Benchmarking in Elixir

Antonín Hackenberg

Developer: JavaScript, Elixir at Blueberry

Gardener

Traveller

Email: antonin.hackenberg@gmail.com
LinkedIn: https://cz.linkedin.com/in/antoninhackenberg
Github: https://github.com/TondaHack

Elixir

- 2012 (José Valim)
- Erlang bytecode
- Erlang
  - Amazon, Yahoo, Facebook, WhatsApp, T-Mobile, Motorola, Ericsson, World of Warcraft
- Low-latency, Distributed, Fault-tolerant

Elixir

- Amazing Tooling
- Good documentation
- Metaprogramming
- Millions users
- Lonely planet, Pinterest, Bet365, Cabify

Elixir - use-cases

- Web applications - Pheonix Framework
- Web APIs (JSON and GraphQL) - Absinthe
- Real-time web - Channels

Elixir - use-cases

- Stateful web - OTP
- Distributed systems (Erlang Distribution Protocol :rpc)
- Internet of things - Nerves project

Benchmarking

What?

Where?

How?

80/20

Benchmarking

Benchmark vs. Profile

Micro vs. Macro

Correctness

Tooling

Let’s Benchmark Tail Call Optimization

Sum recursion - No Tail Call Optimization

```
defmodule NoTCO do
 def sum_no_tco([]), do: 0

 def sum_no_tco([head | tail]) do
   head + sum_no_tco(tail)
 end
end
```

Sum function - Without Tail Call

```
NoTCO.sum_no_tco([1, 2, 3, 4])
```

```
defmodule NoTCO do
 def sum_no_tco([]), do: 0

 def sum_no_tco([head | tail]) do
   head + sum_no_tco(tail)
 end
end
```

Sum function - Without Tail Call

```
NoTCO.sum_no_tco([1, 2, 3, 4])
```

1 + sum_no_tco([2, 3, 4])

```
defmodule NoTCO do
 def sum_no_tco([]), do: 0

 def sum_no_tco([head | tail]) do
   head + sum_no_tco(tail)
 end
end
```

Sum function - Without Tail Call

```
NoTCO.sum_no_tco([1, 2, 3, 4])

1 + sum_no_tco([2, 3, 4])
  + 2 + sum_no_tco([3, 4])
```

```
defmodule NoTCO do
 def sum_no_tco([]), do: 0

 def sum_no_tco([head | tail]) do
   head + sum_no_tco(tail)
 end
end
```

Sum function - Without Tail Call

```
NoTCO.sum_no_tco([1, 2, 3, 4])

1 + sum_no_tco([2, 3, 4])
  + 2 + sum_no_tco([3, 4])
      + 3 + sum_no_tco([4])
```

```
defmodule NoTCO do
 def sum_no_tco([]), do: 0

 def sum_no_tco([head | tail]) do
   head + sum_no_tco(tail)
 end
end
```

Sum function - Without Tail Call

```
NoTCO.sum_no_tco([1, 2, 3, 4])

1 + sum_no_tco([2, 3, 4])
  + 2 + sum_no_tco([3, 4])
      + 3 + sum_no_tco([4])
          + 4 + sum_no_tco([])
```

```
defmodule NoTCO do
 def sum_no_tco([]), do: 0

 def sum_no_tco([head | tail]) do
   head + sum_no_tco(tail)
 end
end
```

Sum function - Without Tail Call

```
NoTCO.sum_no_tco([1, 2, 3, 4])

1 + sum_no_tco([2, 3, 4])
  + 2 + sum_no_tco([3, 4])
      + 3 + sum_no_tco([4])
          + 4 + sum_no_tco([])
              + 0
```

```
defmodule NoTCO do
 def sum_no_tco([]), do: 0

 def sum_no_tco([head | tail]) do
   head + sum_no_tco(tail)
 end
end
```

Sum function - Without Tail Call

```
NoTCO.sum_no_tco([1, 2, 3, 4])

1 + sum_no_tco([2, 3, 4])
  + 2 + sum_no_tco([3, 4])
      + 3 + sum_no_tco([4])
          + 4 + 0
```

```
defmodule NoTCO do
 def sum_no_tco([]), do: 0

 def sum_no_tco([head | tail]) do
   head + sum_no_tco(tail)
 end
end
```

