Deep and Shallow Types

Thesis Defense

Ben Greenman 2020-12-17

Matthias Felleisen

Amal Ahmed

Jan Vitek

Shriram Krishnamurthi

Fritz Henglein

Sam Tobin-Hochstadt

Deep and Shallow Types

Thesis Defense

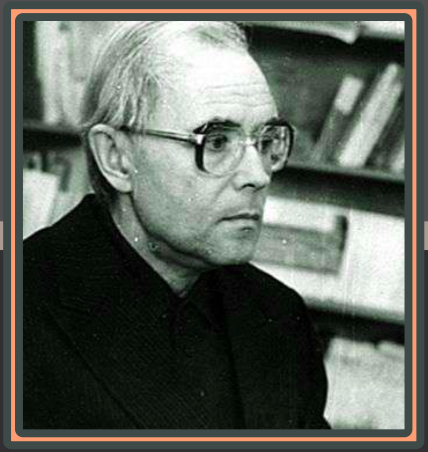
Ben Greenman   2020-12-17

On Great Ideas

If I reproduce somebody's guess
in my work ...

me living far away ...

it means that
there really is something in it.


Ershov

Great Idea:

mixing typed and untyped code

Gradual Typing

Migratory Typing

Multi-Language Semantics

Hybrid Typing

The Basics

typed code 🟩🟩🟩🟩🟩

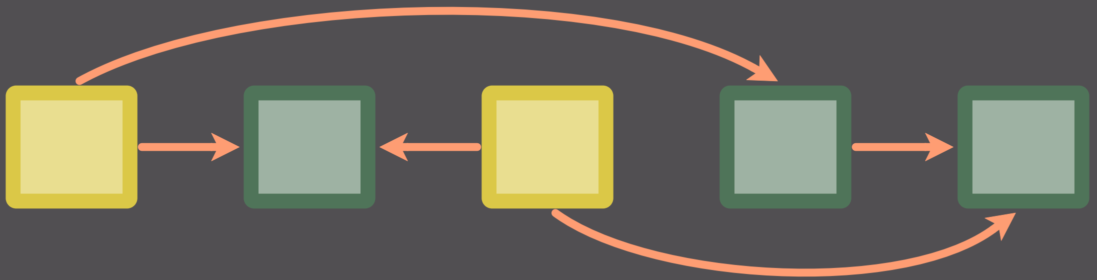
more constraints, strong guarantees

untyped code 🟨🟨🟨🟨🟨

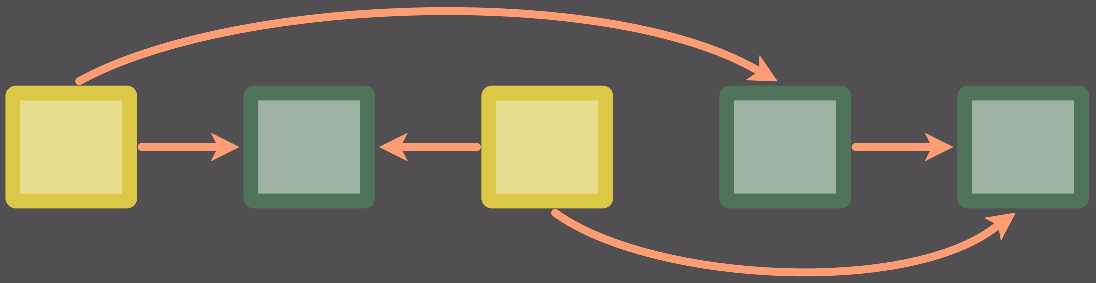
more freedom, for better or worse

mixed-typed code 🟨🟩🟨🟩🟩

combine both ... somehow

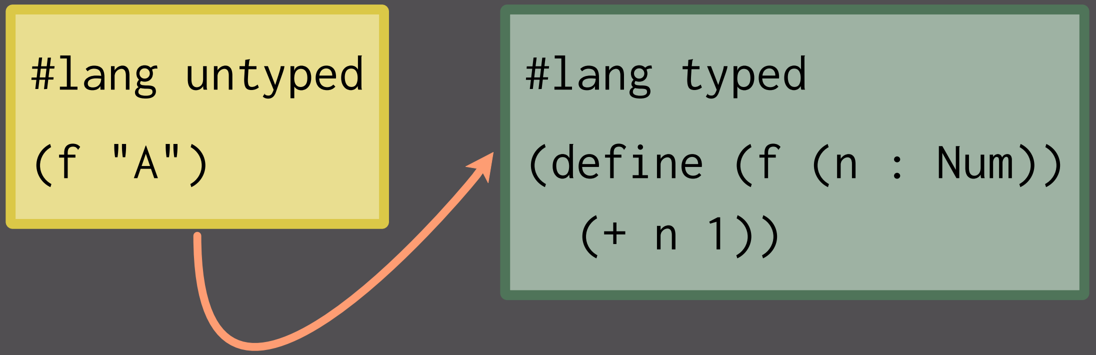


Q. What happens at the boundaries?



Q. What happens at the boundaries?

Does the type `Num` keep out the letter `"A"` ?



```
#lang untyped
(f "A")
```

```
#lang typed
(define (f (n : Num))
  (+ n 1))
```

Gradual Typing

Migratory Typing

Multi-Language Semantics

Hybrid Typing

research landscape

Gradual Typing

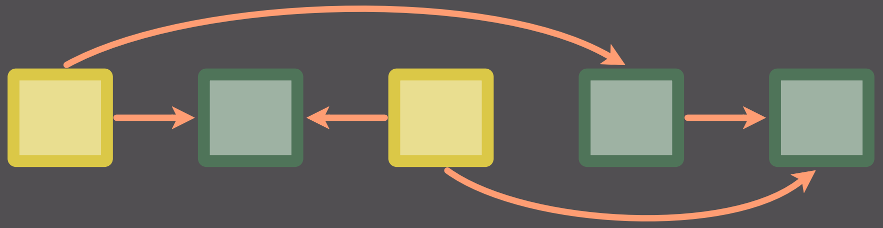
Migratory Typing

Multi-Language Semantics

Hybrid Typing

research landscape ... over 200 publications

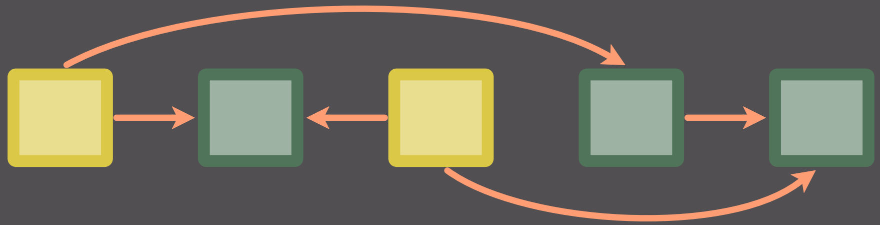
research landscape ... over 200 publications



Q. What happens at the boundaries?

research landscape ... over 200 publications

6+ ideas for boundaries



Q. What happens at the boundaries?

research landscape

research landscape

language landscape ... many implementations

Mixed-Typed Design Space

```
#lang untyped
(f "A")
```

```
#lang typed
(define (f (n : Num))
  (+ n 1))
```

Q. Does the type `Num` keep out the letter `"A"` ?

hack Lua PYRE THORN

```
#lang untyped

(f "A")
```

```
#lang typed

(define (f (n : Num))
  (+ n 1))
```

Q. Does the type `Num` keep out the letter `"A"` ?

A. Yes!

A. No

```
#lang untyped

(f (λ "A"))
```

```
#lang typed

(define (f (x : (-> Num)))
  (g x))
```

```
#lang untyped

(define (g y)
  (.... y))
```

Q. Can the type `(-> Num)` detect bad functions?

```
#lang untyped

(f (λ "A"))
```

```
#lang typed

(define (f (x : (-> Num)))
  (g x))
```

```
#lang untyped

(define (g y)
  (.... y))
```

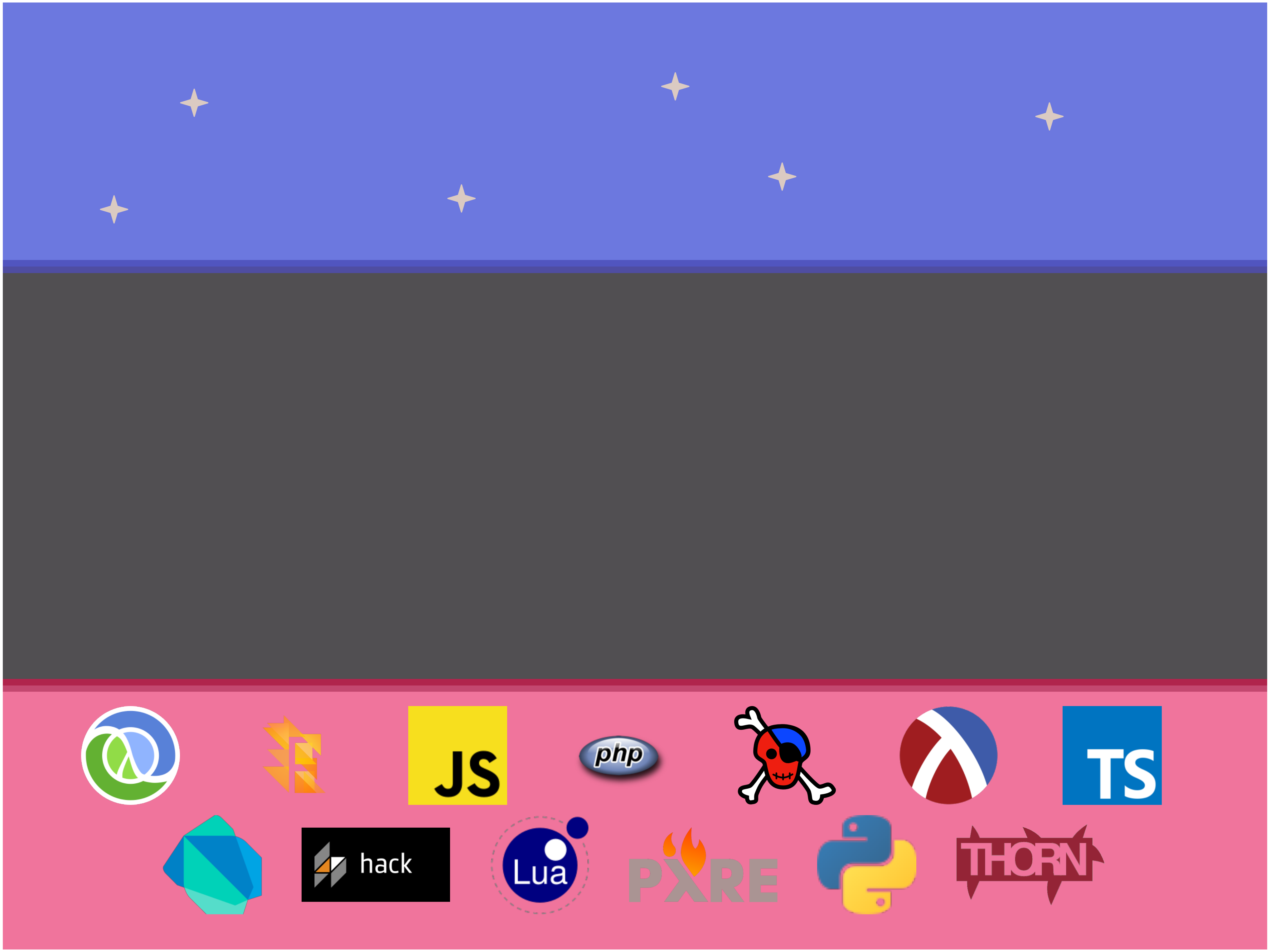
Q. Can the type `(-> Num)` detect bad functions?

A. Yes

A. No

Q. What happens at the boundaries?

A. Nothing A. Spot-checks A. Everything! A. ...


JS
php
TS
hack
Lua
PYRE
THORN

Q. Why?

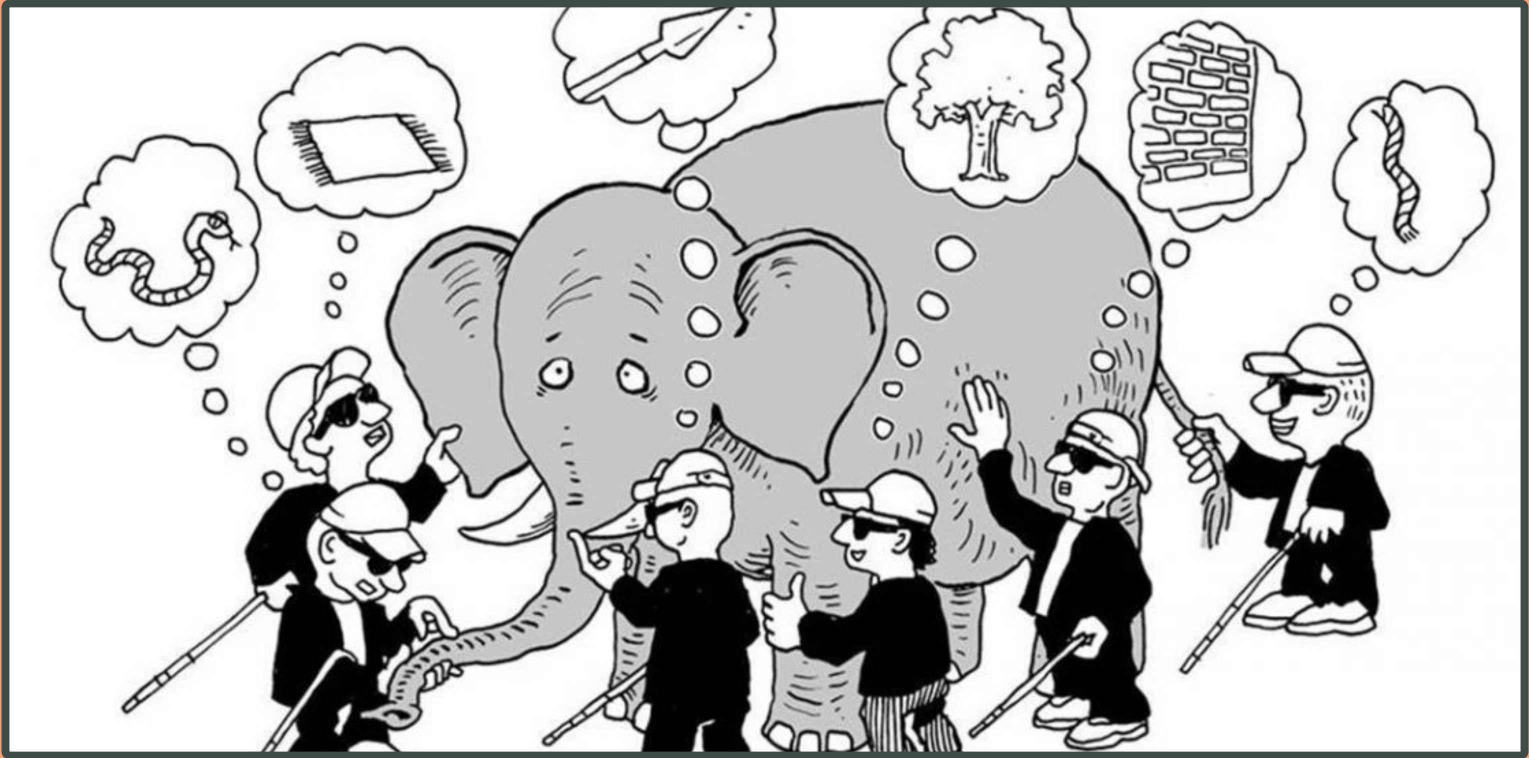
Q. Why? A. Performance!

Q. Why? A. Performance!

Q. Where's the data?

Mixed-Typed Design Space

Lively, but Disorganized!



JS php TS

hack Lua PYRE THORN

My Research

brings order to the design space

* How to assess type guarantees
* How to measure performance

* How to measure performance

  (the problem)

* How to assess type guarantees

  (solution space)

* How to measure performance

  (the problem)

* How to assess type guarantees

  (solution space)

Thesis Preview:

Deep and Shallow types can interoperate

* How to measure performance

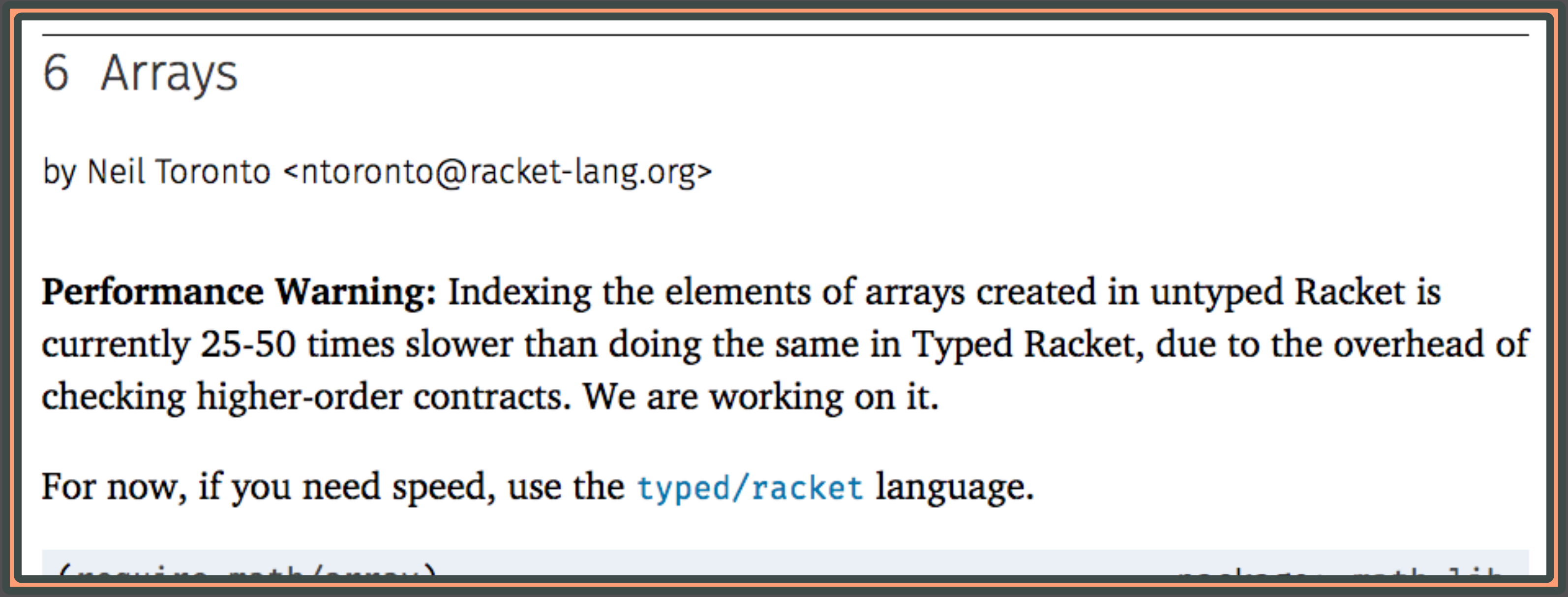
  (the problem)

Typed Racket

- Mature, strong mixed-typed language
- Home of severe performance costs

Costs ...

25x to 50x



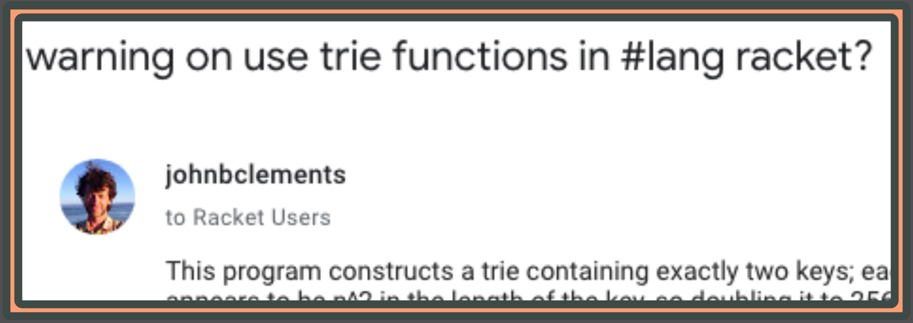
6 Arrays

by Neil Toronto <ntoronto@racket-lang.org>

Performance Warning: Indexing the elements of arrays created in untyped Racket is currently 25-50 times slower than doing the same in Typed Racket, due to the overhead of checking higher-order contracts. We are working on it.

For now, if you need speed, use the `typed/racket` language.

