A Transient Semantics for Typed Racket

Ben Greenman
Lukas Lazarek
Christos Dimoulas
Matthias Felleisen

2022-04-12

<Programming> 6.2

Context = Gradual Typing

High-level goal: mix typed and untyped code

Context = Gradual Typing

High-level goal: mix typed and untyped code

Typed Function

```
function add1(n : Num)
  n + 1
```

Untyped Caller

```
add1("hola")
```

Context = Gradual Typing

High-level goal: mix typed and untyped code

Typed Function

```
function add1(n : Num)
  n + 1
```

Untyped Caller

```
add1("hola")
```

Central question: what should types mean at run-time?

What Should Types Mean?

Three leading strategies:

What Should Types Mean?

Three leading strategies:

Guarded	**Transient**	**Optional**
Types enforce behaviors	Types enforce top-level shapes	Types enforce nothing

Example

Typed Function

```
function add1(n : Num)
  n + 1
```

Untyped Caller

```
add1("hola")
```

Example

Typed Function

```
function add1(n : Num)
  n + 1
```

Untyped Caller

```
add1("hola")
```

Guarded and **Transient**

```
Error: expected Num
```

Optional

```
"hola" + 1
```

Example 2

Untyped Array

```
arr = ["A", 3]
```

Typed Client

```
nums : Array(Num) = arr
nums[0]
```

Example 2

Guarded and **Transient** agree, but for different reasons ...

Untyped Array

```
arr = ["A", 3]
```

Typed Client

```
nums : Array(Num) = arr
nums[0]
```

Guarded and **Transient**

```
Error: expected Array(Num)
```

Optional

```
"A"
```

Example 2+

Guarded and **Transient** agree, but for different reasons ...

... and they disagree for an untyped client

Example 2+

Guarded and **Transient** agree, but for different reasons ...

... and they disagree for an untyped client

Untyped Array

```
arr = ["A", 3]
```

Typed Interface

```
nums : Array(Num) = arr
```

Unyped Client

```
nums[0]
```

Example 2+

Guarded and **Transient** agree, but for different reasons ...

... and they disagree for an untyped client

Untyped Array

```
arr = ["A", 3]
```

Typed Interface

```
nums : Array(Num) = arr
```

Unyped Client

```
nums[0]
```

Guarded

```
Error: expected Array(Num)
```

Transient and **Optional**

```
"A"
```

Guarded

Types enforce behaviors

Transient

Types enforce top-level shapes

Optional

Types enforce nothing

Typed Racket has **Guarded** types … and a big problem

Guarded	**Transient**	**Optional**
Types enforce behaviors	Types enforce top-level shapes	Types enforce nothing

Guarded Types are Expensive!

Guarded Types are Expensive!

Typed Racket

- **Strong types**: Type soundness + Complete monitoring
- **High overheads** are common on the GTP Benchmarks
- **Worst cases**: **25x**, **1400x**

Guarded Types are Expensive!

Typed Racket

- **Strong types**: Type soun
  + Complete monitoring
- **High overheads** are common on the GTP Benchmarks
- **Worst cases**: **25x**, **1400x**

Q. Is Sound Gradual Typing Dead?

Guarded gradual types are too slow

What to do?

Guarded gradual types are too slow

What to do?

1. **Improve the compiler**

   Collapsible Contracts `[OOPSLA'18]`

Guarded gradual types are too slow

What to do?

1. **Improve the compiler**

   Collapsible Contracts `[OOPSLA'18]`

2. **Remove checks statically**

   Corpse Reviver `[POPL'21]`

Guarded gradual types are too slow

What to do?

1. **Improve the compiler**

   Collapsible Contracts `[OOPSLA'18]`

2. **Remove checks statically**

   Corpse Reviver `[POPL'21]`

3. **Build a new compiler**

   Pycket `[OOPSLA'17]`

Guarded gradual types are too slow

What to do?

1. **Improve the compiler**

   Collapsible Contracts `[OOPSLA'18]`

2. **Remove checks statically**

   Corpse Reviver `[POPL'21]`

3. **Build a new compiler**

   Pycket `[OOPSLA'17]`

4. **Use weaker types**

   Today!

Hope to **reduce costs across the board**
without changing the surface language

- Same code, types, and type checker
- Different run-time behavior

4. **Use weaker types**

Today!

The Inspiration: Reticulated Python

Transient sematics ~ enforce types with tag checks

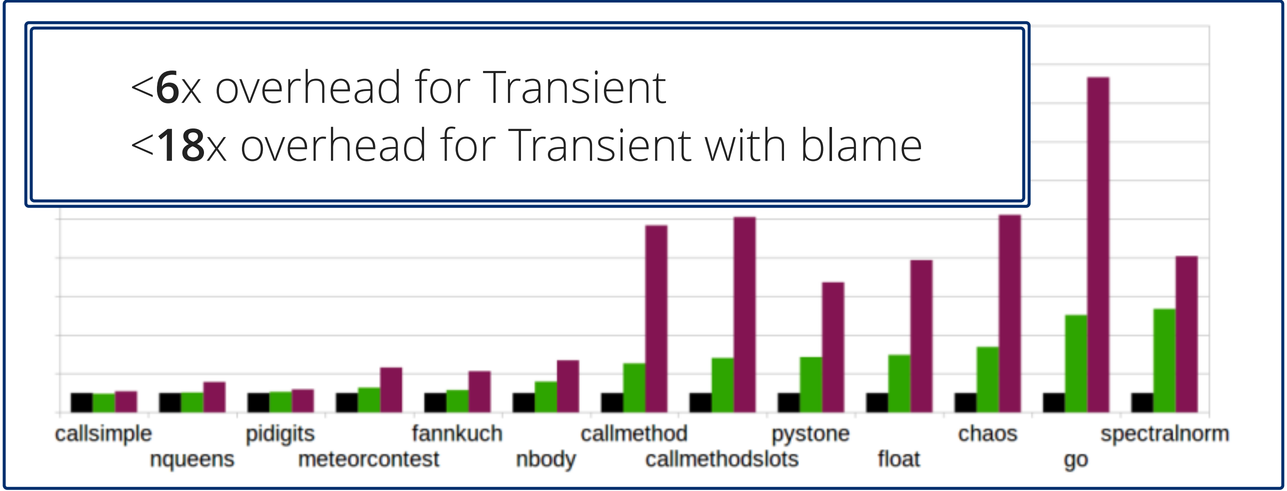
No contract wrappers

The Inspiration: Reticulated Python

Transient sematics ~ enforce types with tag checks

No contract wrappers

Performance is not bad! `[POPL'17]`



Research Questions

RQ0. How to add **transient types** to Typed Racket?

