IS SOUND GRADUAL TYPING DEAD?

Asumu Takikawa Daniel Feltey Ben Greenman

Max S. New Jan Vitek Matthias Felleisen

GRADUAL TYPING THESIS

1. People write untyped code
2. Static types help maintain software
3. *Sound types* can be added *incrementally*
4. Types *respect existing code* & the result is *runnable*

SOUND TYPES

UNSOUND TYPED

```
#lang typed/racket/unsound

; fact.rkt

(provide fact)

(: fact (-> Integer Integer))
(define (fact n)
  (if (zero? n)
      1
      (* n (sub1 n))))
```

UNTYPED

```
#lang racket

; use.rkt

(require "fact.rkt")

(fact "ill-typed call")
```

UNSOUND TYPED

```
#lang typed/racket/unsound

; fact.rkt

(provide fact)

(: fact (-> Inte
(define (fact n)
  (if (zero? n)
      1
      (* n (sub1 n))))
```

UNTYPED

```
#lang racket

; use.rkt

                fact.rkt")

                -typed call")
```

```
; zero?: contract violation
;   expected: number?
;   given: "ill-typed call"
; [,bt for context]
```

~~UNSOUND~~ TYPED

```
#lang typed/racket

; fact.rkt

(provide fact)

(: fact (-> Integer Integer))
(define (fact n)
  (if (zero? n)
      1
      (* n (sub1 n))))
```

UNTYPED

```
#lang racket

; use.rkt

(require "fact.rkt")

(fact "ill-typed call")
```

~~UNSOUND~~ TYPED

```
#lang typed/racket

; fact.rkt

(provide fact)

(: fact (-> Int
(define (fact n
  (if (zero? n)
      1
      (* n (subt n))))
```

UNTYPED

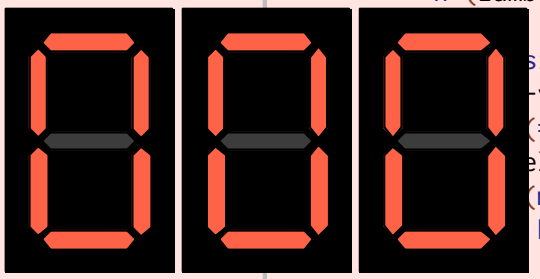
```
#lang racket

act.rkt")

typed call")
```

```
; fact: contract violation
;   expected: Integer
;   given: "ill-typed call"
;   in: the 1st argument of
;       (-> Integer any)
;   contract from: "fact.rkt"
;   blaming: "use.rkt"
```

RESULTS ARE RUNNABLE

Prime number sieve

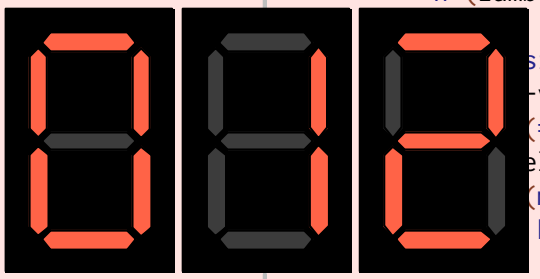
```
#lang racket/base

(provide (struct-out stream)
         make-stream stream-unfold
         stream-get stream-take)
(struct stream (first rest))

(define (make-stream hd thunk)
  (stream hd thunk))

(define (stream-unfold st)
  (values (stream-first st)
          ((stream-rest st))))

(define (stream-get st i)
  (define-values (hd tl)
    (stream-unfold st))
  (cond [(= i 0) hd]
        [else (stream-get tl (sub1 i))]))

(define (stream-take st n)
  (cond [(= n 0) '()]
        [else
         (define-values (hd tl) (stream-unfold st))
         (cons hd (stream-take tl (sub1 n)))]))
```



```
#lang racket/base

(require "streams.rkt")

(define (count-from n)
  (make-stream
   n (lambda () (count-from (add1 n)))))

sift n st)
-values (hd tl) (stream-unfold st))
= 0 (modulo hd n)) (sift n tl)]
lse
make-stream
hd (lambda () (sift n tl)))]))

(define (sieve st)
  (define-values (hd tl)
    (stream-unfold st))
  (make-stream hd (lambda () (sieve (sift hd tl)))))

(define primes (sieve (count-from 2)))

(define (main)
  (printf "The ~a-th prime number is: ~a\n" 100
          (stream-get primes 99)))

(time (main))
```

Prime number sieve

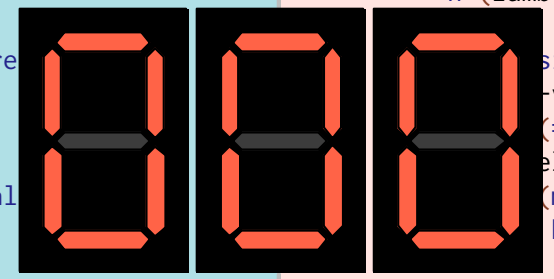
```
#lang racket/base

(provide (struct-out stream)
         make-stream stream-unfold
         stream-get stream-take)
(struct stream (first rest))

(define (make-stream hd thunk)
  (stream hd thunk))

(define (stream-unfold st)
  (values (stream-first st)
          ((stream-rest st))))

(define (stream-get st i)
  (define-values (hd tl)
    (stream-unfold st))
  (cond [(= i 0) hd]
        [else (stream-get tl (sub1 i))]))

(define (stream-take st n)
  (cond [(= n 0) '()]
        [else
         (define-values (hd tl) (stream-unfold st))
         (cons hd (stream-take tl (sub1 n)))]))
```



```
#lang racket/base

(require "streams.rkt")

(define (count-from n)
  (make-stream
   n (lambda () (count-from (add1 n)))))

sift n st)
-values (hd tl) (stream-unfold st))
= 0 (modulo hd n)) (sift n tl)]
else
(make-stream
 hd (lambda () (sift n tl)))]))

(define (sieve st)
  (define-values (hd tl)
    (stream-unfold st))
  (make-stream hd (lambda () (sieve (sift hd tl)))))

(define primes (sieve (count-from 2)))

(define (main)
  (printf "The ~a-th prime number is: ~a\n" 100
          (stream-get primes 99)))

(time (main))
```

Prime number sieve

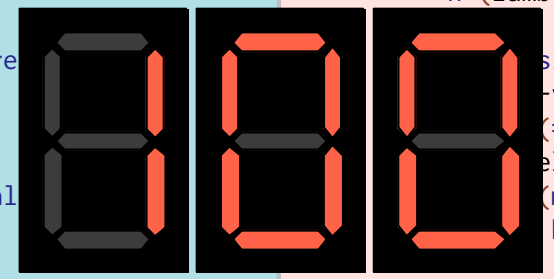
```
#lang typed/racket/base

(provide (struct-out stream)
         make-stream stream-unfold
         stream-get stream-take)
(struct: stream ([first : Natural]
                 [rest : (-> stream)]))

(: make-stream (-> Natural (-> stream) stre
(define (make-stream hd thunk)
  (stream hd thunk))

(: stream-unfold (-> stream (values Natural
(define (stream-unfold st)
  (values (stream-first st)
          ((stream-rest st))))

(: stream-get (-> stream Natural Natural))
(define (stream-get st i)
  (define-values (hd tl)
    (stream-unfold st))
  (cond [(= i 0) hd]
        [else (stream-get tl (sub1 i))]))

(: stream-take (-> stream Natural (Listof Natural)))
(define (stream-take st n)
  (cond [(= n 0) '()]
        [else
         (define-values (hd tl) (stream-unfold st))
         (cons hd (stream-take tl (sub1 n)))]))
```



```
#lang racket/base

(require "streams.rkt")

(define (count-from n)
  (make-stream
   n (lambda () (count-from (add1 n)))))

                sift n st)
               -values (hd tl) (stream-unfold st))
               (= 0 (modulo hd n)) (sift n tl)]
               else
               (make-stream
                hd (lambda () (sift n tl)))]))

(define (sieve st)
  (define-values (hd tl)
    (stream-unfold st))
  (make-stream hd (lambda () (sieve (sift hd tl)))))

(define primes (sieve (count-from 2)))

(define (main)
  (printf "The ~a-th prime number is: ~a\n" 100
          (stream-get primes 99)))

(time (main))
```

Prime number sieve

```
#lang typed/racket/base

(provide (struct-out stream)
         make-stream stream-unfold
         stream-get stream-take)
(struct: stream ([first : Natural]
                 [rest : (-> stream)]))

(: make-stream (-> Natural (-> stream) stre
(define (make-stream hd thunk)
  (stream hd thunk))

(: stream-unfold (-> stream (values Natural
(define (stream-unfold st)
  (values (stream-first st)
          ((stream-rest st))))

(: stream-get (-> stream Natural Natural))
(define (stream-get st i)
  (define-values (hd tl)
    (stream-unfold st))
  (cond [(= i 0) hd]
        [else (stream-get tl (sub1 i))]))

(: stream-take (-> stream Natural (Listof Natural)))
(define (stream-take st n)
  (cond [(= n 0) '()]
        [else
         (define-values (hd tl) (stream-unfold st))
         (cons hd (stream-take tl (sub1 n)))]))
```



```
#lang racket/base

(require "streams.rkt")

(define (count-from n)
  (make-stream
   n (lambda () (count-from (add1 n)))))

                sift n st)
               -values (hd tl) (stream-unfold st))
               (= 0 (modulo hd n)) (sift n tl)]
               else
               (make-stream
                hd (lambda () (sift n tl)))]))

(define (sieve st)
  (define-values (hd tl)
    (stream-unfold st))
  (make-stream hd (lambda () (sieve (sift hd tl)))))

(define primes (sieve (count-from 2)))

(define (main)
  (printf "The ~a-th prime number is: ~a\n" 100
          (stream-get primes 99)))

(time (main))
```

10x slowdown could make the software undeliverable

Anecdotes from users

"The end-product appears to be a 50%-performance hybrid due to boundary contracts" 2x

2.5x "At this point, about one-fifth of my code is now typed. Unfortunately, this version is 2.5 times slower"

"On my machine, it takes *twelve seconds* ... ... the time taken is 1ms" 12,000x

“As a practitioner, there are costs associated with using TR, therefore it has to provide equivalent performance improvements to be worthwhile at all.”

— Matthew Butterick

Why is it slow?

Bad programming / isolated incidents?

Bad implementation / design?

Fundamental issue with gradual typing?

To answer, we need an evaluation method

CONTRIBUTIONS OF OUR PAPER

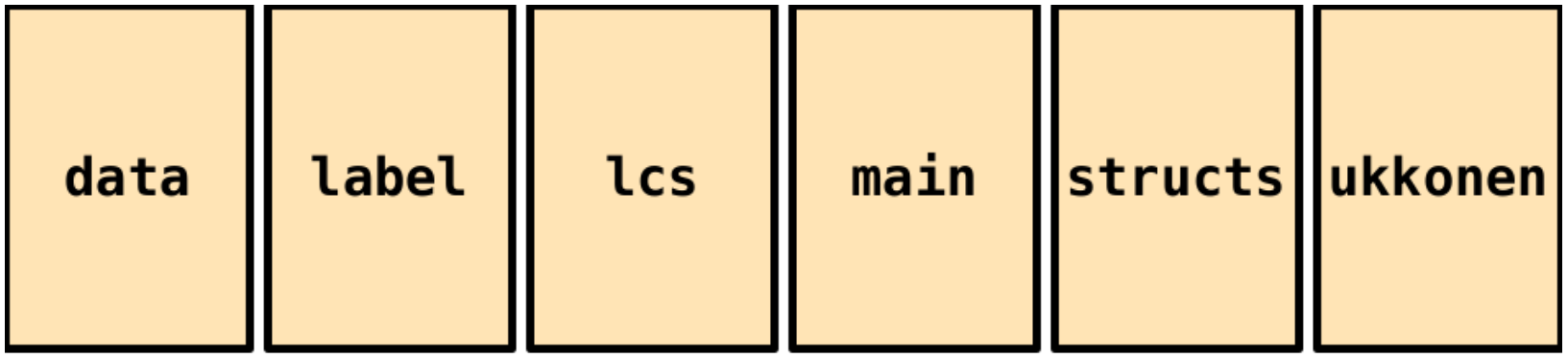
- **Evaluation method** for language implementors
- Idea for **graphically summarizing** evaluation results
- Results of **evaluating Typed Racket** using the method