… More Costs



warning on use trie functions in #lang racket?

johnbclements

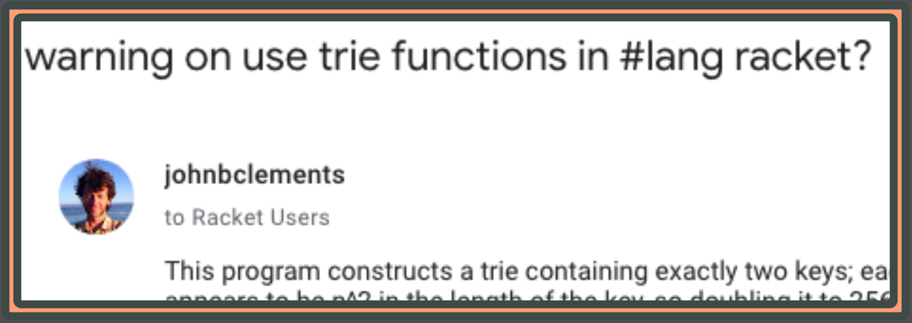
to Racket Users

This program constructs a trie containing exactly two keys; ea

```
#lang untyped
(require pfds/trie)

(define t (trie ....))
(time (bind t ....))
```

12 seconds

… More Costs


warning on use trie functions in #lang racket?

johnbclements

to Racket Users

This program constructs a trie containing exactly two keys; ea
appears to be n^2 in the length of the key, so doubling it to 256

```
#lang untyped

(require pfds/trie)


(define t (trie ....))
(time (bind t ....))
```

12 seconds

```
#lang typed

(require pfds/trie)


(define t (trie ....))
(time (bind t ....))
```

1 ms!

Typed Racket, Performance

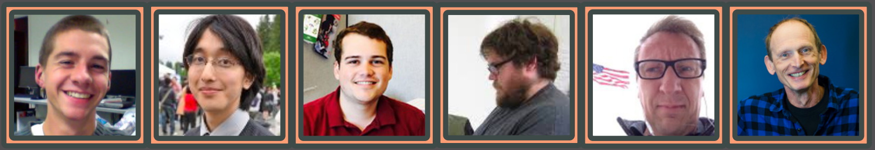
- Clearly, problems exist

Typed Racket, Performance

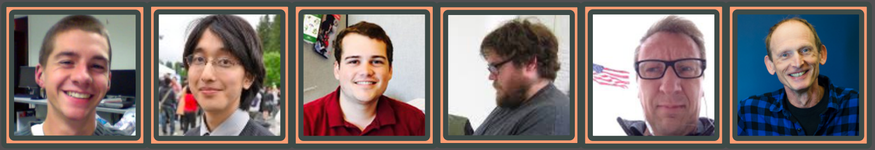
- Clearly, problems exist

Need a way to measure!

Step 1: Benchmarks

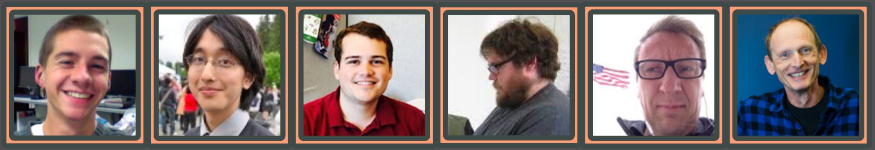


Step 1: Benchmarks



Collected small, useful programs

Step 1: Benchmarks

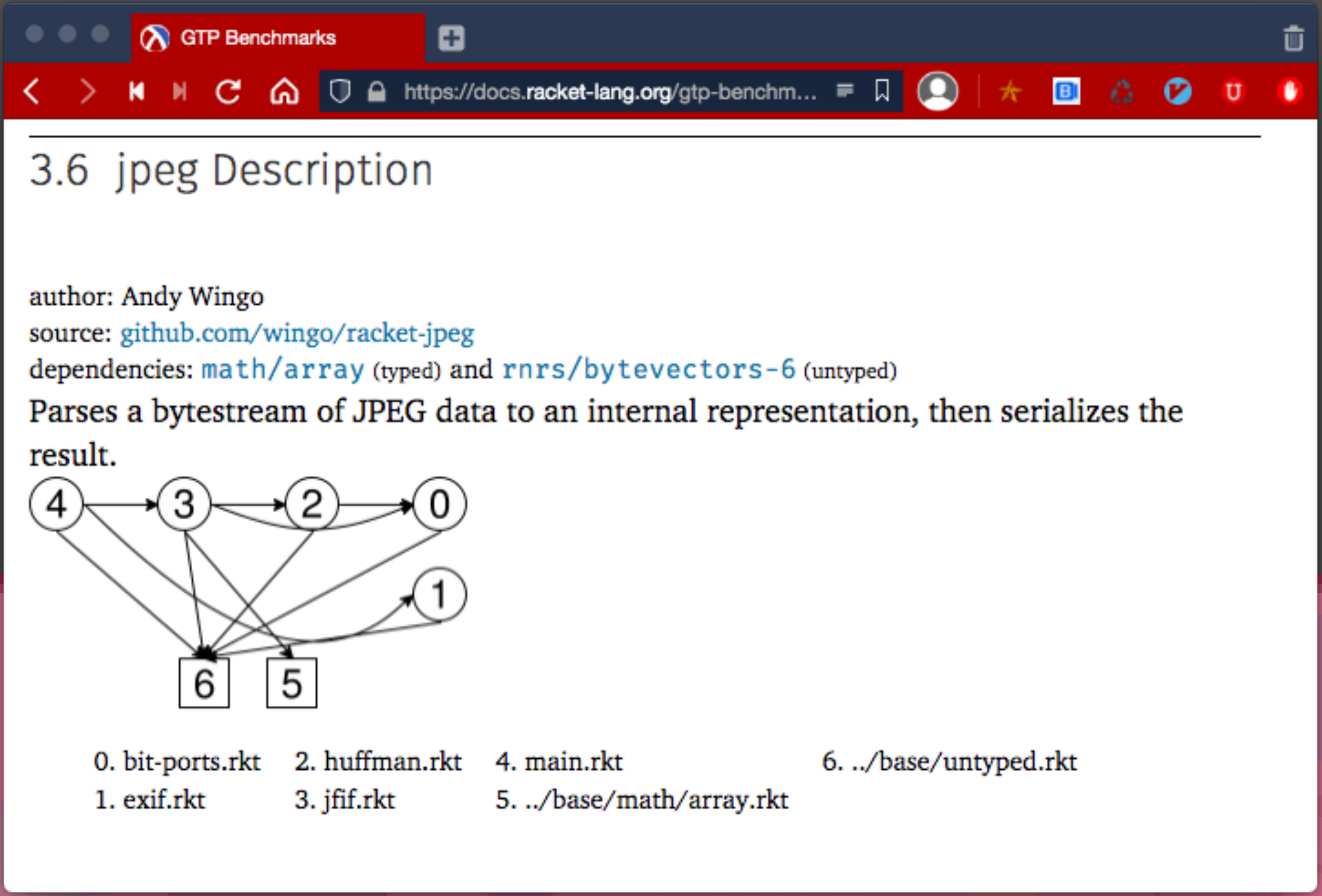


Collected small, useful programs

Added types, if missing

Step 1: Benchmarks

🟨🟨🟨🟨🟨 + 🟩🟨



3.6 jpeg Description

author: Andy Wingo
source: github.com/wingo/racket-jpeg
dependencies: `math/array` (typed) and `rnrs/bytevectors-6` (untyped)

Parses a bytestream of JPEG data to an internal representation, then serializes the result.

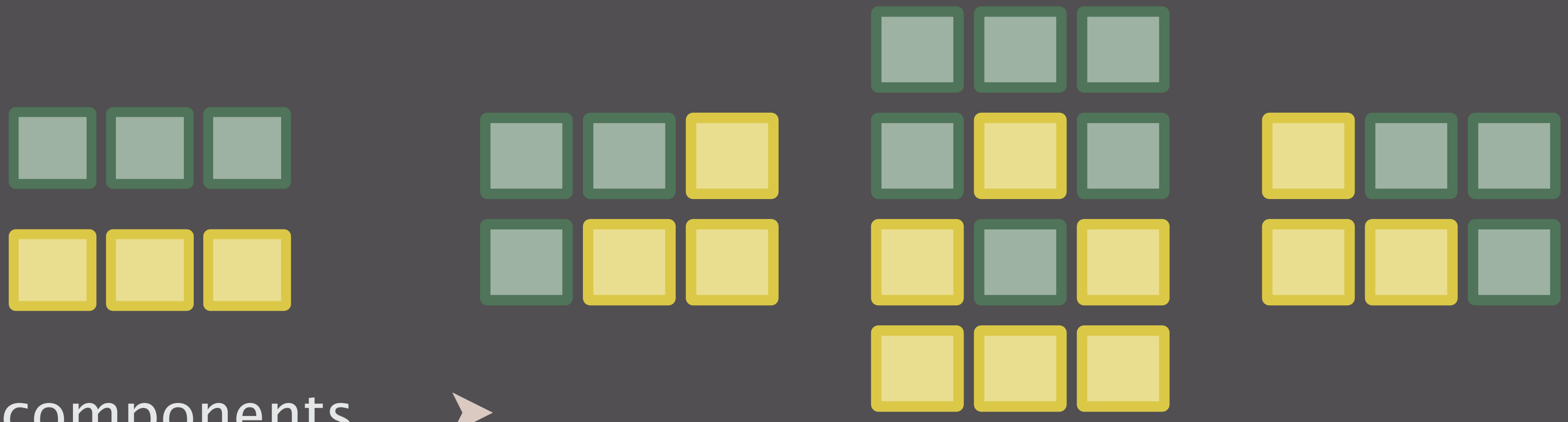
$$\begin{tikzcd}
4 \arrow[r] \arrow[drr] \arrow[dr] & 3 \arrow[r] \arrow[rr, bend right] \arrow[d] \arrow[dr] & 2 \arrow[r] \arrow[dl] & 0 \arrow[dll] \\
 & 6 \arrow[rr, bend left] & 5 & 1 \arrow[ll]
\end{tikzcd}$$

0. bit-ports.rkt	2. huffman.rkt	4. main.rkt	6. ../base/untyped.rkt
1. exif.rkt	3. jfif.rkt	5. ../base/math/array.rkt	

Step 2: How to Measure

Step 2: How to Measure

What to measure = all configurations

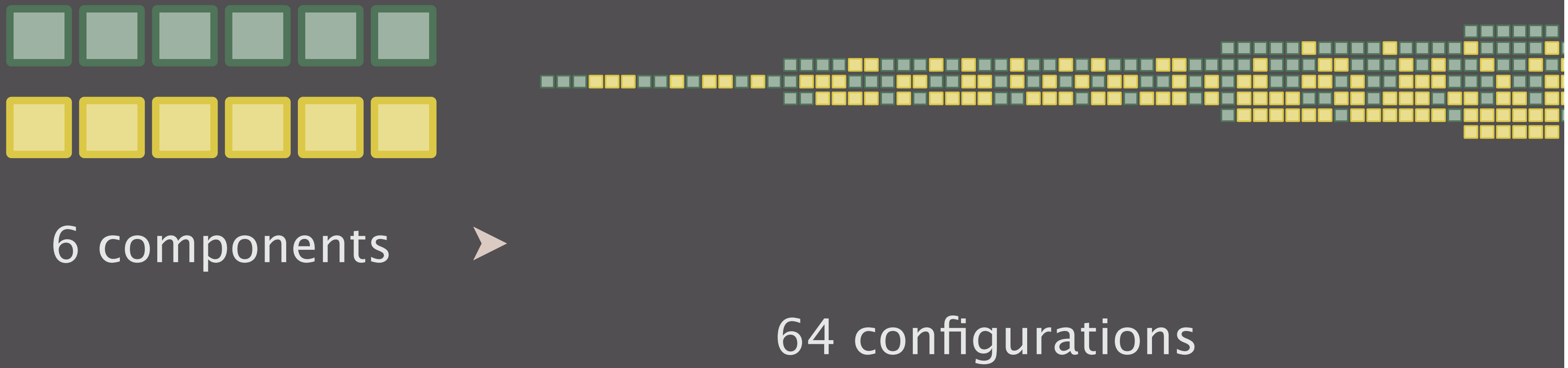


3 components ➤

8 configurations

Step 2: How to Measure

What to measure = all configurations



Step 2: How to Measure

What to measure = all configurations

Q. How to study?

Q. How to scale?

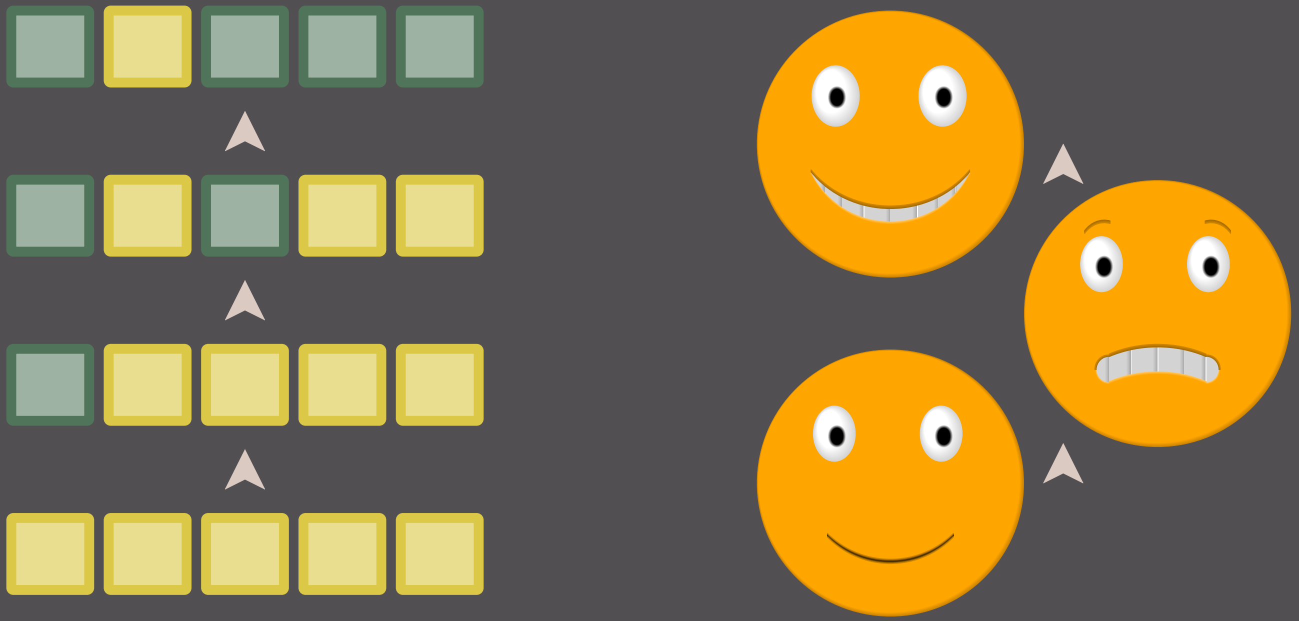
Step 2: How to Measure

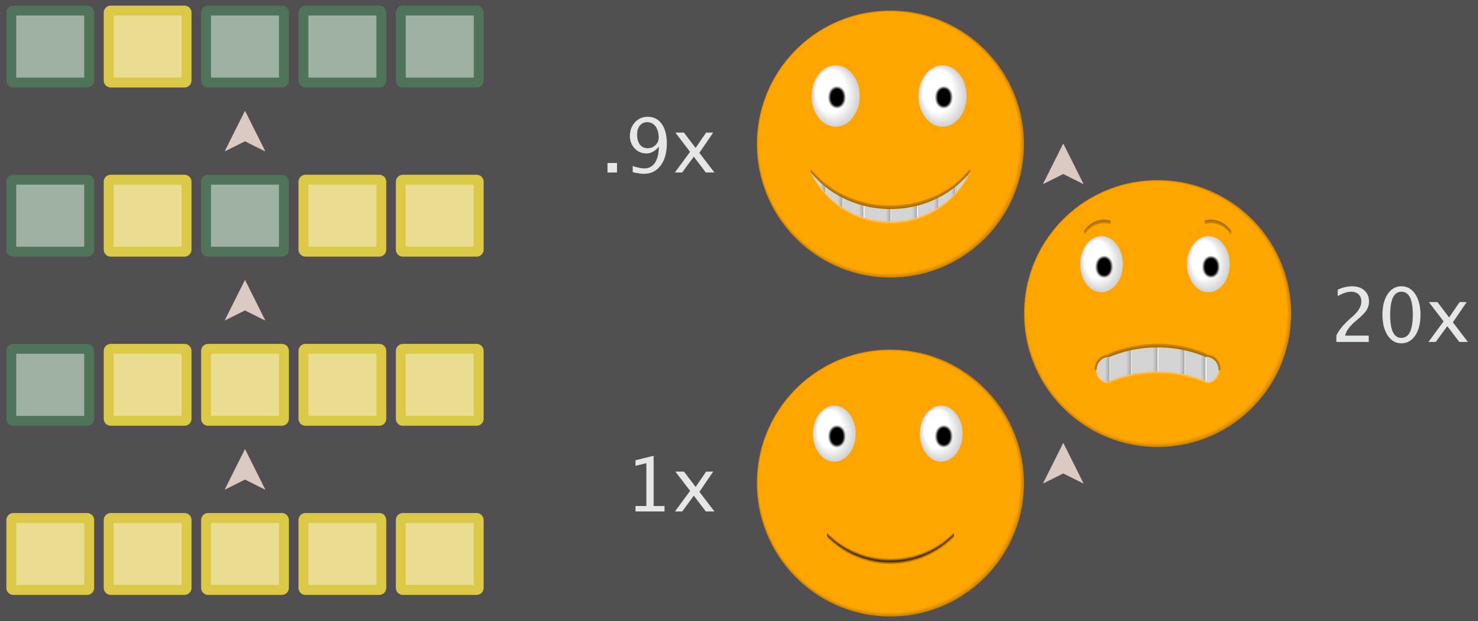
What to measure = all configurations

Q. How to study?

Q. How to scale?

A. Focus on the programmer ...




.9x
20x
1x

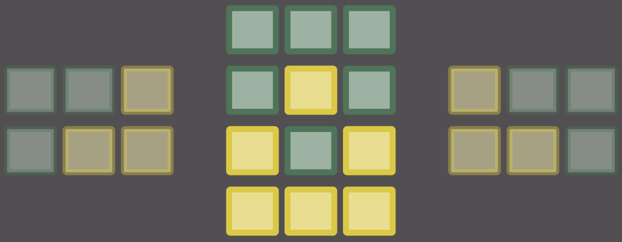
Step 2: How to Measure

A. Count **D**-deliverable configs

Step 2: How to Measure

A. Count **D**–deliverable configs

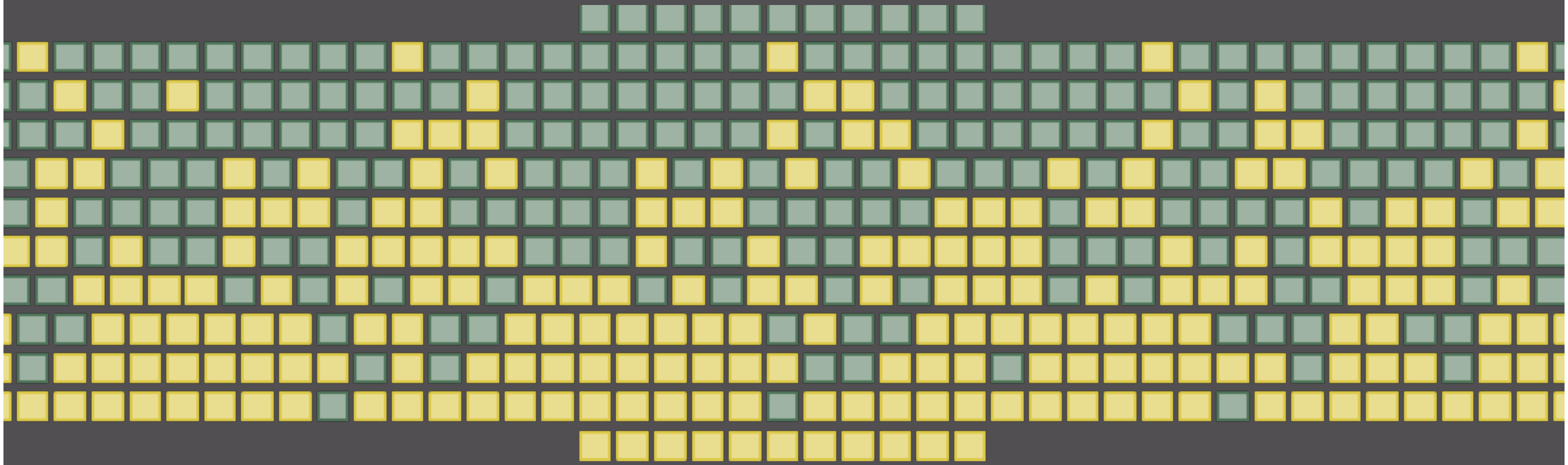
If **D**=4, then count configs with at most 4x overhead