Research Questions

RQ0. How to add **transient types** to Typed Racket?

RQ1. Can Transient **scale to a rich type system**?

RQ2. Can we **adapt an existing complier** to do so?

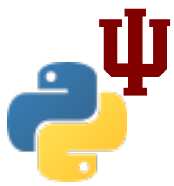
Research Questions

Implications for other gradual languages,
especially **Optional** ones that wish to **strengthen** their types

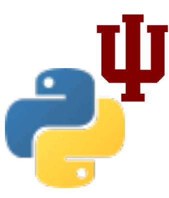
RQ1. Can Transient **scale to a rich type system**?

RQ2. Can we **adapt an existing complier** to do so?

Two Type Systems



Two Type Systems



```
T := Dynamic
     Int
     Ref T
     T -> T
     Class { T ... }
     .... a few more
```

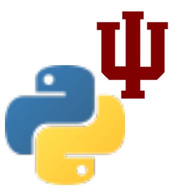
```
T :=
```

Two Type Systems

```
T := Dynamic
     Int
     Ref T
     T -> T
     Class { T ... }
     .... a few more
```

T := **Any (the top type)**

Two Type Systems



```
T := Dynamic
     Int
     Ref T
     T -> T
     Class { T ... }
     .... a few more
```

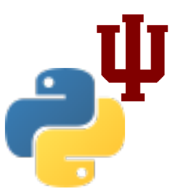
```
T := Any (the top type)
     Integer
     Natural
```

Two Type Systems

```
T := Dynamic
     Int
     Ref T
     T -> T
     Class { T ... }
     .... a few more
```

```
T := Any (the top type)
     Integer
     Natural
     (Vectorof T)
     (Vector T ...)
```

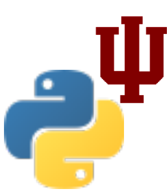
Two Type Systems



```
T := Dynamic
     Int
     Ref T
     T -> T
     Class { T ... }
     .... a few more
```

```
T := Any (the top type)
     Integer
     Natural
     (Vectorof T)
     (Vector T ...)
     (-> T ... (Values T ...))
```

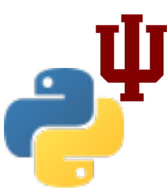
Two Type Systems



```
T := Dynamic
     Int
     Ref T
     T -> T
     Class { T ... }
     .... a few more
```

```
T := Any (the top type)
     Integer
     Natural
     (Vectorof T)
     (Vector T ...)
     (-> T ... (Values T ...))
     (Class T ...)
```

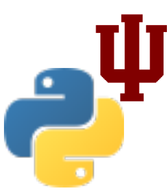
Two Type Systems



```
T := Dynamic
     Int
     Ref T
     T -> T
     Class { T ... }
     .... a few more
```

```
T := Any (the top type)
     Integer
     Natural
     (Vectorof T)
     (Vector T ...)
     (-> T ... (Values T ...))
     (Class T ...)
     (All X T)
     (Union T ...)
     (Rec X T)
```

Two Type Systems



```
T := Dynamic
     Int
     Ref T
     T -> T
     Class { T ... }
     .... a few more
```

```
T := Any (the top type)
     Integer
     Natural
     (Vectorof T)
     (Vector T ...)
     (-> T ... (Values T ...))
     (Class T ...)
     (All X T)
     (Union T ...)
     (Rec X T)
     .... many more
```

