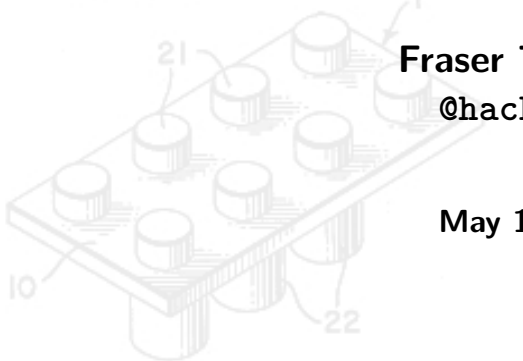
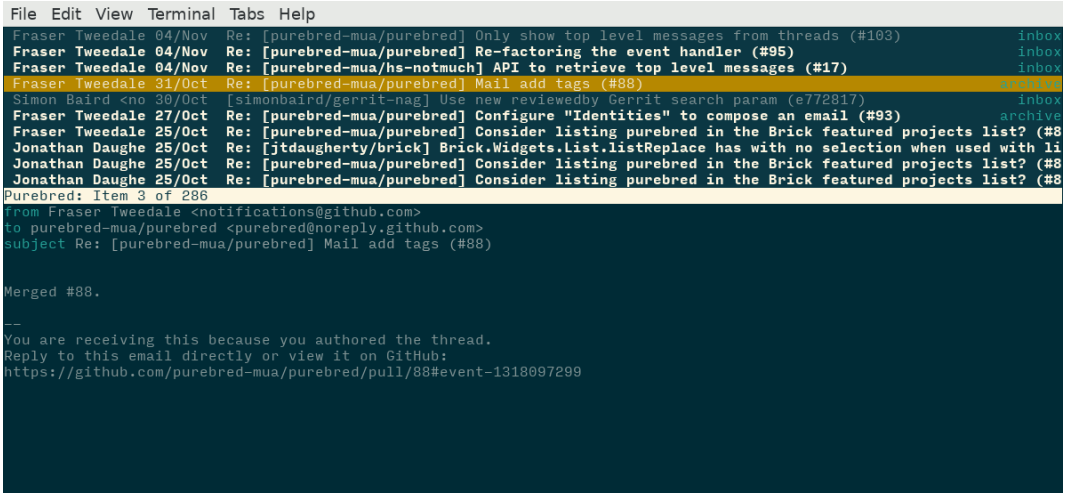
Infinite Scroll

Lazy lists in the Brick TUI library

Fraser Tweedale
@hackuador

May 15, 2019




File Edit View Terminal Tabs Help
Fraser Tweedale 04/Nov Re: [purebred-mua/purebred] Only show top level messages from threads (#103) inbox
Fraser Tweedale 04/Nov Re: [purebred-mua/purebred] Re-factoring the event handler (#95) inbox
Fraser Tweedale 04/Nov Re: [purebred-mua/hs-notmuch] API to retrieve top level messages (#17) inbox
Fraser Tweedale 31/Oct Re: [purebred-mua/purebred] Mail add tags (#88) archive
Simon Baird <no 30/Oct [simonbaird/gerrit-nag] Use new reviewedby Gerrit search param (e772817) inbox
Fraser Tweedale 27/Oct Re: [purebred-mua/purebred] Configure "Identities" to compose an email (#93) archive
Fraser Tweedale 25/Oct Re: [purebred-mua/purebred] Consider listing purebred in the Brick featured projects list? (#8
Jonathan Daughe 25/Oct Re: [jtdaugherty/brick] Brick.Widgets.List.listReplace has with no selection when used with li
Jonathan Daughe 25/Oct Re: [purebred-mua/purebred] Consider listing purebred in the Brick featured projects list? (#8
Jonathan Daughe 25/Oct Re: [purebred-mua/purebred] Consider listing purebred in the Brick featured projects list? (#8
Purebred: Item 3 of 286
from Fraser Tweedale <notifications@github.com>
to purebred-mua/purebred <purebred@noreply.github.com>
subject Re: [purebred-mua/purebred] Mail add tags (#88)
Merged #88.
--
You are receiving this because you authored the thread.
Reply to this email directly or view it on GitHub:
https://github.com/purebred-mua/purebred/pull/88#event-1318097299