= 50%

Step 2: How to Measure

A. Count **D**–deliverable configs

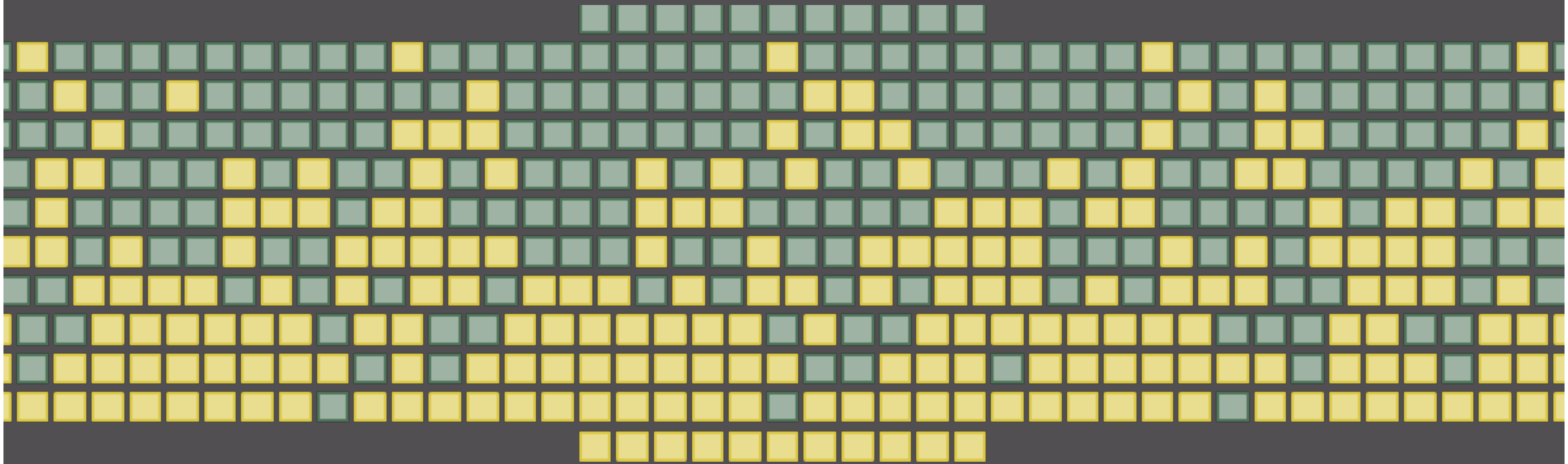


Step 2: How to Measure

A. Count **D**-deliverable configs

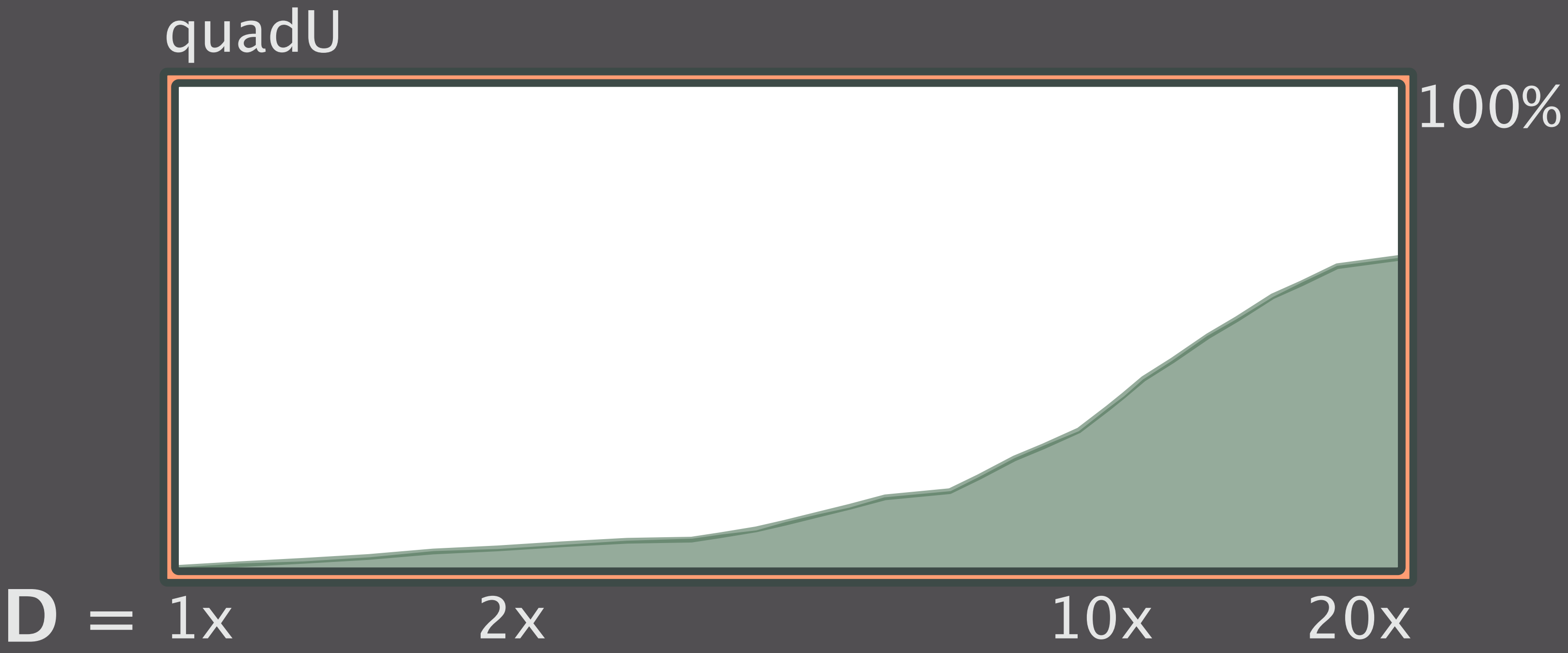
D-deliverable ~ Bernoulli random variable

linear-size sampling works

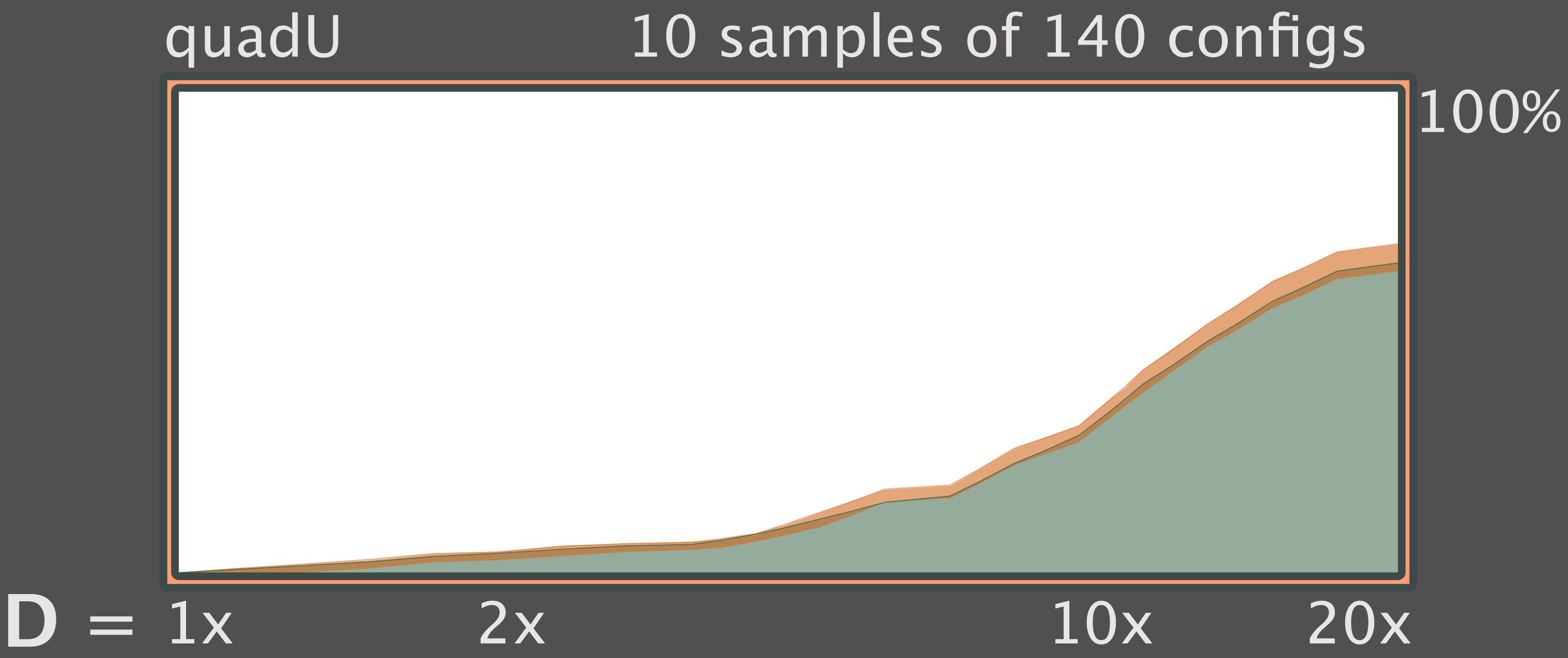


Step 3: Summarize with a Picture

Step 3: Summarize with a Picture



Step 3: Summarize with a Picture



Performance Method

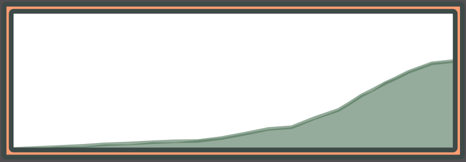
Performance Method

1. collect mixed-typed benchmarks

2. count **D**-deliverable configs
   (or sample)

✓ ✗

3. plot results



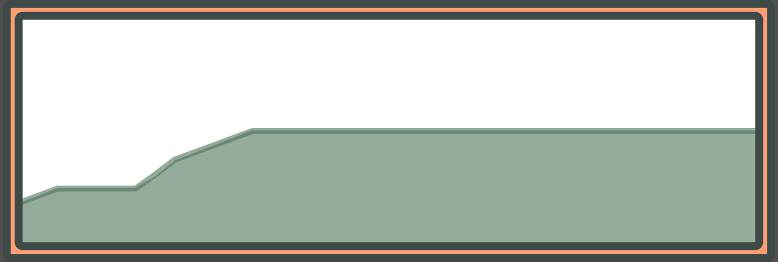
Applications:

Typed Racket

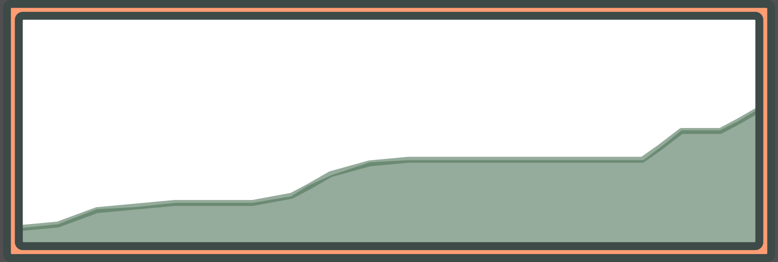
Reticulated Python

Typed Racket some results from our 21 benchmarks

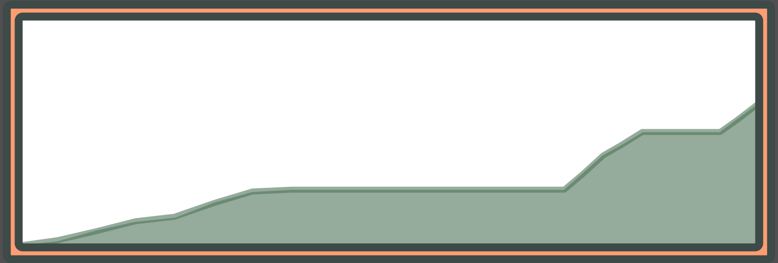
jpeg



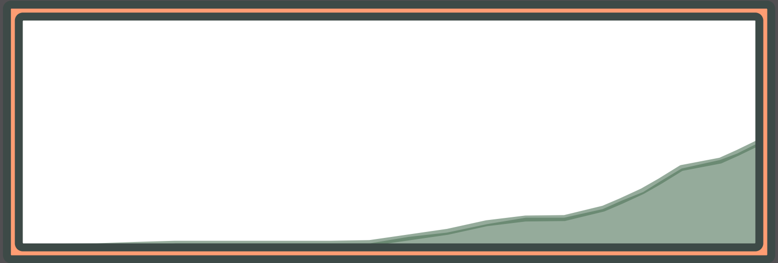
suffixtree



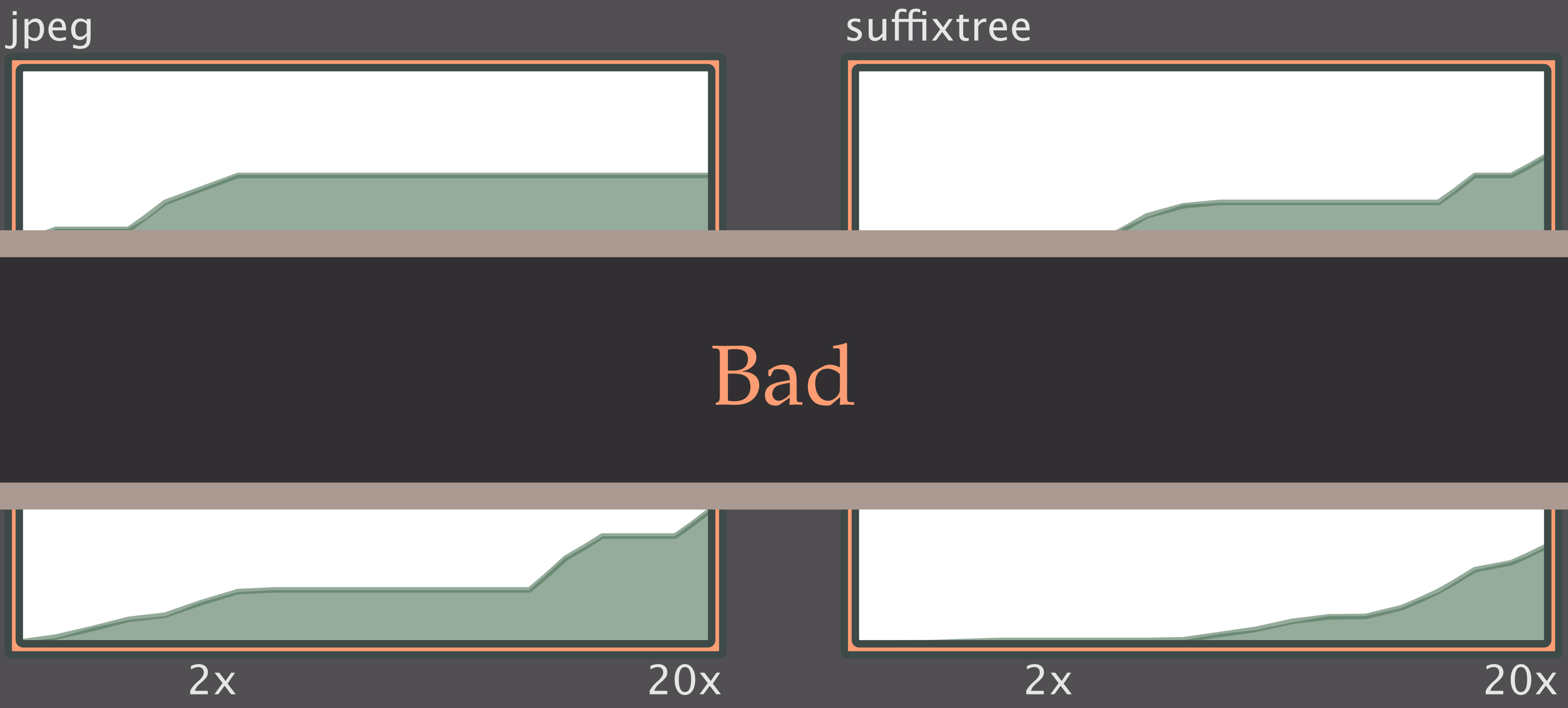
take5



synth



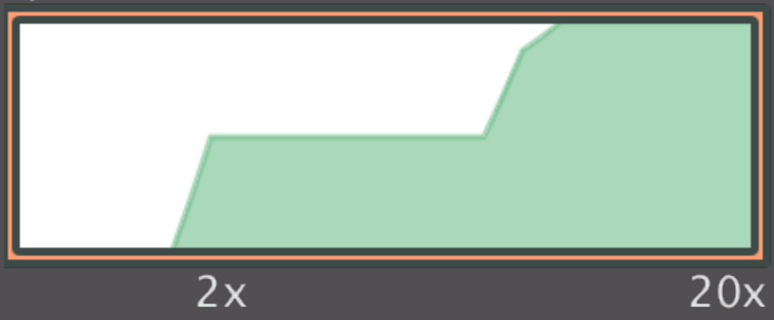
Typed Racket some results from our 21 benchmarks