Sum function - Without Tail Call

```
NoTCO.sum_no_tco([1, 2, 3, 4])

1 + sum_no_tco([2, 3, 4])
  + 2 + sum_no_tco([3, 4])
      + 3 + 4
```

```
defmodule NoTCO do
 def sum_no_tco([]), do: 0

 def sum_no_tco([head | tail]) do
   head + sum_no_tco(tail)
 end
end
```

Sum function - Without Tail Call

```
NoTCO.sum_no_tco([1, 2, 3, 4])

1 + sum_no_tco([2, 3, 4])
  + 2 + 7
```

```
defmodule NoTCO do
 def sum_no_tco([]), do: 0

 def sum_no_tco([head | tail]) do
   head + sum_no_tco(tail)
 end
end
```

Sum function - Without Tail Call

```
NoTCO.sum_no_tco([1, 2, 3, 4])
```

1 + 9

```
defmodule NoTCO do
 def sum_no_tco([]), do: 0

 def sum_no_tco([head | tail]) do
   head + sum_no_tco(tail)
 end
end
```

Sum function - Without Tail Call

```
NoTCO.sum_no_tco([1, 2, 3, 4])
```

= 10

```
defmodule NoTCO do
 def sum_no_tco([]), do: 0

 def sum_no_tco([head | tail]) do
   head + sum_no_tco(tail)
 end
end
```

Sum function - Tail Call Optimization

```
defmodule TCO do
 def sum([], acc), do: acc

 def sum([head | tail], acc) do
   sum(tail, head + acc)
 end
end
```

Sum function - Tail Call Optimization

```
TCO.sum([1, 2, 3, 4], 0)
```

```
defmodule TCO do
 def sum([], acc), do: acc

 def sum([head | tail], acc) do
   sum(tail, head + acc)
 end
end
```

Sum function - Tail Call Optimization

```
TCO.sum([1, 2, 3, 4], 0)
```

```
sum([2, 3, 4], 1)
```

```
defmodule TCO do
 def sum([], acc), do: acc

 def sum([head | tail], acc) do
   sum(tail, head + acc)
 end
end
```

Sum function - Tail Call Optimization

```
TCO.sum([1, 2, 3, 4], 0)
```

```
sum([3, 4], 3)
```

```
defmodule TCO do
 def sum([], acc), do: acc

 def sum([head | tail], acc) do
   sum(tail, head + acc)
 end
end
```

Sum function - Tail Call Optimization

```
TCO.sum([1, 2, 3, 4], 0)
```

sum([4], 6)

```
defmodule TCO do
 def sum([], acc), do: acc

 def sum([head | tail], acc) do
   sum(tail, head + acc)
 end
end
```

Sum function - Tail Call Optimization

```
TCO.sum([1, 2, 3, 4], 0)
```

```
sum([], 10)
```

```
defmodule TCO do
 def sum([], acc), do: acc

 def sum([head | tail], acc) do
   sum(tail, head + acc)
 end
end
```

Sum function - Tail Call Optimization

```
TCO.sum([1, 2, 3, 4], 0)
```

```
10
```

```
defmodule TCO do
 def sum([], acc), do: acc

 def sum([head | tail], acc) do
   sum(tail, head + acc)
 end
end
```

Difference

```
defmodule NoTCO do
 def sum_no_tco([]), do: 0
 def sum_no_tco([head | tail]) do
   head + sum_no_tco(tail)
 end
end
```

```
defmodule TCO do
 def sum([], acc), do: acc
 def sum([head | tail], acc) do
   sum(tail, head + acc)
 end
end
```