Two Compilers

Two Compilers

Typecheck + Elaborate

Python

Typecheck + Elaborate

Python

Expand

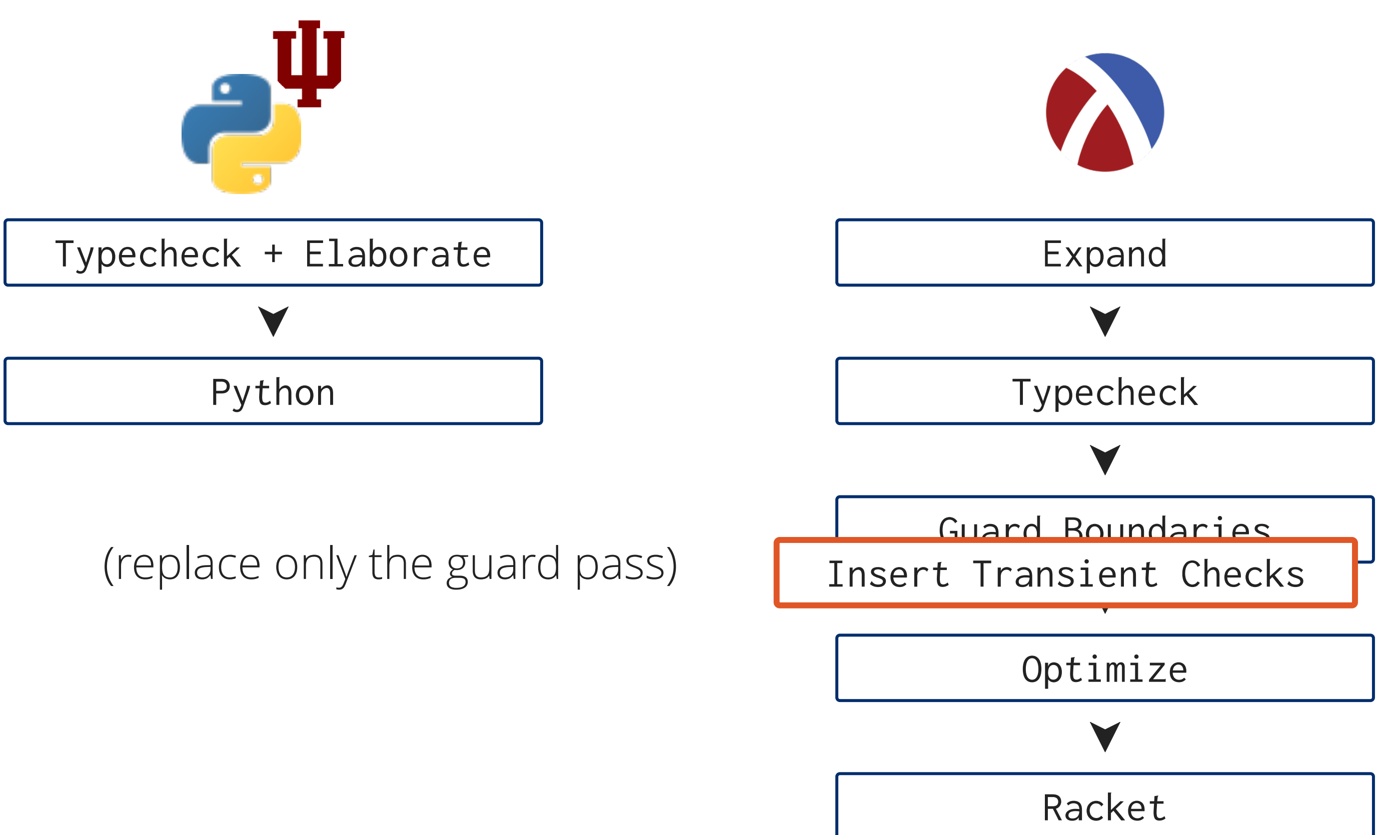
Typecheck

Guard Boundaries

Optimize

Racket

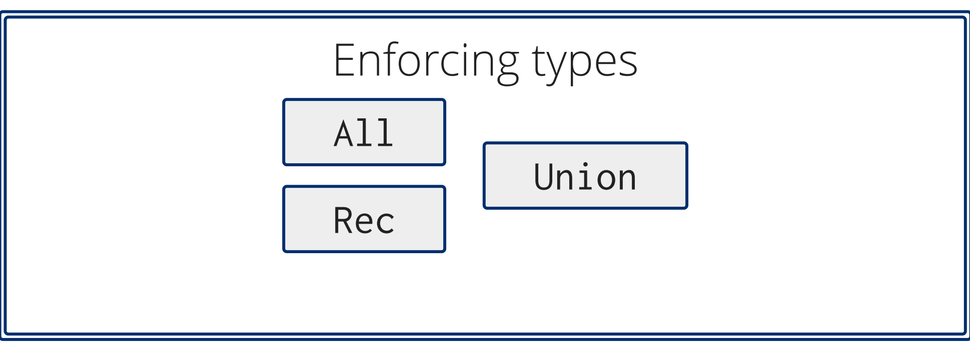
Two Compilers



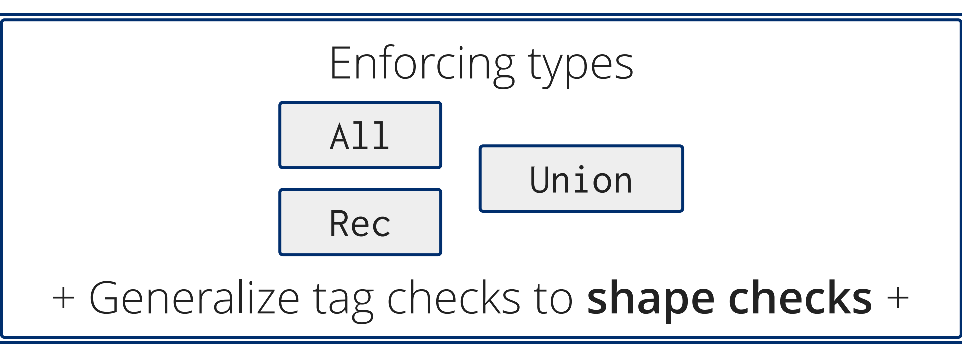
(replace only the guard pass)

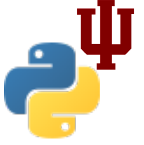
Challenges

Challenges

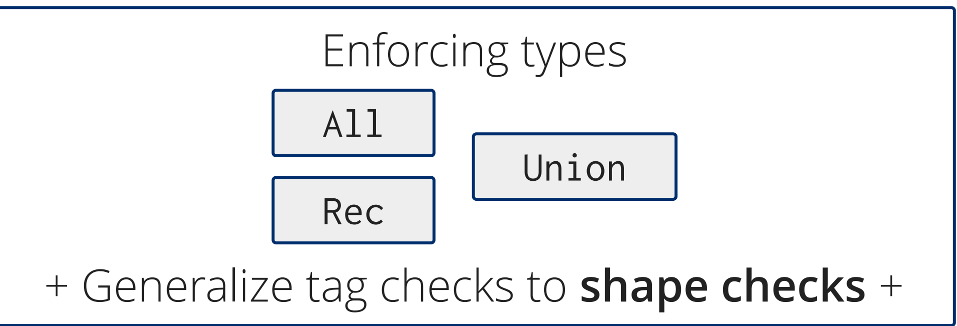


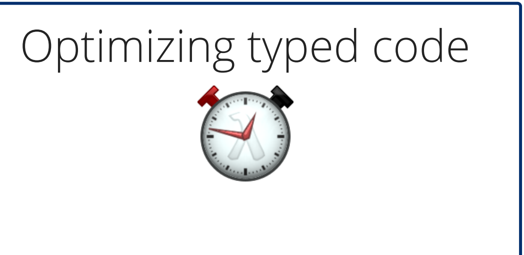
Challenges



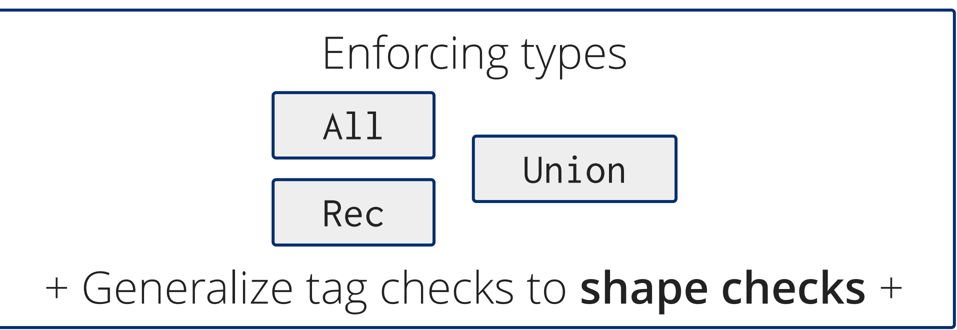


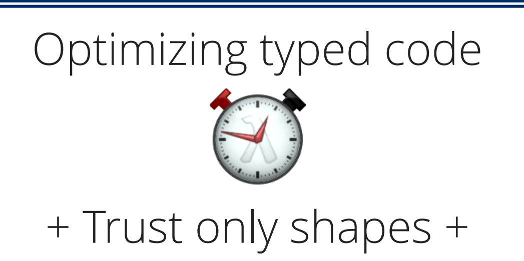
Challenges





Challenges





Challenges

Enforcing types

`All` `Rec` `Union`

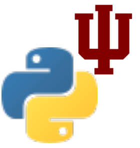
+ Generalize tag checks to **shape checks** +

