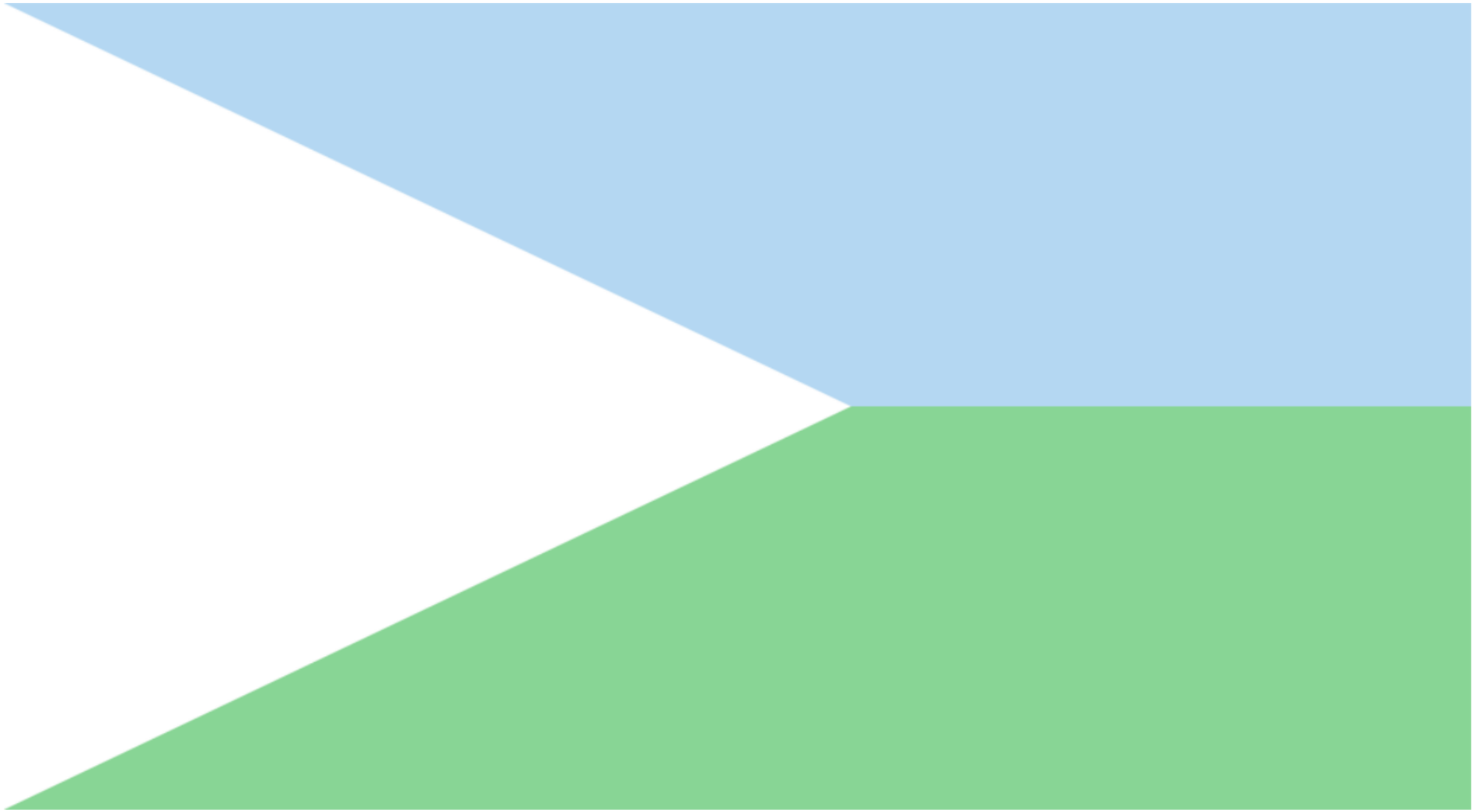
The Typed Racket Optimizer vs. Transient