Demo: basic Brick list

Brick.Widget.List API

```
data List n e

list :: k -> Vector e -> Int -> List n e

listMoveTo :: Int -> List n e -> List n e
listMoveBy :: Int -> List n e -> List n e

listInsert :: Int -> e -> List n e -> List n e
listRemove :: Int ->      List n e -> List n e
```

Brick.Widget.List internals

```
data List n e = List
  { listElements :: Vector e
  , listSelected :: Maybe Int
  , listName :: n
  , listItemHeight :: Int
  }
  deriving (Functor, Foldable, Traversable)

listElementsL :: Lens' (List n e) (Vector e)
listSelectedL :: Lens' (List n e) (Maybe Int)
```

Demo: Purebred thread list

Can we lazily load items?

- Only need to evaluate list up to displayed items
- `Vector` is strict...

Can we lazily load items?

- Only need to evaluate list up to displayed items
- `Vector` is strict...
- but not all container types are!

The plan

1. Engage upstream
2. Ensure adequate regression tests
3. Implementation

Regression tests

```
prop_insertSize :: (Eq a) => Int -> a -> List n a -> Bool
prop_insertSize i a l =
  length (listInsert i a l ^. listElementsL)
    == length (l ^. listElementsL) + 1

prop_insert :: (Eq a) => Int -> a -> List n a -> Bool
prop_insert i a l =
  i >= 0 && i <= length (l ^. listElementsL) ==>
    listSelectedElement (listMoveTo i (listInsert i a l)
    == Just (i, a)
```

Regression tests

```
data ListMoveOp a
  = MoveUp
  | MoveDown
  | MoveBy Int
  | MoveTo Int
  | MoveToElement a

data ListOp a
  = Insert Int a
  | Remove Int
  | Replace Int [a]
  | Clear
  | ListMoveOp (ListMoveOp a)
```

Regression tests

```
prop_listOpsMaintainValidSelection
  :: (Eq a) => [ListOp a] -> List n a -> Bool

prop_moveUpReachesBeginning
  :: (Eq a) => [ListOp a] -> List n a -> Bool
```

Implementation (before)

```
data List n e = List
  { listElements :: Vector e
  , listSelected :: Maybe Int
  , listName :: n
  , listItemHeight :: Int
  }
  deriving (Functor, Foldable, Traversable)
```

Implementation (after)

```
data GenericList n t e = List
  { listElements :: t e
  , listSelected :: Maybe Int
  , listName :: n
  , listItemHeight :: Int
  }
  deriving (Functor, Foldable, Traversable)

type List n e = GenericList n Vector e
```

Implementation (after)

```haskell
class Splittable t where
    {-# MINIMAL splitAt #-}

    -- Equivalent to '(take n xs, drop n xs)'
    splitAt :: Int -> t a -> (t a, t a)

    -- Equivalent to '(take n . drop i) xs'
    slice :: Int -> Int -> t a -> t a
    slice i n = fst . splitAt n . snd . splitAt i

instance Splittable Vector where
  -- | /O(1)/
  splitAt = Data.Vector.splitAt
```

Implementation (before)

```
listMoveTo
  :: ()
  => Int -> List n e -> List n e
listMoveTo pos l =
  let
    len = length l
    i = if pos < 0 then len - pos else pos
  in
    l & listSelectedL .~ if null l
          then Nothing
          else Just $ clamp 0 (len - 1) i
```