Benchmarking

```
iex(1)> :timer.tc fn -> NoTCO.sum_no_tco(Enum.to_list(1..100_000)) end
{17673, 5000050000}

iex(2)> :timer.tc fn -> TCO.sum(Enum.to_list(1..100_000), 0) end
{10518, 5000050000}
```

```
inputs = %{
 "1_000 list size"    => Enum.to_list(1..1_000),
 "100_000 list size" => Enum.to_list(1..1000_000),
 "1000_000 list size" => Enum.to_list(1..1000_000),
}

Benchee.run(%{
 "Tail Call Optimizitation"  => fn(list) -> TCO.sum(list, 0) end,
 "No TCO" => fn(list) -> NoTCO.sum_no_tco(list) end,
 "Enum sum" => fn(list) -> Enum.sum(list) end
},
 time: 10,
 memory_time: 2,
 inputs: inputs,
 formatters: [
   Benchee.Formatters.HTML,
   Benchee.Formatters.Console
 ],
 formatter_options: [html: [file: "lib/report/tco.html"]]
)
```

Comparison

Data Table

Name	Iterations per Second	Average	Deviation	median	minimum	maximum	Sample size
Enum sum	41.88	23.88 ms	±5.57%	23.39 ms	22.52 ms	32.09 ms	418
No TCO	108.14	9.25 ms	±11.85%	8.95 ms	8.52 ms	37.81 ms	1079
Tail Call Optimizitation	169.85	5.89 ms	±6.81%	5.78 ms	5.51 ms	9.87 ms	1696

Average Iterations per Second (1000_000 list size)

benchee report 1000_000 list size

System info

System info

- Elixir: 1.7.0-dev
- Erlang: 20.2.2
- Operating system: macOS

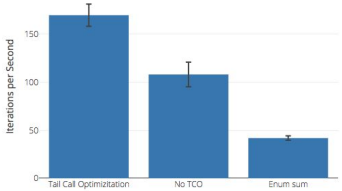
Available memory: 16 GB
CPU Information: Intel(R) Core(TM) i5-6267U CPU @ 2.90GHz
Number of Available Cores: 4

Comparison

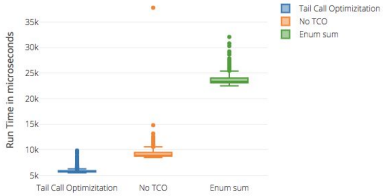
Data Table

Name	Iterations per Second	Average	Deviation	median	minimum	maximum	Sample size
Enum sum	41.88	23.88 ms	±5.57%	23.39 ms	22.52 ms	32.09 ms	418
No TCO	108.14	9.25 ms	±11.85%	8.95 ms	8.52 ms	37.81 ms	1079
Tail Call Optimizitation	169.85	5.89 ms	±6.81%	5.78 ms	5.51 ms	9.87 ms	1696

Average Iterations per Second (1000_000 list size)



Run Time Boxplot (1000_000 list size)