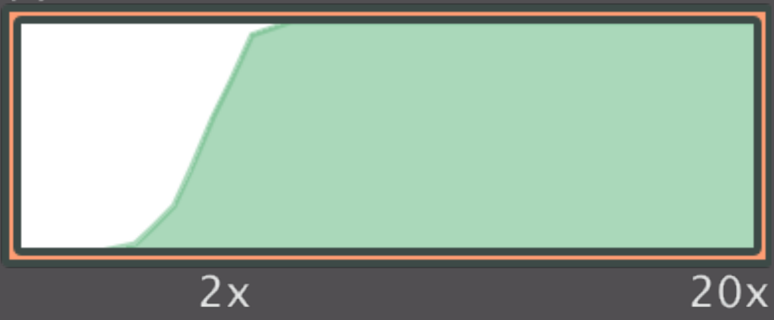
Bad

Reticulated Python different benchmarks

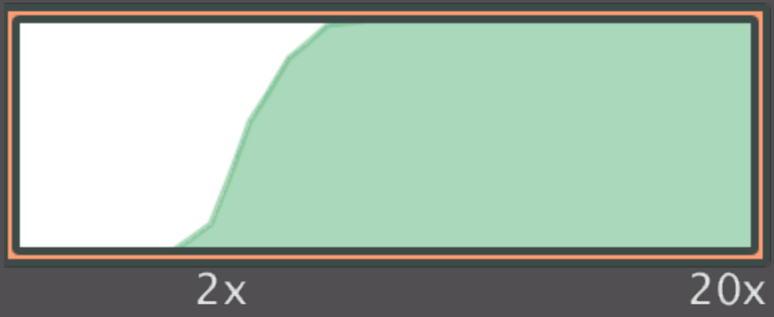
spectralnorm



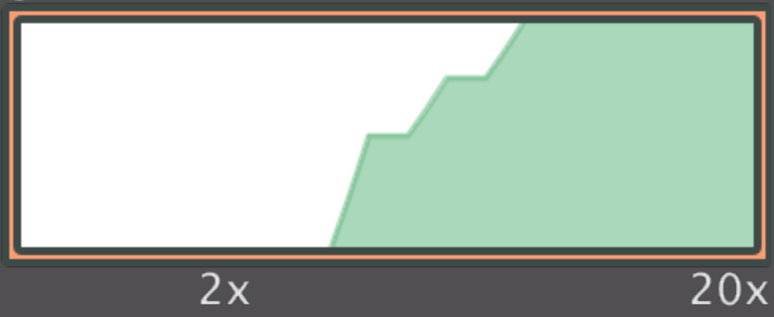
pystone



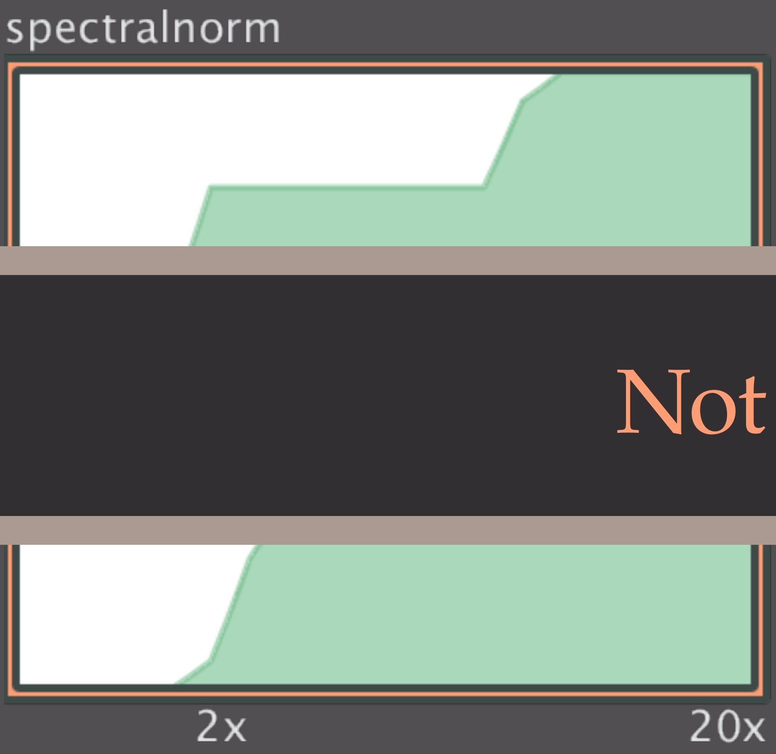
chaos

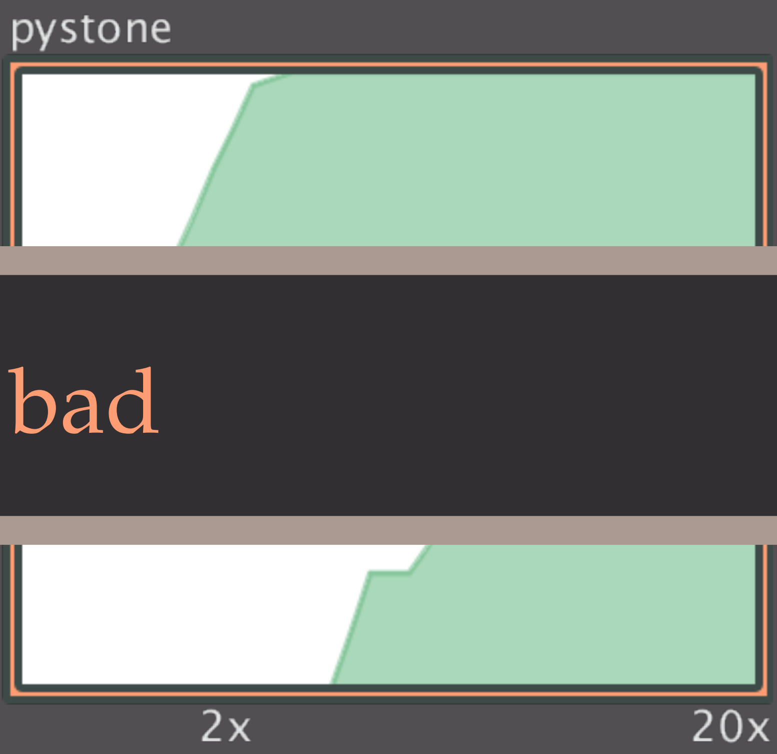


go



Reticulated Python different benchmarks





Not so bad

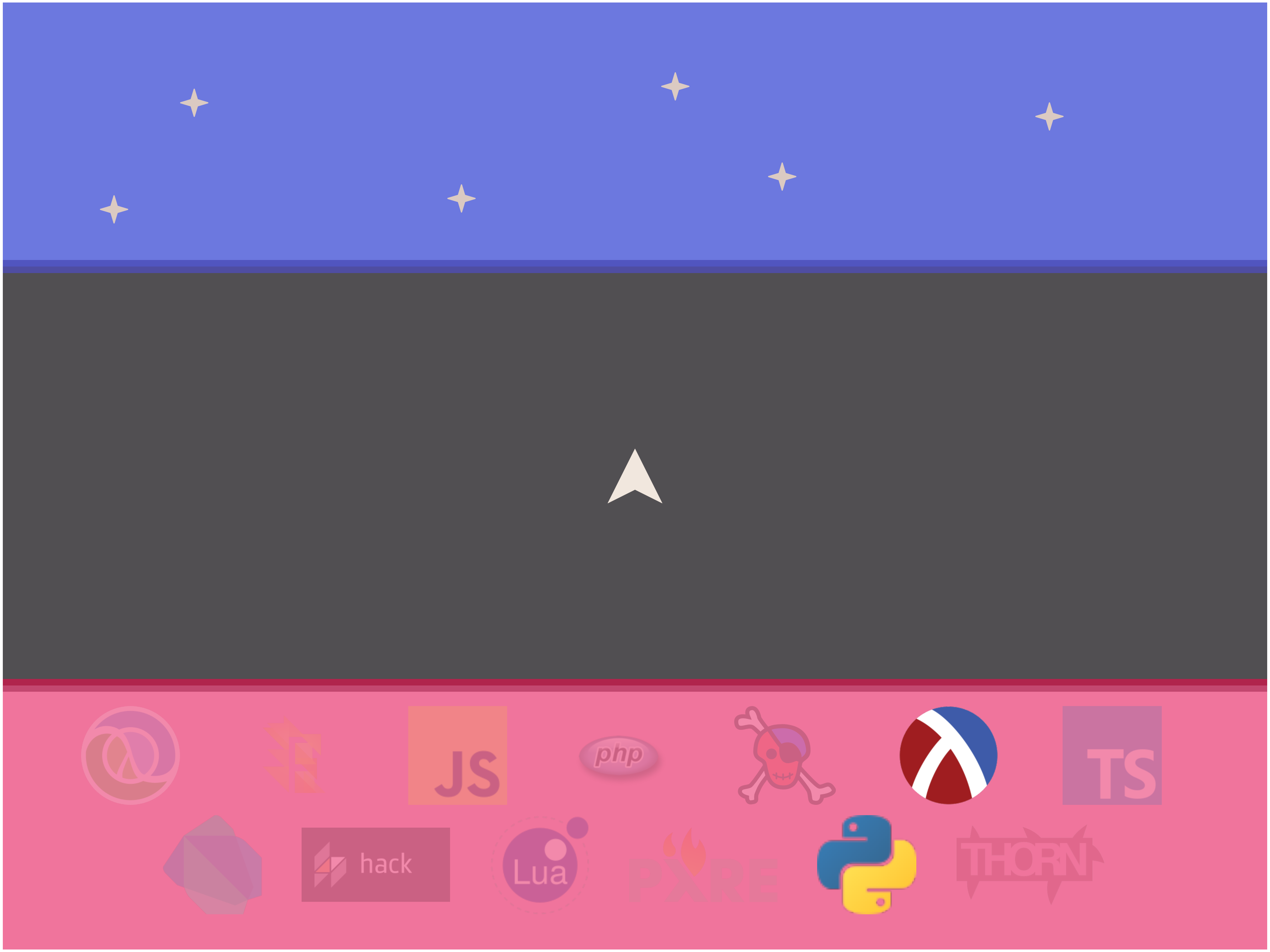
Bad

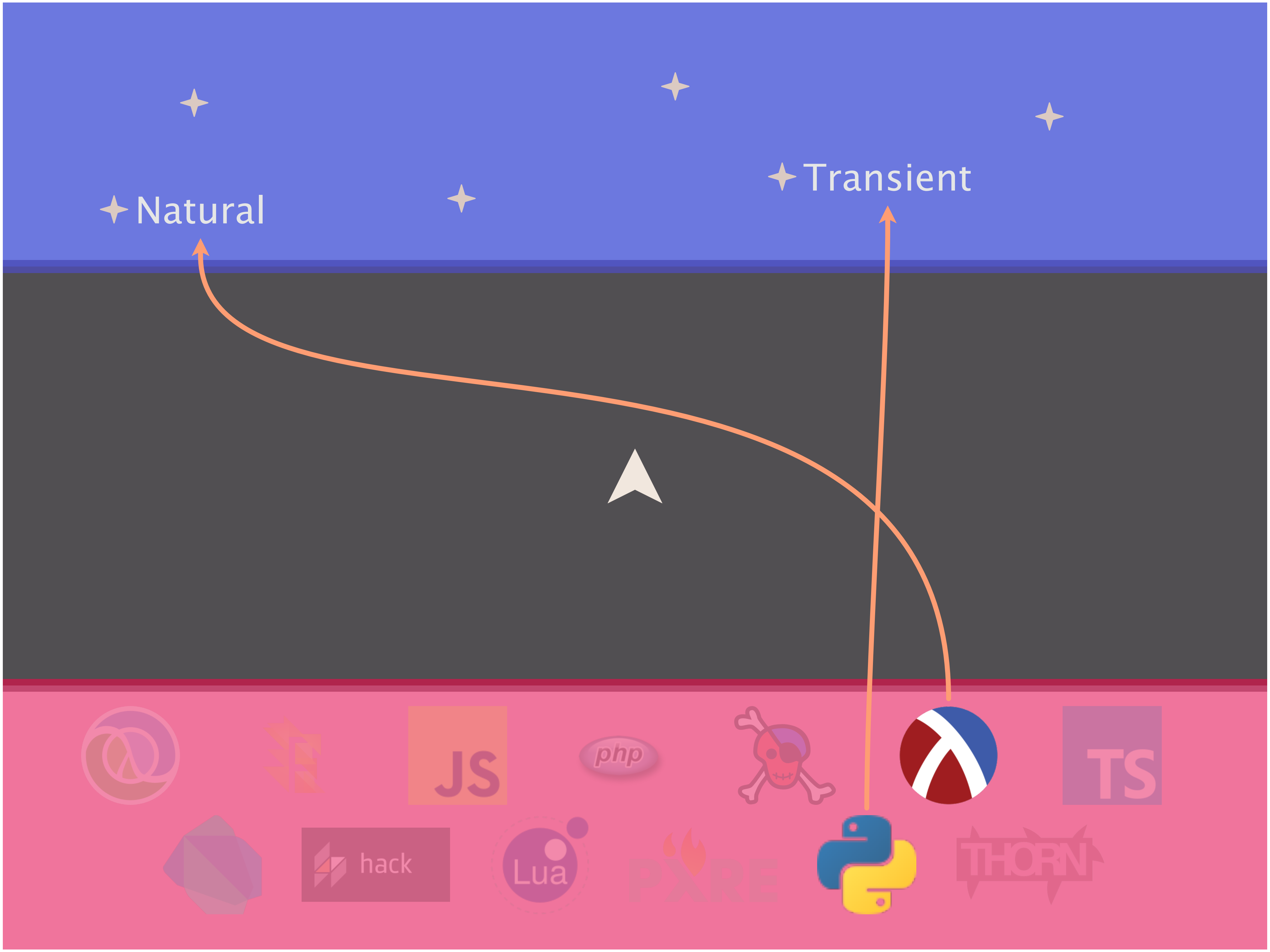
Not bad

Bad

Not bad

Q. Is Reticulated better, overall?


JS
php
TS
hack
Lua
PYRE
THORN


Natural
Transient
JS
php
TS
hack
Lua
PYRE
THORN

✦ Natural ✦ Transient

	Natural	Transient
`type soundness`		
`gradual guarantee`		
`blame theorem`		

Natural

Transient

	Natural	Transient
type soundness	✓	✓
gradual guarantee	✓	✓
blame theorem	✓	✓

✦ Natural ✦ Transient

```
#lang untyped

(f (λ "A"))
```

```
#lang typed

(define (f (x : (-> Num)))
  (g x))
```

```
#lang untyped

(define (g y)
  (.... y))
```

Q. Can the type `(-> Num)` detect bad functions?

✦ Natural ✦ Transient

```
#lang untyped

(f (λ "A"))
```

```
#lang typed

(define (f (x : (-> Num)))
  (g x))
```

```
#lang untyped

(define (g y)
  (.... y))
```

Q. Can the type `(-> Num)` detect bad functions?

A. Natural = Yes

A. Transient = No

expects Num , Str ...

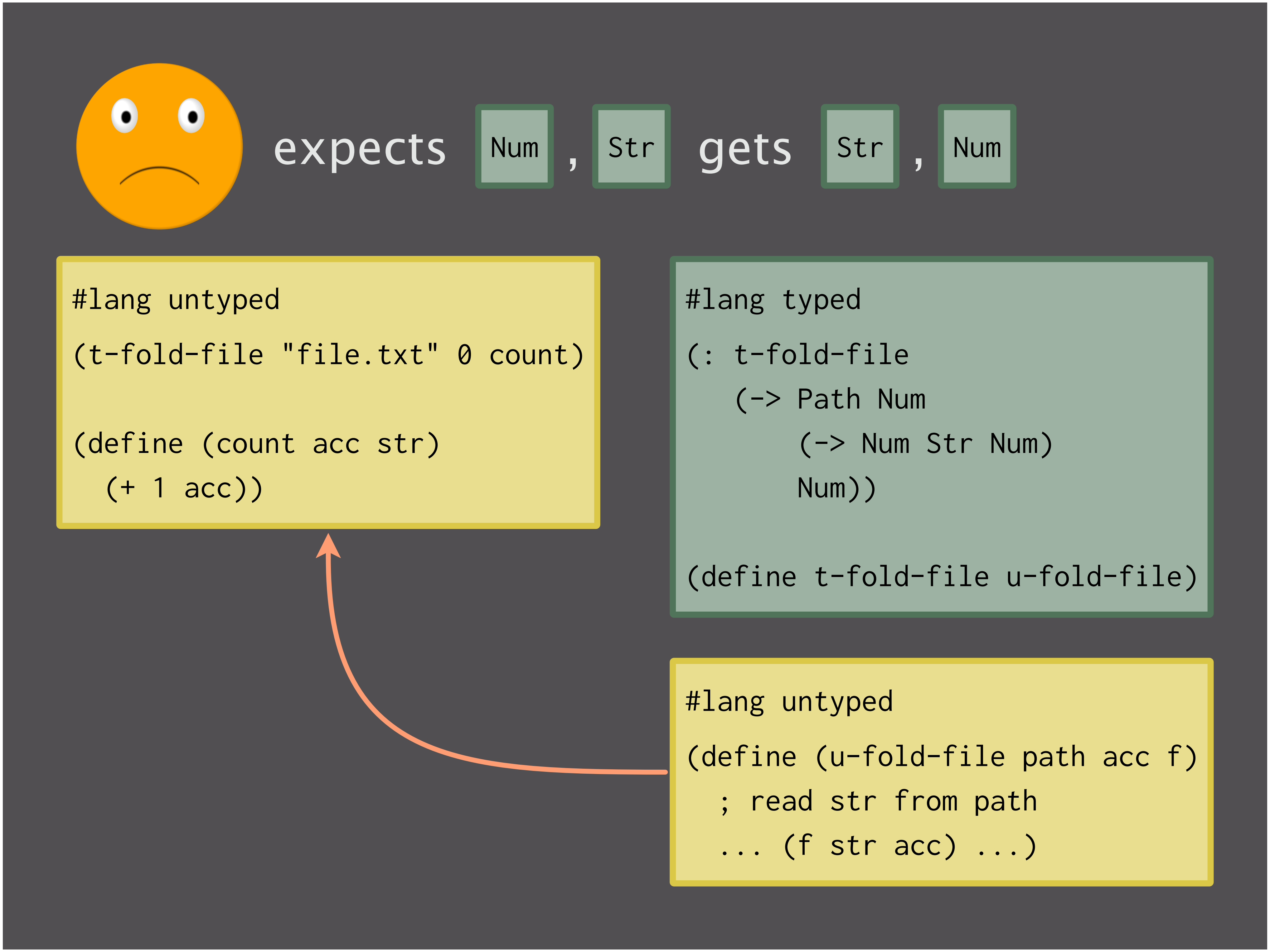
```
#lang untyped

(t-fold-file "file.txt" 0 count)

(define (count acc str)
  (+ 1 acc))
```

```
#lang typed

(: t-fold-file
   (-> Path Num
       (-> Num Str Num)
       Num))

(define t-fold-file u-fold-file)
```

expects `Num` , `Str` gets `Error: + bad input`

```
#lang untyped

(t-fold-file "file.txt" 0 count)


(define (count acc str)
  (+ 1 acc))
```

```
#lang typed

(: t-fold-file
   (-> Path Num
       (-> Num Str Num)
       Num))


(define t-fold-file u-fold-file)
```



expects Num , Str gets Str , Num

```
#lang untyped

(t-fold-file "file.txt" 0 count)

(define (count acc str)
  (+ 1 acc))
```

```
#lang typed

(: t-fold-file
   (-> Path Num
       (-> Num Str Num)
       Num))

(define t-fold-file u-fold-file)
```

```
#lang untyped

(define (u-fold-file path acc f)
  ; read str from path
  ... (f str acc) ...)
```

expects Num , Str gets Str , Num

```
#lang untyped

(t-fold-file "file.txt" 0 count)


(define (count acc str)
  (+ 1 acc))
```

```
#lang typed

(: t-fold-file
   (-> Path Num
       (-> Num Str Num)
       Num))

(define t-fold-file u-fold-file)
```

Q. Do types protect the callback?

A. Transient = No

A. Natural = Yes

```
#lang untyped

(define (u-fold-file path acc f)
  ; read str from path
  ... (f str acc) ...)
```

✦ Natural ✦ Transient

	Natural	Transient
`type soundness`	✓	✓
`gradual guarantee`	✓	✓
`blame theorem`	✓	✓

- But Natural and Transient disagree

Natural

Transient

	Natural	Transient
type soundness	✓	✓

Need to measure type guarantees

- But Natural and Transient disagree

Natural

Transient

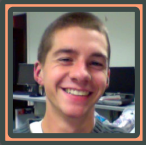
* How to assess type guarantees

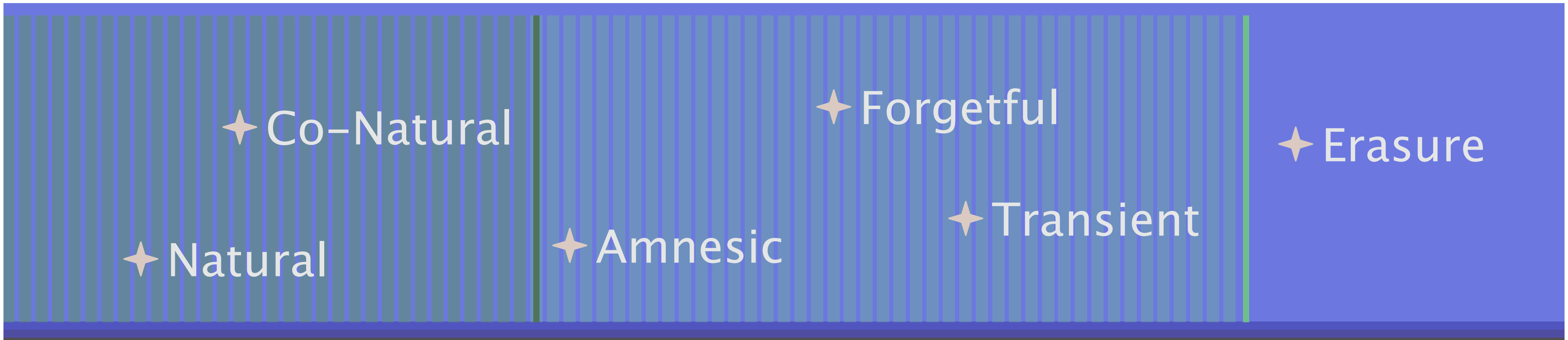
- Co–Natural
- Forgetful
- Erasure
- Transient
- Natural
- Amnesic

✦ Co–Natural ✦ Forgetful ✦ Erasure

✦ Natural ✦ Amnesic ✦ Transient

0. before = sound vs. unsound





0. before = sound vs. unsound

1. Complete Monitoring ~ types guard all boundaries

Complete Monitoring vs. Type Soundness

Complete Monitoring vs. Type Soundness

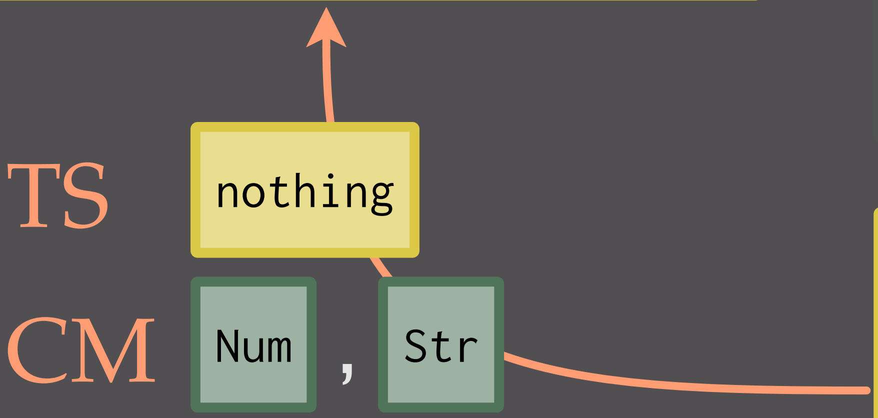
```
#lang untyped

(t-fold-file "file.txt" 0 count)

(define (count acc str)
  (+ 1 acc))
```

```
#lang typed

(: t-fold-file
   (-> Path Num
       (-> Num Str Num)
       Num))

(define t-fold-file u-fold-file)
```

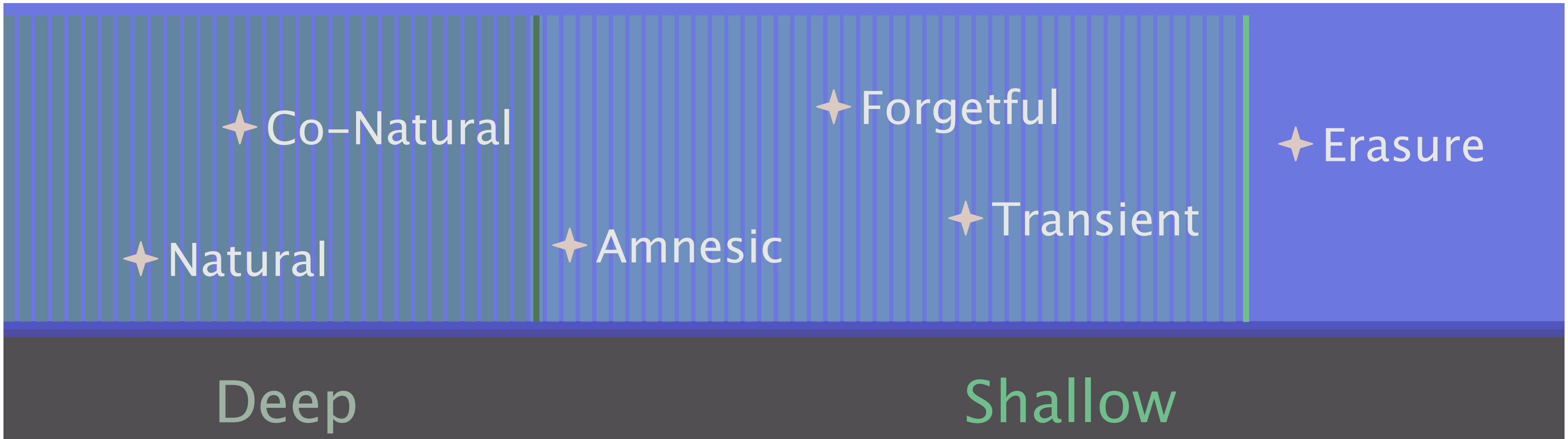
Q. Do types protect the callback?