Navigating expanded code

Optimizing typed code

+ Trust only shapes +

Challenges



Enforcing types

`All` `Rec` `Union`

+ Generalize tag checks to **shape checks** +

Navigating expanded code

+ Typechecker must leave **evidence** +

Optimizing typed code

+ Trust only shapes +

Challenges

Enforcing types

`All` `Rec` `Union`

+ Generalize tag checks to **shape checks** +

Navigating expanded code

+ Typechecker must leave **evidence** +

Optimizing typed code

+ Trust only shapes +

Minimizing costs

Challenges

Enforcing types

`All` `Rec` `Union`

+ Generalize tag checks to **shape checks** +

Navigating expanded code

+ Typechecker must leave **evidence** +

Optimizing typed code

+ Trust only shapes +

Minimizing costs

+ E.g. reduce codegen +

Challenges

Enforcing types

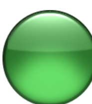
`All` `Union` `Rec`

+ Generalize tag checks to **shape checks** +

Navigating expanded code

+ Typechecker must leave **evidence** +

Minimizing costs



+ E.g. reduce codegen +

Optimizing typed code

+ Trust only shapes +

Navigating expanded code

+ Typechecker must leave **evidence** +

```
(for/sum ([byte (open-input-file "my.txt")])
  byte)
```

Navigating expanded code

+ Typechecker must leave **evidence** +

```
(for/sum ([byte (open-input-file "my.txt")])
  byte)
```

```
(define seq (make-seq (open-input-file "my.txt")))
(define (for-loop result pos)
  (if (not (seq.use-pos? pos))
    result
    (let ([byte (seq.get-val pos)])
      (for-loop (if (or (not seq.use-val?)
                        (seq.use-val? byte))
                  (+ result byte)
                  result)
                (seq.next-pos pos)))))
(for-loop 0 seq.init)
```

Navigating expanded code

+ Typechecker must leave **evidence** +

```
(for/sum ([byte (open-input-file "my.txt")])
  byte)
```

```
(define seq (make-seq (open-input-file "my.txt")))
(define (for-loop result pos)
  (if (not (seq.use-p
    result
    (let ([byte (seq.
      (for-loop (if (or (not seq.use-val?)
                         (seq.use-val? byte))
                     (+ result byte)
                     result)
                 (seq.next-pos pos)))))
(for-loop 0 seq.init)
```

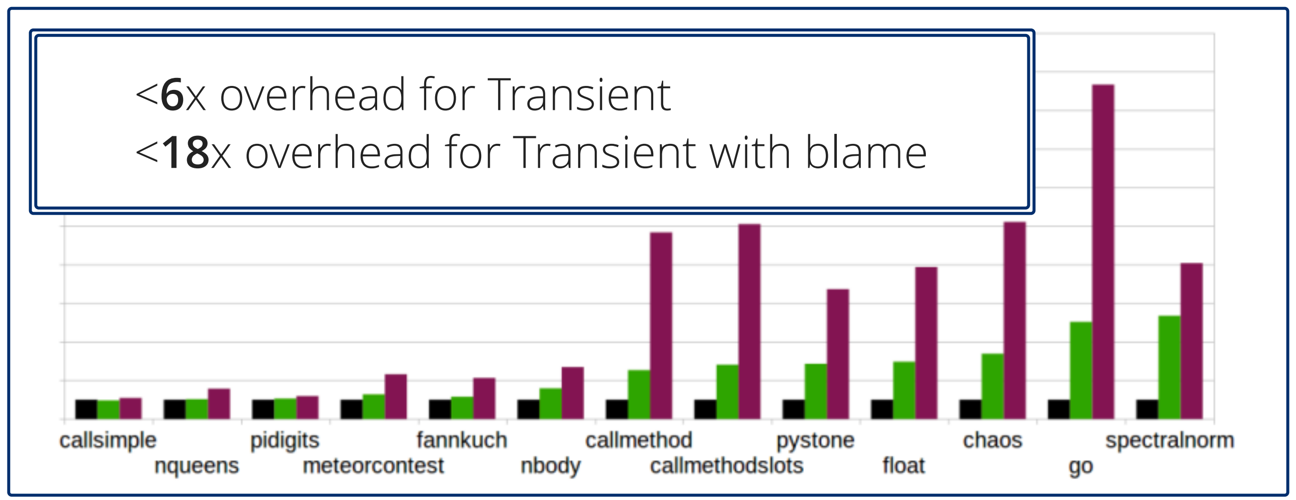
Navigating expanded code

+ Typechecker must leave **evidence** +

Don't want to check **every** function call!

How's performance?

Does it match the worst cases for Reticulated?



Worst Case Overhead vs. Untyped

Worst Case Overhead vs. Untyped

	Transient
kcfa	1x
morsecode	3x
sieve	4x
snake	8x
suffixtree	6x
tetris	10x
acquire	1x
dungeon	5x
forth	6x
fsm	2x
fsmoo	4x

	Transient
gregor	2x
jpeg	2x
lnm	1x
mbta	2x
quadT	7x
quadU	8x
synth	4x
take5	3x
zombie	31x
zordoz	3x

Worst Case Overhead vs. Untyped

	Transient	T+Blame
kcfa	1x	>540x
morsecode	3x	>250x
sieve	4x	>220x
snake	8x	>1000x
suffixtree	6x	>190x
tetris	10x	>720x
acquire	1x	34x
dungeon	5x	75x
forth	6x	48x
fsm	2x	230x
fsmoo	4x	100x

	Transient	T+Blame
gregor	2x	23x
jpeg	2x	38x
lnm	1x	29x
mbta	2x	37x
quadT	7x	34x
quadU	8x	320x
synth	4x	220x
take5	3x	33x
zombie	31x	560x
zordoz	3x	220x