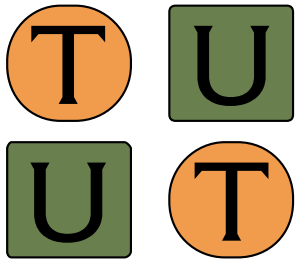
Ben Greenman

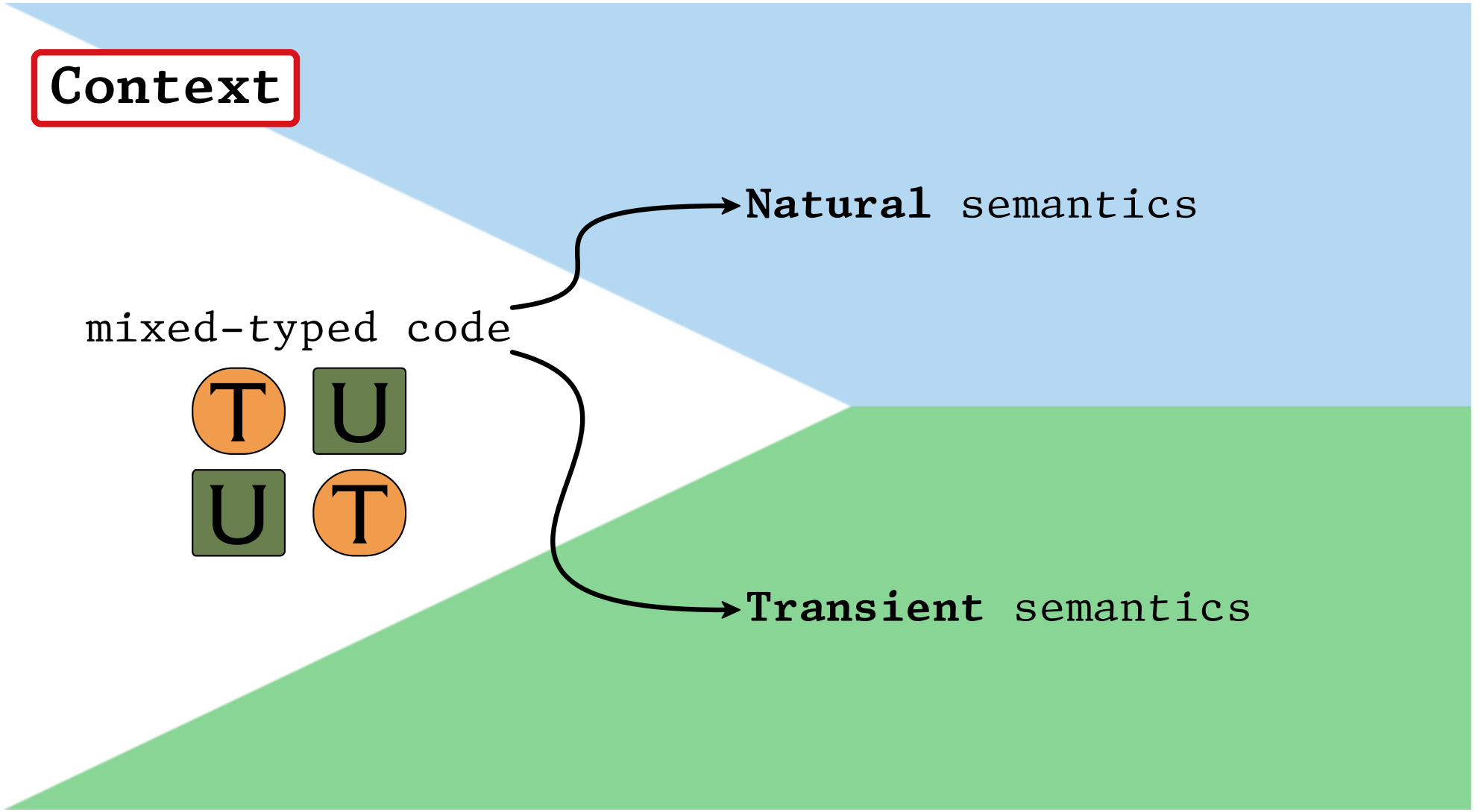
2019-11-11

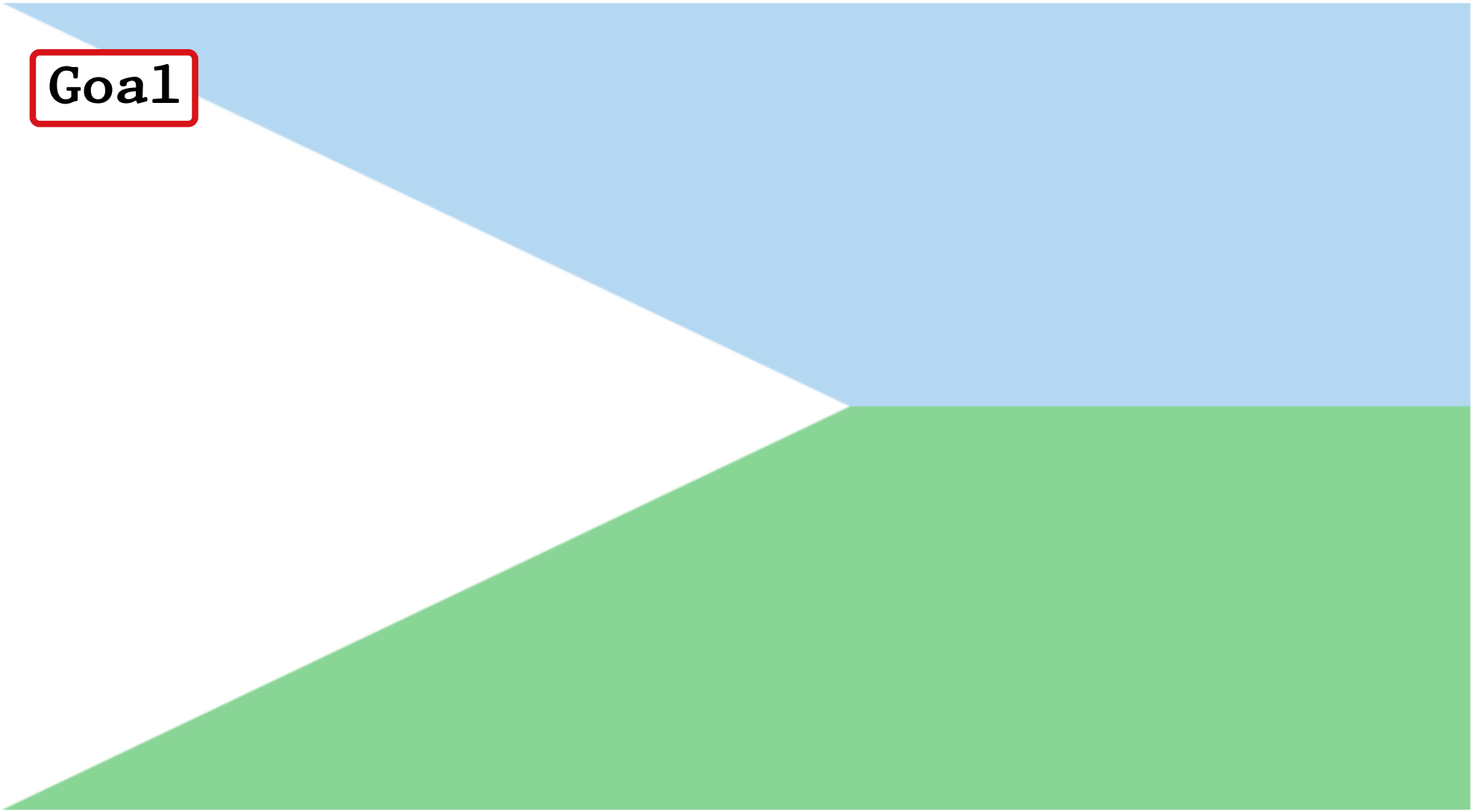


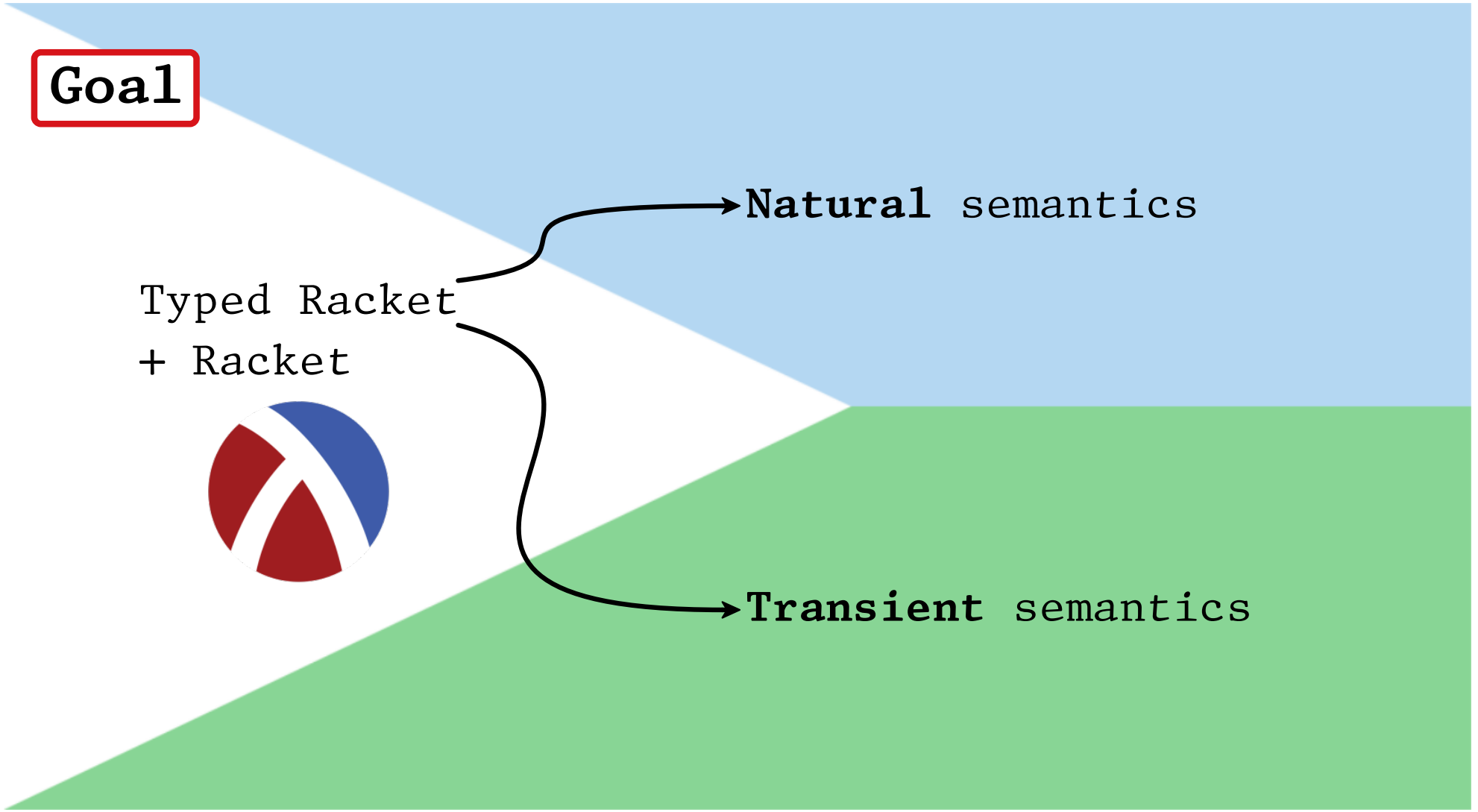
Context

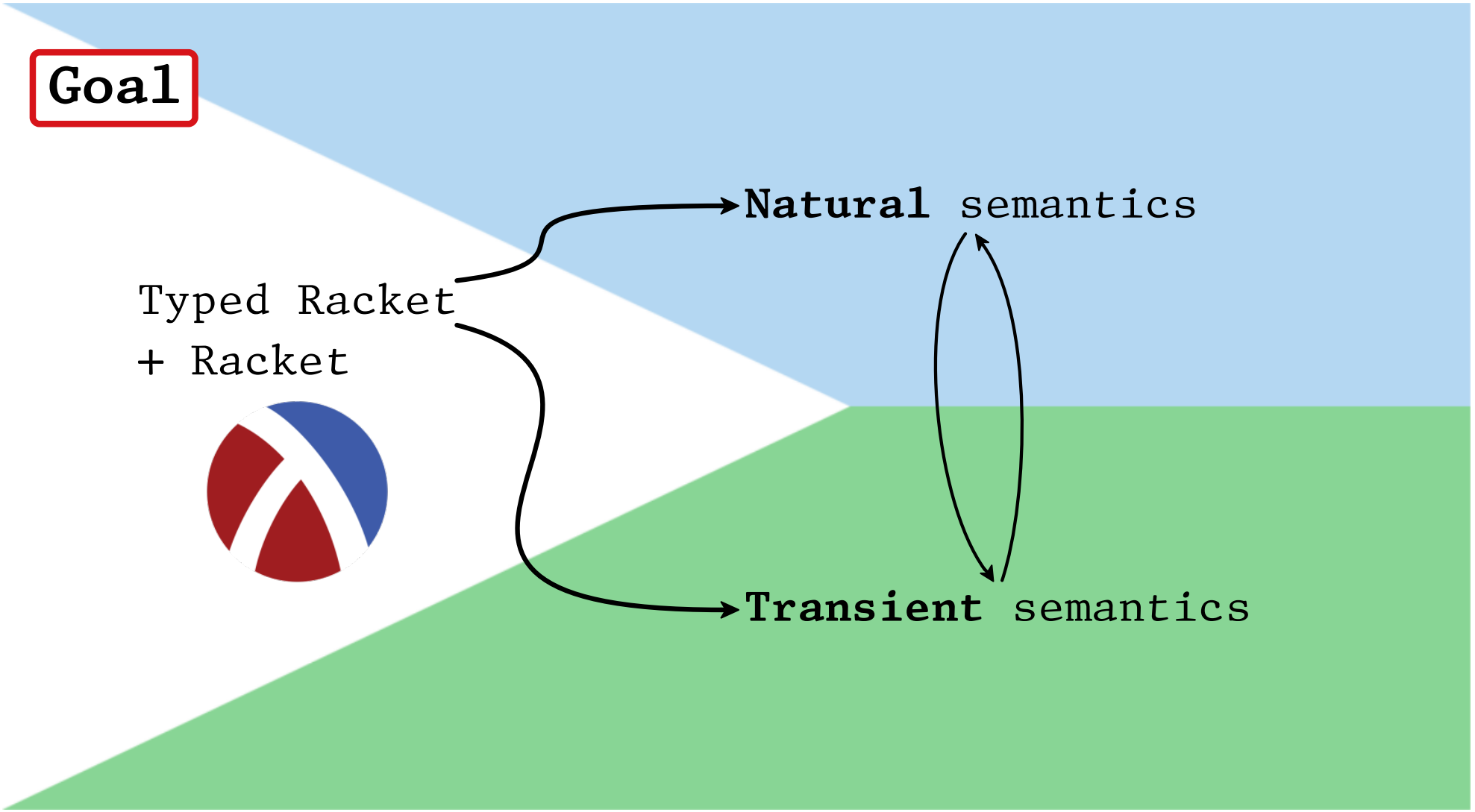