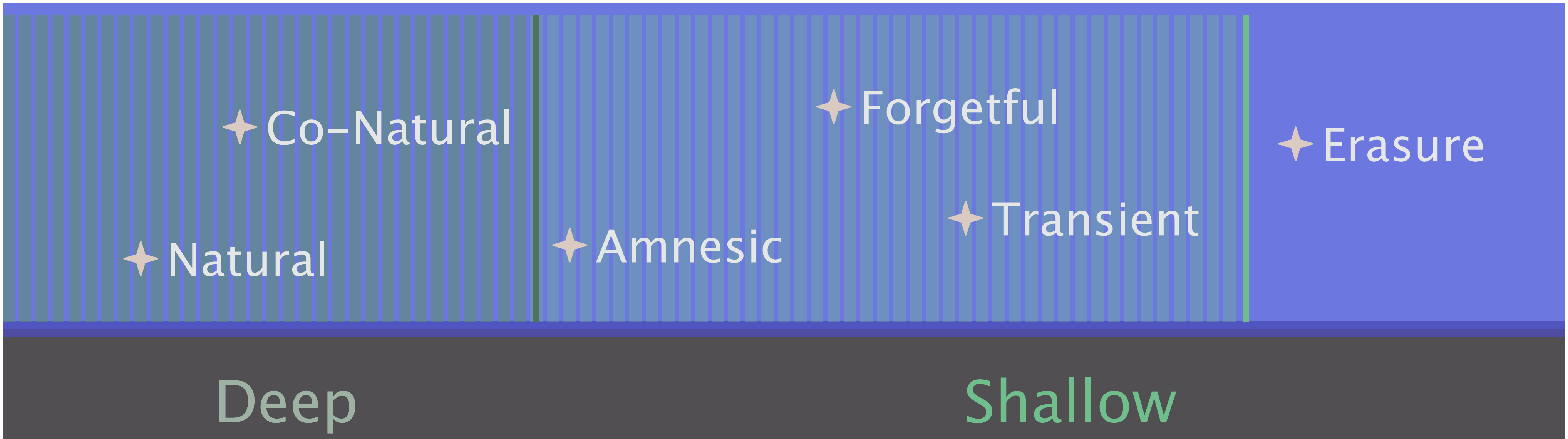
TS =/> Yes

CM => Yes



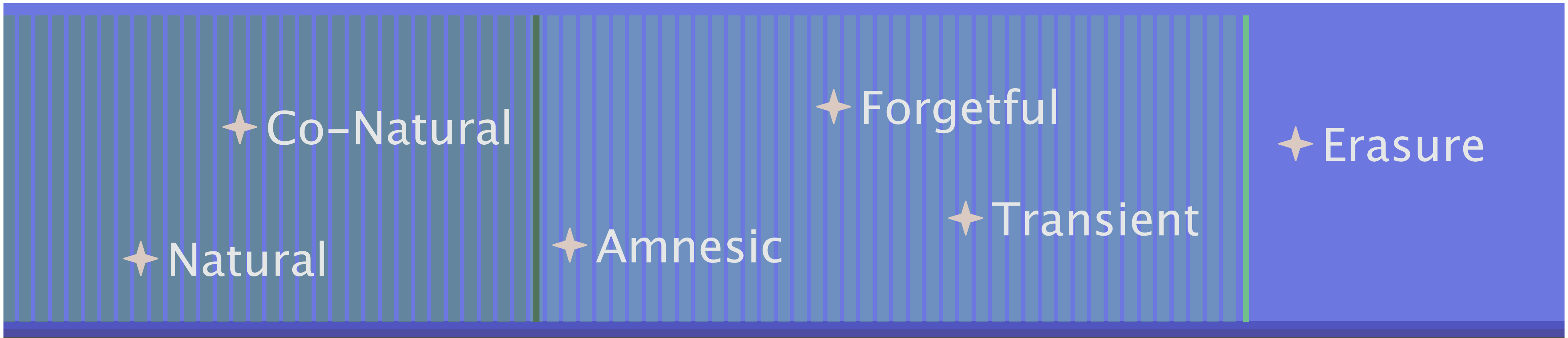
```
#lang untyped

(define (u-fold-file path acc f)
  ; read str from path
  ... (f str acc) ...)
```


Co-Natural
Natural
Forgetful
Amnesic
Transient
Erasure
Deep
Shallow



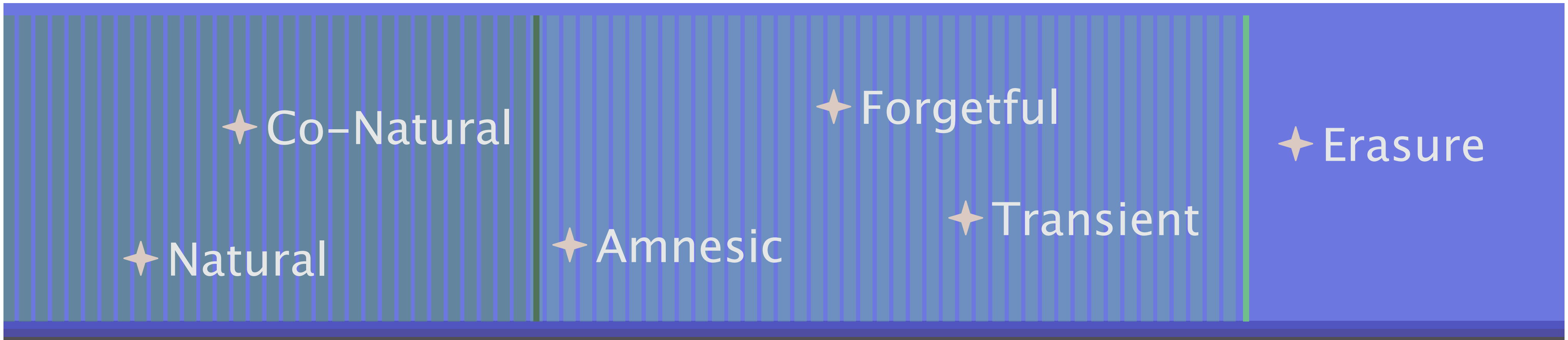
Shallow types are sound.

Deep types protect untyped code, too.



0. before = sound vs. unsound

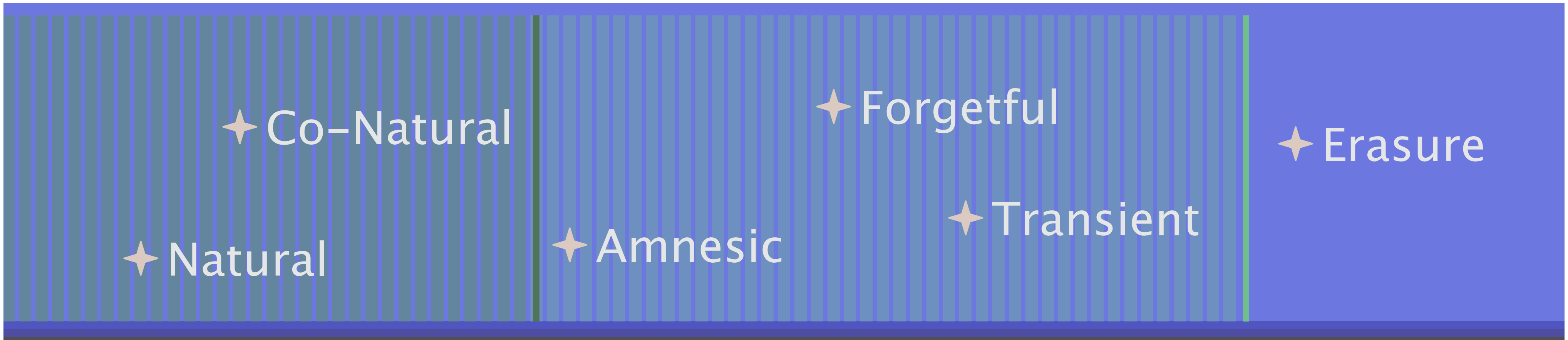
1. Complete Monitoring ~ types guard all boundaries



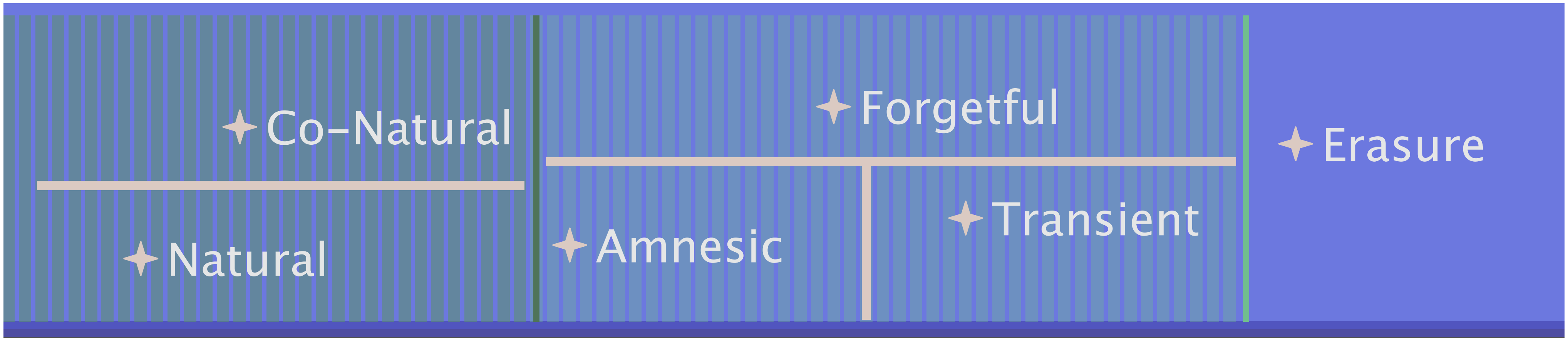
0. before = sound vs. unsound

1. Complete Monitoring ~ types guard all boundaries

2. Blame Soundness ~ errors are accurate



0. before = sound vs. unsound
1. Complete Monitoring ~ types guard all boundaries
2. Blame Soundness ~ errors are accurate
3. Blame Completeness ~ errors are exhaustive



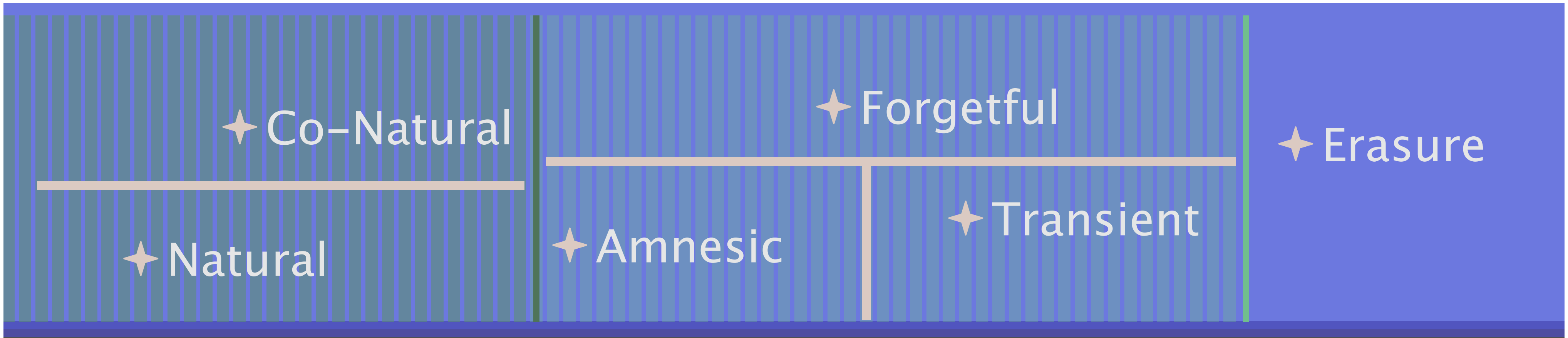
0. before = sound vs. unsound

1. Complete Monitoring ~ types guard all boundaries

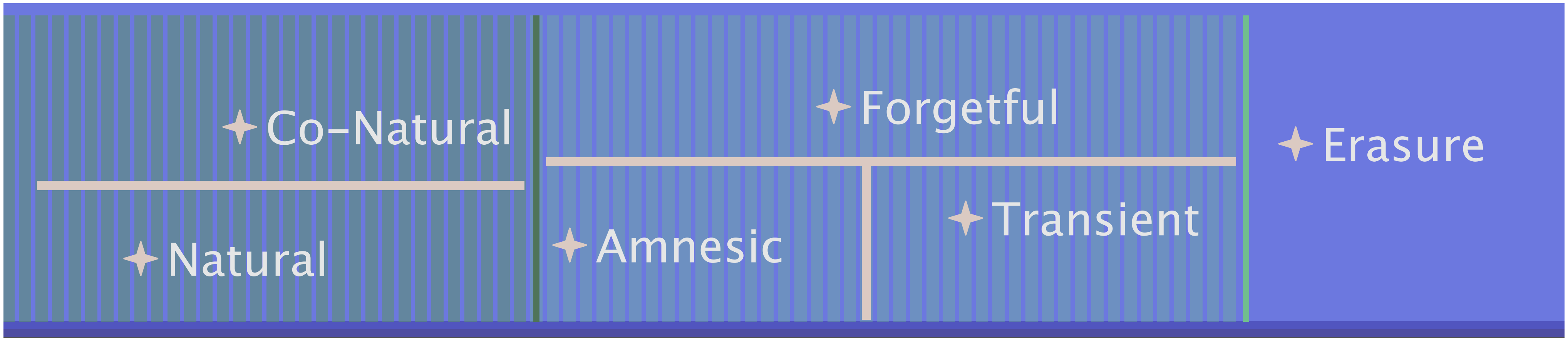
2. Blame Soundness ~ errors are accurate

3. Blame Completeness ~ errors are exhaustive

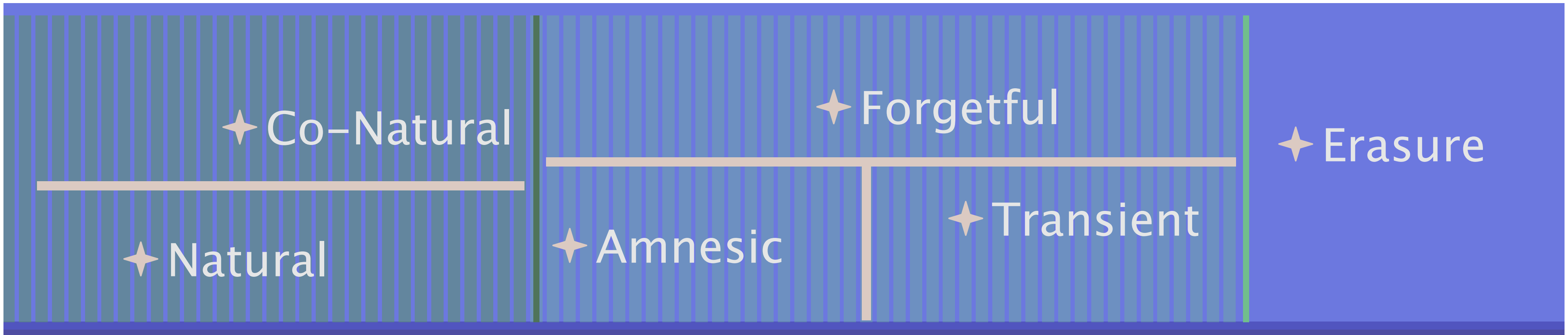
4. Error Preorder ~ head-to-head test



	Natural	C	F	Transient	A	E
type soundness						
complete monitoring						
blame soundness						
blame completeness						
error preorder						



	Natural	C	F	Transient	A	E
type soundness	✓	✓	✓	y	✓	✗
complete monitoring	✓	✓	✗	✗	✗	✗
blame soundness	✓	✓	✓	h	✓	0
blame completeness	✓	✓	✓	✗	✓	✗
error preorder	Natural $<$	C $<$	F $<$	Transient $=$	A $<$	E

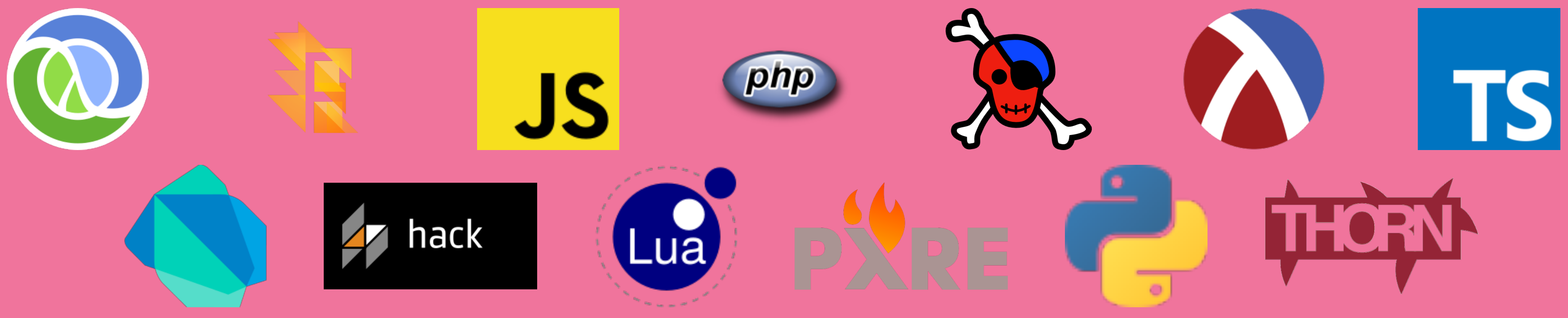


My Research

brings order to the design space

* How to assess type guarantees

* How to measure performance



Goal: mixed-typed code with **strong** guarantees

Problem: high performance overhead

Goal: mixed-typed code with **strong** guarantees

Problem: high performance overhead

Q. What to do?

Goal: mixed-typed code with **strong** guarantees

Problem: high performance overhead

Q. What to do?

a. build a new language

a. build a new compiler

a. improve the current compiler

Goal: mixed-typed code with **strong** guarantees

Problem: high performance overhead

Q. What to do?

a. build a new language

a. build a new compiler

✓ a. improve the current compiler

Goal: mixed-typed code with **strong** guarantees

Problem: high performance overhead

Q. What to do?

a. build a new language

a. build a new compiler

✓ a. improve the current compiler
- re-use type system
- add new semantics

Thesis Statement

Deep and Shallow types can interoperate.