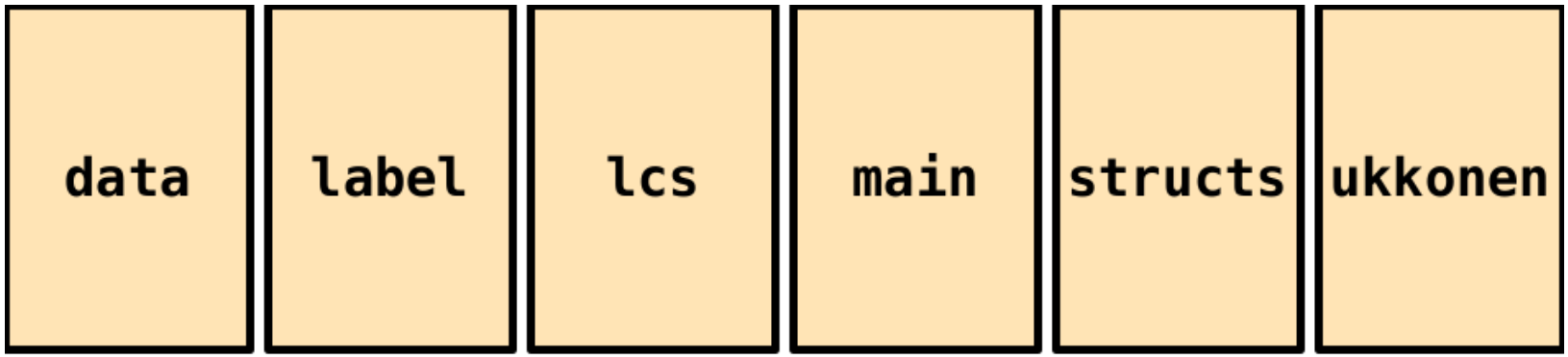
KEY CONCEPTS

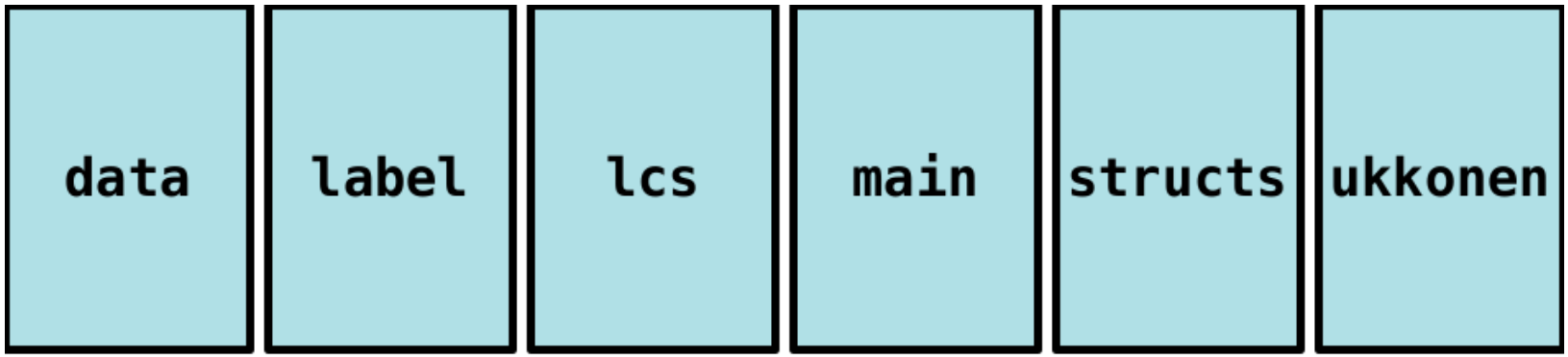
Programmers add types incrementally

so should the evaluation method

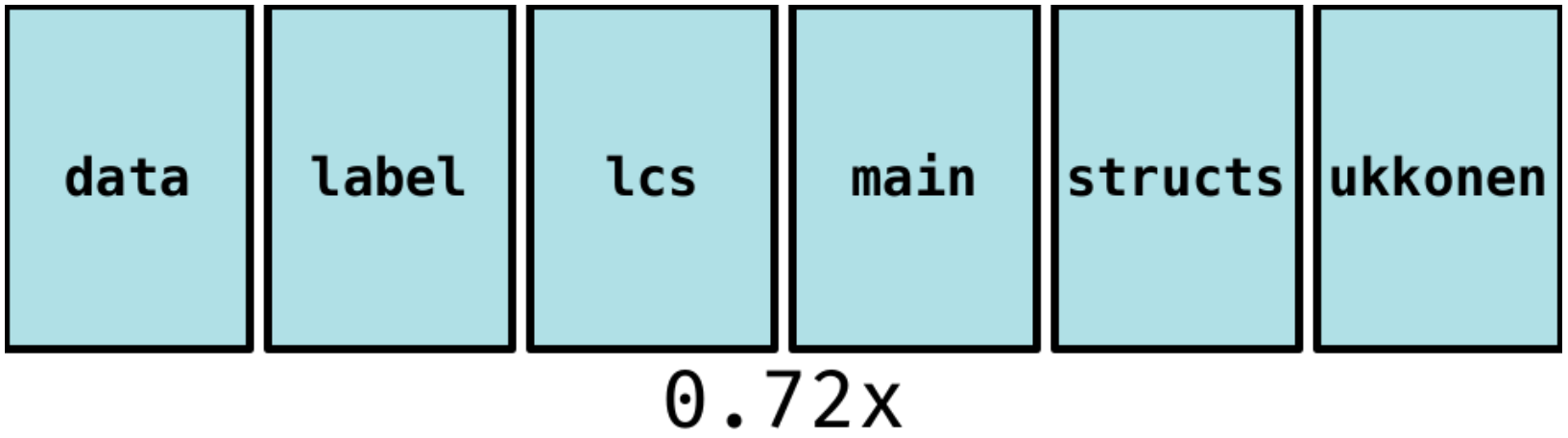
Suffixtree benchmark with 6 modules

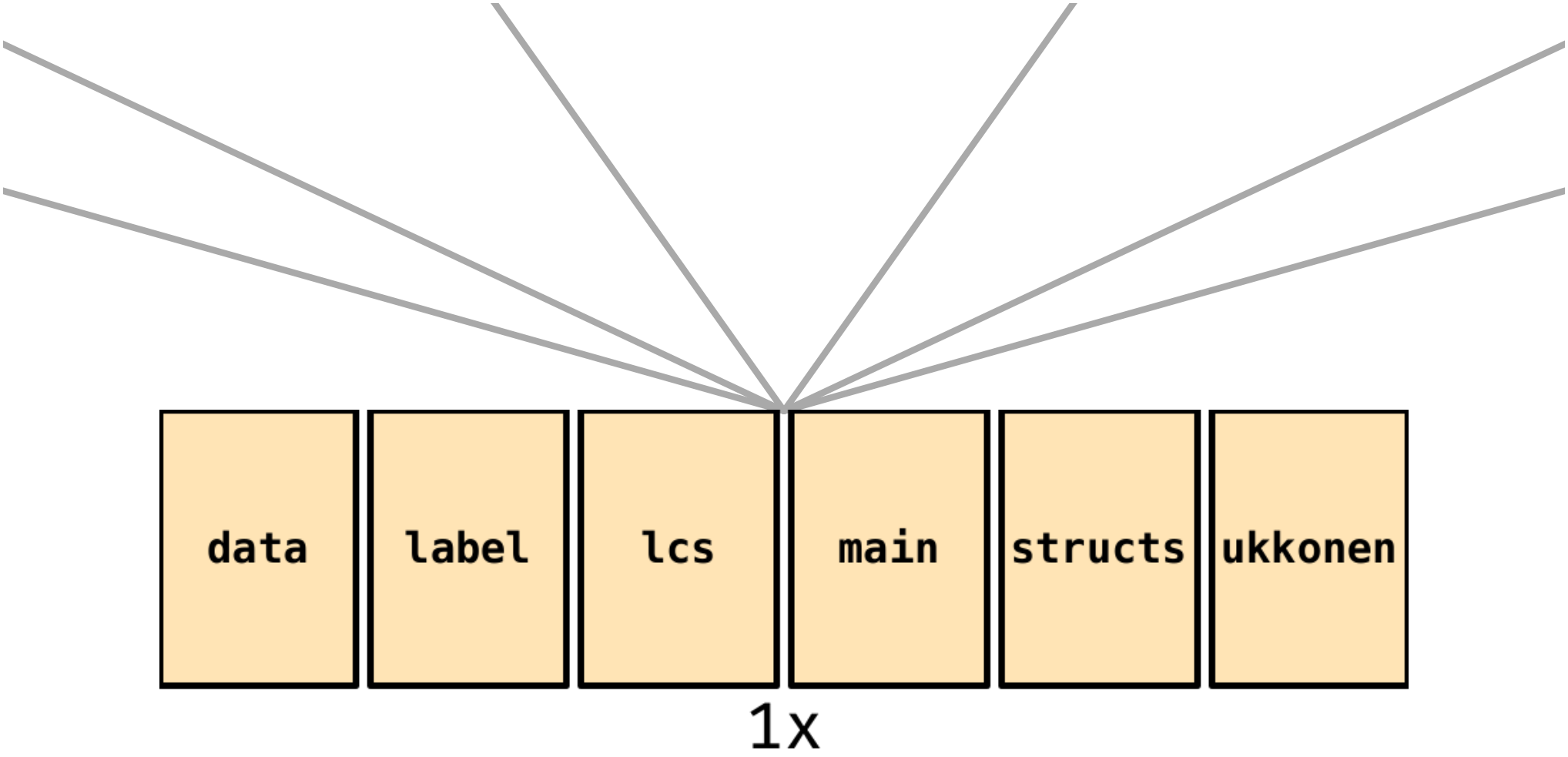


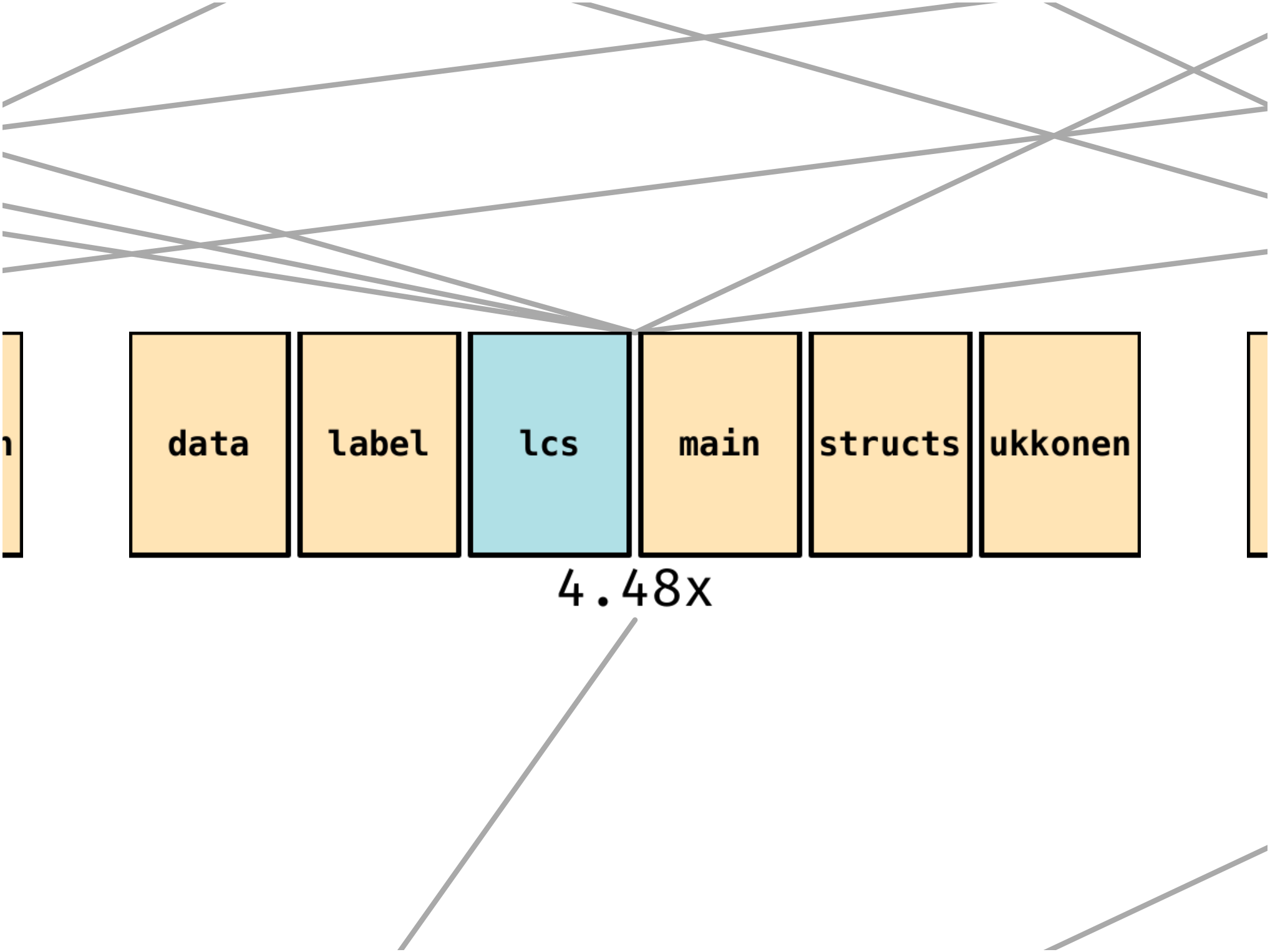

data
label
lcs
main
structs
ukkonen
2610 ms / 1x

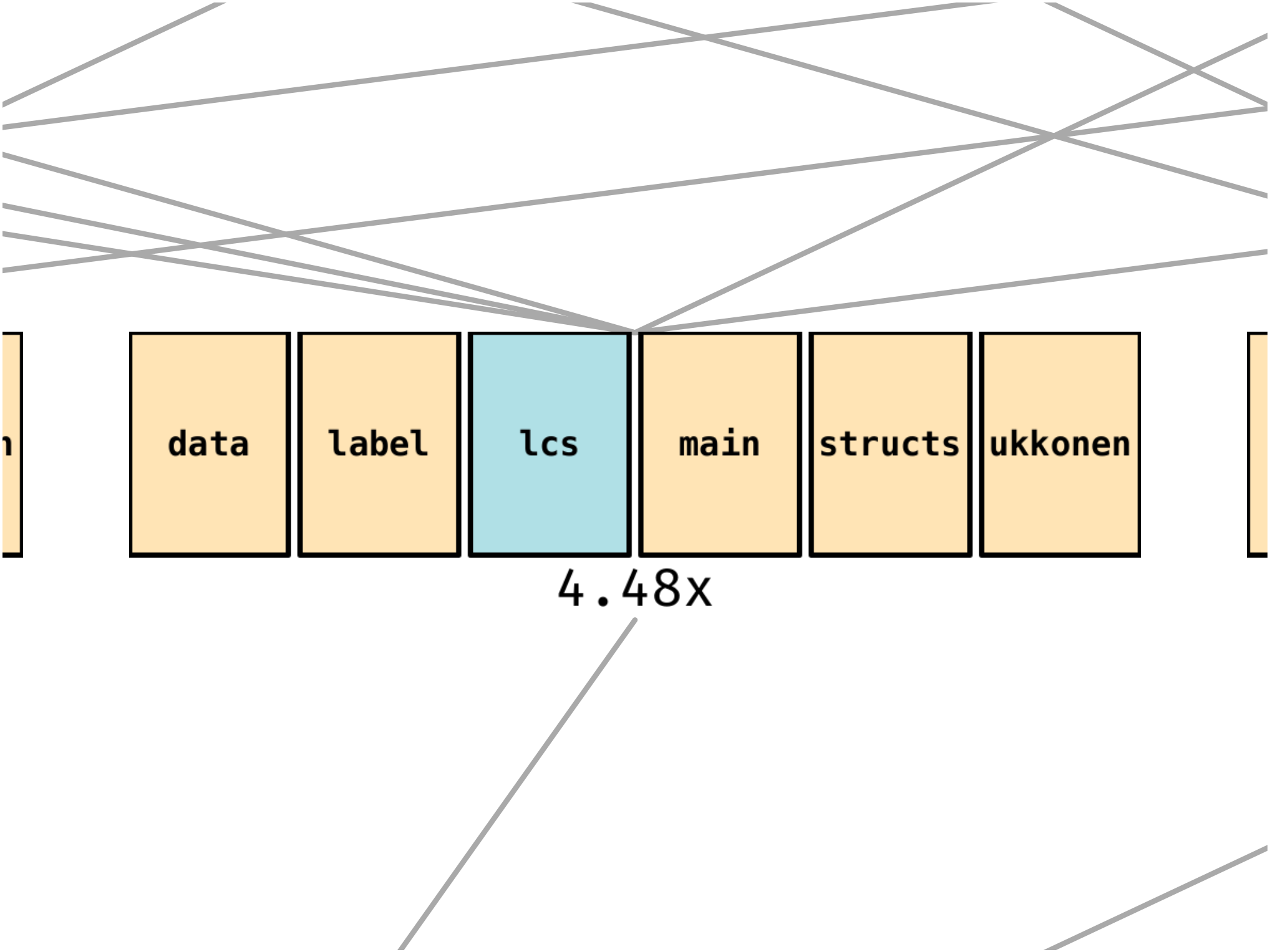

data
label
lcs
main
