrew Lelechenko
andrew.lelechenko@gmail.com

MuniHac, 13.09.2025

List concatenation is linear . . .

```
type String = [Char]

output :: String
output = longString ++ veryLongString ++ extraLongString

output :: String
output = (longString ++ veryLongString) ++ extraLongString

infixr 5  ++
(++) :: [a] → [a] → [a]
(++) []       ys = ys
(++) (x : xs) ys = x : xs ++ ys
```

List concatenation is linear only in its first argument

```
type String = [Char]

good :: String
good = longString ++ veryLongString ++ extraLongString

good' :: String
good' = longString ++ (veryLongString ++ extraLongString)

bad :: String
bad = (longString ++ veryLongString) ++ extraLongString

ghci> length (replicate 100000000 'x' ++ ("foo" ++ "bar"))
100000006
(1.18 secs, 11,200,068,064 bytes)
ghci> length ((replicate 100000000 'x' ++ "foo") ++ "bar")
100000006
(1.84 secs, 16,800,068,064 bytes)
```

How to define instances of `class Show`?

```
data User = User
  { name    :: String
  , address :: String
  , phone   :: String }

instance Show User where
  show User{..} = name ++ address ++ phone

data Order = Order
  { user    :: User
  , product :: String
  , date    :: String  }

instance Show Order where
  show Order{..} = show user ++ product ++ date
```

Define showsPrec, not show

```
class Show a where
  show :: a → String
  showsPrec :: Int → a → (String → String)
  {-# MINIMAL show | showsPrec #-}

instance Show User where
  showsPrec _ User{..} rest =
    name ++ address ++ phone ++ rest

instance Show Order where
  showsPrec p Order{..} rest =
    showsPrec p user (product ++ date ++ rest)
```

Compose functions instead of concatenating data

```haskell
instance Show User where
  -- showsPrec _ User{..} rest =
  --   name ++ address ++ phone ++ rest
  showsPrec _ User{..} =
    (name ++) . (address ++) . (phone ++)

instance Show Order where
  -- showsPrec p Order{..} rest =
  --   showsPrec p user (product ++ date ++ rest)
  showsPrec p Order{..} =
    showsPrec p user . (product ++) . (date ++)
```

Which is faster?

```
(long ++) . ((veryLong ++) . (extraLong ++)) $ []

((long ++) . (veryLong ++)) . (extraLong ++) $ []

((long ++) . (veryLong ++)) . (extraLong ++) $ []
  = ((long ++) . (veryLong ++)) $ (extraLong ++) $ []
    = (long ++) $ (veryLong ++) $ (extraLong ++) $ []
```

Builders for lists

Efficient concatenation is possible, but requires diligence to add new chunks from the left only.

```
newtype Builder = Builder (String → String)

fromString :: String → Builder
fromString xs = Builder (xs ++)

toString :: Builder → String
toString (Builder f) = f []

instance Semigroup Builder where
  Builder f <> Builder g = Builder (f . g)
```

Builder allows to concatenate left and right, although has an increased constant factor.

~~`String`~~

`StrictText`

Concatenation of StrictText is linear . . .

```
data StrictText = StrictText
  { buffer :: ByteArray
  , offset :: Int
  , length :: Int }
  -- len(buffer) could be /= offset + length

concatRight :: StrictText
concatRight = longText ++ (veryLongText ++ extraLongText)

concatLeft :: StrictText
concatLeft = (longText ++ veryLongText) ++ extraLongText
```

Concatenation of StrictText is linear in both arguments

```
data StrictText = StrictText
  { buffer :: ByteArray
  , offset :: Int
  , length :: Int }
  -- len(buffer) could be /= offset + length

concatRight :: StrictText
concatRight = longText ++ (veryLongText ++ extraLongText)

concatLeft :: StrictText
concatLeft = (longText ++ veryLongText) ++ extraLongText
```

Data.Text.Lazy.Builder sidesteps the issue

```
data LazyText = Empty | Chunk StrictText LazyText

newtype Builder = Builder {
  forall s. (Buffer s → ST s [StrictText])
         → (Buffer s → ST s [StrictText]) }

data Buffer s = Buffer
  { buffer :: MutableByteArray s
  , offset :: Int
  , used   :: Int
  , unused :: Int }
  -- len(buffer) = offset + used + unused
```

TextBuilder precomputes the total length

```
data TextBuilder = TextBuilder
  -- Estimated maximum size of the byte array to allocate.
  Int
  -- Function that populates a preallocated bytearray
  -- of the estimated maximum size specified above provided
  -- an offset into it and producing the offset after.
  (forall s. MutableByteArray s → Int → ST s Int)

instance Semigroup TextBuilder where
  TextBuilder lenL writeL <> TextBuilder lenR writeR =
    TextBuilder
      (lenL + lenR)
      (\array offset → do
        offsetAfter1 ← writeL array offset
        writeR array offsetAfter1
      )
```

What would Java do?

```
data Builder = Builder
  { buffer :: ByteArray
  , used   :: Int }

(++) :: Builder → Text → Builder
Builder arr used ++ Text srcArr srcOff srcLen = runST $ do
  let unused = sizeofByteArray arr - used
  if unused ⩾ srcLen then do
    mutArr ← unsafeThawByteArray arr
    arr' ← unsafeFreezeByteArray mutArr
    pure $ Builder arr' (used + srcLen)
  else do
    mutArr ← newByteArray ((used + srcLen) * 2)
    copyByteArray mutArr 0 arr 0 used
    copyByteArray mutArr used srcArr srcOff srcLen
    arr' ← unsafeFreezeByteArray mutArr
    pure $ Builder arr' (used + srcLen)
```

Trailblazing attoparsec

```
data Builder = Builder
  { gen    :: Int
  , buffer :: ByteArray -- ^ also stores 'gen' at start
  , used   :: Int }
```

Commit 62856d6 by @bos on May 30, 2014

The fact of having a mutable buffer really helps with performance, but . . . it does have a consequence: if someone misuses [it] . . . they could overwrite data.

. . . we use two generation counters (one mutable, one immutable) to track the number of appends to a mutable buffer. If the counters ever get out of sync, someone is appending twice to a mutable buffer, so we duplicate the entire buffer in order to preserve the immutability of its older self.

While we could go a step further and gain protection against API abuse on a multicore system, by use of an atomic increment instruction to bump the mutable generation counter, that would be very expensive. . . Clients should never call a continuation more than once; **we lack a linear type system** that could enforce this. . .

Thank you!

Bodigrim

@ andrew.lelechenko@gmail.com

github.com/Bodigrim/linear-builder

hackage.haskell.org/package/text-builder-linear