Linear Haskell for string builders

Andrew Lelechenko
andrew.lelechenko@gmail.com

MuniHac, 13.09.2025

List concatenation is linear . . .

```
type String = [Char]

concatRight :: String
concatRight = long ++ veryLong ++ extraLong

concatLeft :: String
concatLeft = (long ++ veryLong) ++ extraLong

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

List concatenation is linear only in its first argument

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

> length (foldr (++) [] (map (:[]) ['\0'..'\10000']))
10001
(0.01 secs, 2,945,344 bytes)
> length (foldl (++) [] (map (:[]) ['\0'..'\10000']))
10001
(0.84 secs, 4,299,915,184 bytes)
```

How to define instances of class `Show`?

```haskell
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 ++)) $ []
```

vs.

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

There is no big difference!

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

List concatenation: recap

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

```
newtype DList = DList (String → String)

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

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

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

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

~~String~~ (Strict)Text

Concatenation of Text is linear . . .

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

concatRight :: Text
concatRight = longText <> (veryLongText <> extraLongText)

concatLeft :: Text
concatLeft = (longText <> veryLongText) <> extraLongText
```

Concatenation of Text is linear in both arguments

```
> let x = T.replicate 100000000 "x" in T.length (x <> (x <> x))
300000000
(0.14 secs, 600,067,864 bytes)
> let x = T.replicate 100000000 "x" in T.length ((x <> x) <> x)
300000000
(0.12 secs, 600,067,864 bytes)

> T.length (foldr (<>) mempty (map T.singleton ['\0'..'\10000']))
10001
(0.16 secs, 151,491,368 bytes)
> T.length (foldl (<>) mempty (map T.singleton ['\0'..'\10000']))
10001
(0.14 secs, 133,854,104 bytes)
```

Efficient concatenation requires us to guess the size of the final result

Data.Text.Lazy.Builder sidesteps the issue

```
newtype Builder = Builder {
  ∀ s. (Buffer s → ST s [Text])
    → (Buffer s → ST s [Text]) }

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

TextBuilder by Nikita Volkov

```
data TextBuilder = TextBuilder
  -- Estimated max size of the bytearray to allocate.
  Int
  -- Function that populates a preallocated bytearray
  -- of the estimated max size specified above provided
  -- an offset into it and producing the offset after.
  (∀ 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
      )
```

Java-style string builder

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

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

Honest mutable Buffer

```haskell
data MutBuffer s = MutBuffer
  { buffer :: MutableByteArray s, used :: Int }

(++) :: MutBuffer s → Text → ST s (MutBuffer s)
MutBuffer mutArr used ++ Text srcArr srcOff srcLen = do
  size ← getSizeofMutableByteArray mutArr
  let unused = size - used
  if unused ⩾ srcLen then do
    copyByteArray mutArr used srcArr srcOff srcLen
    pure $ MutBuffer mutArr (used + srcLen)
  else do
    let newSize = (used + srcLen) * 2
    mutArr' ← resizeMutableByteArray mutArr newSize
    copyByteArray mutArr' used srcArr srcOff srcLen
    pure $ MutBuffer mutArr' (used + srcLen)
```

attoparsec used linear types before linear types

```haskell
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...

Linear and unlifted types for the rescue

```
data Buffer :: TYPE ('BoxedRep 'Unlifted) where
  Buffer :: {-# UNPACK #-} !Text → Buffer
  -- ^ constructor is not exported

appendBounded
  :: Int -- ^ Upper bound for the number of bytes to write
  → (∀ s. MutableByteArray s → Int → ST s Int)
  -- ^ Write bytes starting from the given offset
  -- and return an actual number of bytes written.
  → Buffer ⊸ Buffer

(▷) :: Buffer ⊸ Text → Buffer

runBuffer :: (Buffer ⊸ Buffer) ⊸ Text
runBufferBS :: (Buffer ⊸ Buffer) ⊸ StrictByteString
```

Appending letters

```
(▷) :: Buffer ⊸ Text → Buffer
(◁) :: Text → Buffer ⊸ Buffer
(><) :: Buffer ⊸ Buffer ⊸ Buffer

> runBuffer (\b → b ▷ "foo" ▷ "bar")
"foobar"

(▷#) :: Buffer ⊸ Addr# → Buffer
(#◁) :: Addr# → Buffer ⊸ Buffer

> runBuffer (\b → b ▷# "foo"# ▷# "bar"#)
"foobar"

(▷.) :: Buffer ⊸ Char → Buffer
(.◁) :: Char → Buffer ⊸ Buffer
> runBuffer (\b → b ▷. 'q' ▷. 'w')
"qw"
```

Appending numbers

```
(▷$) :: (Integral a, FiniteBits a) ⇒ Buffer ⊸ a → Buffer
($◁) :: (Integral a, FiniteBits a) ⇒ a → Buffer ⊸ Buffer
> runBuffer (\b → b ▷$ (42 :: Int))
"42"

(▷$$) :: Integral a ⇒ Buffer ⊸ a → Buffer
($$◁) :: Integral a ⇒ a → Buffer ⊸ Buffer
runBuffer (\b → b ▷$$ (1e50 :: Integer))
"100000000000000000000000000000000000000000000000000"

(▷&) :: (Integral a, FiniteBits a) ⇒ Buffer ⊸ a → Buffer
(&◁) :: (Integral a, FiniteBits a) ⇒ a → Buffer ⊸ Buffer
> runBuffer (\b → b ▷& (42 :: Int))
"2a"

(▷%) :: Buffer ⊸ Double → Buffer
(%◁) :: Double → Buffer ⊸ Buffer
> runBuffer (\b → b ▷% 123.456)
"123.456"
```

No linear types? No problem

```haskell
newtype Builder = Builder { unBuilder :: Buffer ⊸ Buffer }

fromText :: Text → Builder
fromChar :: Char → Builder
fromAddr :: Addr# → Builder
fromDec :: (Integral a, FiniteBits a) ⇒ a → Builder
fromUnboundedDec :: Integral a ⇒ a → Builder
fromHex :: (Integral a, FiniteBits a) ⇒ a → Builder
fromDouble :: Double → Builder

runBuilder :: Builder → Text
runBuilderBS :: Builder → StrictByteString
```

Benchmarks with GHC 9.12 on aarch64

	`text`	`text-builder`		This package	
Text					
1000	80.5 μs	26.7 μs	0.33x	23.1 μs	0.29x
1000000	216 ms	107 ms	0.49x	22.9 ms	0.11x
Char					
1000	35.4 μs	18.4 μs	0.52x	7.68 μs	0.22x
1000000	175 ms	178 ms	1.02x	10.5 ms	0.06x
Decimal					
1000	148 μs	738 μs	5.00x	106 μs	0.72x
1000000	334 ms	2.803 s	8.40x	108 ms	0.32x
Hexadecimal					
1000	862 μs	141 μs	0.16x	44.6 μs	0.05x
1000000	1.502 s	228 ms	0.15x	45.9 ms	0.03x
Double					
1000	14.2 ms	71.9 ms	5.05x	671 μs	0.05x
1000000	14.366 s	101.342 s	7.05x	689 ms	0.05x

Thank you!

Bodigrim

andrew.lelechenko@gmail.com

github.com/Bodigrim/linear-builder

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