mixed-typed code




Context
Natural semantics
mixed-typed code
T
U
U
T
Transient semantics


Goal


Goal
Natural semantics
Typed Racket
+ Racket
Transient semantics

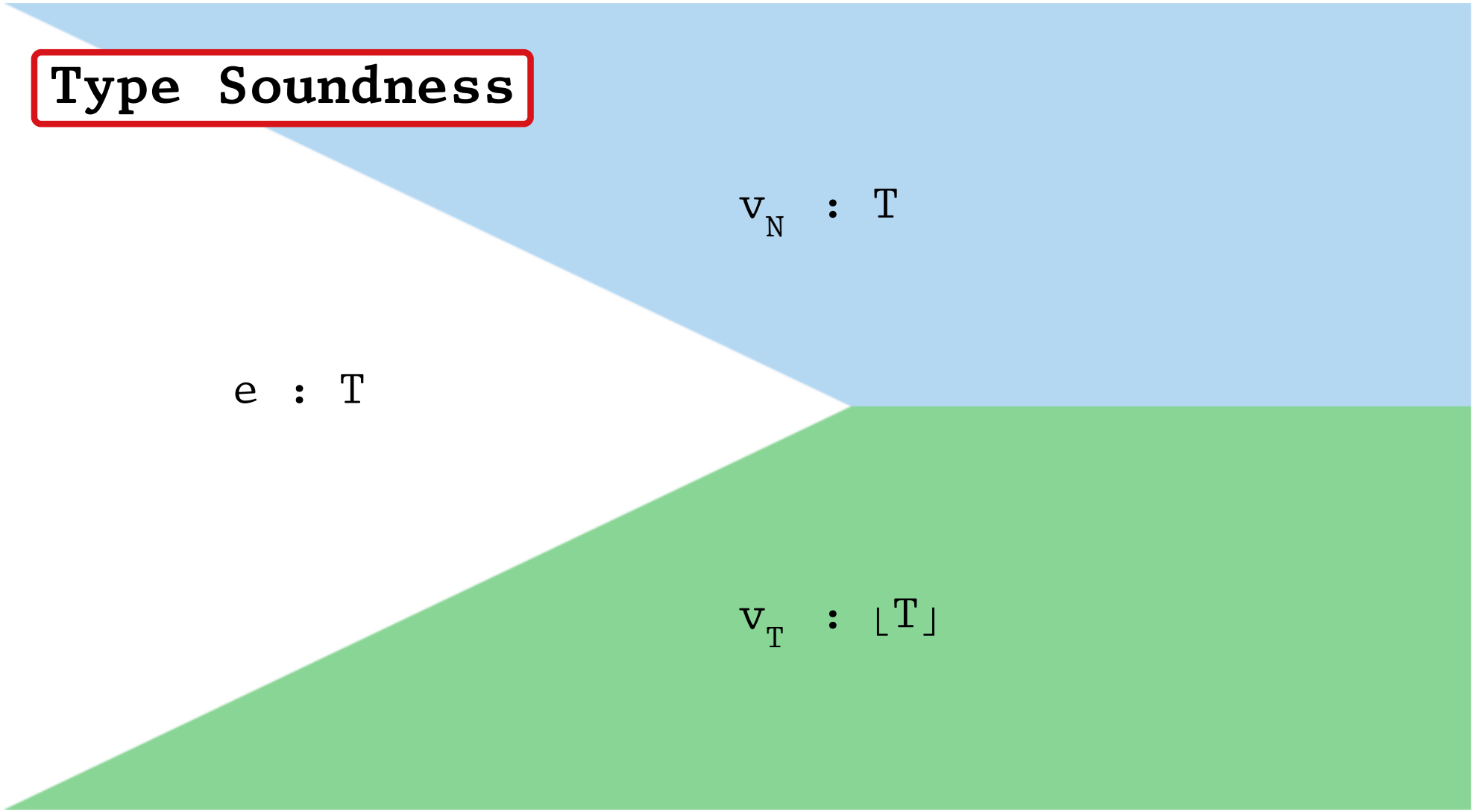

Goal
Natural semantics
