tasty-bench:
featherlight benchmark framework

Andrew Lelechenko
1@dxdy.ru

Haskell Love, 10.09.2021

How dare you to fragment Haskell ecosystem?

- Why was I unsatisfied with existing solutions?
- How does `tasty-bench` solve my issues?
- What makes it valuable for testing of core libraries?

criterion output

The very first example from `criterion` tutorial:

```
benchmarking fib/1
time                    23.91 ns   (23.30 ns .. 24.54 ns)
                        0.994 R²   (0.991 R² .. 0.997 R²)
mean                    24.36 ns   (23.77 ns .. 24.97 ns)
std dev                 2.033 ns   (1.699 ns .. 2.470 ns)
variance introduced by outliers: 88% (severely inflated)
```

- 13 quantities!
- Which quantity is the benchmark?
- What is characterized by bounds?
- Is this R^2 good enough?
- Is outliers' influence too bad?

criterion output

The very first example from `criterion` tutorial:

```
benchmarking fib/1
time                    23.91 ns   (23.30 ns .. 24.54 ns)
                        0.994 R²   (0.991 R² .. 0.997 R²)
mean                    24.36 ns   (23.77 ns .. 24.97 ns)
std dev                 2.033 ns   (1.699 ns .. 2.470 ns)
variance introduced by outliers: 88% (severely inflated)
```

- Understanding results should not require a PhD in statistics.
- If results are flawed, do not bother showing them...
- ...and rerun automatically until desired quality is achieved.

What time are we talking about?

By default `criterion` reports **wall-clock time**

- which is directly affected by any other application...
- ...and system service...
- ...and watching Youtube...
- ...and scrolling Facebook.

There are two possible solutions:

- either exercise iron willpower and restraint from delights,
- or measure per-process **CPU time.**

Comparing competing solutions

```
benchmarking mergesort
time                 23.91 ns   (23.30 ns .. 24.54 ns)
                     0.994 R²   (0.991 R² .. 0.997 R²)
mean                 24.36 ns   (23.77 ns .. 24.97 ns)
std dev              2.033 ns   (1.699 ns .. 2.470 ns)
variance introduced by outliers: 88% (severely inflated)

benchmarking quicksort
time                 25.61 ns   (24.91 ns .. 26.45 ns)
                     0.993 R²   (0.990 R² .. 0.998 R²)
mean                 25.96 ns   (25.15 ns .. 26.51 ns)
std dev              1.700 ns   (1.563 ns .. 1.922 ns)
variance introduced by outliers: 85% (severely inflated)
```

How much faster is `mergesort` than `quicksort`?

Comparing against baseline

Imagine yourself reviewing a PR:

- Ask a contributor to run benchmarks.
- Receive a wall of numbers.
- Ask to run benchmarks before and after the patch.
- Receive two walls of numbers.
- Ask to generate CSV reports.
- Good luck comparing them.

Can we replace manual performance testing with automated?

Living on the edge

How to test performance against GHC HEAD?

- `gauge` has few dependencies.
- But recently they are not updated in time.
- `criterion` has a lot of well-maintained dependencies.
- But building them all takes ages. . .
- . . . and they most certainly include the package you are interested to benchmark.
- Good luck to cut circular dependencies.

Running benchmarks only once GHC is released is already too late!

Writing a new benchmark framework

Reinventing the wheel:

- Support a hierarchy of benchmarks.
- Manage resources.
- Handle exceptions gracefully.
- Generate pretty console output.
- List all available benchmarks.
- Filter benchmarks by name.
- Provide an extensible CLI.

Solution:
use a mature testing framework to provide all of this and more.

Benchmarks as tests

Everything can be expressed as `tasty` plugins:

- `type Benchmark = TestTree`
- There is a test provider with `instance IsTest Benchmarkable`.
- Resources and exceptions are managed by `tasty`.
- Console output is a usual `consoleTestReporteer` extended with a hook to emit additional information.
- CSV and SVG reporters are normal `Ingredients`.
- CLI, listing and filtering are all native `tasty` functionality.

We can concentrate solely on great benchmarking experience.

