Automated Performance Measurements

Johannes Waldmann, HTWK Leipzig

August 16, 2016

Abstract

We reify the API of a library, so we can represent nested calls as terms. Using a `smallcheck`-like enumeration of terms and contexts, we find problematic API usages that take a lot of time. As an example, we investigate the standard `pretty` library.

1 Example and Concepts

The `pretty` library [3] provides an abstract data type `Doc` for two-dimensional documents, with a concrete implementation that is hidden behind an API (application programming interface). For instance, there are functions

- (constructor) `char :: Char -> Doc`
- (combinators) `cat, sep :: [Doc] -> Doc`
- (destructor) `render :: Doc -> String`

We reify this API and obtain a data type that models nested API function calls. Terms can be evaluated by replacing each constructor with the corresponding function.

```
data Term = Char Char | Op Op [Term] ; data Op = Cat | Sep ; eval :: Term -> Doc
```

The cost of a term `t :: Term` can be determined experimentally by measuring the time for its (complete) evaluation, e.g., by forcing `length (render (eval t))`.

Measuring time puts us in `IO`. The actual number of evaluation steps should be a pure function of the term, but actual time for execution may be influenced by garbage collection, which may happen "out of the blue". So we should average over several samples.

We are interested in how the cost behaves as a function of term size, in the worst case. For that, we can enumerate all terms by size, using `smallcheck` [2]. We derive a generic `Serial` instance, obtained from `deriving Generics` for `Term`.

Ignoring for the moment that this will enumerate by depth, not by size, we still find that this approach is infeasible: small terms will be evaluated quickly, and there are just too many of them, so we never get to see anything interesting.

We use iterated contexts. A context is a term with a hole. A context can be applied to a term (the hole is substituted by the argument)

```
data Context = Hole | Cop Op [Term] Context [Term] ; apply :: Context -> Term -> Term
```

Iterated application creates a sequence of terms

```
s :: (Context, Term) -> [ Term ] ; s (c, t) = iterate (apply c) t
```

where the size is a linear function of the index.

We derive a `Serial` instance for `Context`, and enumerate pairs `p :: (Context,Term)`. For a fixed `n :: Int`, e.g., `n = 100`, and each `p`, we measure `cost (s p !! n)`, and investigate closer those that have high cost.

Each `p` gives a one-parametric family of test cases, and we can, for instance, plot cost as a function of size (that is, index). By eyeballing that graph, or more detailed statistical analysis, we can form an opinion on the asymptotic class of the function (linear, quadratic, ...). This can be compared to our expectation, and to the actual implementation.

2 Results

With my implementation [4] I found that (after `import Text.Pretty.HughesPJ`) this expression

```
length $ render
  $ iterate (\ hole -> sep [text "l", cat [hole], text "l"]) (text  "l") !! 1000
```

takes 10 seconds to evaluate, where the output string has length 4001 only.

I suspect that `pretty` behaves quadratically, see `https://github.com/haskell/pretty/issues/32#issuecomment-223312412`.

I also measured performance for wl-pprint [1]. Incredibly, this little program

```
import Text.PrettyPrint.Leijen
main = putDoc $ iterate ( \ hole ->  hsep [hole, sep []] ) (text  "l") !! 100
```

takes 3 minutes (on a standard desktop computer). The same happens when `sep []` is replaced by `group empty`. If we just write `empty` there, then all is fine.

Also, `wl-pprint` follows a different specification than `pretty` and produces different layouts. When we use `wl-pprint` to approximate `pretty`, we should wrap each combinator application in an `align`.

3 Discussion

My implementation works for the special case that there is only one result type `Doc` and all combinators have identical type `[Doc] -> Doc`.

The method should be extended for APIs that use more types. The reified data type (and interpreter) should be obtained in an automated way, from the names and types of the functions in the API. The context type should be obtained in an automated way from the underlying term type.

But even without any extensions, the method is useful: it highlights a little-known feature (namely, quadratic cost) of a standard library, and a serious bug in another.

References

[1] Daan Leijen. wl-pprint. `https://hackage.haskell.org/package/wl-pprint-1.2`, 2015.

[2] Colin Runciman and Roman Cheplyaka. smallcheck. `https://hackage.haskell.org/package/smallcheck`, 2016.

[3] David Terei. pretty. `https://hackage.haskell.org/package/pretty-1.1.3.4`, 2016.

[4] Johannes Waldmann. pretty-test. `https://github.com/jwaldmann/pretty-test`, 2016.