structs
ukkonen
0.72x

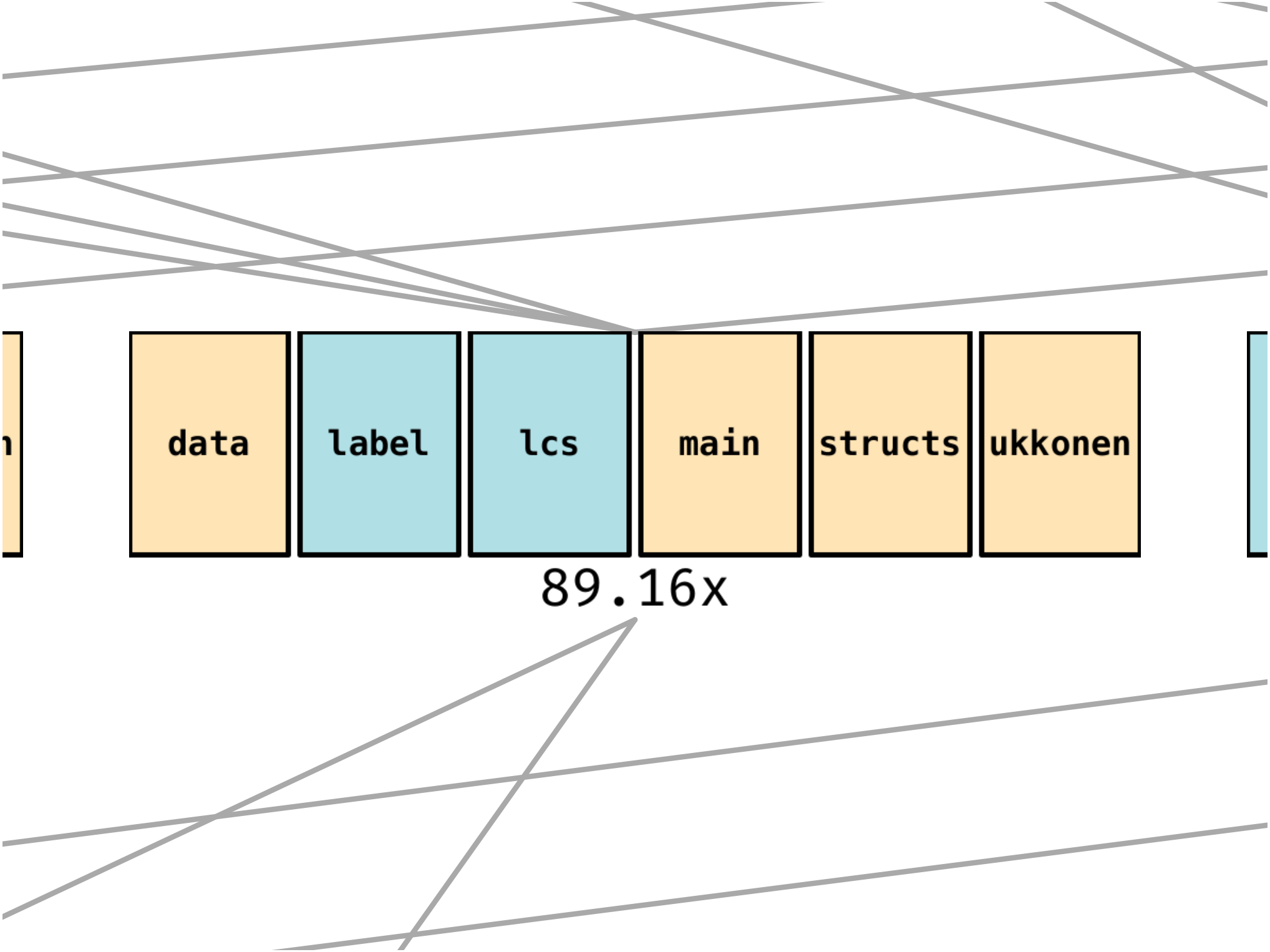
Reminder: incremental addition of types

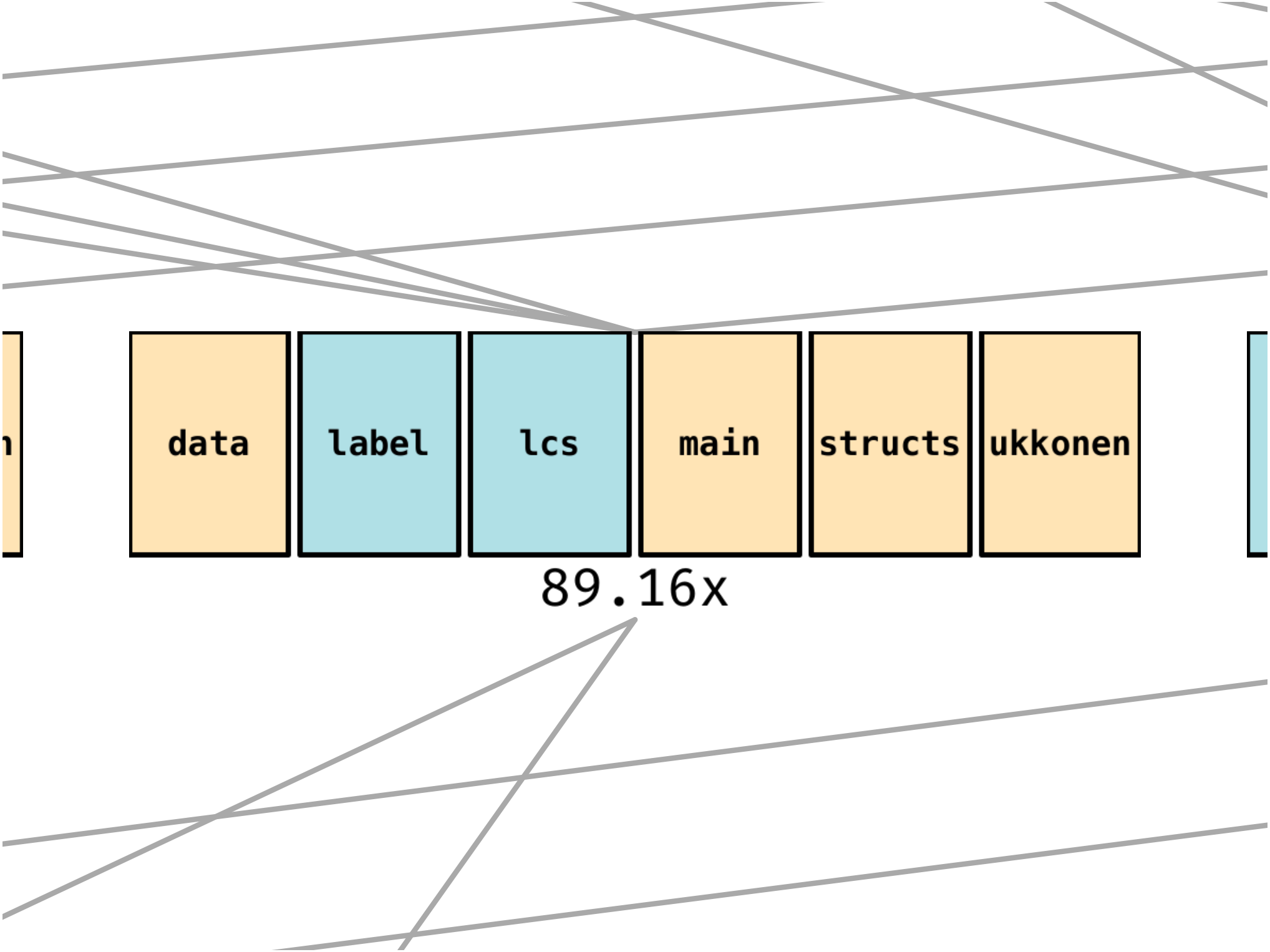


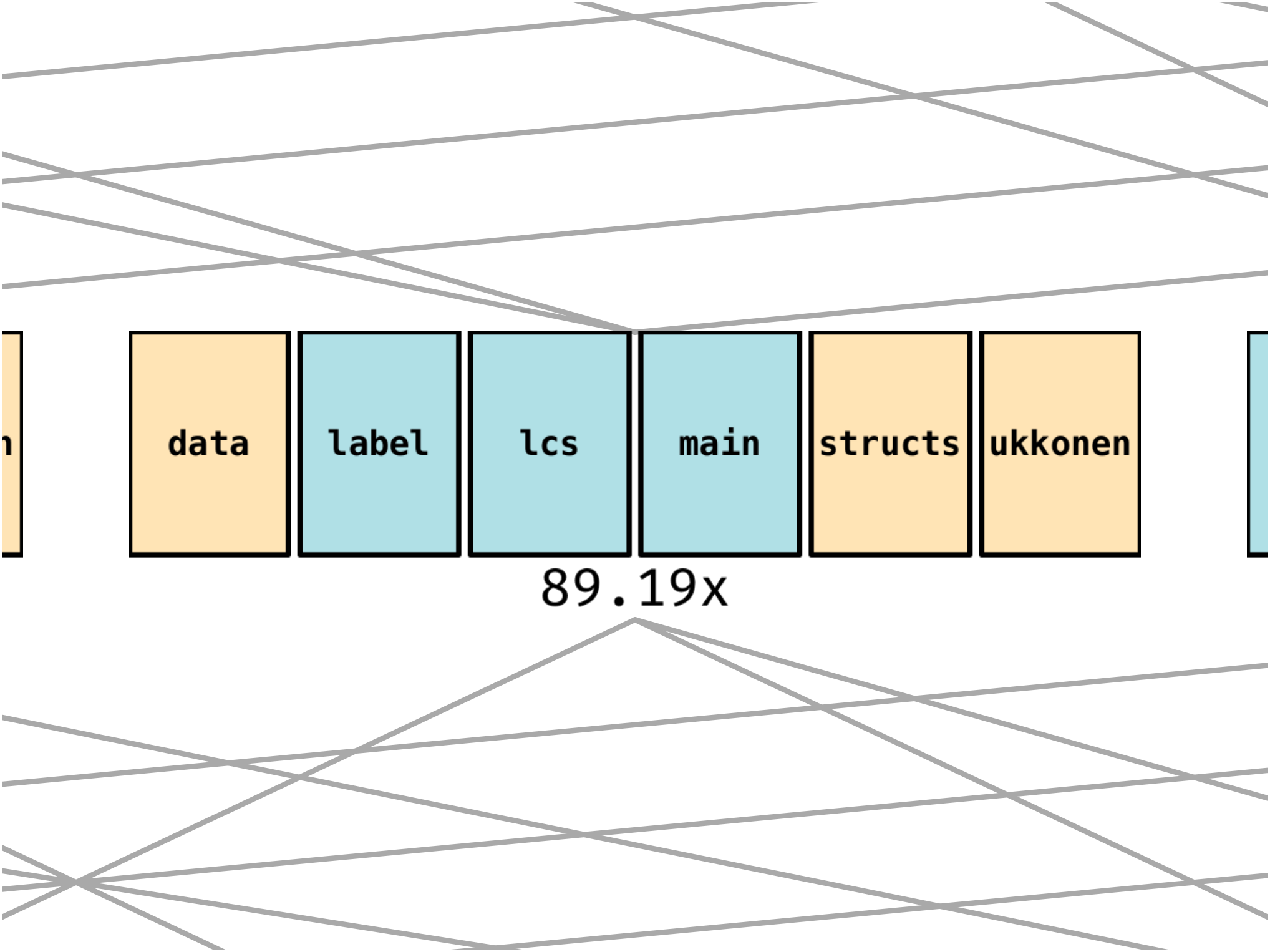

data
label
lcs
main
structs
ukkonen
1x

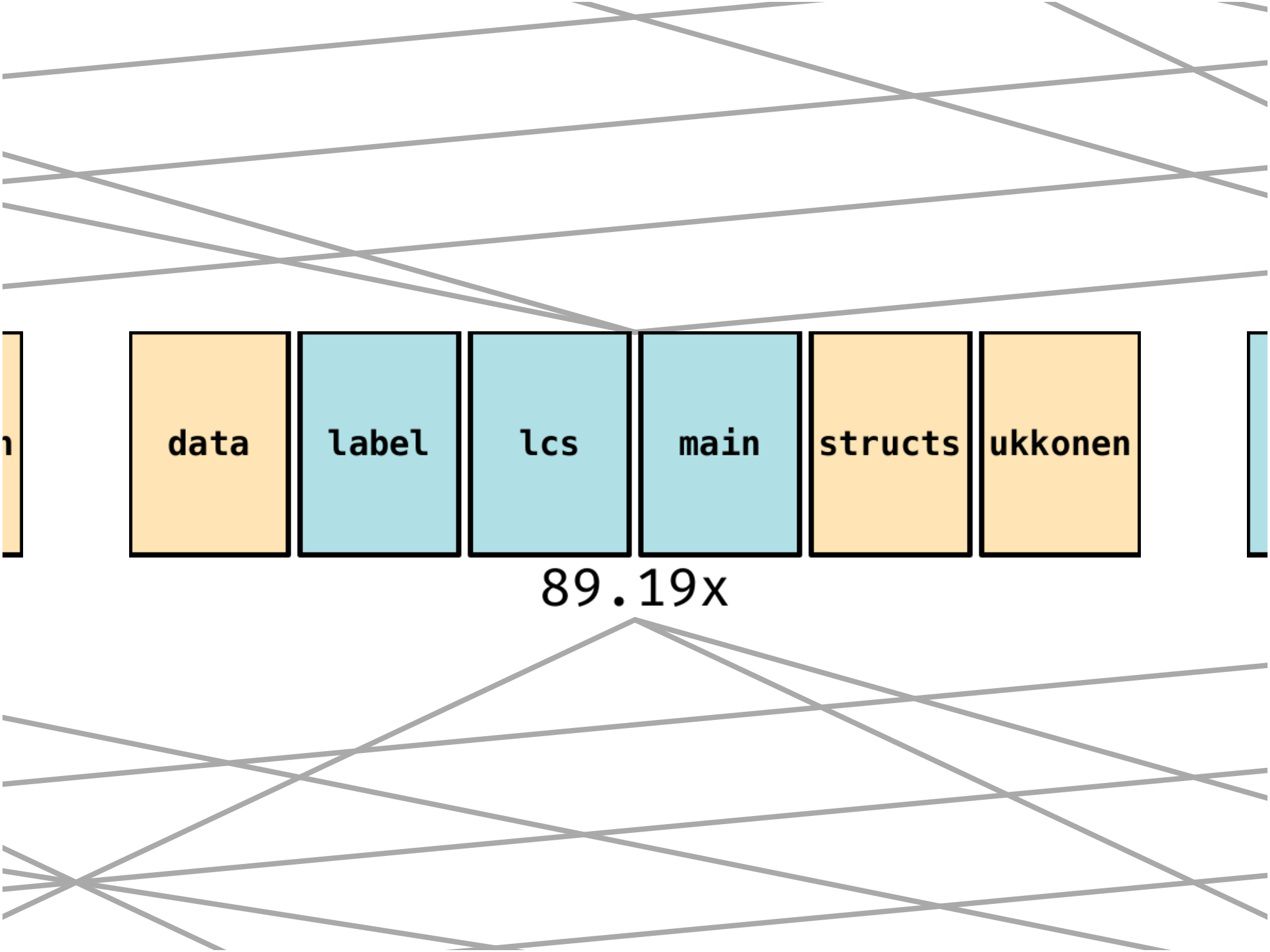

data
label
lcs
main
structs
ukkonen
4.48x

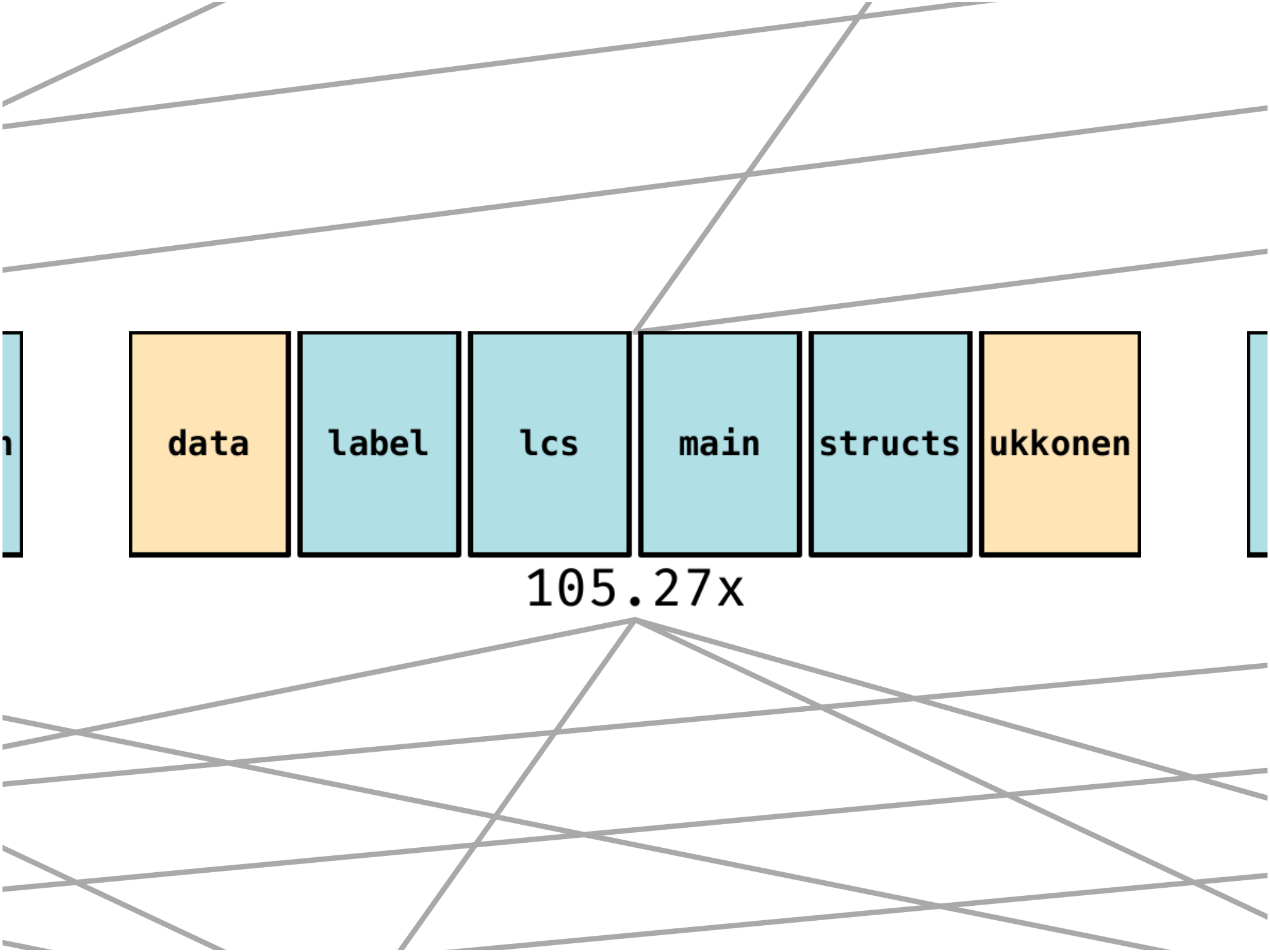

data
label
lcs
main
structs
ukkonen
4.48x

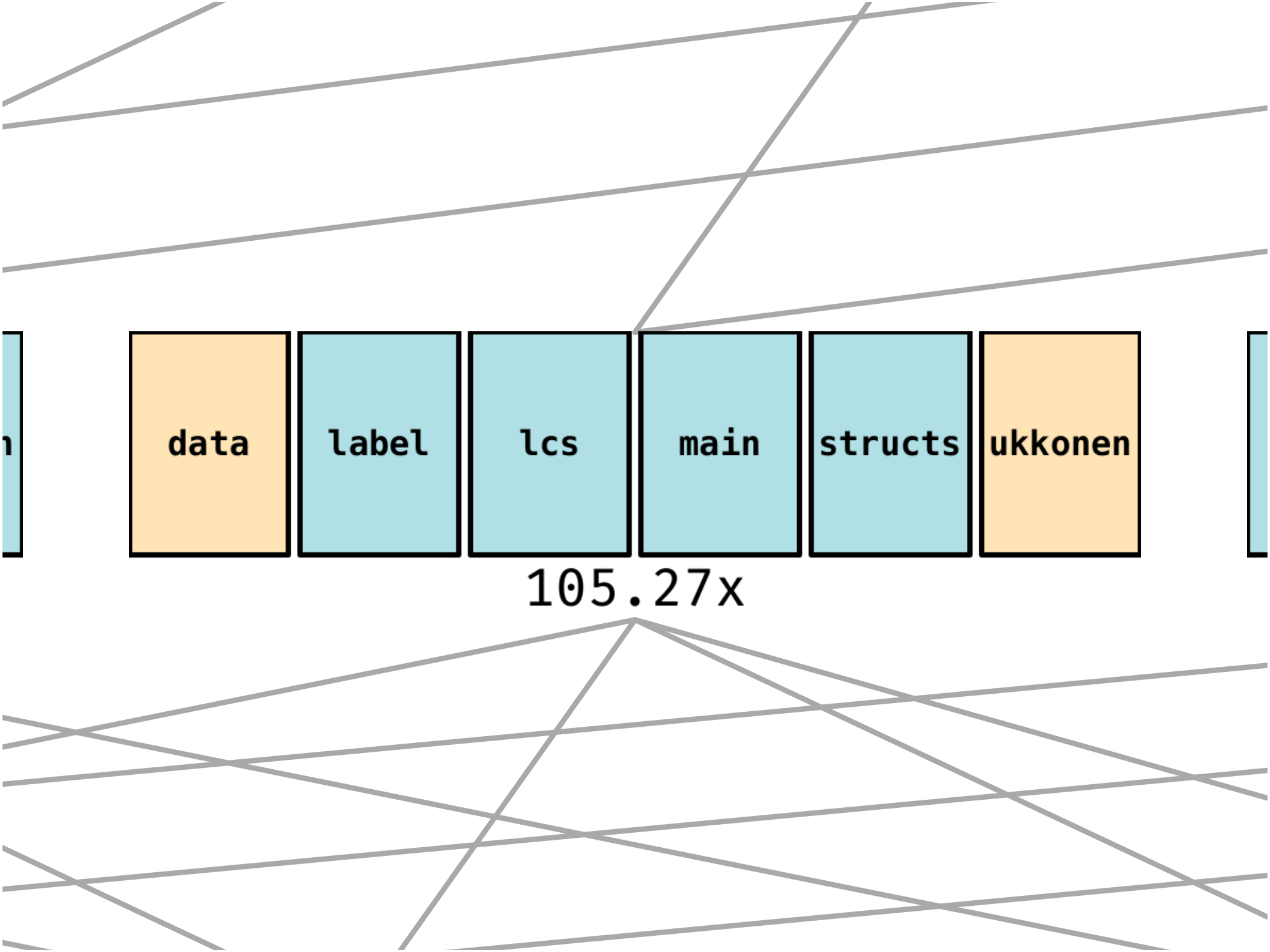