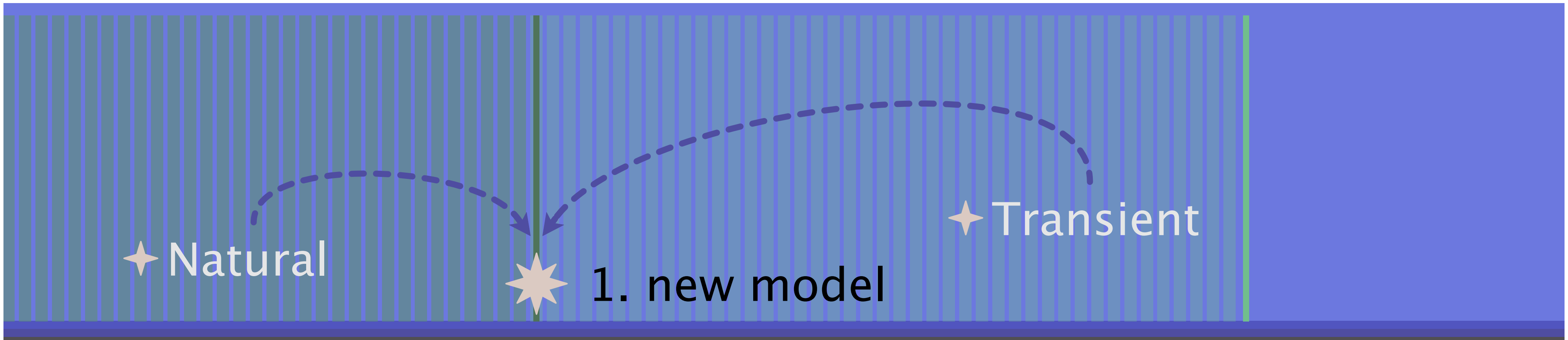
preserving their formal properties

Programmers can use these types to:

- strengthen Shallow guarantees
- avoid unimportant Deep errors
- lower runtime costs

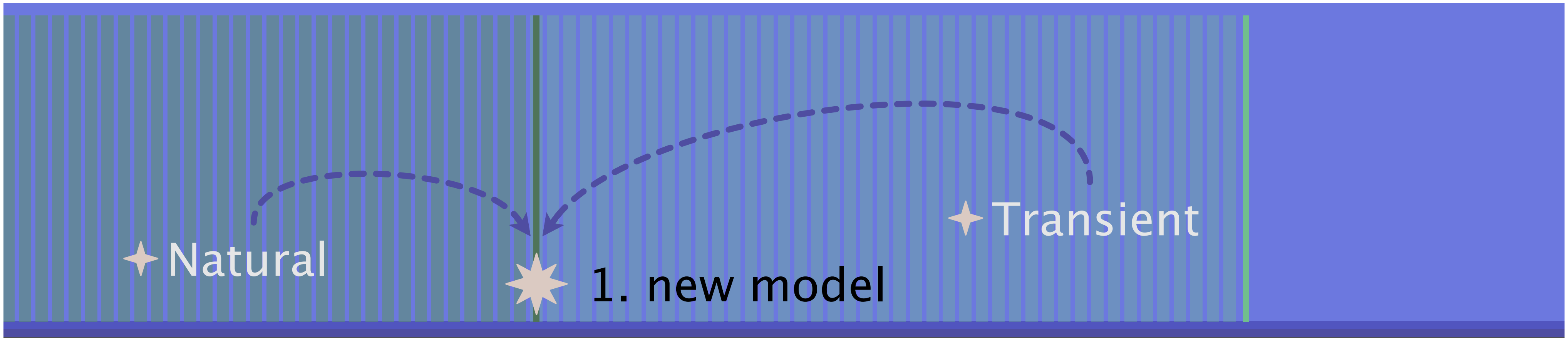
Unpublished Results

Plan:



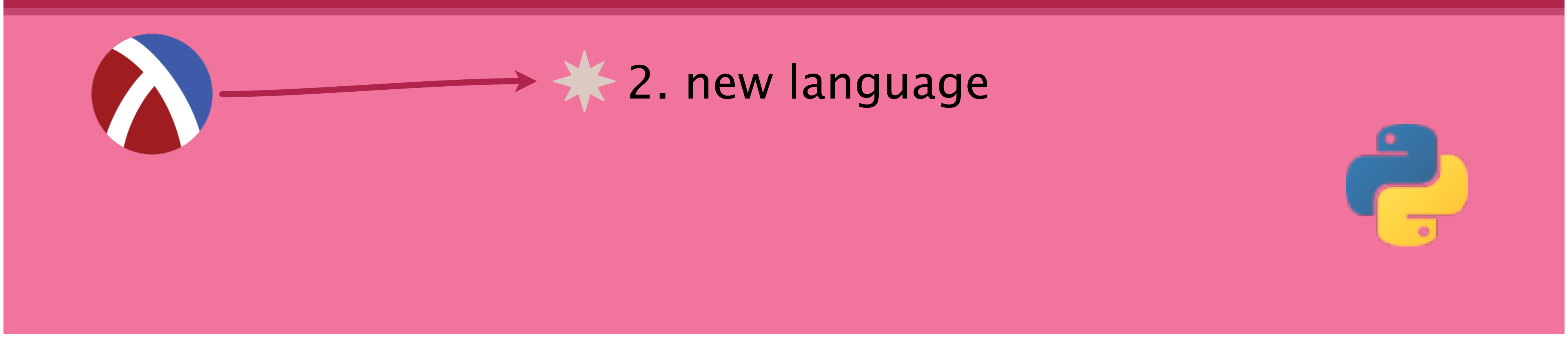
Plan:

- combine Natural + Transient



Plan:

- combine Natural + Transient
- extend TR



Model Deep + Shallow + Untyped

```
s = x | i | (s, s) | λx. s | λx:T. s |
    ....


T = .....

L = ....
```

Model Deep + Shallow + Untyped

```
s = x | i | (s, s) | λx. s | λx:T. s |
    unop s | binop s s | app s s |
    ....

T = ....

L = ....
```

Model Deep + Shallow + Untyped

```
s = x | i | (s, s) | λx. s | λx:T. s |
    unop s | binop s s | app s s |
    module L s

T = ....

L = ....
```

Model Deep + Shallow + Untyped

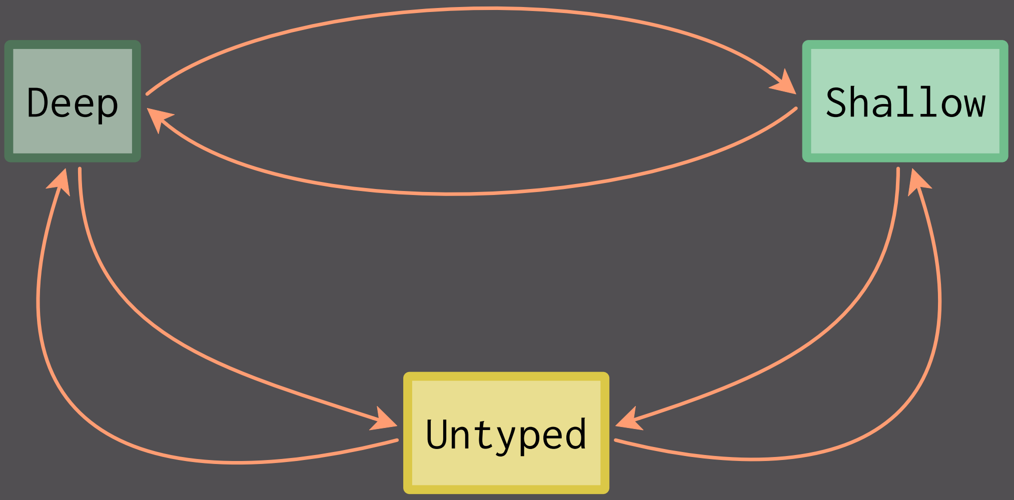
```
s = x | i | (s, s) | λx. s | λx:T. s |
    unop s | binop s s | app s s |
    module L s

T = Nat | Int | T x T | T -> T

L = ....
```

Model Deep + Shallow + Untyped

```
s = x | i | (s, s) | λx. s | λx:T. s |
    unop s | binop s s | app s s |
    module L s

T = Nat | Int | T x T | T -> T

L = Deep | Shallow | Untyped
```

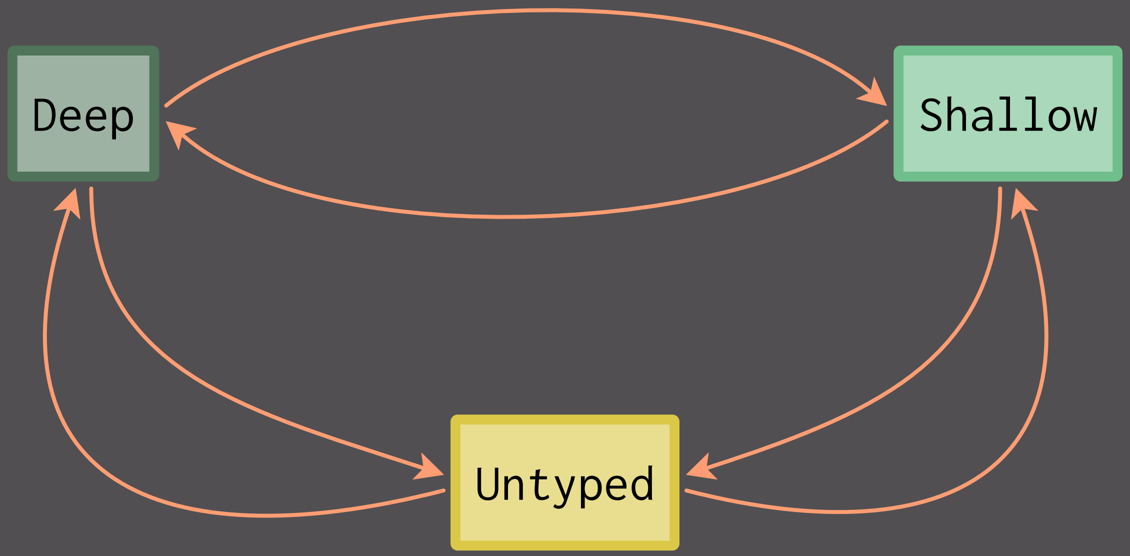
Model Deep + Shallow + Untyped

```
s = x | i | (s, s) | λx. s | λx:T. s |
    unop s | binop s s | app s s |
    module L s

T = Nat | Int | T x T | T -> T

L = Deep | Shallow | Untyped
```

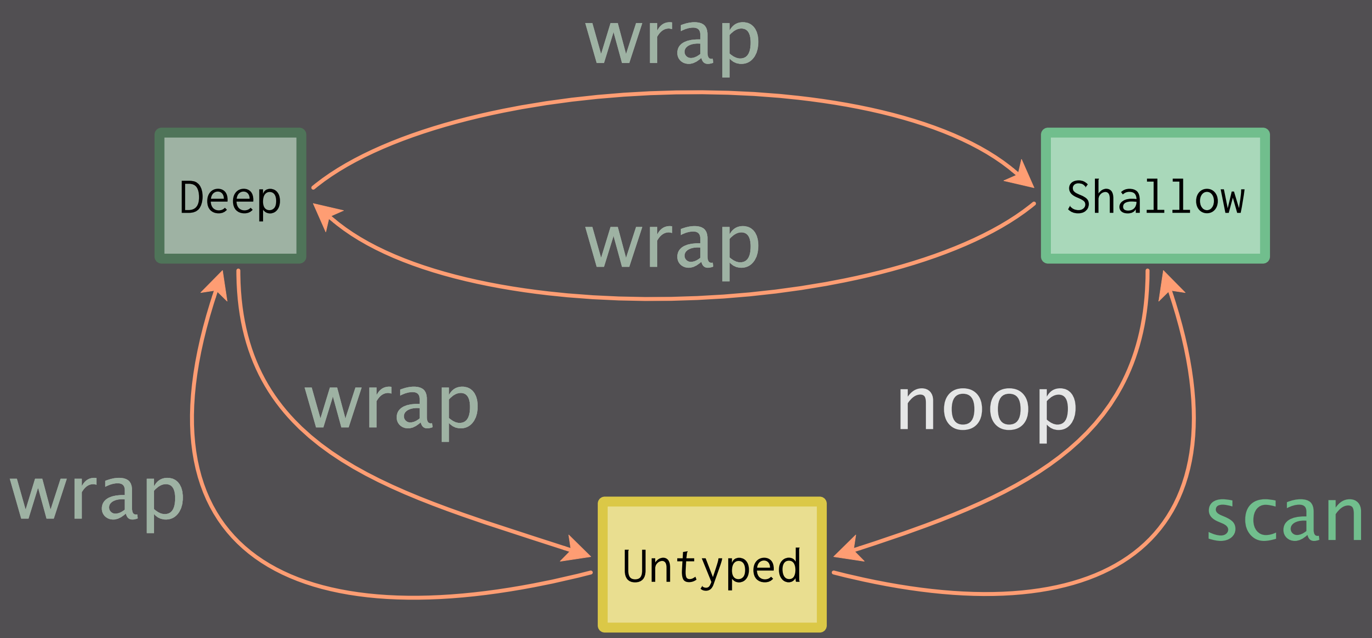


$$\begin{tikzcd}
\text{Deep} \arrow[rr, bend left] \arrow[dr, bend right] & & \text{Shallow} \arrow[ll, bend left] \arrow[dl, bend left] \\
& \text{Untyped} \arrow[ul, bend left] \arrow[ur, bend right] &
\end{tikzcd}$$

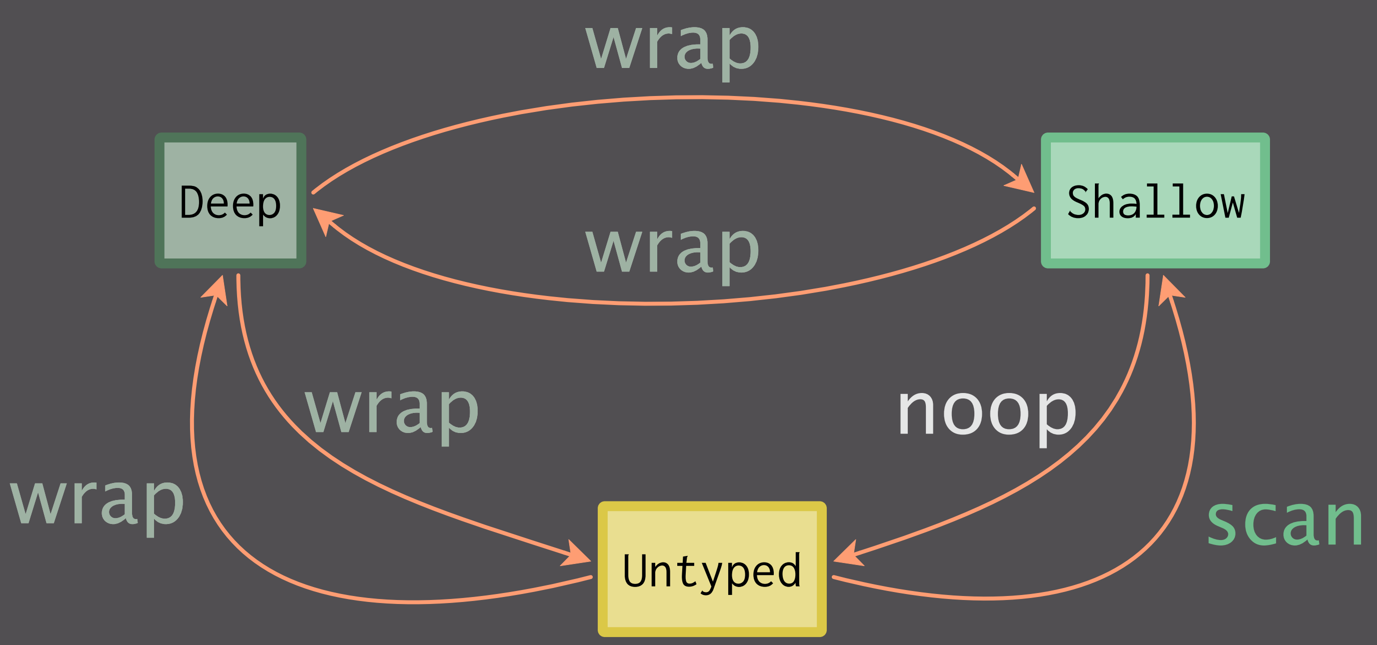
Model Boundaries



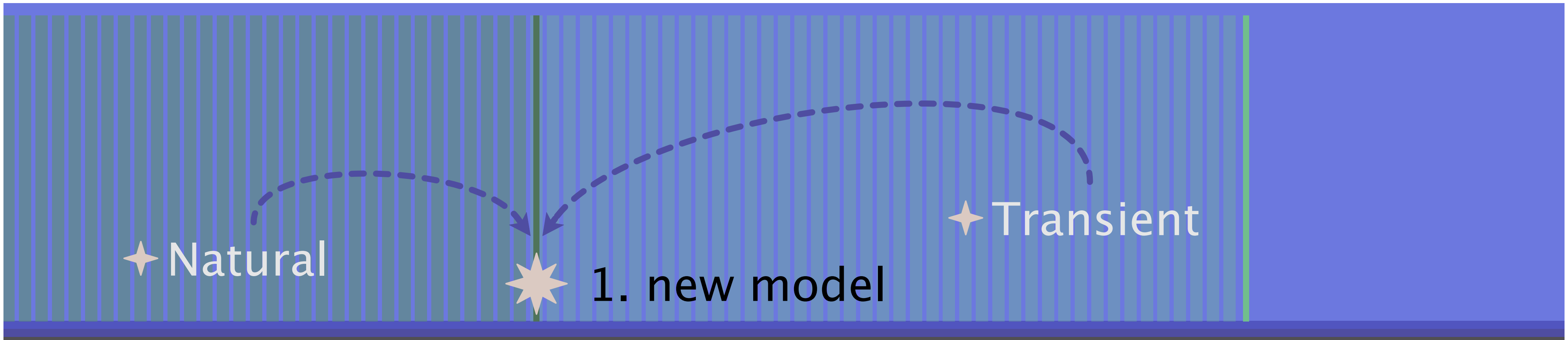
Model Boundaries

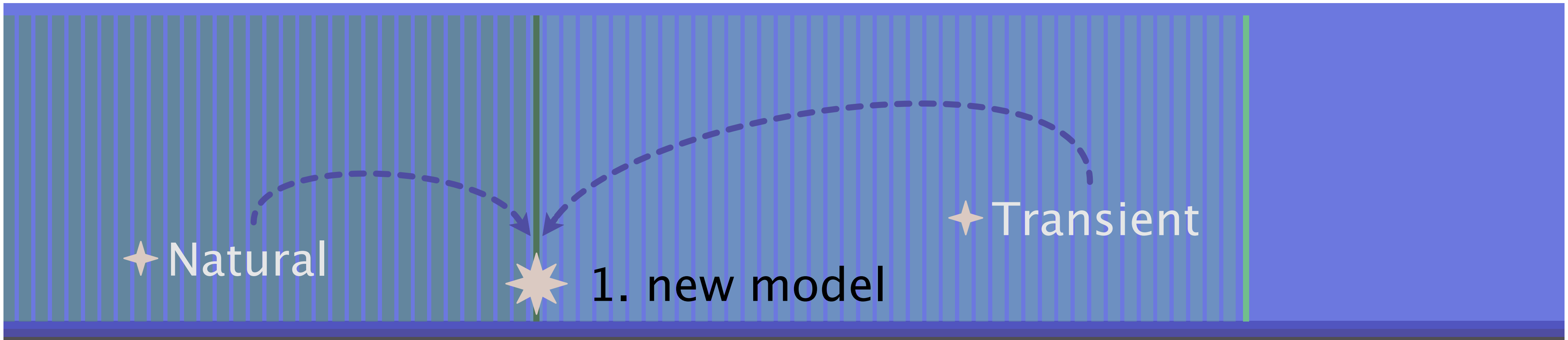


Model Boundaries



Deep	=	wrap, or fully check
Shallow	=	spot-check inputs


Natural
Transient
1. new model



✓ Type Soundness

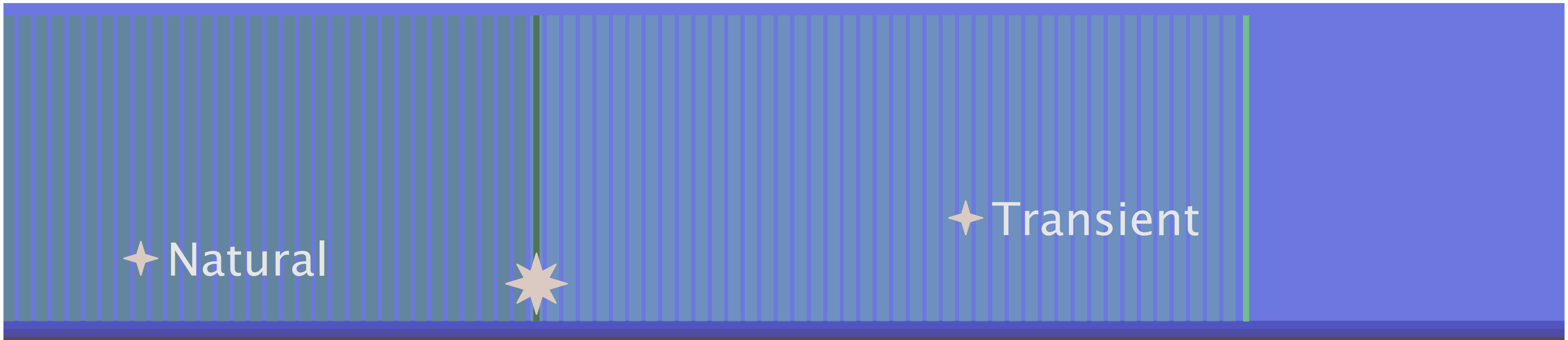
types predict outcomes

Deep · Shallow

✓ Complete Monitoring

Deep types predict behaviors

Deep



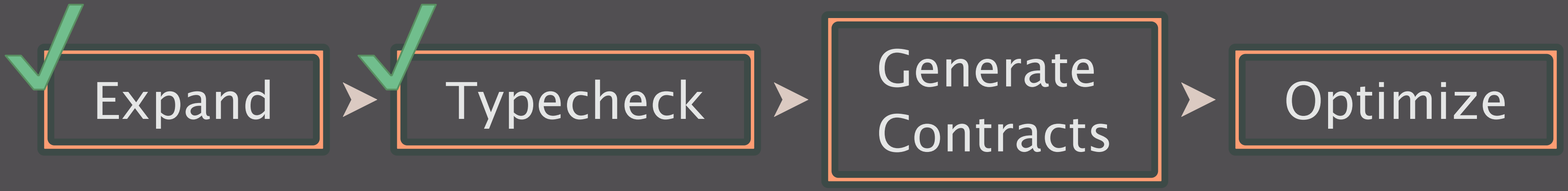
2. new language

Typed Racket Compiler

$$\begin{tikzcd} \boxed{\text{Expand}} \arrow[r] & \boxed{\text{Typecheck}} \arrow[r] & \boxed{\begin{array}{l}\text{Generate}\\ \text{Contracts}\end{array}} \arrow[r] & \boxed{\text{Optimize}} \end{tikzcd}$$