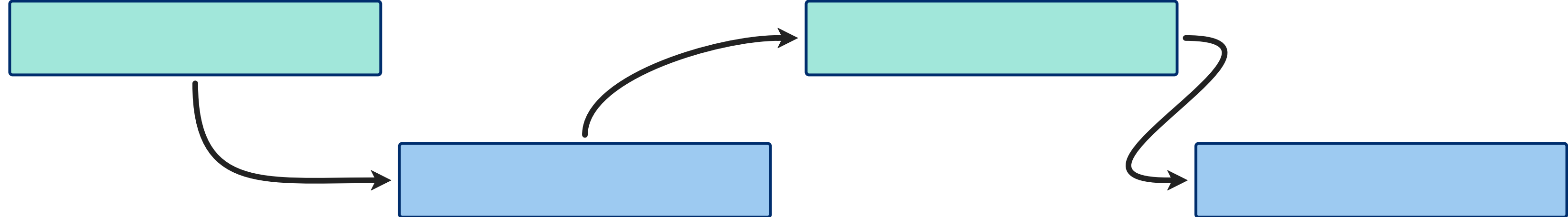
Worst Case Overhead vs. Untyped

	Transient	T+Blame		Transient	T+Blame
kcfa	1x	>540x	gregor	2x	23x
morsecode	3x	>250x	jpeg	2x	38x
sieve	4x	>2		1x	29x
snake	8x	>10		2x	37x
suffixtree	6x	>1		7x	34x
tetris	10x	>7		8x	320x
acquire	1x			4x	220x
dungeon	5x	75x	take5	3x	33x
forth	6x	48x	zombie	31x	560x
fsm	2x	230x	zordoz	3x	220x
fsmoo	4x	100x			

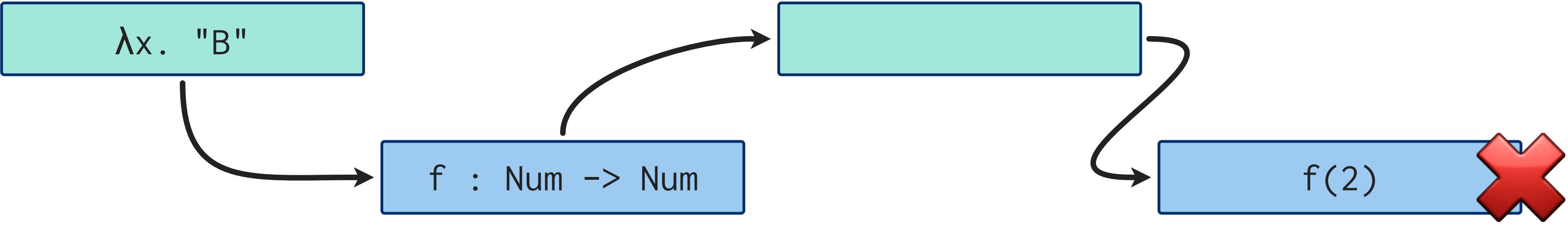
Transient alone is **not so bad**

T+Blame gets **expensive**

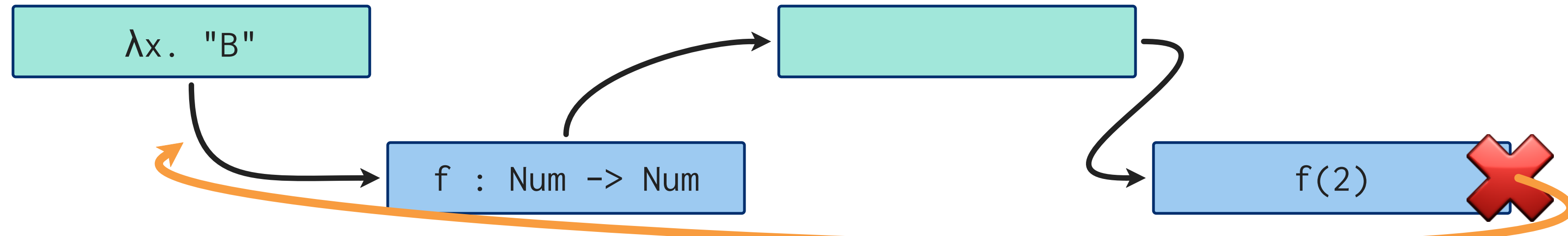
Blame: The Idea



Blame: The Idea

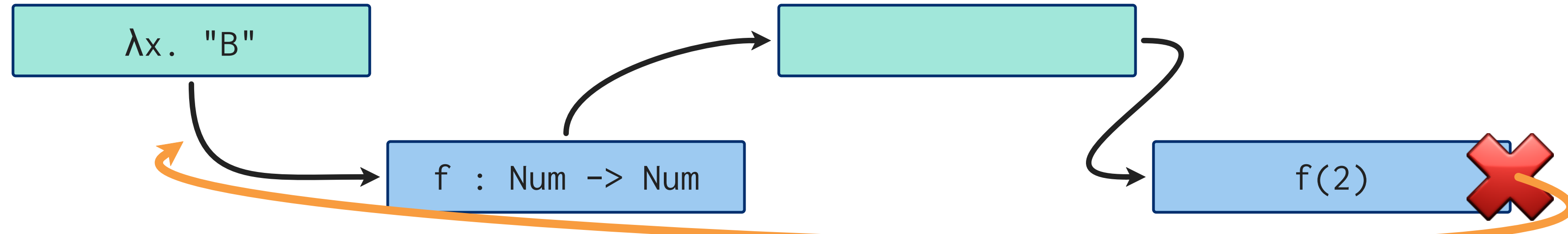


Blame: The Idea



When a typed/untyped interaction goes wrong,
blame shows where to start debugging

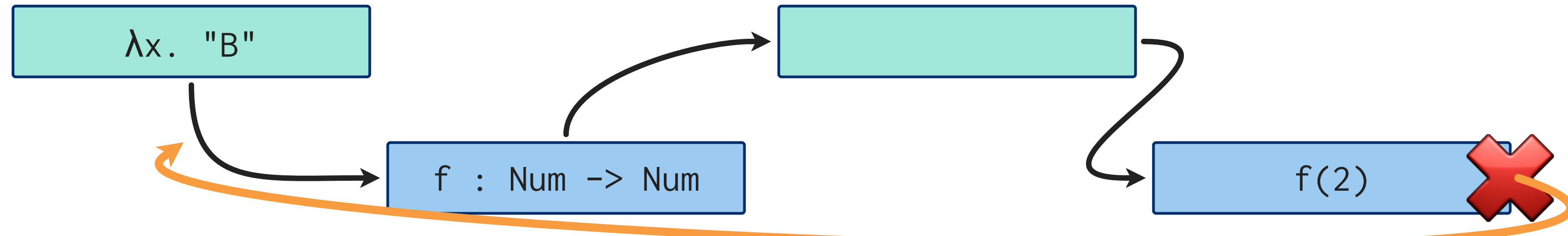
Blame: The Idea



When a typed/untyped interaction goes wrong,
blame shows where to start debugging