data
label
lcs
main
structs
ukkonen
89.16x

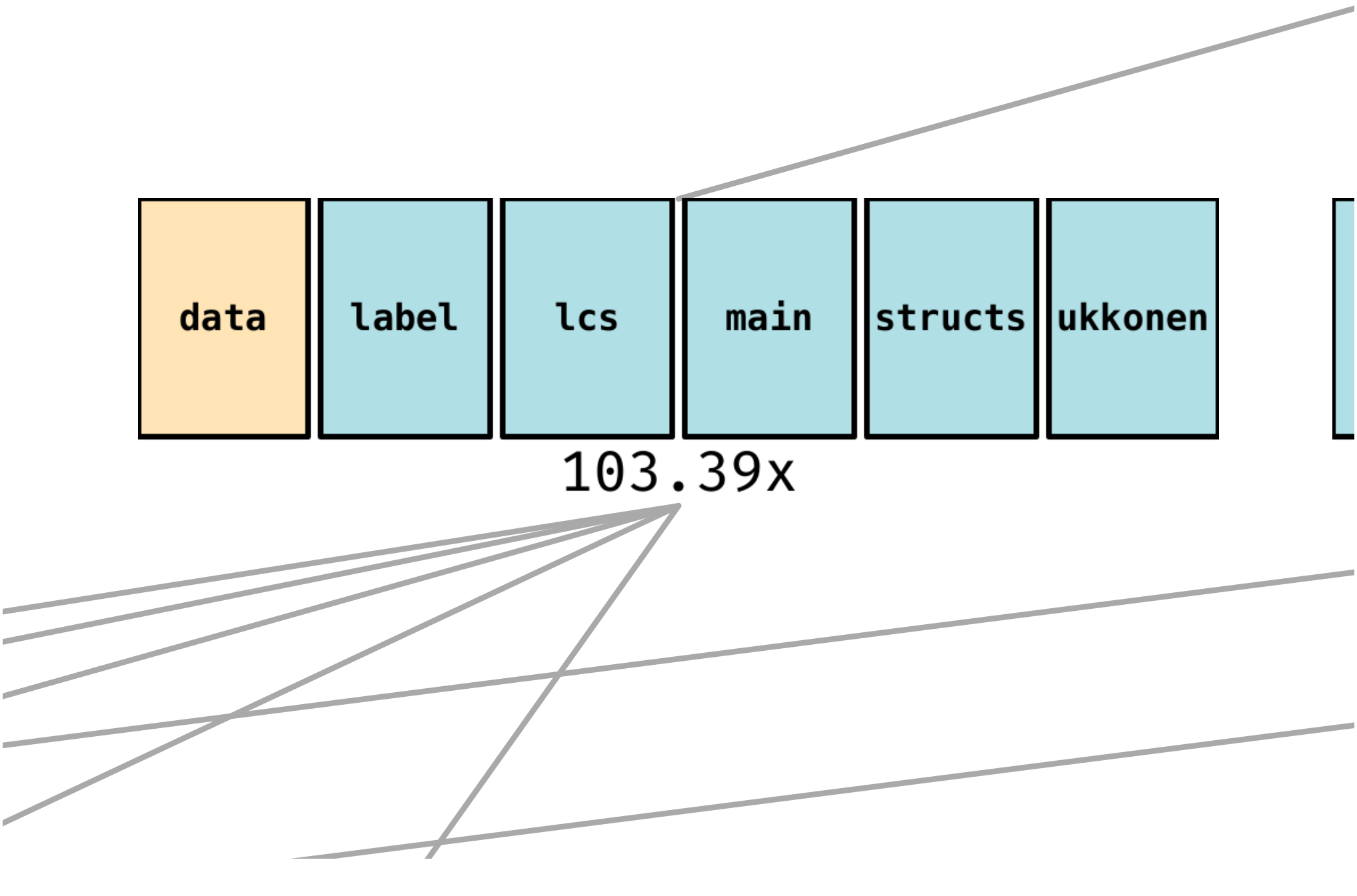

data
label
lcs
main
structs
ukkonen
89.16x

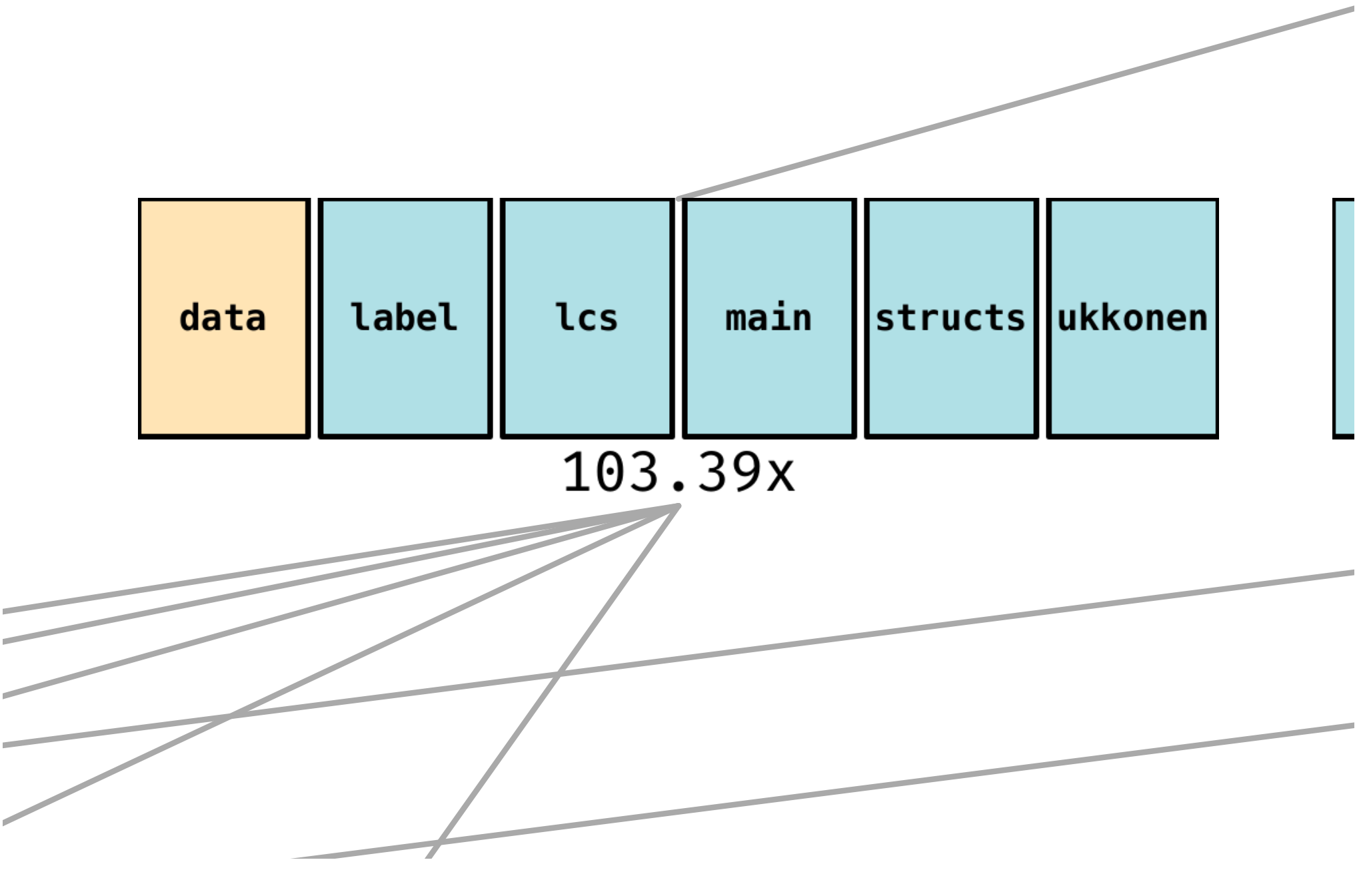

data
label
lcs
main
structs
ukkonen
89.19x

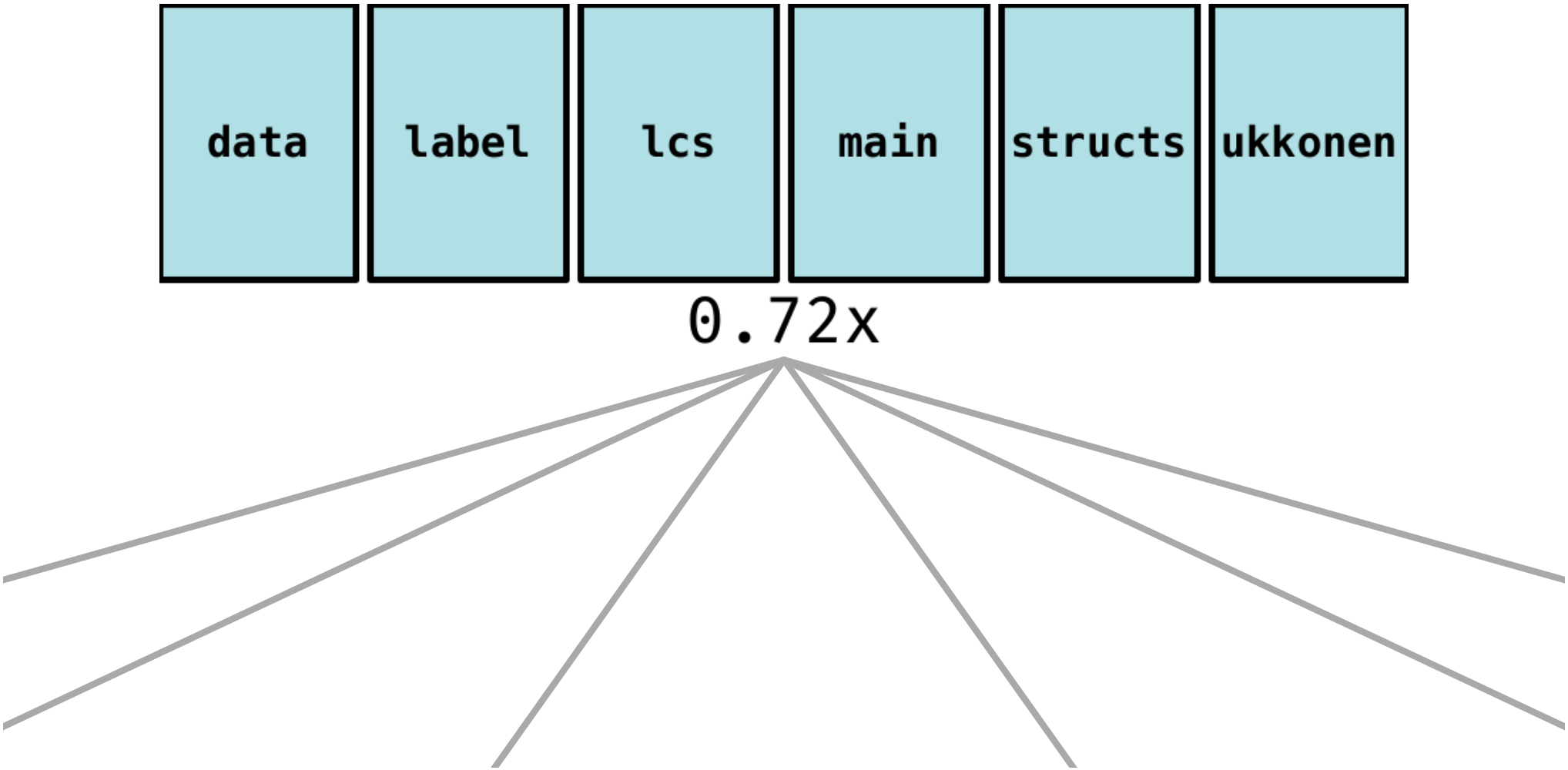

data
label
lcs
main
structs
ukkonen
89.19x

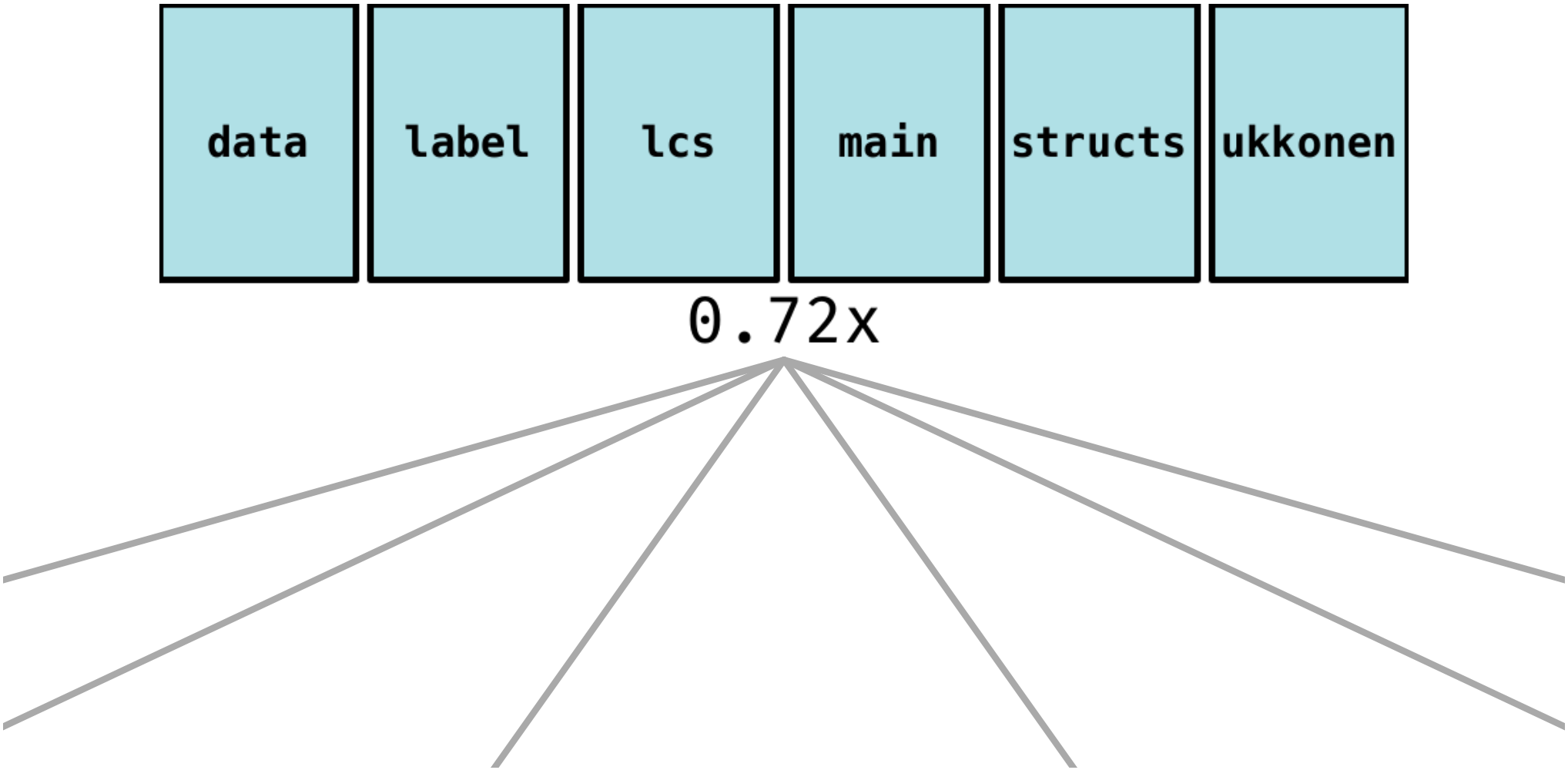

data
label
lcs
main
structs
ukkonen
105.27x

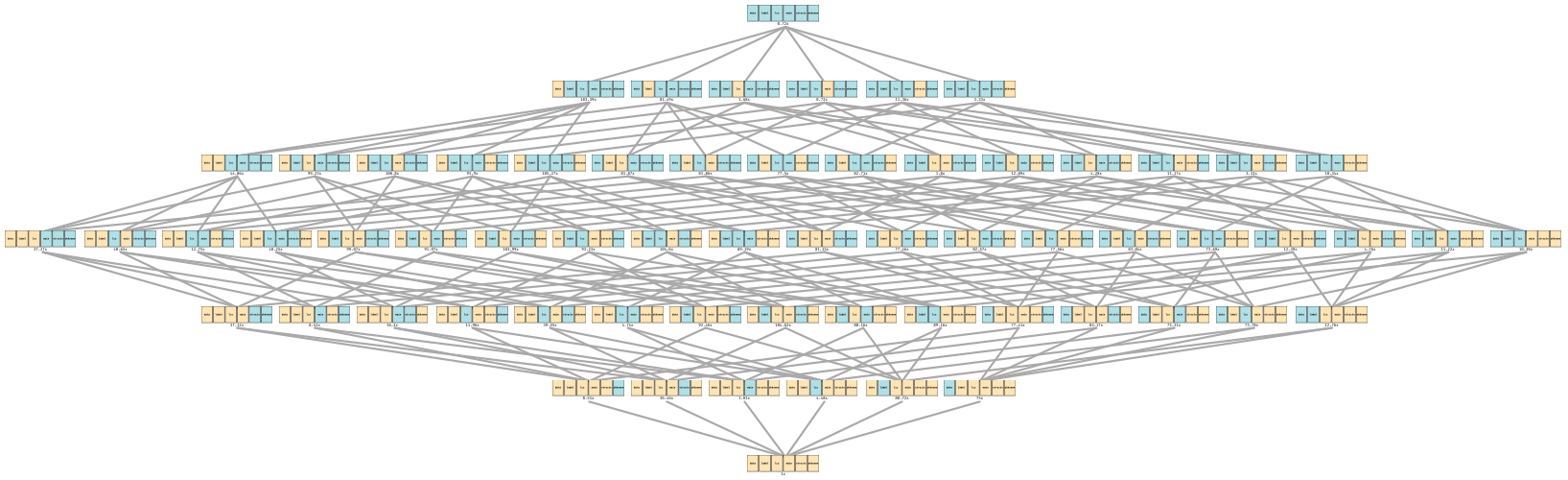

data
label