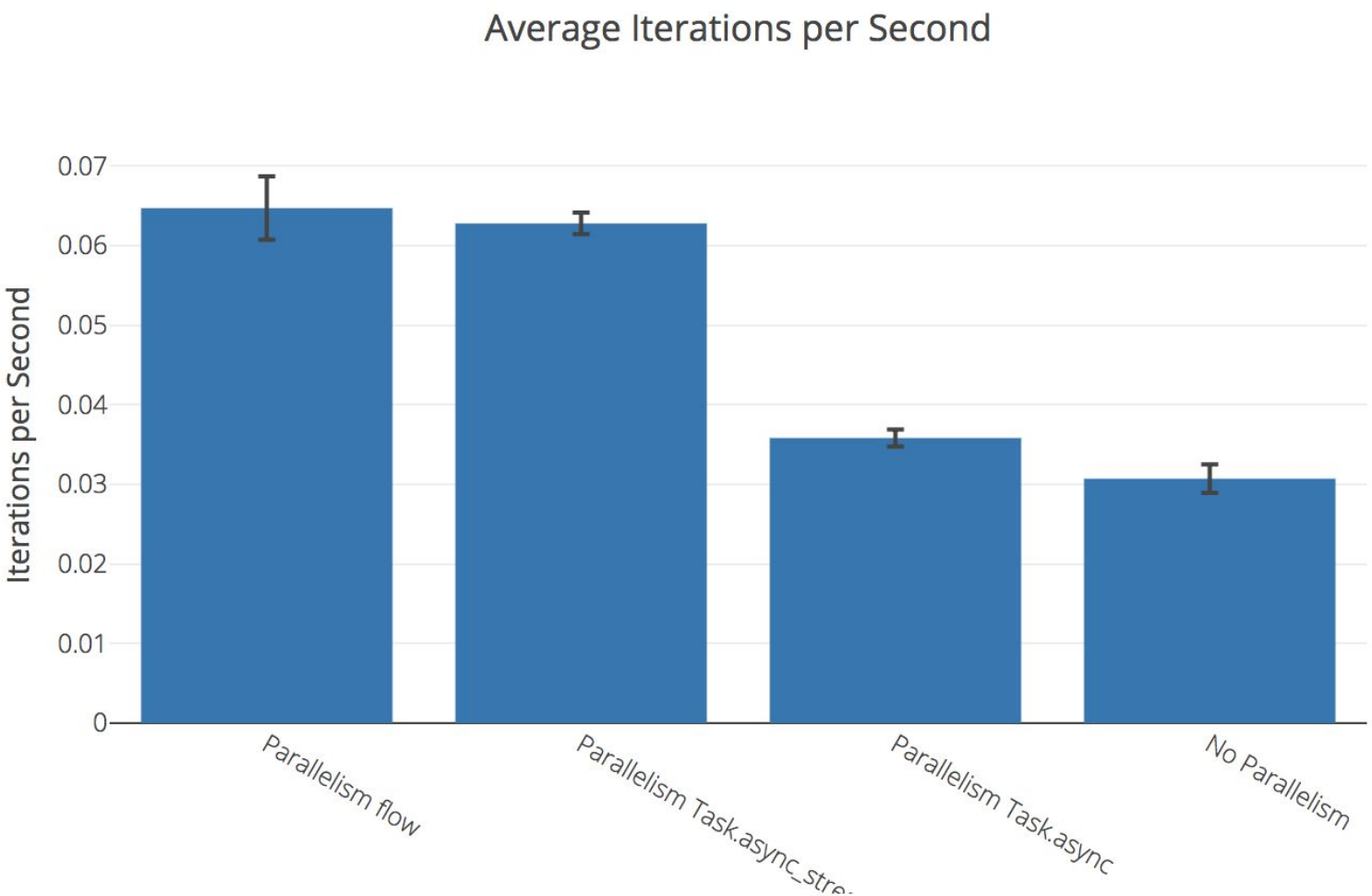


benchee version: 0.13.1
benchee_html version: 0.5.0

Real example

```
def parse(file_path, modifiers \\ %{}, options \\ %{}, take)  do
  file_path
  |> File.stream!()
  |> CsvParser.decode_csv()
  |> Stream.take(take)
  |> Stream.map(&process_item(&1, options, modifiers))
  |> Stream.filter(&filter_items(&1, options["elastic_search_index"]))
  |> Stream.chunk_every(1000)
  |> Stream.each(&store_items(&1, options["elastic_search_index"]))
  |> Stream.run()
end
```

```elixir
[modifiers, file_path] = PpcbeeElixirBackend.Enhance.download(191)

Benchee.run(
  %{
    "No Parallelism" => fn ->
      PpcbeeElixirBackend.DataParser.parse(file_path, modifiers, %{}, 10_000)
    end,
    "Parallelism Task.async_stream" => fn ->
      PpcbeeElixirBackend.DataParser.parse_async_stream(file_path, modifiers, %{},10_000)
    end,
    "Parallelism Task.async" => fn ->
      PpcbeeElixirBackend.DataParser.parse_async_task(file_path, modifiers, %{}, 10_000)
    end,
    "Parallelism flow" => fn ->
      PpcbeeElixirBackend.DataParser.parse_flow(file_path, modifiers, %{}, 10_000)
    end
  },
  time: 5 * 60,
  formatters: [
    Benchee.Formatters.HTML,
  ],
  formatter_options: [html: [file: "lib/benchmark/report/parallelism/report.html"]]
)
```


Average Iterations per Second
Iterations per Second
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
Parallelism flow
Parallelism Task.async_stre
Parallelism Task.async
No Parallelism

Ruby vs. Elixir - Tested on one machine

Ruby - previous service after optimization

- 200 / items per seconds
- CPU 100%
- RAM 560 mb

Elixir - before optimization

- 310 (no parallelism), 620 (paralellism)
- CPU 100%, 300%
- RAM 65 mb

Profiling - eprof, ExProf

"The module eprof provides a set of functions for time profiling of Erlang programs to find out how the execution time is used. The profiling is done using the Erlang trace BIFs. Tracing of local function calls for a specified set of processes is enabled when profiling is begun, and disabled when profiling is stopped."