Guarded wrappers can attach precise blame info to values

Blame: The Idea

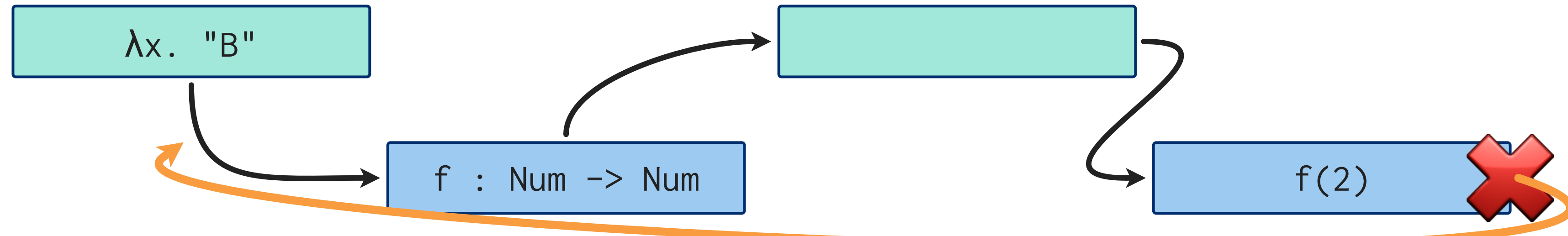


When a typed/untyped interaction goes wrong,
blame shows where to start debugging

Guarded wrappers can attach precise blame info to values

Transient has no wrappers, but keeps a **global map on the side**

Blame: The Idea



When a typed/untyped interaction goes wrong,
blame shows where to start debugging

Guarded wrappers can attach precise blame info to values

Transient has no wrappers, but keeps a **global map on the side**

... a large map gets expensive

Worst Case Overhead vs. Untyped

	Transient	T+Blame
kcfa	1x	>540x
morsecode	3x	>250x
sieve	4x	>220x
snake	8x	>1000x
suffixtree	6x	>190x
tetris	10x	>720x
acquire	1x	34x
dungeon	5x	75x
forth	6x	48x
fsm	2x	230x
fsmoo	4x	100x

	Transient	T+Blame
gregor	2x	23x
jpeg	2x	38x
lnm	1x	29x
mbta	2x	37x
quadT	7x	34x
quadU	8x	320x
synth	4x	220x
take5	3x	33x
zombie	31x	560x
zordoz	3x	220x

Worst Case Overhead vs. Untyped

	Transient	T+Blame		Transient	T+Blame
kcfa	1x	>540x	gregor	2x	23x
morsecode	3x	>250x	jpeg	2x	38x
sieve					29x
snake					37x
suffixtree					34x
tetris	1				320x
acquire					220x
dungeon	5x	75x	take5	3x	33x
forth	6x	48x	zombie	31x	560x
fsm	2x	230x	zordoz	3x	220x
fsmoo	4x	100x			

Why is T+Blame so much worse than Reticulated?

1. Larger, longer-running benchmarks
2. No dynamic type

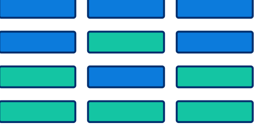
Future: can run-time support reduce the cost?