lcs
main
structs
ukkonen
105.27x


data
label
lcs
main
structs
ukkonen
103.39x


data
label
lcs
main
structs
ukkonen
103.39x

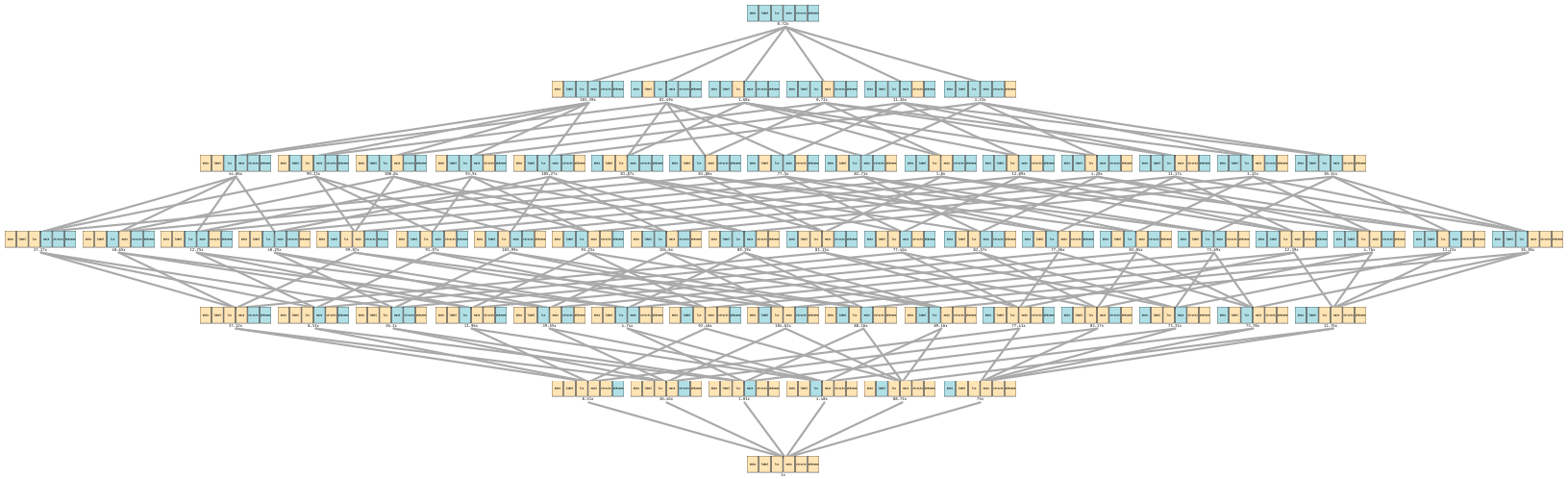

data
label
lcs
main
structs
ukkonen
0.72x


data
label
lcs
main
structs
ukkonen
0.72x



The performance lattice

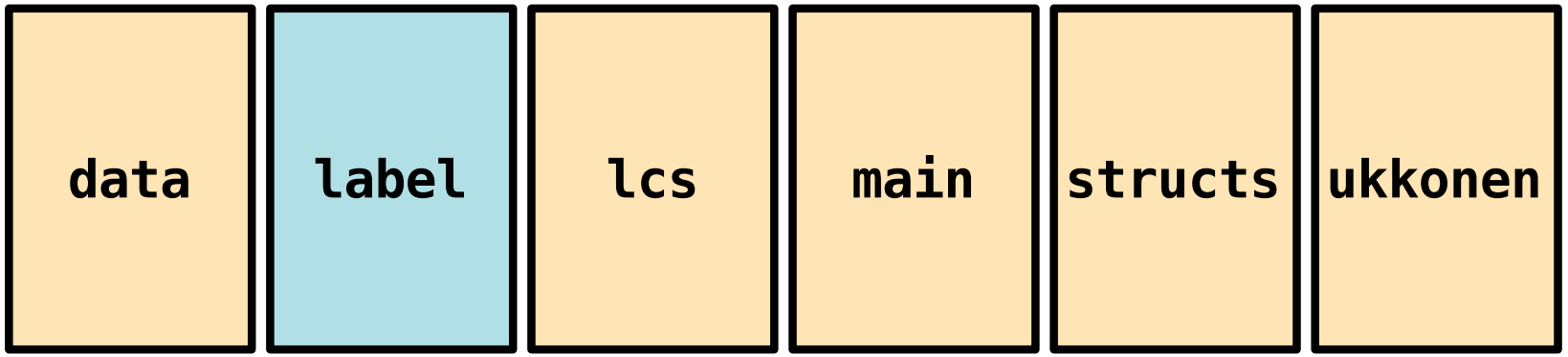
Paths in lattice are gradual migration paths



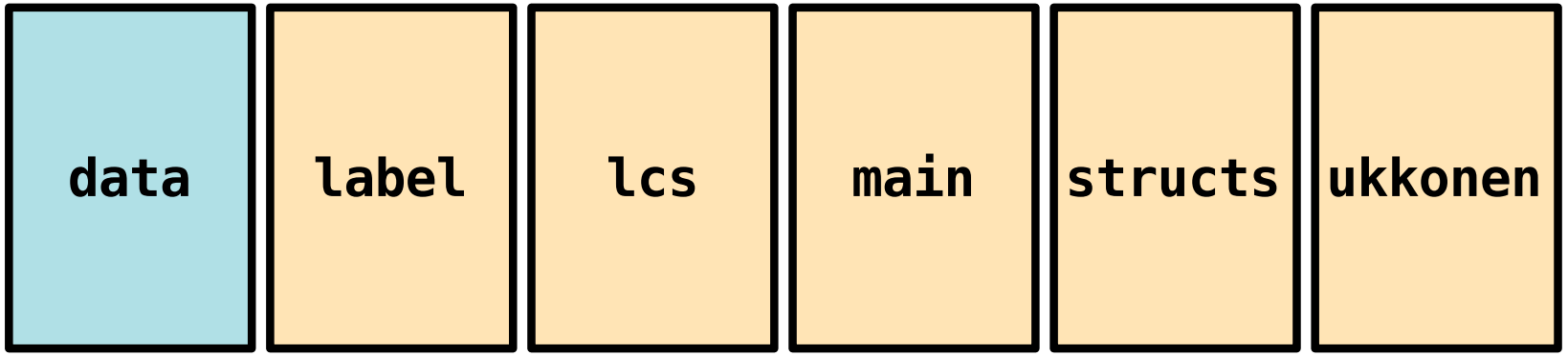
Why are the configs useful?