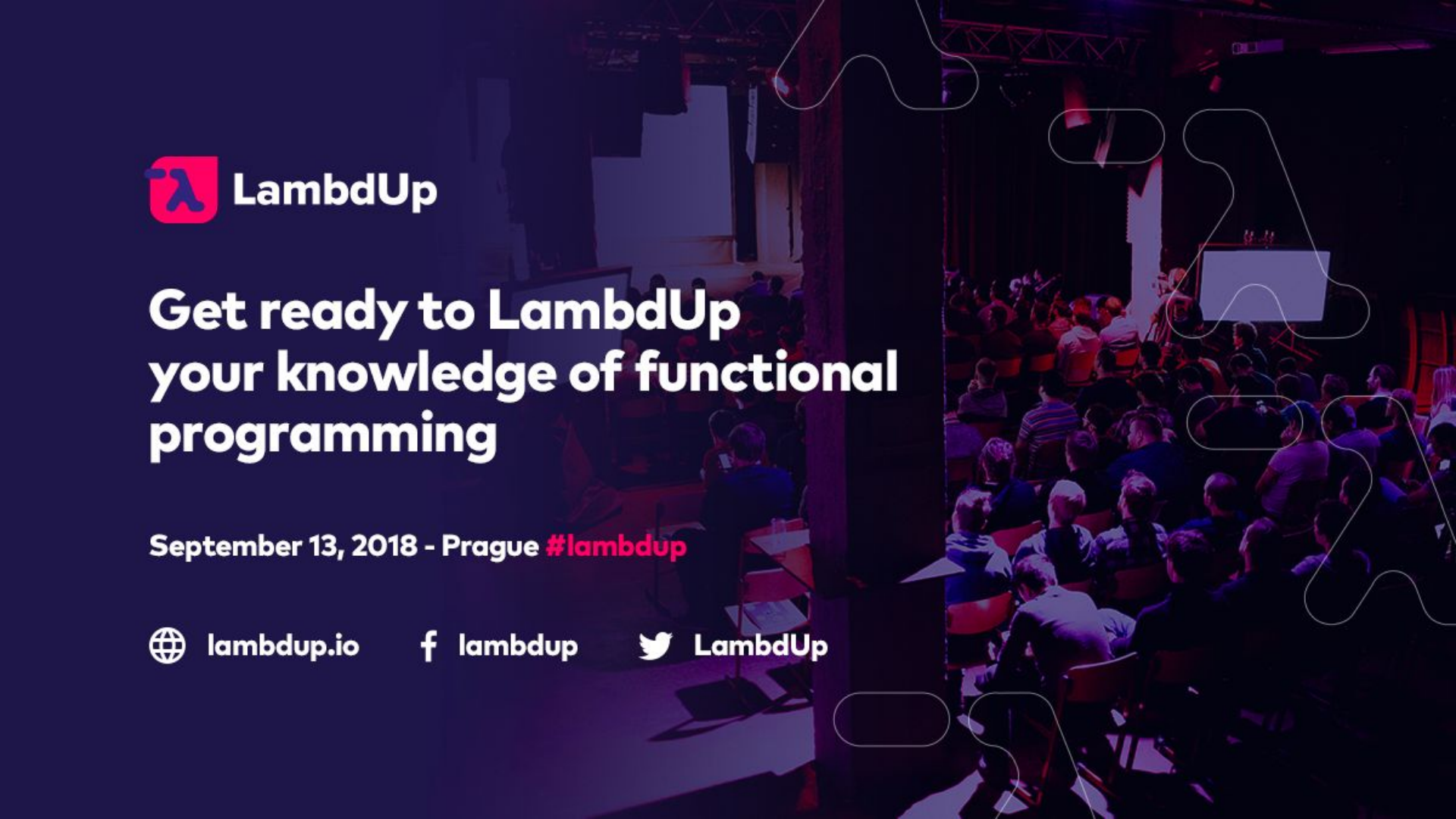
http://erlang.org/doc/man/eprof.html

```
iex(1)> SpecialRunner.run_it
message profile

FUNCTION                                                    CALLS        %       TIME  [uS / CALLS]
--------                                                    -----  -------       ----  [----------]
io:o_request/3                                                  1     0.00          0  [      0.00]
'Elixir.File.Stream':'__build__'/3                              1     0.00          0  [      0.00]
prim_file:drv_close/1                                           2     0.00          0  [      0.00]

...

re:check_for_unicode/2                                   11542821     1.73    7683520  [      0.67]
re:postprocess/5                                         11712821     1.77    7847086  [      0.67]
re:grun2/3                                               11542821     1.86    8236662  [      0.71]
'Elixir.PpcbeeElixirBackend.StringFormat':normalize_downcase/1  10467017     1.86    8250051  [      0.79]
'Elixir.Keyword':get/2                                   11451509     1.87    8275974  [      0.72]
'Elixir.Regex':'regex?'/1                                11381509     1.89    8381970  [      0.74]
'Elixir.Regex':precompile_replacement/1                  11681001     1.94    8571924  [      0.73]
'Elixir.Access':get/3                                    11893330     1.95    8635278  [      0.73]
re:grun/3                                                11542821     2.02    8959633  [      0.78]
lists:keyfind/3                                          23524084     2.10    9283667  [      0.39]
re:loopexec/7                                            15547467     2.85   12608011  [      0.81]
'Elixir.String.Unicode':next_extend_size/3               48675779     4.14   18342173  [      0.38]
'Elixir.String.Normalizer':normalize_nfd/2               56494052     5.53   24497157  [      0.43]
'Elixir.String.Casing':downcase/2                        53056724     5.58   24707051  [      0.47]