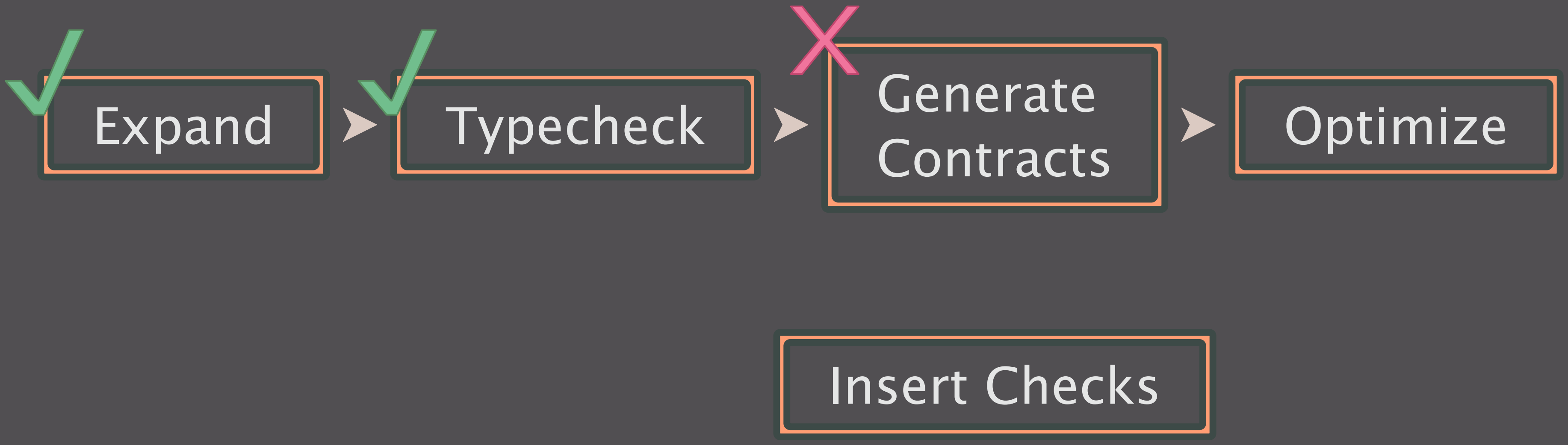
2. new language

Shallow Racket



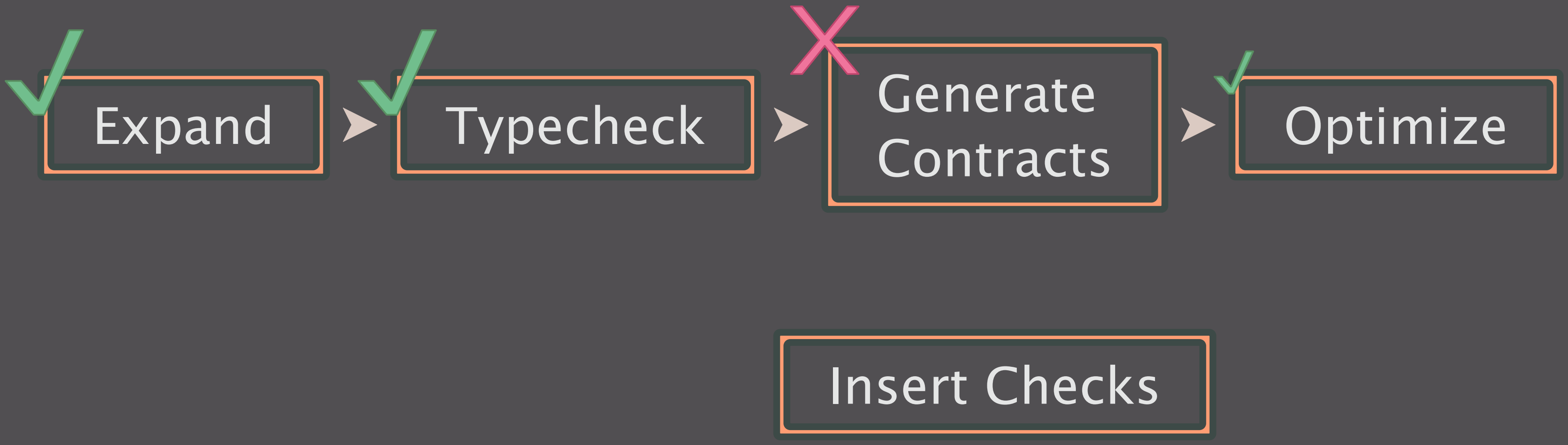
2. new language

Shallow Racket



2. new language

Shallow Racket



2. new language

Insert Checks types to shapes

design choice: enforce full type constructors

Insert Checks types to shapes

design choice: enforce full type constructors

Type	shape
`Num`	`number?`
`(Listof Num)`	`list?`
`(U Num Sym)`	`(or number? symbol?)`
`(-> Num Num)`	`(and procedure? (arity-includes 1))`

Optimize

apply

box

dead-code

extflonum

fixnum

float-complex

float

list

number

pair

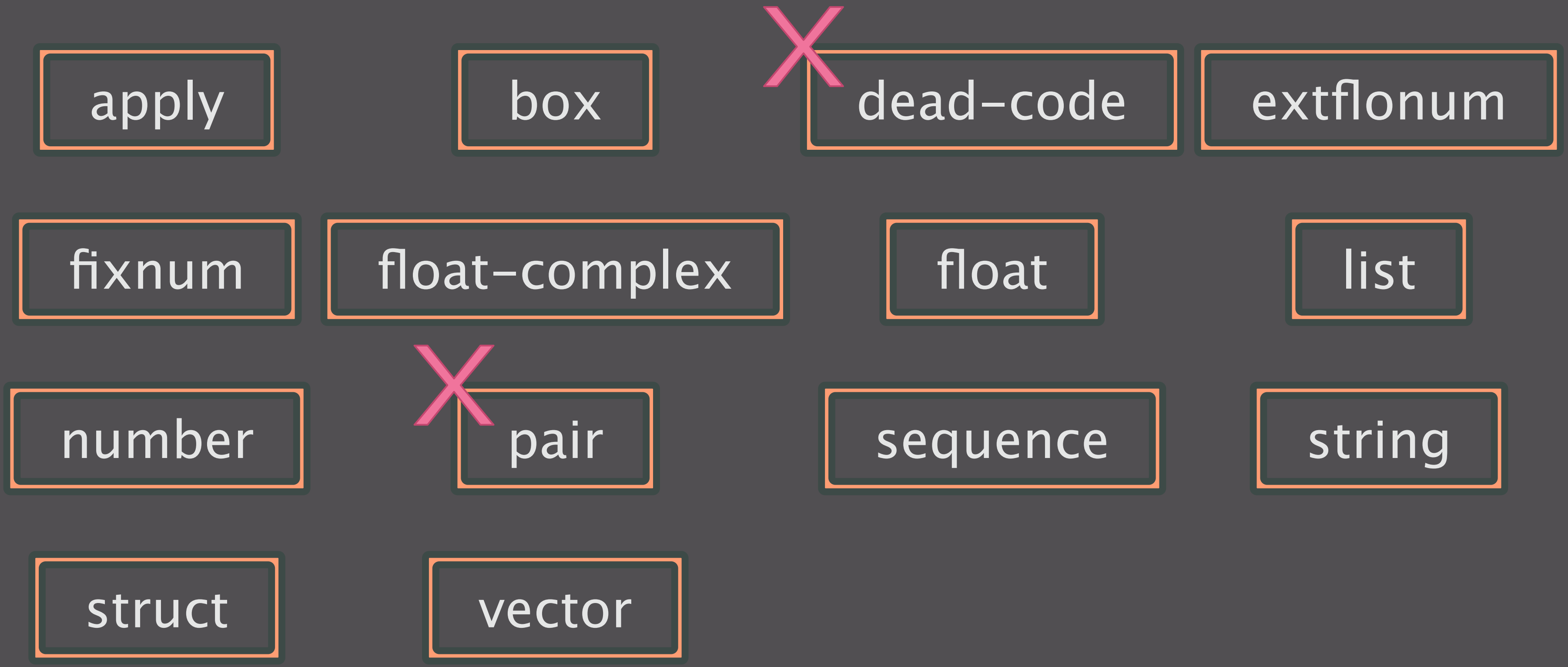
sequence

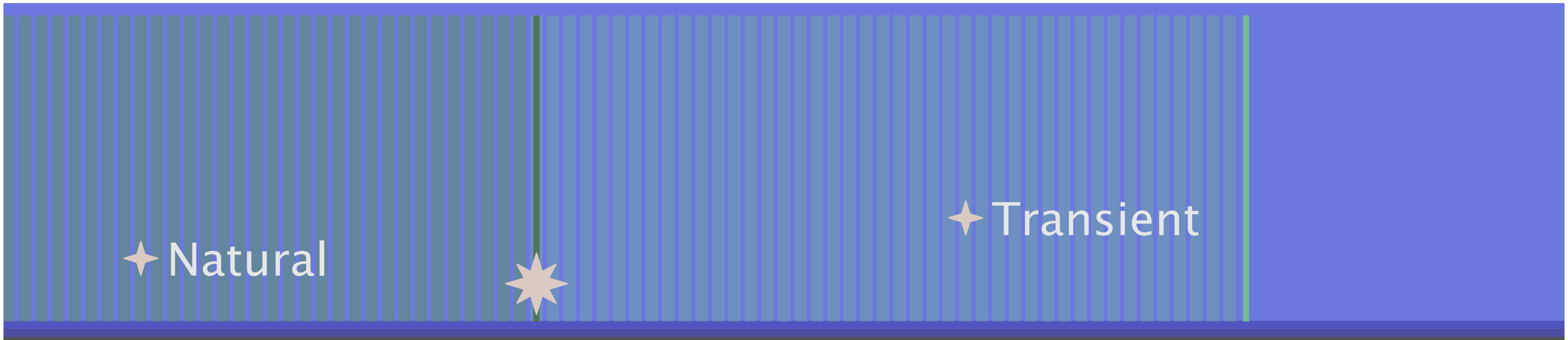
string

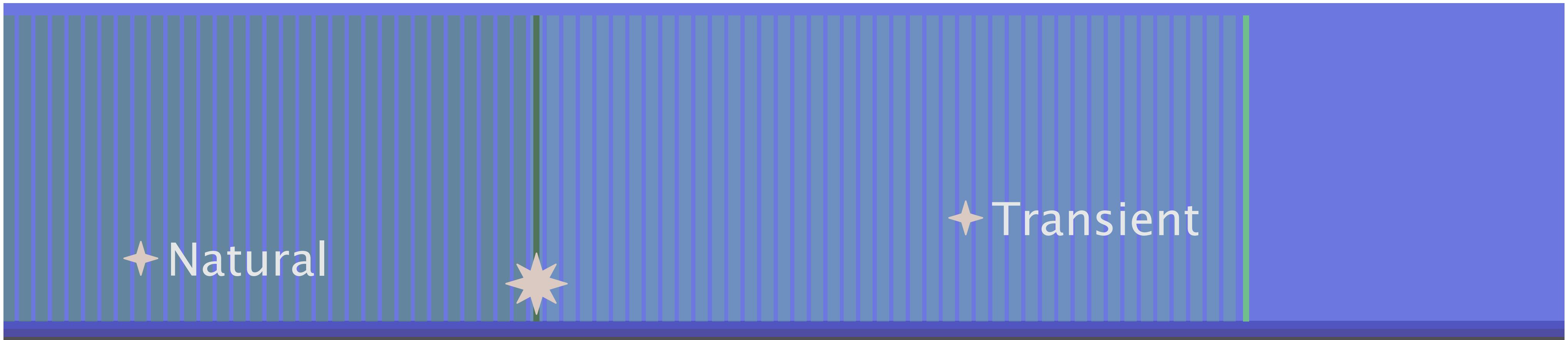
struct

vector

Optimize


apply
box
X
dead-code
extflonum
fixnum
float-complex
float
list
number
X
pair
sequence
string
struct
vector


Natural
Transient



- strengthen Shallow guarantees
- avoid unimportant Deep errors
- lower runtime costs

Shallow to Deep = stronger guarantees

Shallow to Deep = stronger guarantees

```
#lang untyped

(t-fold-file "file.txt" 0 count)

(define (count acc str)
  (+ 1 acc))
```

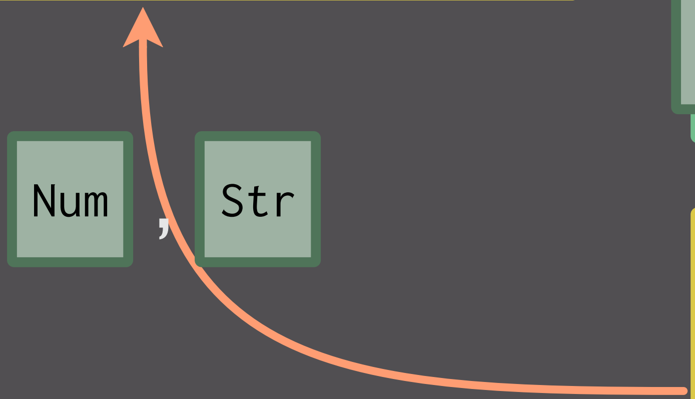
```
#lang shallow

(: t-fold-file
   (-> Path Num
       (-> Num Str Num)
       Num))

(define t-fold-file u-fold-file)
```

nothing

```
#lang untyped

(define (u-fold-file path acc f)
  ; read str from path
  ... (f str acc) ...)
```

Shallow to Deep = stronger guarantees

```
#lang untyped

(t-fold-file "file.txt" 0 count)

(define (count acc str)
  (+ 1 acc))
```



```
#lang deep

(: t-fold-file
   (-> Path Num
       (-> Num Str Num)
       Num))

(define t-fold-file u-fold-file)
```

```
#lang untyped

(define (u-fold-file path acc f)
  ; read str from path
  ... (f str acc) ...)
```

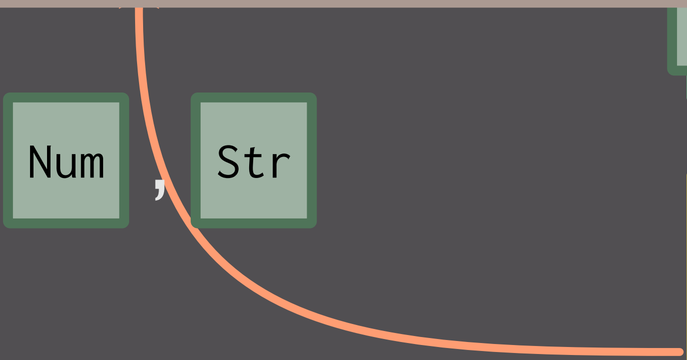
Shallow to Deep = stronger guarantees

```
#lang untyped
(t-fold-file "file.txt" 0 count)
```

```
#lang deep
(: t-fold-file
   (-> Path Num
```

Deep protects all boundaries

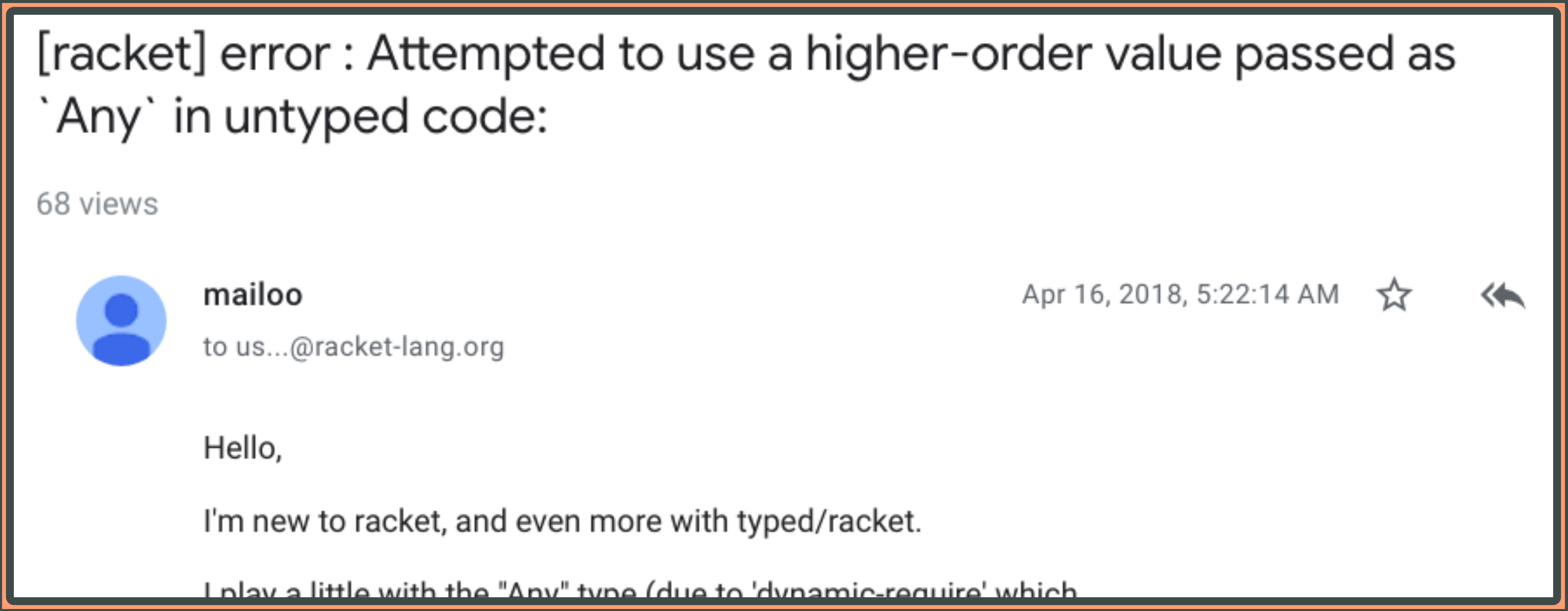
```
(define t-fold-file u-fold-file)
```



```
#lang untyped
(define (u-fold-file path acc f)
  ; read str from path
  ... (f str acc) ...)
```

Deep to Shallow = fewer errors

Deep to Shallow = fewer errors



[racket] error : Attempted to use a higher-order value passed as \`Any\` in untyped code:

68 views

mailoo
to us...@racket-lang.org

Apr 16, 2018, 5:22:14 AM

Hello,

I'm new to racket, and even more with typed/racket.

I play a little with the "Any" type (due to 'dynamic-require' which

Deep to Shallow = fewer errors

```
#lang deep

(: b Any)
(define b (box 42))
```

```
#lang untyped


(set-box! b 0)
```

Deep to Shallow = fewer errors

```
#lang deep

(: b Any)
(define b (box 42))
```

```
#lang untyped


(set-box! b 0)
```

Error: attempted to use higher-order value passed as `Any`

Deep to Shallow = fewer errors

```
#lang deep

(: b Any)
(define b (box 42))
```

```
#lang untyped


(set-box! b 0)
```

Error: attempted to use higher-order value passed as `Any`

```
#lang shallow

(: b Any)
(define b (box 42))
```

```
#lang untyped


(set-box! b 0)
```

OK

Deep to Shallow = fewer errors

Shallow can run almost all type-correct code

Error: attempted to use higher-order value passed as `Any`

```
#lang shallow

(: b Any)
(define b (box 42))
```

```
#lang untyped


(set-box! b 0)
```

OK

Better Performance

```
#lang untyped
....
```

```
#lang untyped
....
```

~ 2 sec.

Untyped baseline

Better Performance

```
#lang untyped      #lang untyped
....               ....
```

~ 2 sec.