Reveals the cost of boundaries in gradual programs

Shows paths from untyped to typed

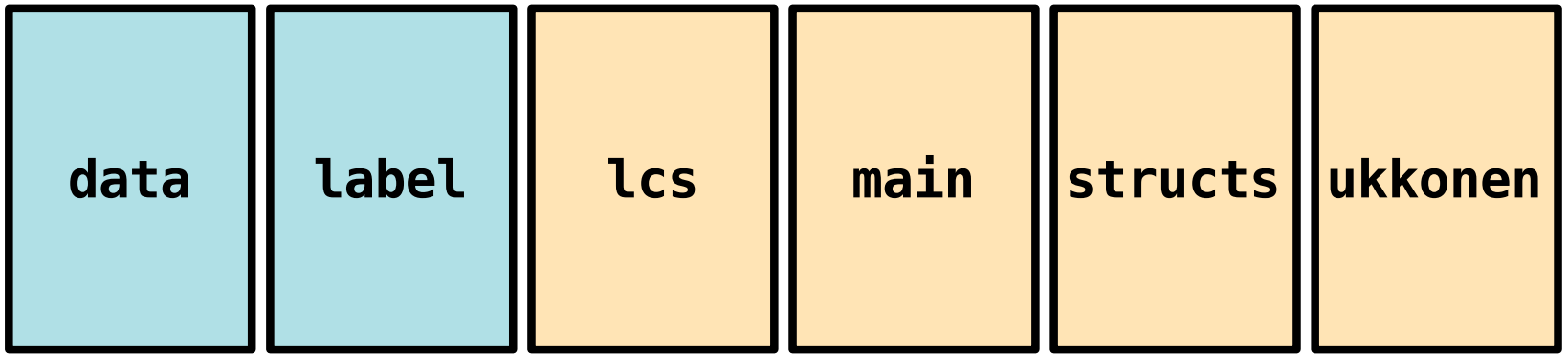


231570 ms / 88.72x

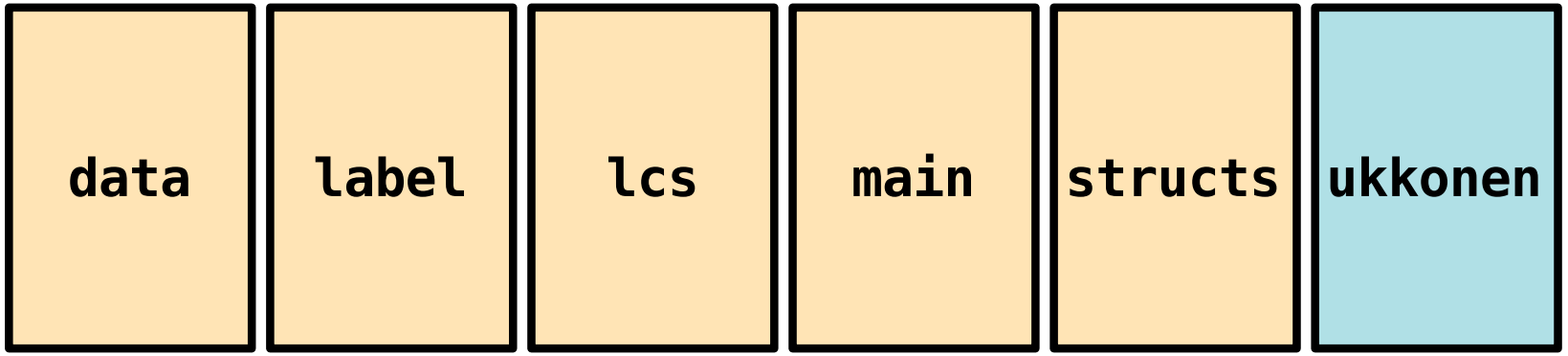


195755 ms / 75x

Data / Label boundary is costly



33294 ms / 12.76x

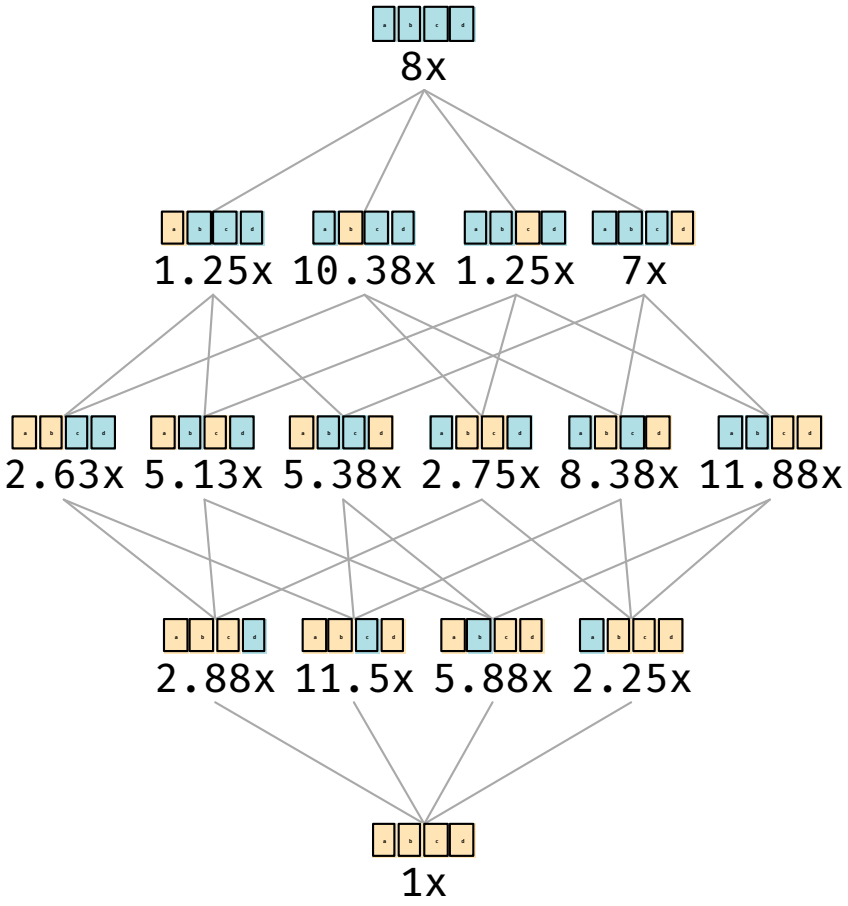


22203 ms / 8.51x

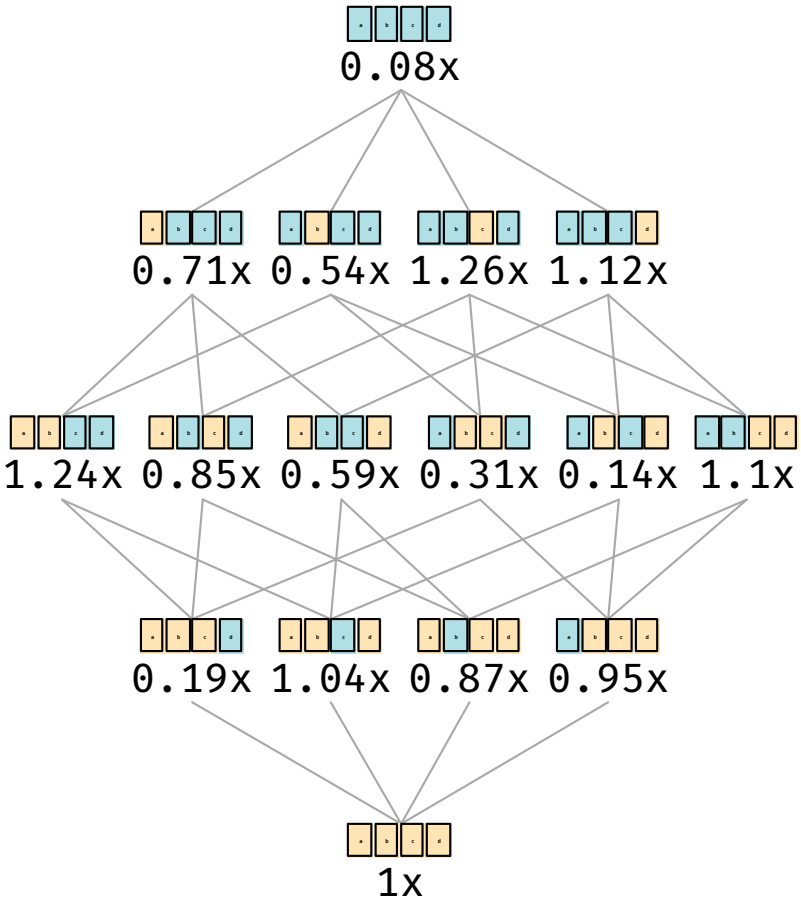
When Data / Label have same color, it's more ok

The visualization has some limitations

Which one is better?

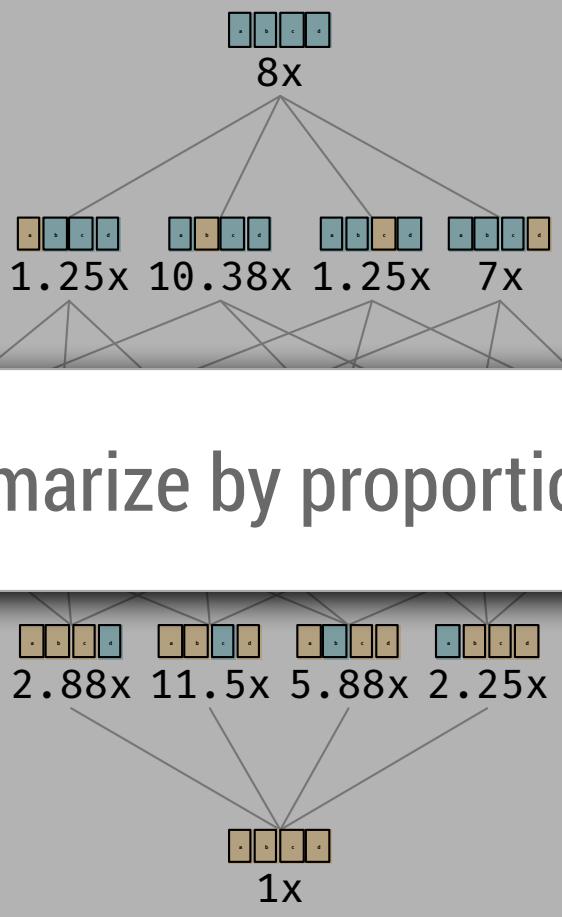


Version 1

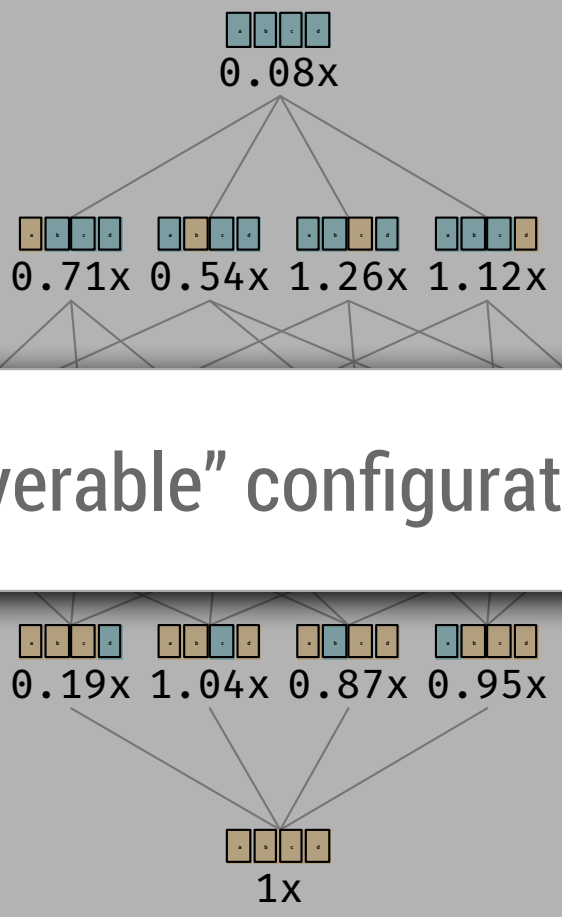


Version 2

Which one is better?



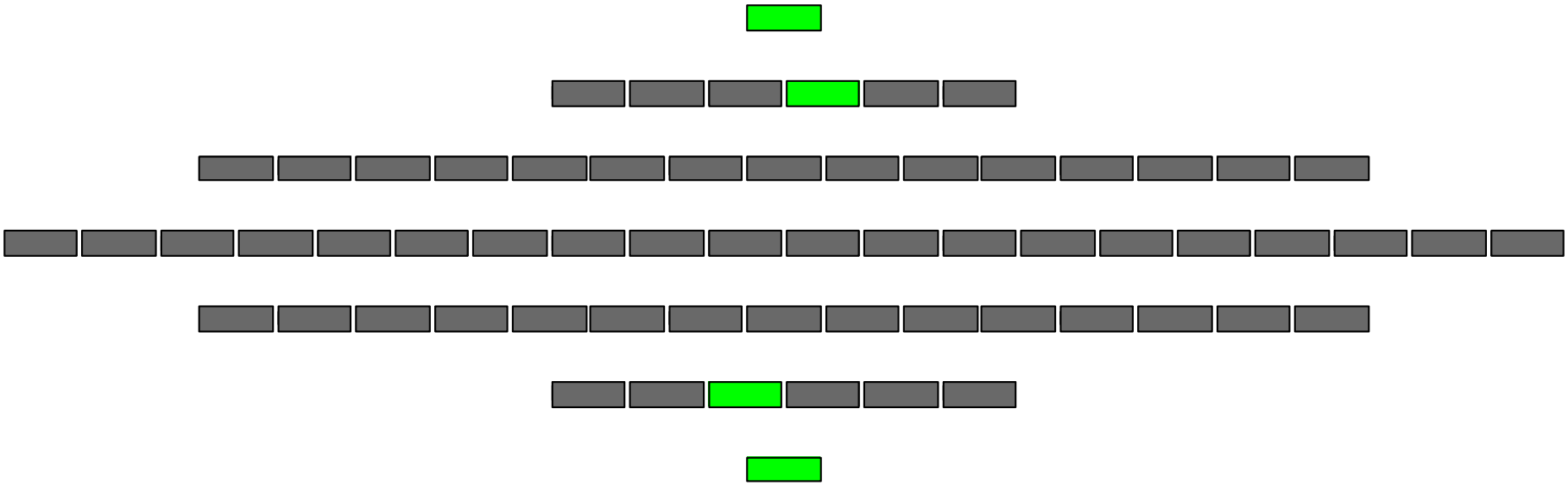
Version 1



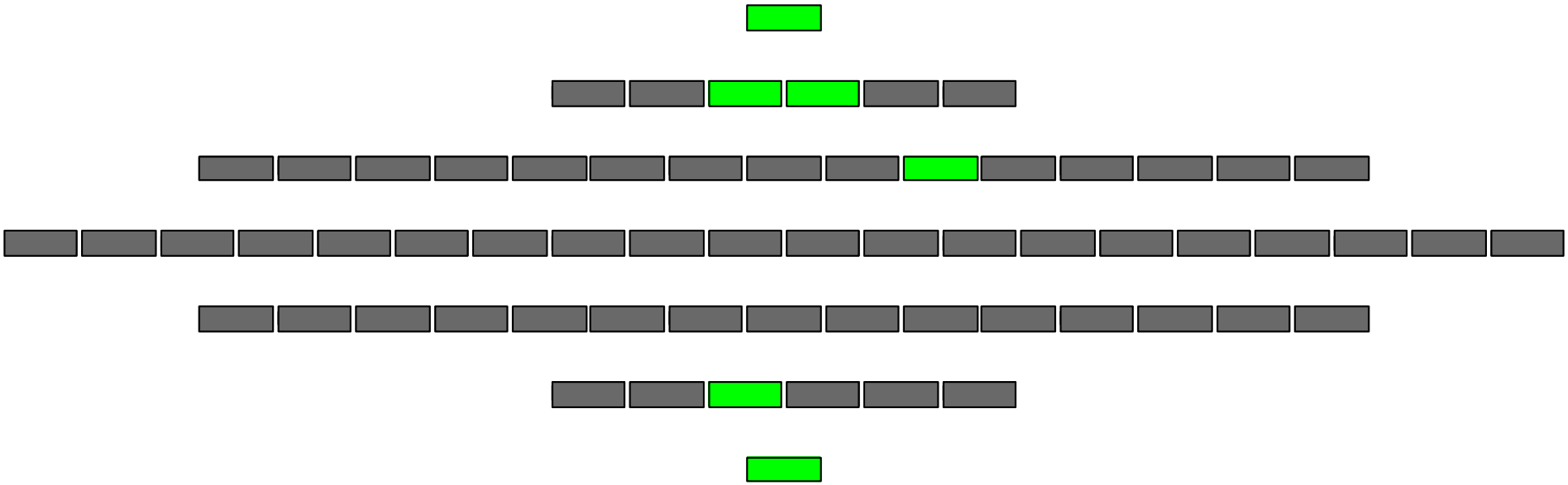
Version 2

Summarize by proportion of “deliverable” configurations

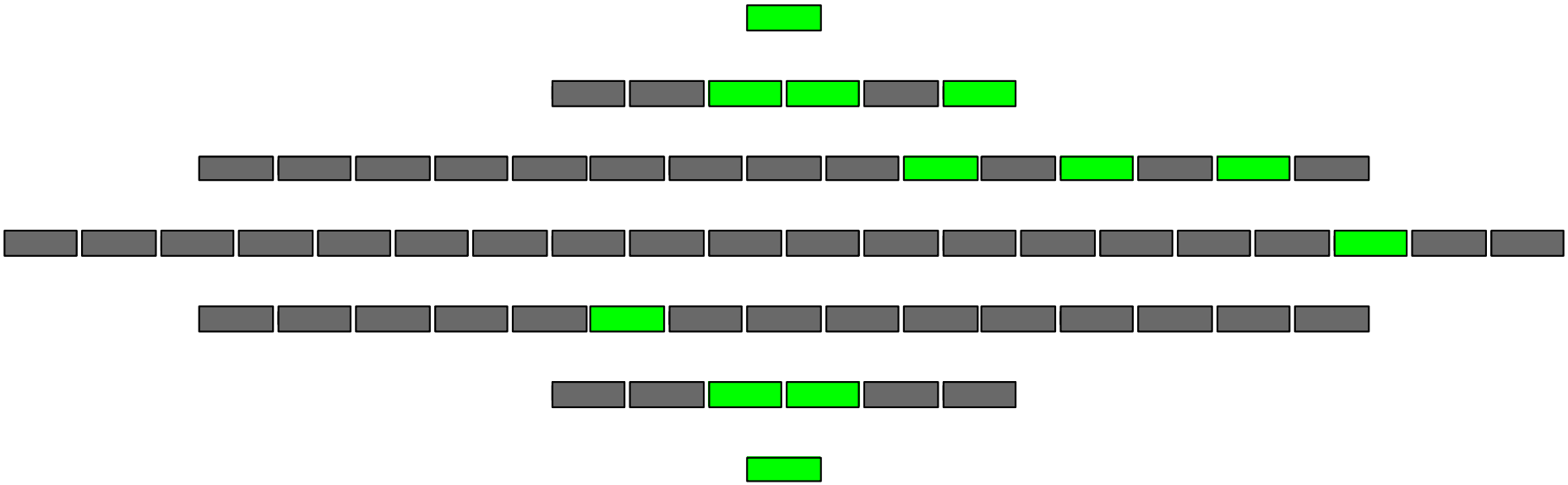
A configuration is **N-deliverable**
if its overhead factor ≤ Nx