re:process_parameters/6                                  46144972     5.89   26088881  [      0.57]
binary:part/3                                            45897035     6.32   28011653  [      0.61]
'Elixir.String.Unicode':next_grapheme_size/1             48925823     8.49   37612282  [      0.77]
re:run/3                                                 27964312    11.38   50392515  [      1.80]
-------------------------------------------------------  ---------  -------  ---------  [----------]
Total:                                                  729210223  100.00%  442943730  [      0.61]
"total change= 99.58999999999999"
```

```
39
40    def remove_all_stopwords(text, _option) do
41      text
42        |> normalize()
43        |> String.split(@spaces_delimiter)
44        |> Enum.reduce([], fn(x, acc) ->
45  -       has_stopword = Enum.any?(@stopwords, &(&1 == downcase(x)))
46
47  -       (has_stopword && acc) || (acc ++ [x])
48        end)
49        |> Enum.join(" ")
50    end
```

```
42
43    def remove_all_stopwords(text, _option) do
44      text
45        |> normalize()
46        |> String.split(@spaces_delimiter)
47        |> Enum.reduce([], fn(x, acc) ->
48  +       has_stopword = String.match?(x, @stopwords_regexp)
49
50  +       if has_stopword, do: acc, else: acc ++ [x]
51        end)
52        |> Enum.join(" ")
53    end
```

Benchmarking is not real user!


LambdUp
Get ready to LambdUp your knowledge of functional programming
September 13, 2018 - Prague #lambdup
lambdup.io
lambdup
LambdUp

Thank you :)

Sources

- https://www.issart.com/blog/benchmarking-best-practice-for-code-optimization
- https://pragtob.wordpress.com/2016/06/16/tail-call-optimization-in-elixir-erlang-not-as-efficient-and-important-as-you-probably-think/
- https://www.youtube.com/watch?v=KSrImdsfjL4
- https://www.amberbit.com/blog/2018/5/15/when-to-use-elixir-language/
- https://github.com/parroty/exprof
- https://github.com/PragTob/benchee
- https://www.ppcbee.com/