API is compatible with criterion and gauge

```
import Test.Tasty.Bench

fibo :: Int → Integer
fibo n
  | n < 2      = toInteger n
  | otherwise = fibo (n - 1) + fibo (n - 2)

main :: IO ()
main = defaultMain
  [ bgroup "fibonacci numbers"
    [ bench "fifth"      $ nf fibo  5
    , bench "tenth"      $ nf fibo 10
    , bench "twentieth" $ nf fibo 20
    ]
  ]
```

Output format

Goal-based (`--stdev N`) measurement of CPU time:

```
All
  fibonacci numbers
    fifth:      OK (2.13s)
       63 ns ± 3.4 ns
    tenth:      OK (1.71s)
      809 ns ±  73 ns
    twentieth: OK (3.39s)
      104 μs ± 4.9 μs
```

CSV:

```
All.fibonacci numbers.fifth,63453,3460
All.fibonacci numbers.tenth,809152,73744
All.fibonacci numbers.twentieth,104369531,4942646
```

tasty-bench can read CSV as well

- Prepare baseline CSV with `--csv`.
- Apply a patch.
- Rerun against baseline with `--baseline`.

```
All
  fibonacci numbers
    fifth:      OK (0.44s)
       53 ns ± 2.7 ns,  8% slower than baseline
    tenth:      OK (0.33s)
      641 ns ±  59 ns
    twentieth: OK (0.36s)
       77 μs ± 6.4 μs,  5% faster than baseline
```

Convert benchmarks to real tests

- `--fail-if-slower N` to mark all slow downs as failures.
- `--fail-if-faster N` to mark all speed ups as failures.
- `--hide-successes` to focus on problematic benchmarks only.

```
All
  fibonacci numbers
    fifth:     FAIL (0.44s)
       53 ns ± 2.7 ns,  8% slower than baseline
    twentieth: FAIL (0.36s)
       77 μs ± 6.4 μs,  5% faster than baseline
```

One can also mix benchmarks with unit/property/etc. tests in the same suite to check performance and correctness simultaneously.

Compare competing implementations

```
import Test.Tasty.Bench

fibo :: Int → Integer
fibo n
  | n < 2      = toInteger n
  | otherwise = fibo (n - 1) + fibo (n - 2)

main :: IO ()
main = defaultMain
  [ bgroup "fibonacci numbers"
    [ bcompare "tenth"  $ bench "fifth"      $ nf fibo  5
    ,                     bench "tenth"      $ nf fibo 10
    , bcompare "tenth"  $ bench "twentieth" $ nf fibo 20
    ]
  ]
```

Compare competing implementations

```
All
  fibonacci numbers
    fifth:      OK (16.56s)
      121 ns ± 2.6 ns, 0.08x
    tenth:      OK (6.84s)
      1.6 μs ±  31 ns
    twentieth: OK (6.96s)
      203 μs ± 4.1 μs, 128.36x
```

Coming soon: portable performance regression tests, which do not depend on baselines.

tasty-bench for Core libraries

- Allows to immerse benchmarks into the main package.
- Simplifies CI setup and turn around.
- Improves contributors' experience.
- Prevents maintainers' burnout.
- Provides answers to important questions faster.
- Supports bleeding-edge GHC and rebuilds quickly.

Current status

- Only 676 lines of portable Haskell code.
- Includes console, CSV and SVG reporters.
- Built-in comparison between items and between runs.
- No dependencies except implied by `tasty`.
- Builds up to a magnitude faster than competitors.
- Supports GHCs from 7.0 to 9.2 and HEAD.
- API is compatible with `criterion` and `gauge`.
- Early adopters include `bytestring`, `text`, `primitive`, `vector`, `random`, `optics`, `effectful`, `fused-effects`, `streamly`, `tar`, `pandoc`, `commonmark`.
- I would never have managed to switch `text` to UTF-8 without `tasty-bench`.

Grand Unified *Testing* Theory

- `tasty-bench` is not about fragmentation of benchmark suites.
- `tasty-bench` is about unifying testing landscape:
- both technically, by being just a plugin for `tasty`,
- and ideologically, by making performance regression testing more accessible than ever.

Thank you!

@ 1@dxdy.ru Bodigrim

github.com/Bodigrim/tasty-bench github.com/Bodigrim/my-talks