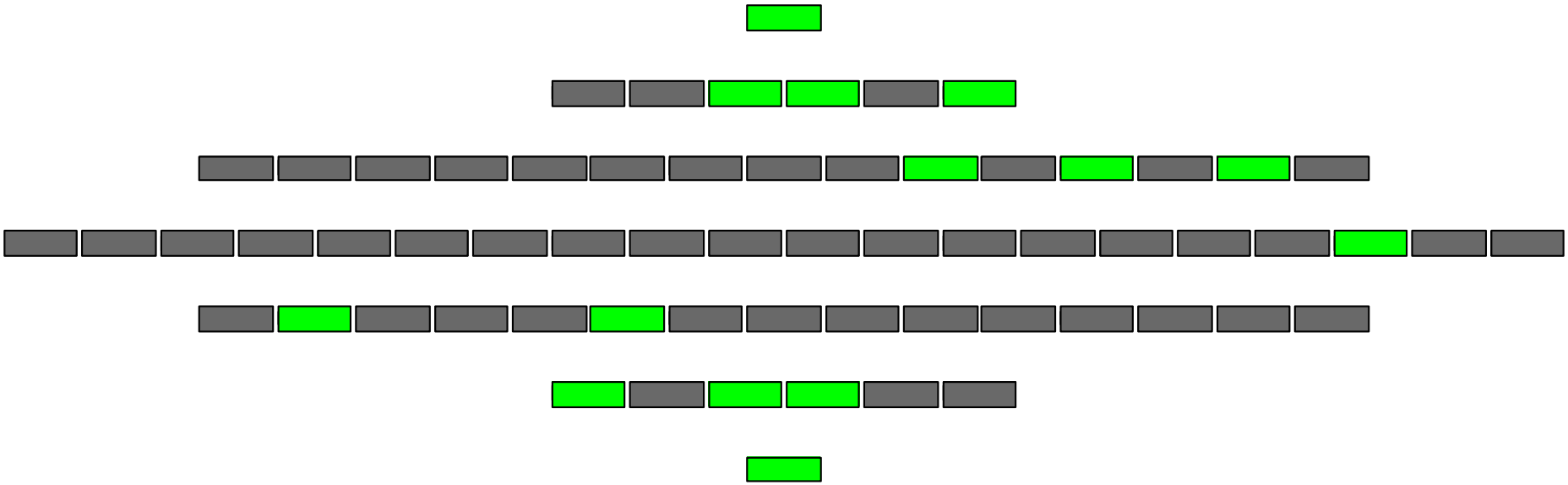
1.1-deliverable proportion: 6%



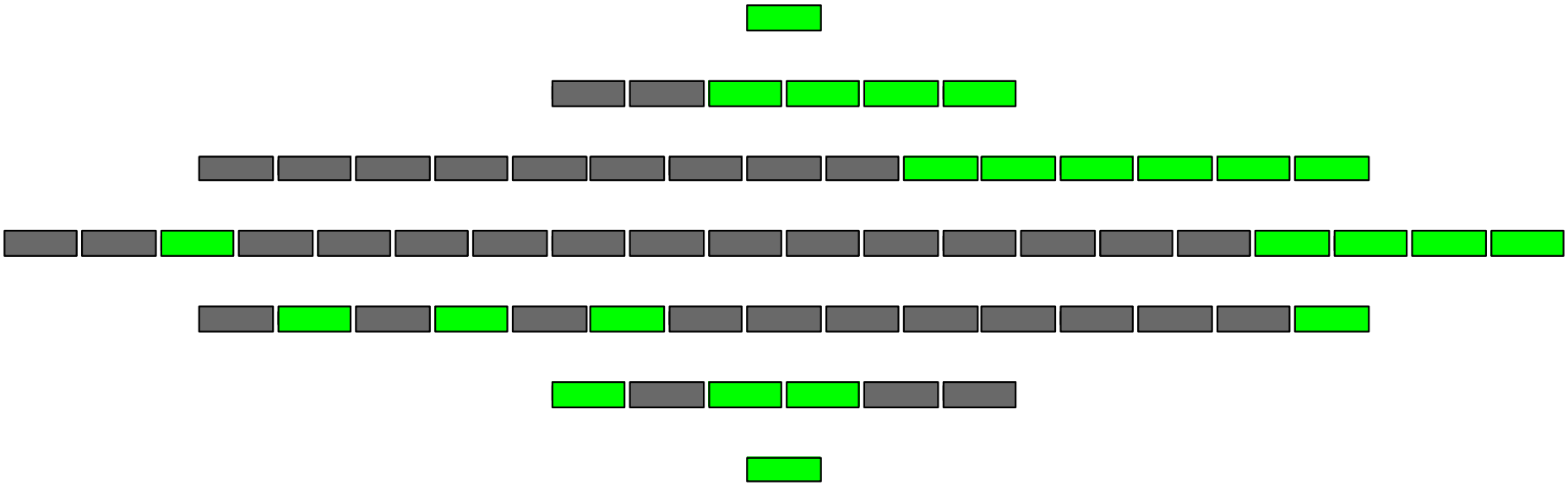
3-deliverable proportion: 9%



5-deliverable proportion: 19%



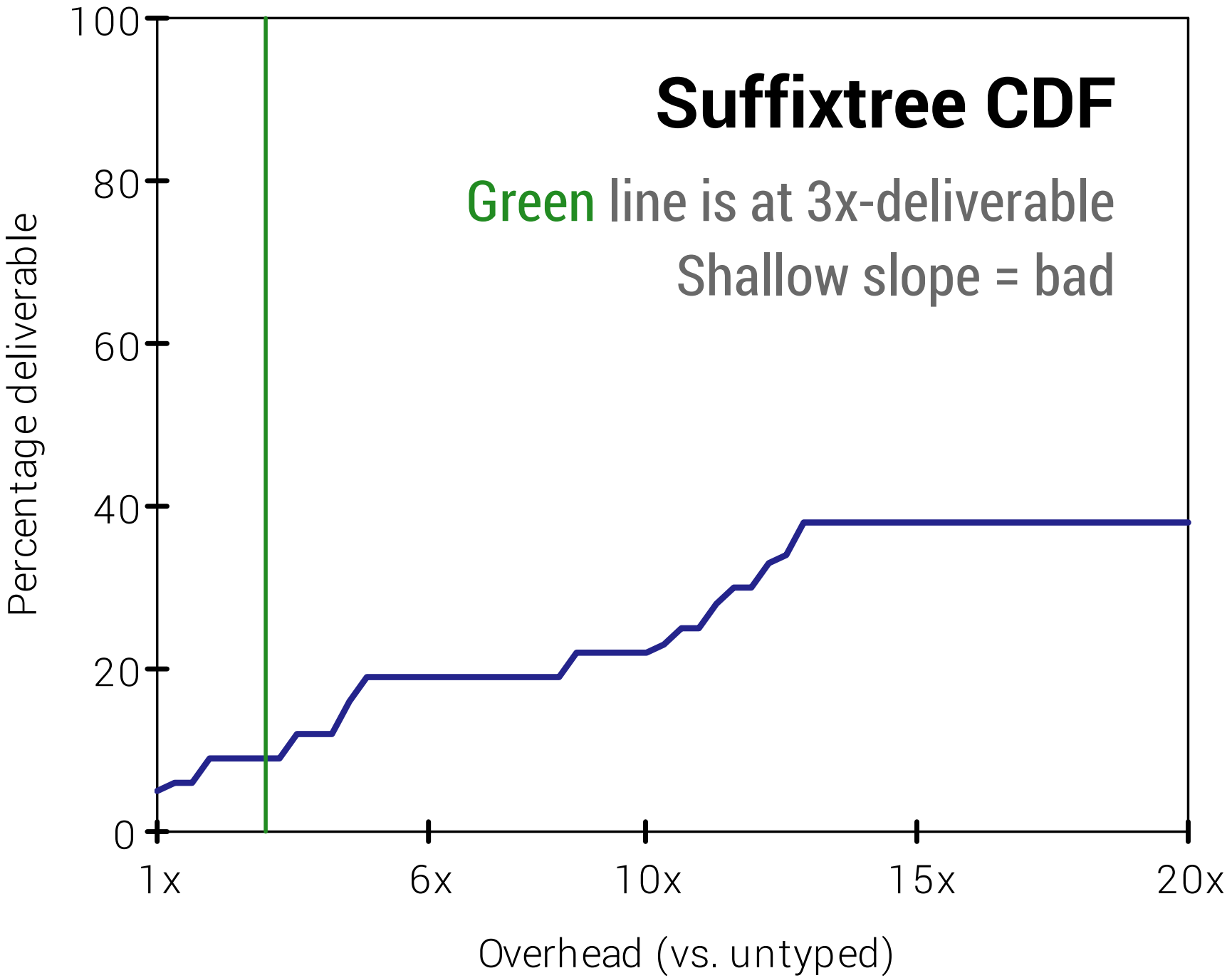
10-deliverable proportion: 22%

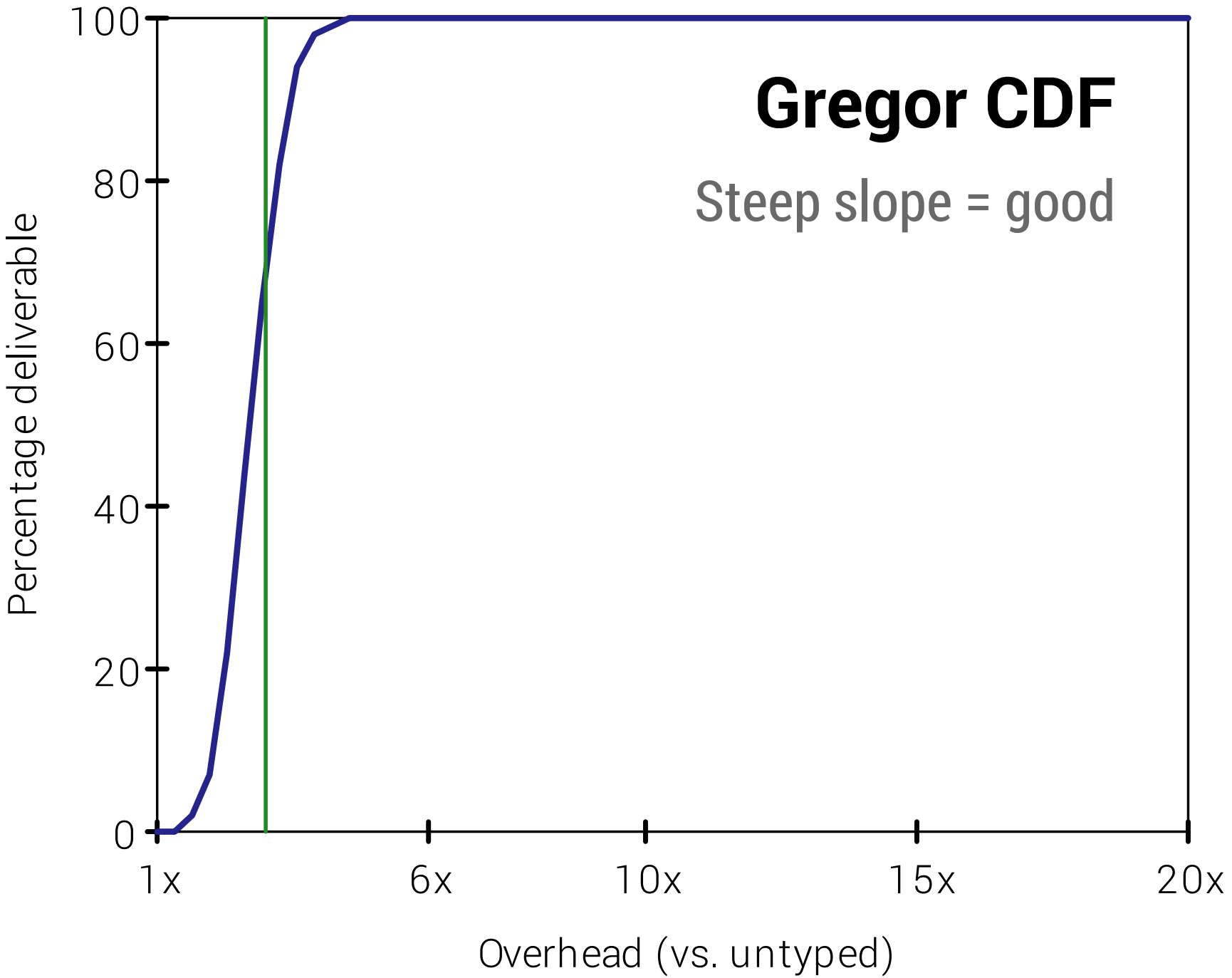


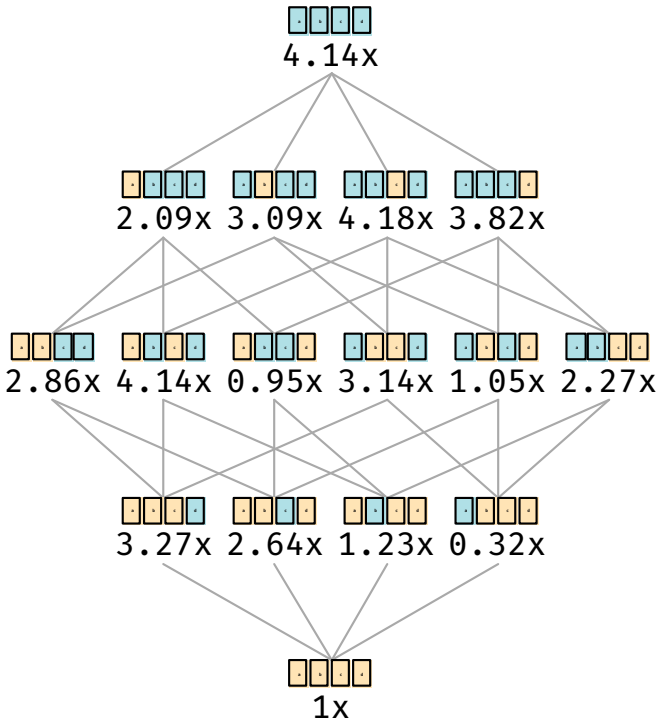
20-deliverable proportion: 38%

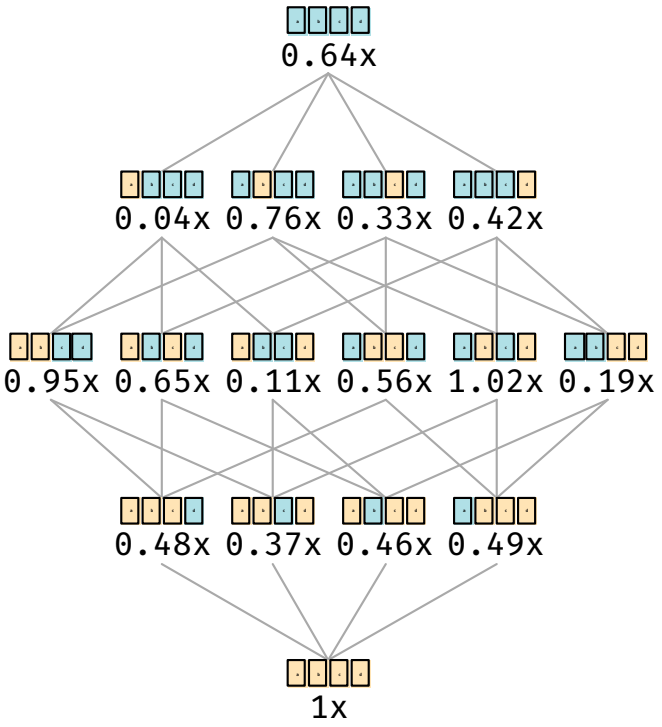
Even at 20x, no paths from untyped to typed

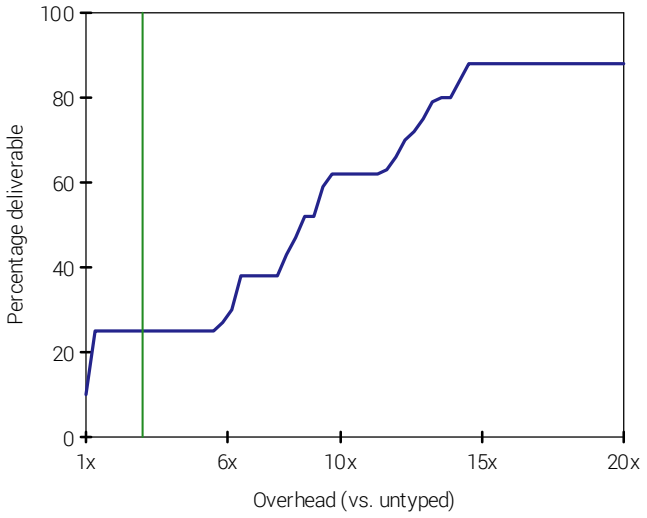
Visualize N-deliverable parameter with a CDF

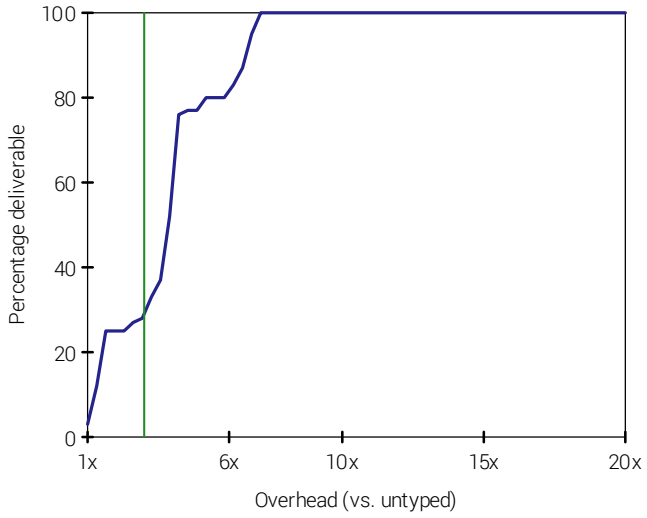

Suffixtree CDF
Green line is at 3x-deliverable
Shallow slope = bad
Percentage deliverable
100
80
60
40
20
0
1x
6x
10x
15x
20x
Overhead (vs. untyped)


Gregor CDF
Steep slope = good
Percentage deliverable
100
80
60
40
20
0
1x
6x
10x
15x
20x
Overhead (vs. untyped)


4.14x
2.09x 3.09x 4.18x 3.82x
2.86x 4.14x 0.95x 3.14x 1.05x 2.27x
3.27x 2.64x 1.23x 0.32x
1x


0.64x
0.04x 0.76x 0.33x 0.42x
0.95x 0.65x 0.11x 0.56x 1.02x 0.19x
0.48x 0.37x 0.46x 0.49x
1x


100
80
60
40
20
0
Percentage deliverable
1x
6x
10x
15x
20x
Overhead (vs. untyped)


100
80
60
40
20
0
Percentage deliverable
1x
6x
10x
15x
20x
Overhead (vs. untyped)

