List Fusion

Anselm Jonas Scholl

Hamburg Haskell Meetup

July 26, 2017

Motivation

- Operations on lists are convenient.
- Lots of syntactic sugar, standard library support, ability to stream large data sets, ...
- And lots of allocation!

Motivation

```
sqr x = x * x

sumSquares = sum . map sqr

numbers from to = take (to - from + 1)
    $ iterate (+1) from

sumSquaresOfNumbers from to = sumSquares
    $ numbers from to

sumSquaresOfNumbers' from to = go 0 from
  where go !acc n
    | n <= to    = go (acc + sqr n) (n + 1)
    | otherwise = acc
```

Motivation

```haskell
sqrRem to x = sqr x `rem` to

maximumSqrRem remBy = maximum . map (sqrRem remBy)

numbers from to = take from
    $ enumFromTo 0 to

maximumSqrRemNumbers from to = maximumSqrRem to
    $ numbers from to

maximumSqrRemNumbers' from !to = go (-1) 0
  where go !acc n
    | n < from  = go (max acc (sqrRem to n)) (n + 1)
    | otherwise = acc
```

Motivation

```haskell
avgHelper (!acc, !len) x = (acc + x, len + 1)

average = uncurry quot . foldl avgHelper (0, 0)

numbers a b count = replicate count a ++ replicate count b

averageOfNumbers a b count = average
    $ numbers a b count

averageOfNumbers' a b count = go1 count 0 0
  where
    go1 0 !acc !len = go2 count acc len
    go1 n !acc !len = go1 (n - 1) (acc + a) (len + 1)
    go2 0 !acc !len = acc `quot` len
    go2 n !acc !len = go2 (n - 1) (acc + b) (len + 1)
```

Motivation

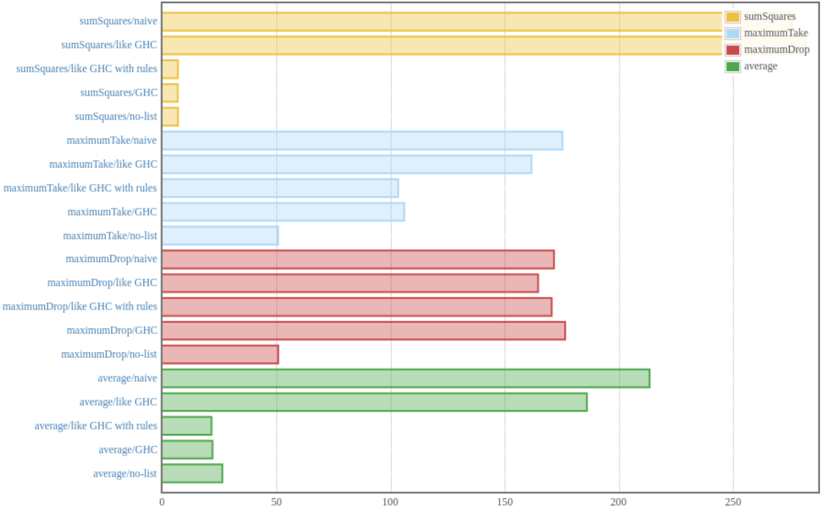

Figure: Performance of custom and GHC list functions

Motivation

- GHC list functions can get optimized to tight loops.
- Even if we copy the implementation of them we still get a lot of allocation.
- Only by also copying the rewrite rules used by GHC we achieve a similar runtime.
- We can also write specialized functions with the same performance, but they do not compose.

List fusion

- Two fundamental functions:
  - ```
    build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
    build g = g (:) []
    ```
  - ```
    foldr :: (a -> b -> b) -> b -> [a] -> b
    foldr k z = go
      where
        go []     = z
        go (y:ys) = y `k` go ys
    ```
- If we see `foldr k z (build g)`, we replace it with `g k z`

→ Avoids constructing the list in the first place

→ Gives GHC the ability to unbox the list elements

→ If done right this gives us a C-like loop

Playing by the Rules

- Instead of hard-coding this rule in the compiler, GHC gives library authors the ability to extend it with additional rewriting rules.
- It searches for terms matching the LHS of a rule and replaces then with the RHS of the rule.
- GHC does not check that LHS and RHS have the same meaning!
- But it makes sure both have the same type.