Untyped baseline

```
#lang untyped      #lang deep
....               ....
```

~ 13 sec.

```
#lang untyped      #lang shallow
....               ....
```

~ 4 sec.

Mixed : Shallow wins

Better Performance

```
#lang untyped    #lang untyped
....             ....
```

~ 2 sec.

Untyped baseline

```
#lang untyped    #lang deep
....             ....
```

~ 13 sec.

```
#lang untyped    #lang shallow
....             ....
```

~ 4 sec.

Mixed : Shallow wins

```
#lang deep       #lang deep
....             ....
```

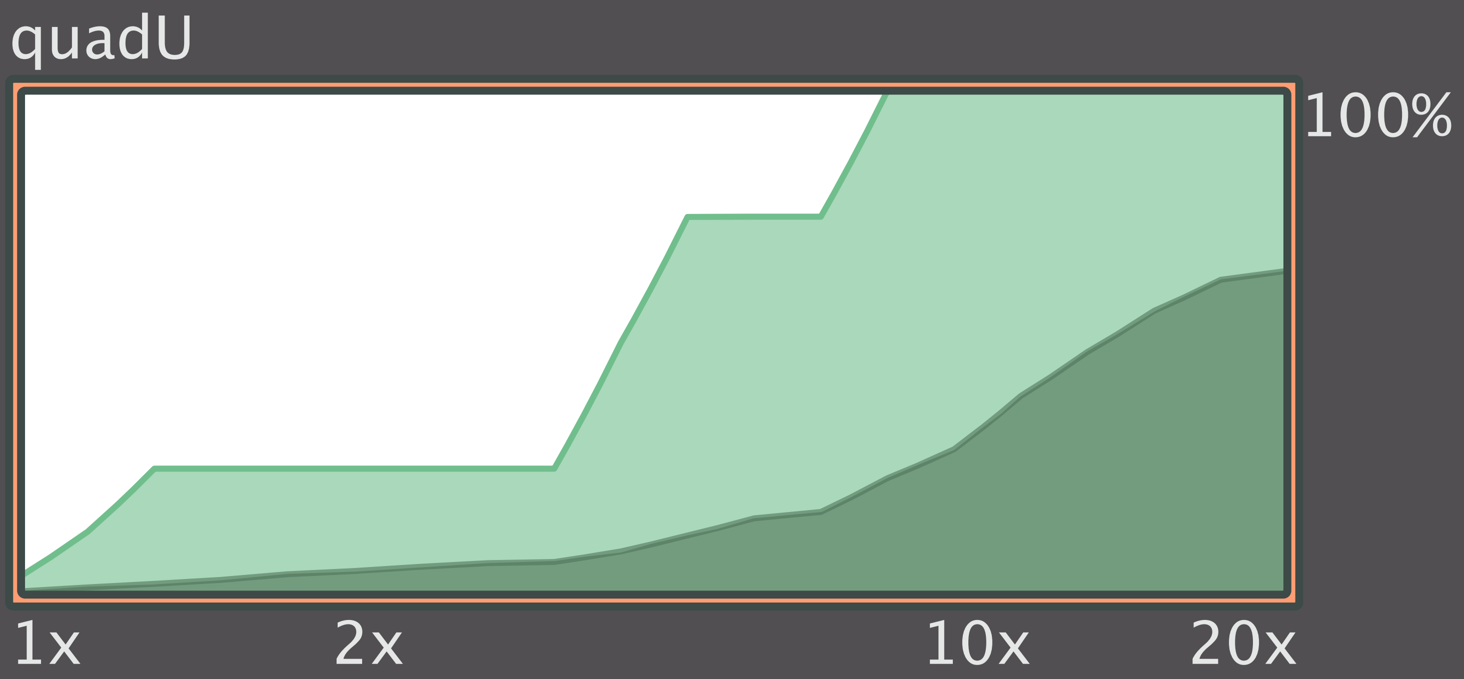
< 2 sec.

```
#lang shallow    #lang shallow
....             ....
```

~ 5 sec.

Typed : Deep wins

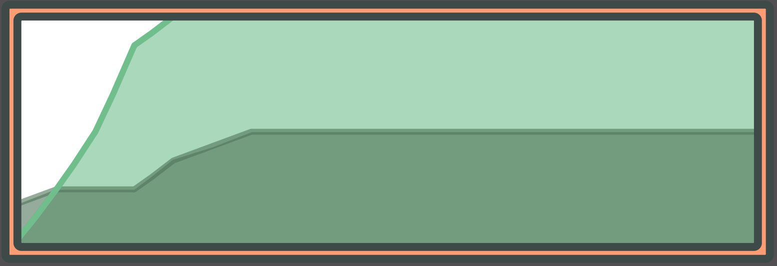
Better Performance



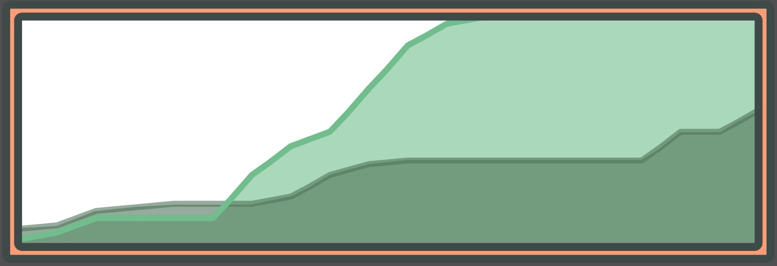
Deep + Shallow = maximize D-deliverable cfgs.

Better Performance

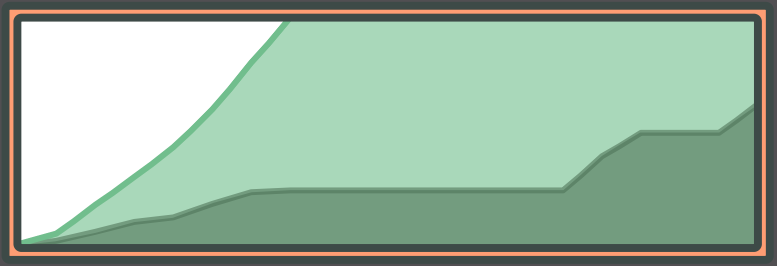
jpeg



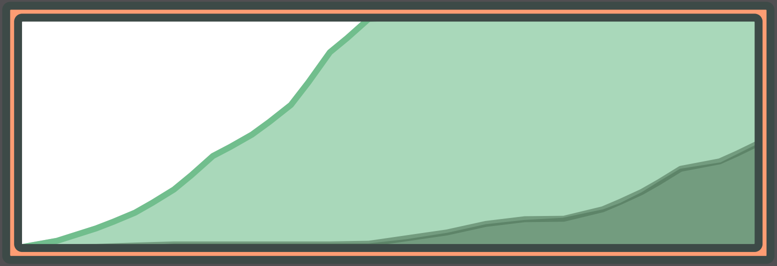
suffixtree



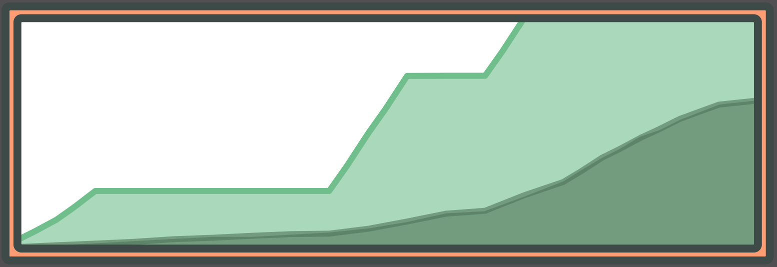
take5



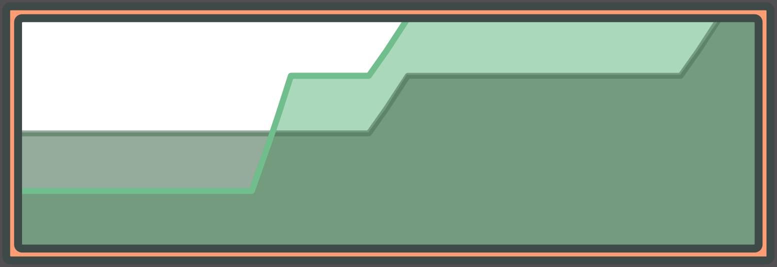
synth



quadU

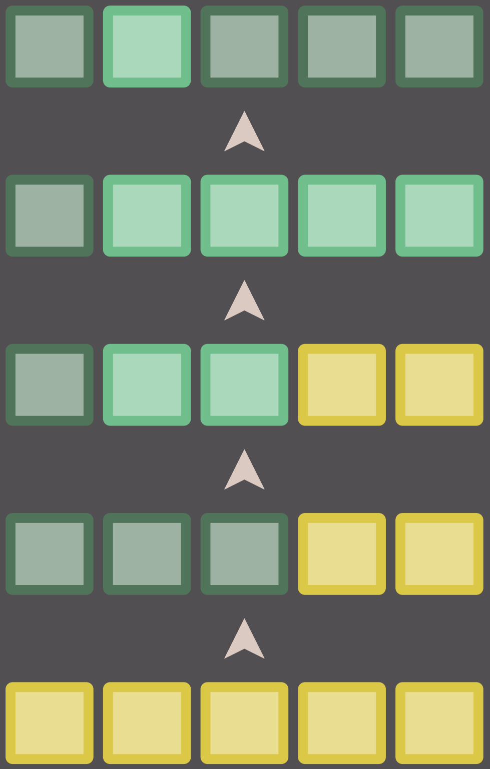


sieve



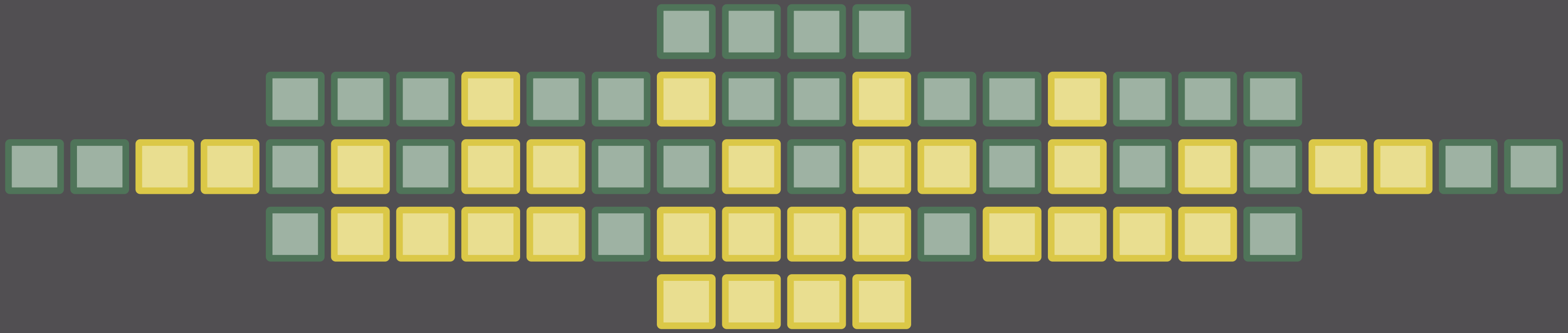
New Migration Plan

1. Deep, until slow
2. Shallow, to fix boundaries
3. Deep, or mix, at end



New Migration Plan

What % of paths are **D**-deliverable
at each step?



New Migration Plan

% of 3-deliverable paths

New Migration Plan

% of 3-deliverable paths

Benchmark	Deep or Shallow	Deep and Shallow
jpeg	100%	100%
suffixt	0%	12%
take5	100%	100%
sieve	0%	100%
fsmoo	0%	50%
dungeon	0%	67%

Better Together

How many configs do best with a mix?

Better Together

How many configs do best with a mix?

Benchmark	D+S ≥ D|S
fsm	37%
morsecode	25%
jpeg	37%
kcfa	55%
zombie	6%
zordoz	46%

Thesis Statement

Deep and Shallow types can interoperate.

preserving their formal properties

Programmers can use these types to:

- strengthen Shallow guarantees
- avoid unimportant Deep errors
- lower runtime costs

Thesis Statement

Deep and Shallow types can interoperate.

✓ preserving their formal properties

Programmers can use these types to:

- strengthen Shallow guarantees
- avoid unimportant Deep errors
- lower runtime costs

Thesis Statement

Deep and Shallow types can interoperate.

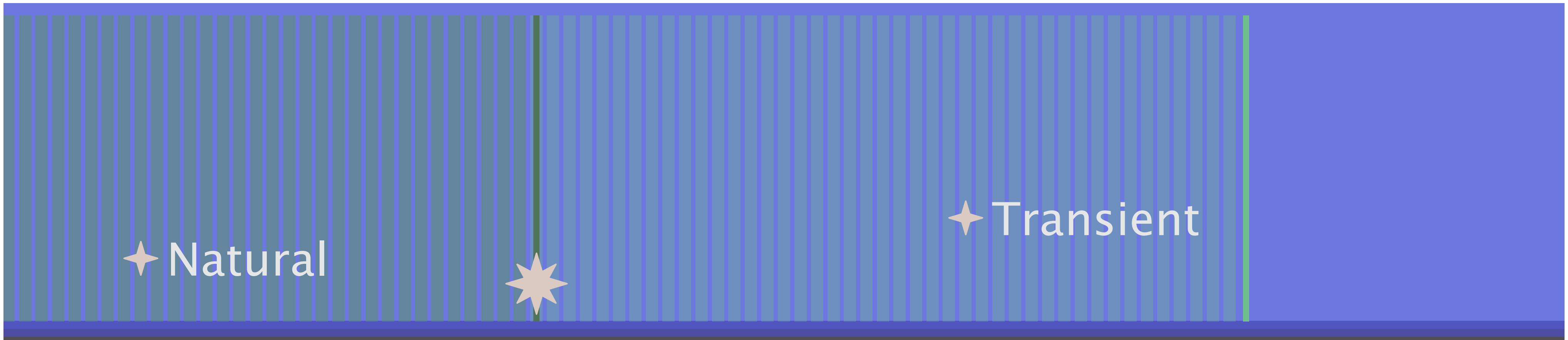
✓ preserving their formal properties

Programmers can use these types to:

✓ strengthen Shallow guarantees

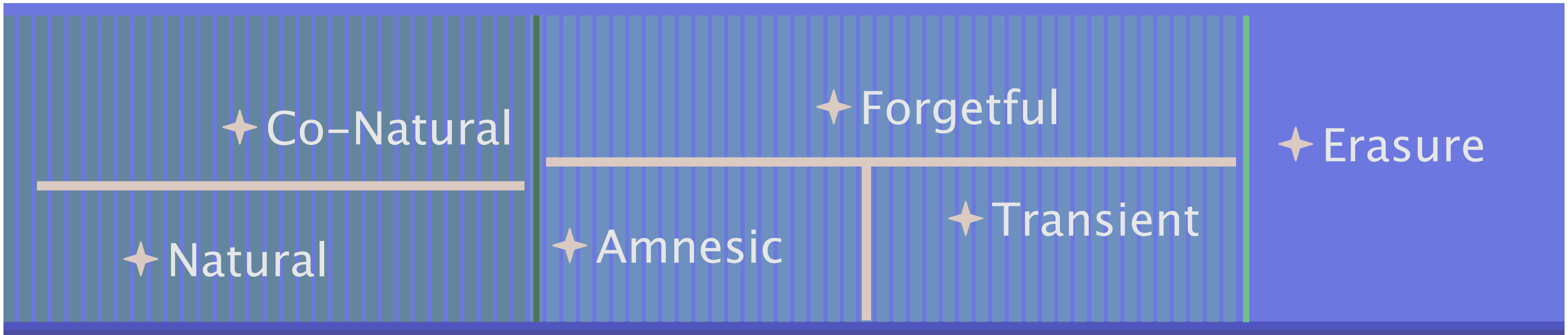
✓ avoid unimportant Deep errors

✓ lower runtime costs



Contributions

1. performance analysis method
2. design analysis method
3. scaled-up Transient
4. Deep + Shallow



	Natural	C	F	Transient	A	E
type soundness	✓	✓	✓	y	✓	✗
complete monitoring	✓	✓	✗	✗	✗	✗
blame soundness	✓	✓	✓	h	✓	0
blame completeness	✓	✓	✓	✗	✓	✗
error preorder	Natural <	C <	F <	Transient =	A <	E

Optimization

apply	box	✗ dead-code	extflonum
fixnum	float-complex	float	list
number	✗ pair	sequence	string
struct	vector		

Better Performance

Benchmark	Worst Deep	Worst Shallow
jpeg	23x	2x
suffixtree	31x	6x
take5	32x	3x
synth	49x	4x
quadU	60x	8x
sieve	10x	2x

Transient Blame Quite Bad!

Benchmark	Shallow Blame	Worst Deep
jpeg	46x	23x
suffixtree	>189x	31x
take5	51x	32x
synth	>1440x	49x
quadU	560x	60x
sieve	out of memory	10x

Shallow cannot run 1/2

problem: inst changes shape

```
#lang deep

(require/typed racket/base
 (cdr (All (A) A)))

(define fake-str : String
  (inst cdr String))

(string-length fake-str)
```

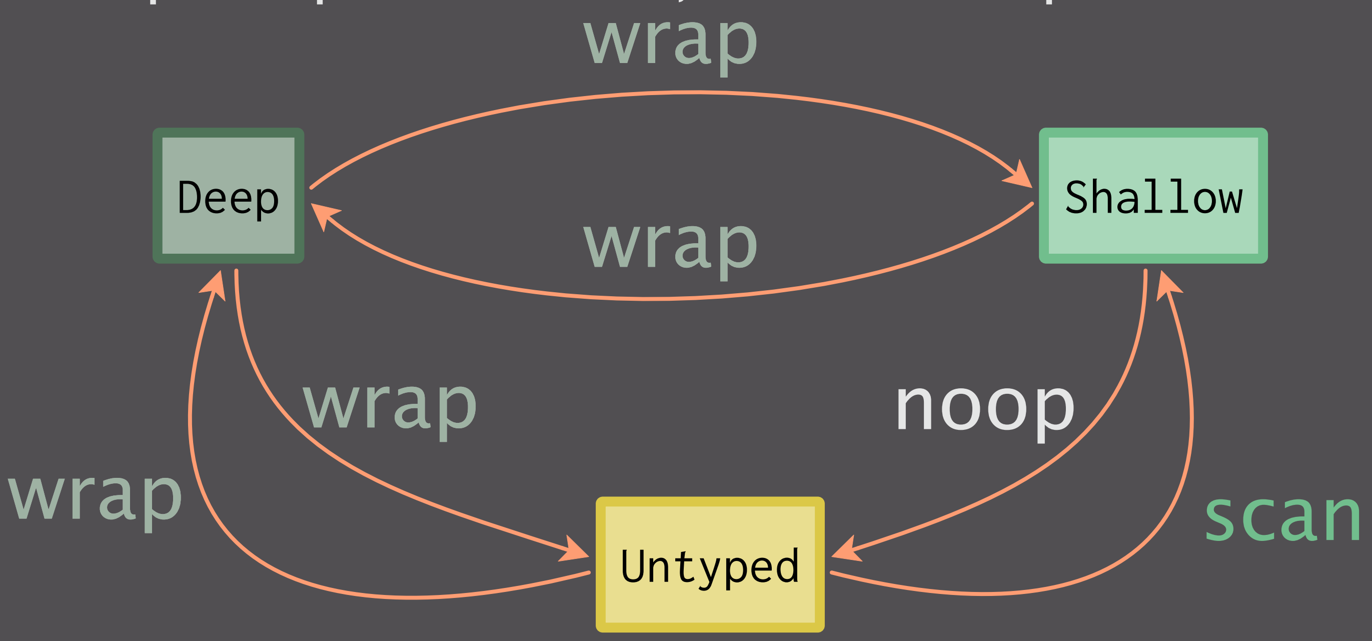
Shallow cannot run 2/2

problem: occurrence-type side effect

```
#lang deep

(require/typed racket/base
 (values (-> Any Any : String)))

(define x : Any 0)

(define fake-str : String
  (if (values x)
    x
    (error 'unreachable)))
```

Model Other Ideas

- conditionally weaken Deep -- Shallow, if escapes
- noop Deep -- Shallow, if S can wrap



Deep to Shallow = simpler behavior

```
#lang untyped
(index-of '(a b) 'a)
```

Untyped `0` Deep `#f` Shallow `0`

Deep to Shallow = simpler behavior

```
#lang deep

(: index-of
   (-> (Listof T) T (Maybe Num)))


(index-of '(a b) 'a)
```

Untyped `0` Deep `#f` Shallow `0`

Deep to Shallow = simpler behavior

```
#lang shallow

(: index-of
   (-> (Listof T) T (Maybe Num)))

(index-of '(a b) 'a)
```

Untyped	Deep	Shallow
`0`	`#f`	`0`

Deep to Shallow = simpler behavior

```
#lang shallow
(: index-of
   (-> (Listof T) T (Maybe Num)))
```

No wrappers = fewer surprises

Untyped `0` Deep `#f` Shallow `0`