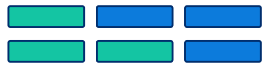
Overall Performance

Overall Performance

Gradual types should support **all** mixed-typed configurations

N components => 2^N configurations

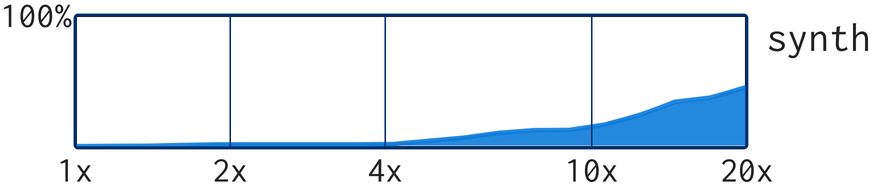




Overall Performance

Gradual types should support **all** mixed-typed configurations

N components => 2^N configurations

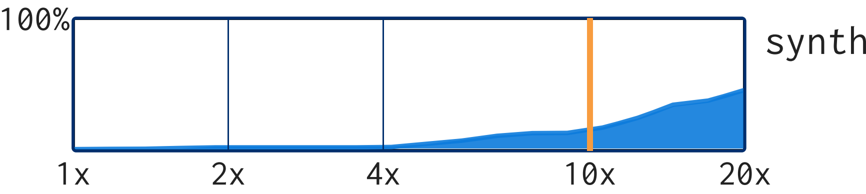


Overall Performance

Gradual types should support **all** mixed-typed configurations

N components => 2^N configurations

✗ ✓ ✗
✗ ✗ ✗
✗ ✓

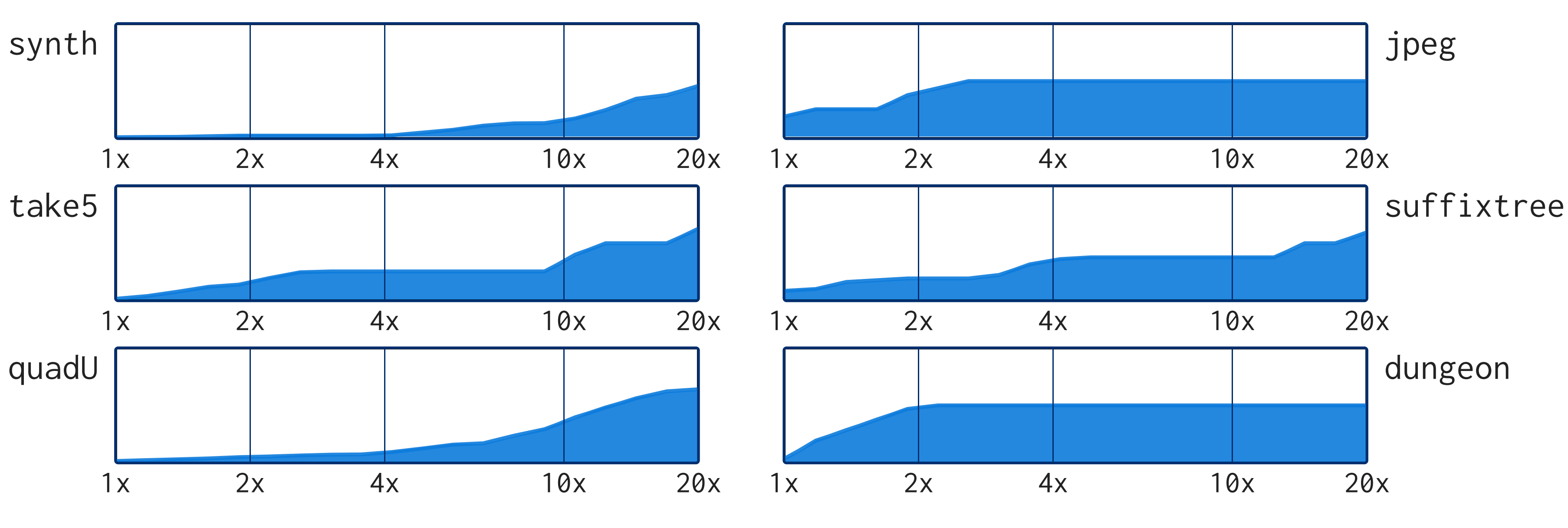


At x=10, count the % of configurations
that run at most 10x slower than untyped

Overall Performance

Overall Performance

Guarded: % of fast-enough points

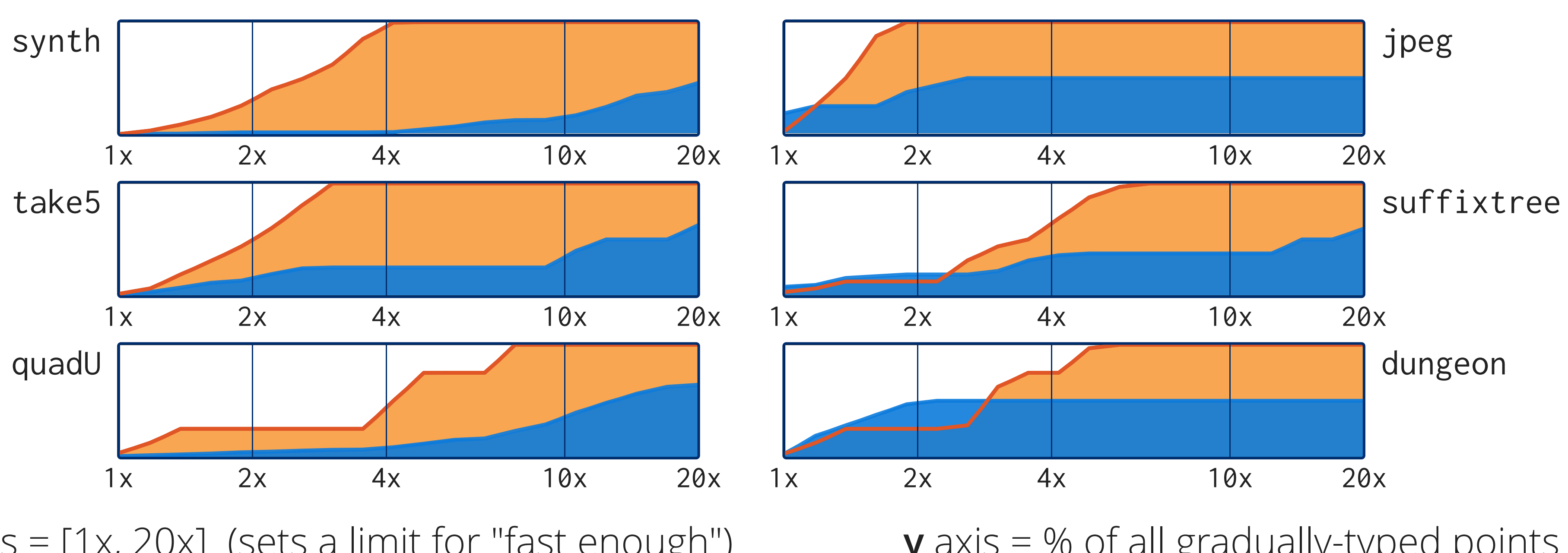


x axis = [1x, 20x] (sets a limit for "fast enough")

y axis = % of all gradually-typed points

Overall Performance

Guarded vs Transient: % of fast-enough points

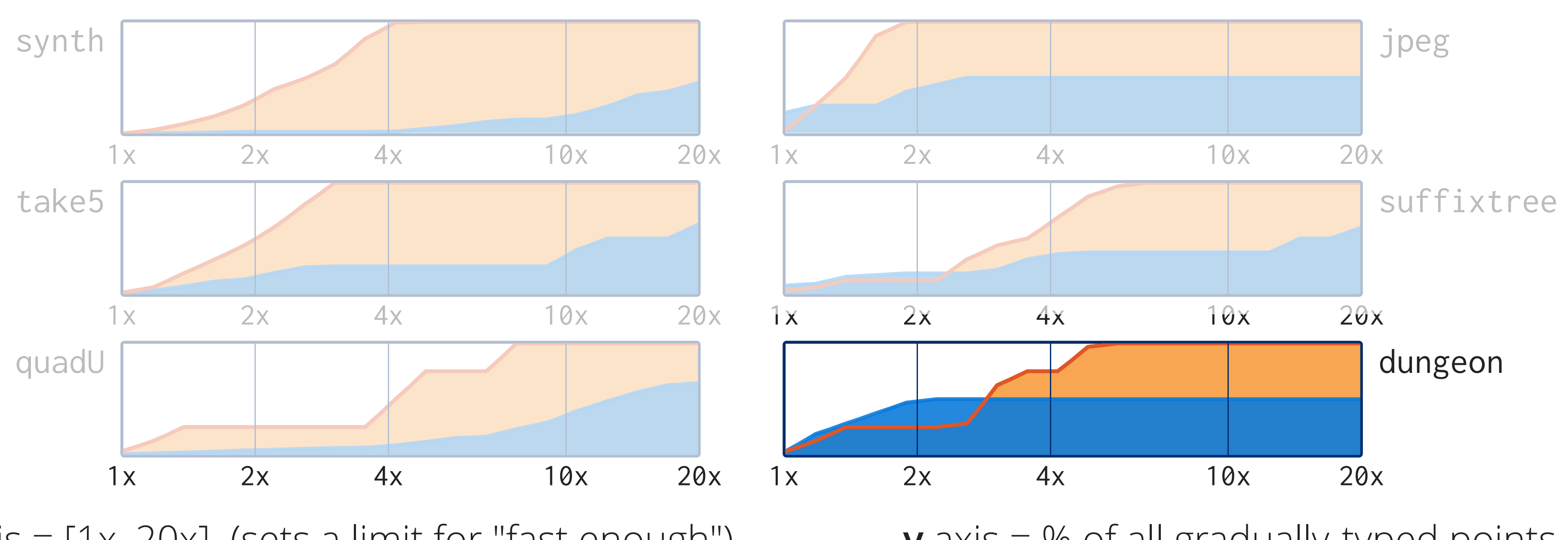


x axis = [1x, 20x] (sets a limit for "fast enough")