Implementation (after)

```
listMoveTo
  :: (Foldable t, Splittable t)
  => Int -> GenericList n t e -> GenericList n t e
listMoveTo pos l =
  let
    len = length l
    i = if pos < 0 then len - pos else pos
  in
    l & listSelectedL .~ if null l
          then Nothing
          else Just $ splitClamp l i
```

Implementation (before/after)

```
clamp :: (Ord a) => a -> a -> a -> a
clamp lo hi = max lo . min hi

splitClamp
  :: (Foldable t, Splittable t)
  => GenericList n t e -> Int -> Int
splitClamp l i =
  let (_, t) = splitAt i (l ^. listElementsL)
  in clamp 0 (if null t then length l - 1 else i) i
```

Data.Sequence.Seq

```
instance Splittable Seq where
  -- | /O(log(min(i,n-1)))/
  splitAt = Data.Sequence.splitAt
```

Testing laziness

```
newtype L a = L [a]
  deriving (Functor, Foldable)

instance Splittable L where ...

prop_moveByPosLazy :: Bool
prop_moveByPosLazy =
  let
    v = L (1:2:3:4:undefined) :: L Int
    l = list () v 1  -- initial selection is 0
    l' = listMoveBy 1 l
  in
    l' ^. listSelectedL == Just 1  -- now it's 1
```

Parametricity

```
-- before
listMoveBy
  :: Int -> List n e -> List n e  -- Vector-based

-- after
listMoveBy
  :: (Foldable t, Splittable t)
  => Int -> GenericList n t e -> GenericList n t e
```

Purebred.LazyVector

```
newtype V a = V [Vector a]  -- linked list of chunks
  deriving (Functor, Foldable, Traversable, Show)

fromList :: Int -> [a] -> V a
fromList chunkSize xs = ...  -- one chunk at a time

-- | O(n/c).  May fragment a chunk.
instance Splittable V where
  splitAt = ... -- might split chunks

-- Eq and Ord ignore chunk boundaries
-- Semigroup and Monoid (<>) do not defragment chunks
```

Searching for threads

```
getThreads
  :: (MonadError Error m, MonadIO m)
  => Notmuch.SearchTerm -> FilePath -> m (V Thread)
getThreads query dbPath =
  withDatabaseReadOnly dbPath $
    flip Notmuch.query query
    >=> Notmuch.threads  -- thread list produced lazily
    >=> liftIO . lazyTraverse processThread
    >=> pure . fromList 128 -- chunk size

lazyTraverse :: (a -> IO b) -> [a] -> IO [b]
lazyTraverse f = foldr
  (\x ys -> (:) <$> f x <*> unsafeInterleaveIO ys)
  (pure [])
```

Demo: Purebred lazy list

List size notification

```
-- compute length in background; emit notification
notifyNumThreads
  :: (Foldable t)
  => BChan PurebredEvent -> t a -> IO ()
notifyNumThreads chan l =
  let
    len = length l
    go = len `seq` writeBChan chan (NotifyNumThreads len)
  in
    forkIO go
```

Demo: Background length compute

Infinite scroll - prune list

```
pruneList
  :: (L.Splittable t)
  => L.GenericList n t a -> L.GenericList n t a
pruneList l =
  case l ^. L.listSelectedL of
    Nothing -> l
    Just i ->
      let
        i' = max 0 (i - 999)
      in
        ($l) $
          over L.listElementsL (snd . L.splitAt i')
          . set L.listSelectedL (Just (i - i'))
```

Infinite scroll - prune list

```
appEvent
  :: L.GenericList () L Int
  -> T.BrickEvent () e
  -> T.EventM () (T.Next (L.GenericList () L Int))
appEvent l ev = case ev of
  T.VtyEvent (V.EvKey V.KEsc []) -> M.halt l
  T.VtyEvent vev ->
    M.continue . pruneList =<< L.handleListEvent vev l
  _ -> M.continue l
```

Demo: Infinite scroll

Questions?

https://speakerdeck.com/frasertweedale

@hackuador

jtdaugherty/brick

purebred-mua/purebred