SUMMARY OF APPROACH

- Construct performance lattices for benchmarks
- Inspect lattices manually when feasible
- Compare lattices with N-deliverable CDF

RESULTS

Measured 12 curated benchmarks on all configs

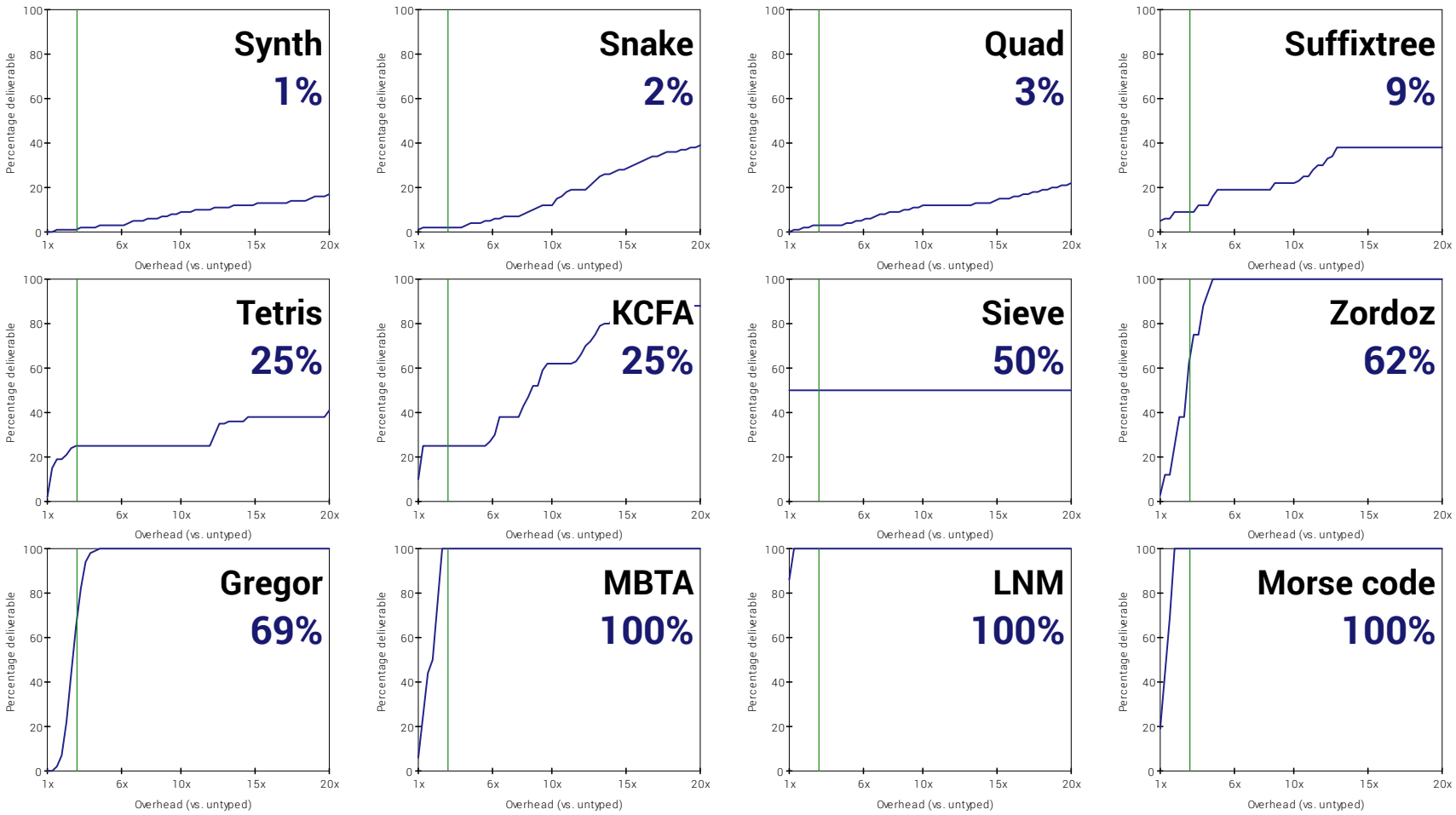
5 are user-written libraries & programs

5 are educational programs

2 were written for this paper

Ran a total of 75844 configurations

Took 3 months to run



3-deliverable proportions

1.1-deliverable configs over all benchmarks

$$\frac{283}{75844} \approx 0.4\%$$

3-deliverable configs over all benchmarks

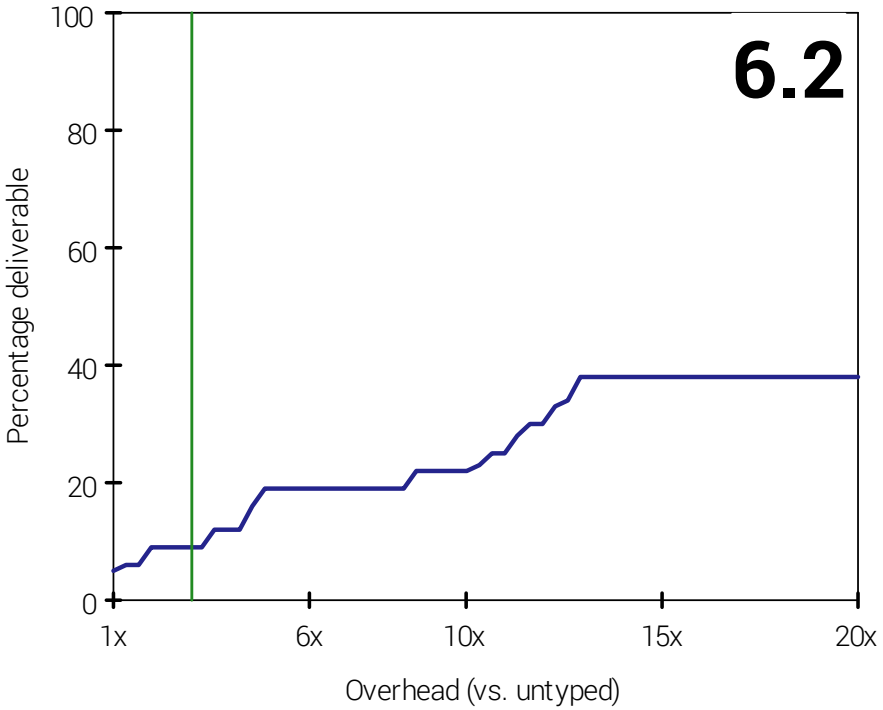
$$\frac{7992}{75844} \approx 10.5\%$$

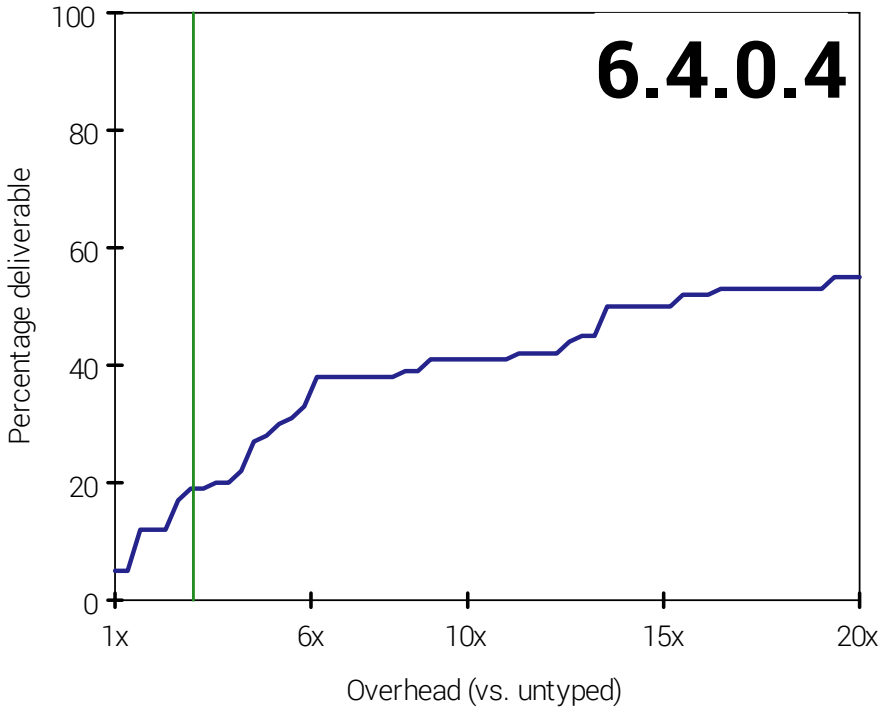
Bottom line: most configs not deliverable

Even with liberal 3x-deliverable criterion

SO, IS THERE HOPE?

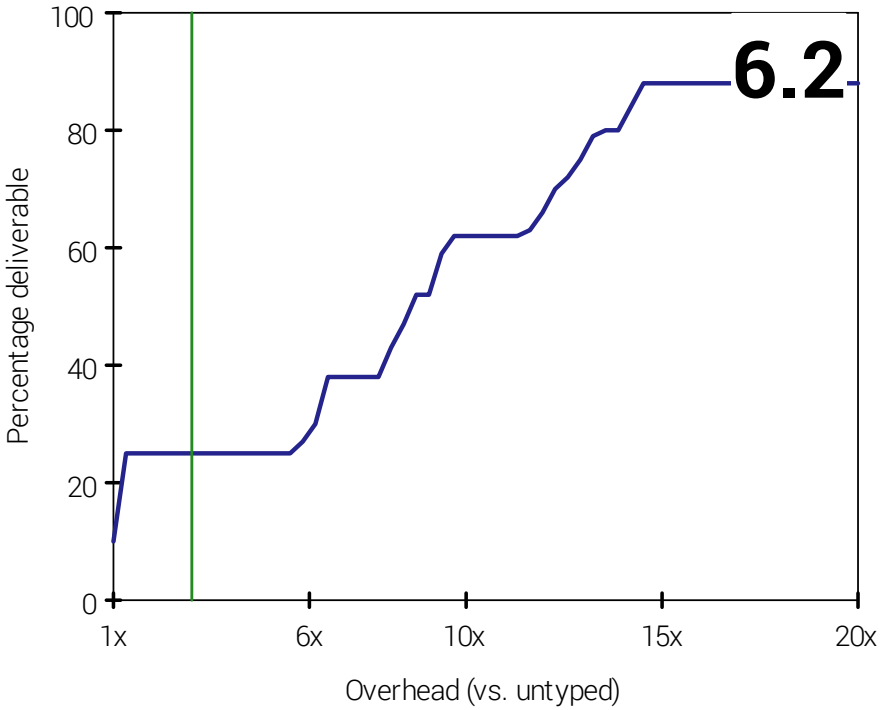
Suffixtree improvement

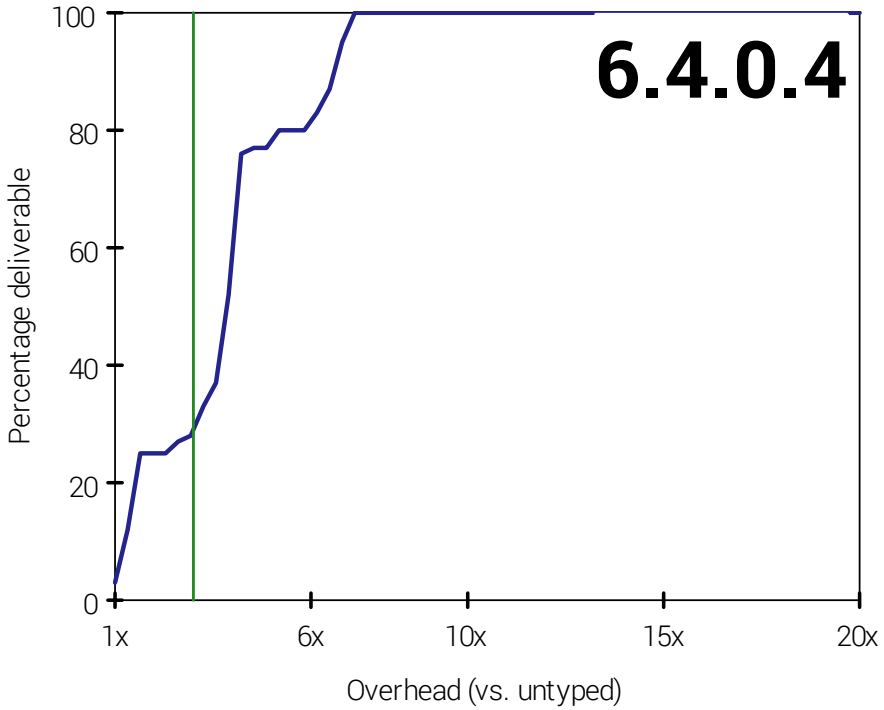




9% to 19% improvement in 3-deliverability

KCFA improvement



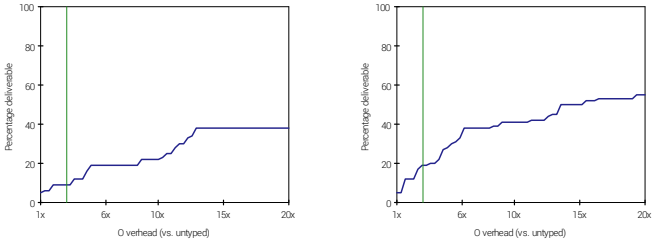


25% to 29% improvement in 3-deliverability

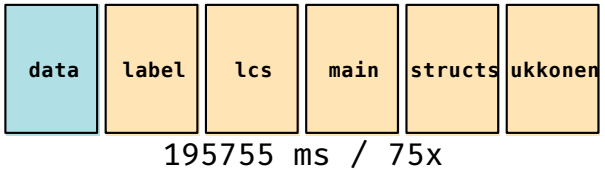
HOPE

Evaluation method helps implementors

Helps measure improvements between versions



Can inspect lattice for bad configs



HOPE

Tools for avoiding GT performance pitfalls

Initial steps: contract profiler [St-Amour et al 2015]

```
Contracts account for 47.35% of running time (286 / 604 ms)
  188 ms : build-matrix          (-> Int Int (-> any any any) Array)
  88 ms : matrix-multiply-data   (-> Array Array [...]))
  10 ms : make-matrix-multiply   (-> Int Int Int (-> any any any) Array)
```

HOPE

Evaluation method helps GT system implementors

Tools for avoiding GT performance pitfalls

Paper & Datasets:

`http://www.ccs.neu.edu/racket/pubs/#popl16-tfgnvf`

HOPE

Evaluation method helps GT system implementors

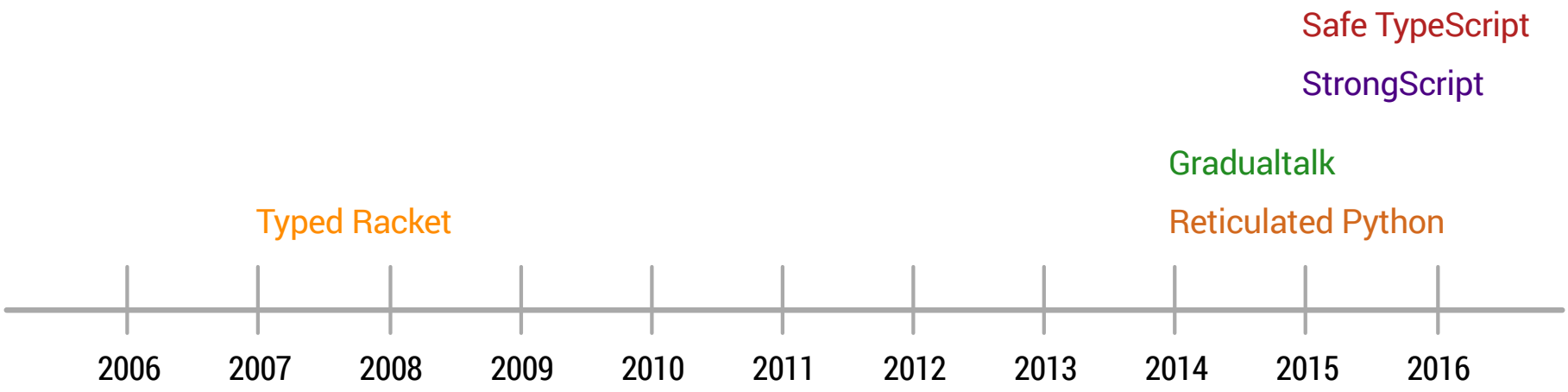
Tools for avoiding GT performance pitfalls

Paper & Datasets:

`http://www.ccs.neu.edu/racket/pubs/#popl16-tfgnvf`

Thank you!

Other research implementations of gradual typing



Challenge: adapt this method to your chosen sound GT system