Rules and inlining

- Rules can only fire on the functions mentioned in a rule.
- If a function inlines too early, the rule won't fire.
- If a function inlines too late, the rule will not see the correct function and also not fire.
- One has to carefully control inlining:
  - Inlining happens in phases: 2, 1 and 0.
  - Each phase the simplifier runs at least once and applies inlining and rewrite rules.
  - A function can be inlined/not inlined before or starting from a phase.
  - A rule can also be active before or starting from a phase.

Rules and inlining - Example

```haskell
{-# NOINLINE[~1] foo #-}
foo = let x = x in x `seq` ()

{-# RULES "foo terminates" [~1] foo = () #-}

{-# INLINE[1] bar #-}
bar = foo
```

- We make sure that `foo` does not inline while the rule is active.
- But `bar` is only allowed to inline starting from phase 1.

→ `bar` will diverge while `foo` can be rewritten to `()`.

Good Producers and Consumers

- A good producer can be fused with a good consumer.
- Good producers:
  - List comprehensions
  - Enumerations of Int, Integer, Char
  - List literals
  - The cons constructor
  - (++), map, take, filter, iterate, repeat, zip, zipWith, ...
- Good consumers:
  - List comprehensions
  - array, (++) on the first argument, foldr, map, take, filter, concat, unzip(1,2,3,4), zip, zipWith, partition, head, and, or, any, all, sequence_, msum, ...

Fusing map and sum

Let's take a look at the rules for map:

```
{-# RULES
"map"       [~1] forall f xs.
  map f xs = build (\c n -> foldr (mapFB c f) n xs)
"mapList"   [1]  forall f.
  foldr (mapFB (:) f) [] = map f
"mapFB"     forall c f g.
  mapFB (mapFB c f) g = mapFB c (f.g)
  #-}
mapFB :: (elt -> lst -> lst)
    -> (a -> elt) -> a -> lst -> lst
{-# INLINE [0] mapFB #-}
mapFB c f = \x ys -> c (f x) ys
```

Fusing map and sum

```
{-# RULES
"map" [~1] forall f xs.
  map f xs = build (\c n -> foldr (mapFB c f) n xs)
  #-}
```

- Declares a new rule named "map".
- The rule is active before phase 1.
- `f` and `xs` are free variables in the rule.
- The rule matches on `map f xs`.
- The rule rewrites this to
  `build (\c n -> foldr (mapFB c f) n xs)`.

Outlook: Stream Fusion

- The vector and text libraries use another fusion method: stream fusion
- Operations are written as `unstream . streamOp . stream`.
- Each pair of stream and unstream is removed.

→ `unstream . opC . stream . unstream . opB . stream . unstream . opA . stream`

→ `unstream . opC . opB . opA . stream`

Outlook: Stream Fusion

```haskell
data Step s a = Done | Skip !s | Yield !a !s

data Stream a = forall s. Stream
  (s -> Step s a) -- stepper function
  !s              -- current state
  !Size           -- size hint

stream :: Text -> Stream Char
unstream :: Stream Char -> Text
```

Outlook: Stream Fusion

```haskell
data Step s a = Done | Skip s | Yield a s

data Stream m a = forall s. Stream (s -> m (Step s a)) s

data Chunk v a = Chunk Int
  (forall m. (PrimMonad m, Vector v a)
  => Mutable v (PrimState m) a -> m ())

data Bundle m v a = Bundle
  { sElems   :: Stream m a
  , sChunks  :: Stream m (Chunk v a)
  , sVector  :: Maybe (v a)
  , sSize    :: Size
  }
```