y axis = % of all gradually-typed points

Overall Performance

Guarded vs **Transient**: % of fast-enough points

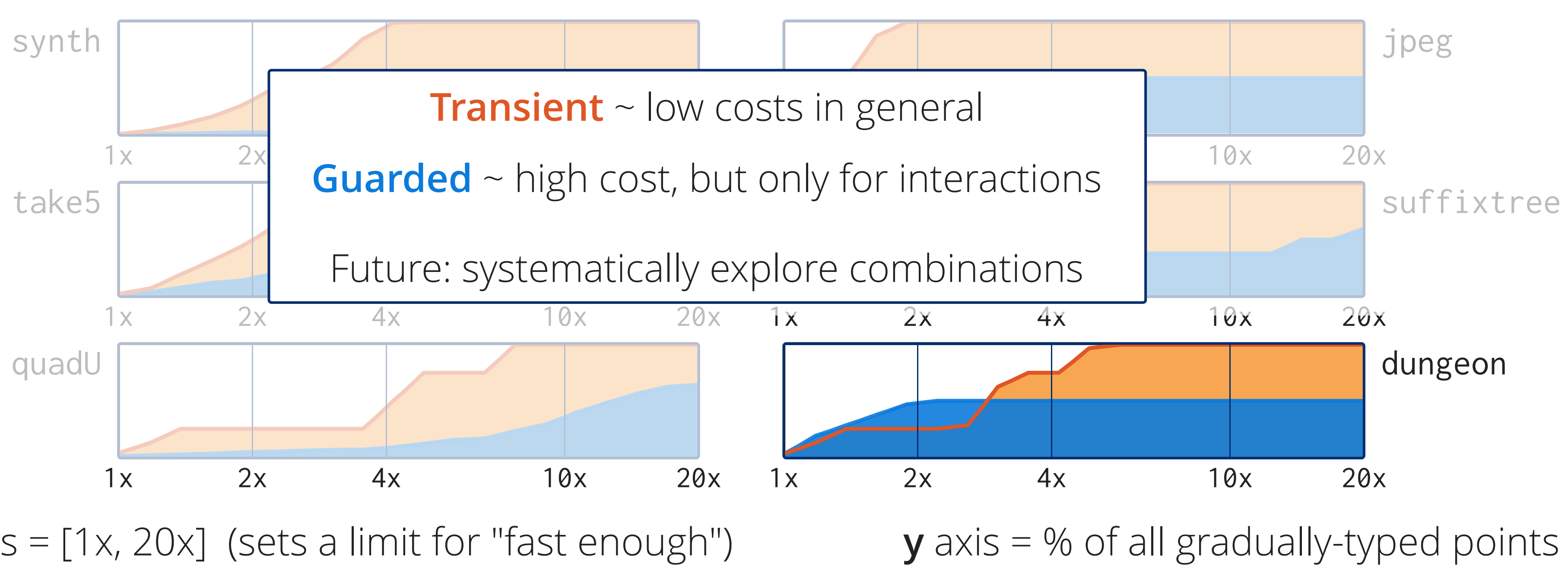


x axis = [1x, 20x] (sets a limit for "fast enough")

y axis = % of all gradually-typed points

Overall Performance

Guarded vs **Transient**: % of fast-enough points



x axis = [1x, 20x] (sets a limit for "fast enough")

y axis = % of all gradually-typed points

In Conclusion

RQ. Can **transient types**:
- scale to a rich type system
- in the context of an existing compiler?

In Conclusion

RQ. Can **transient types**:
- scale to a rich type system
- in the context of an existing compiler?

Yes! ... without blame

In Conclusion

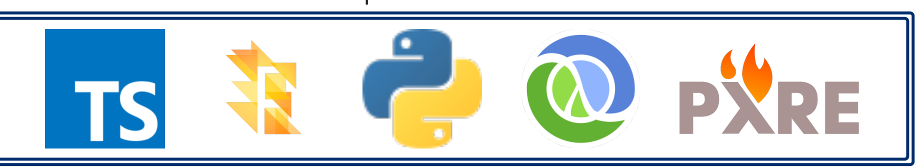
RQ. Can **transient types**:
- scale to a rich type system
- in the context of an existing compiler?

Yes! ... without blame

... and with some tailoring

Reminder: **Transient** is a promising way to **strengthen** unsound **Optional** types

Lots of potential clients!



The End

Worst Case Overhead vs. Untyped

	Transient	T+Blame	Guarded
kcfa	1x	>540x	4x
morsecode	3x	>250x	2x
sieve	4x	>220x	15x
snake	8x	>1000x	12x
suffixtree	6x	>190x	31x
tetris	10x	>720x	12x
acquire	1x	34x	4x
dungeon	5x	75x	15000x
forth	6x	48x	5800x
fsm	2x	230x	2x
fsmoo	4x	100x	420x

	Transient	T+Blame	Guarded
gregor	2x	23x	2x
jpeg	2x	38x	23x
lnm	1x	29x	1x
mbta	2x	37x	2x
quadT	7x	34x	25x
quadU	8x	320x	55x
synth	4x	220x	47x
take5	3x	33x	44x
zombie	31x	560x	46x
zordoz	3x	220x	3x

Optimizations

Topic	Ok for Transient?	Topic	Ok?
apply	y	list	y
box	y	number	y
dead-code	N	pair	N
extflonum	y	sequence	y
fixnum	y	string	y
float-complex	y	struct	y
float	y	vector	y

https://prl.ccs.neu.edu/blog/2020/01/15/the-typed-racket-optimizer-vs-transient

Example: Retic. and Dyn

Most of the local variables get the Dynamic type and skip blame-map updates

```
def permutations(iterable:List(int))->List(List(int)):
    pool = tuple(iterable)
    n = len(pool)
    r = n
    indices = list(range(n))
    cycles = list(range(n-r+1, n+1))[::-1]
    result = [ [pool[i] for i in indices[:r]] ]
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                ....
```

No Wrappers = Simpler

```
(define b : (Boxof Char)
  (box #\X))

(define any : Any b)
```

```
(set-box! any #\Y)
```

Guarded `Error`

Transient `OK`

Limitation

Neither **Guarded** nor **Transient** TR allows occurrence types at a boundary

```
(require/typed racket/function
  (identity (-> Any Boolean : String)))
;; ^ Not permitted!

(define x : Any 0)

(define fake-str : String
  (if (identity x)
      (ann x String)
      (error 'unreachable)))

(string-length fake-str)
```