Typed Racket
+ Racket
Transient semantics

Natural

- strong guarantees
- high runtime overhead

Transient

- weak guarantees
- lower overhead


Type Soundness
$v_N : T$
$e : T$
$v_T : \lfloor T \rfloor$

Example: Int

e : Int

v_T : Int

Example: Listof Str

e : Listof Str

v_T : List

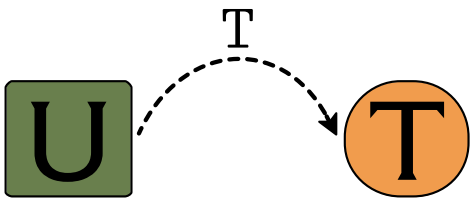
Example: Str -> Str

```
e : Str -> Str
```

v_T : Function

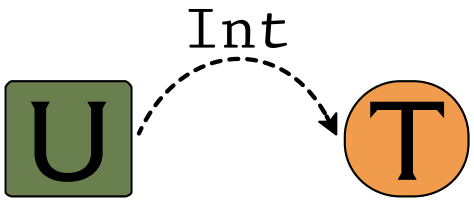
Runtime Checks

check ....



check ....

Example: Int

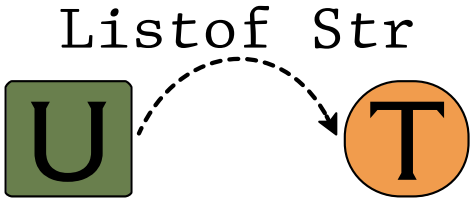



```
check Int
```

```
check Int
```

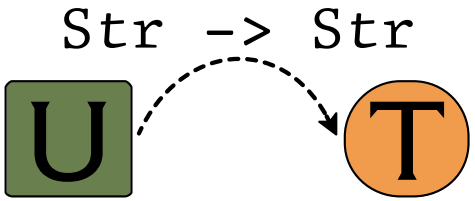
Example: Listof Str

check list
& all elems Str



check list
& protect reads

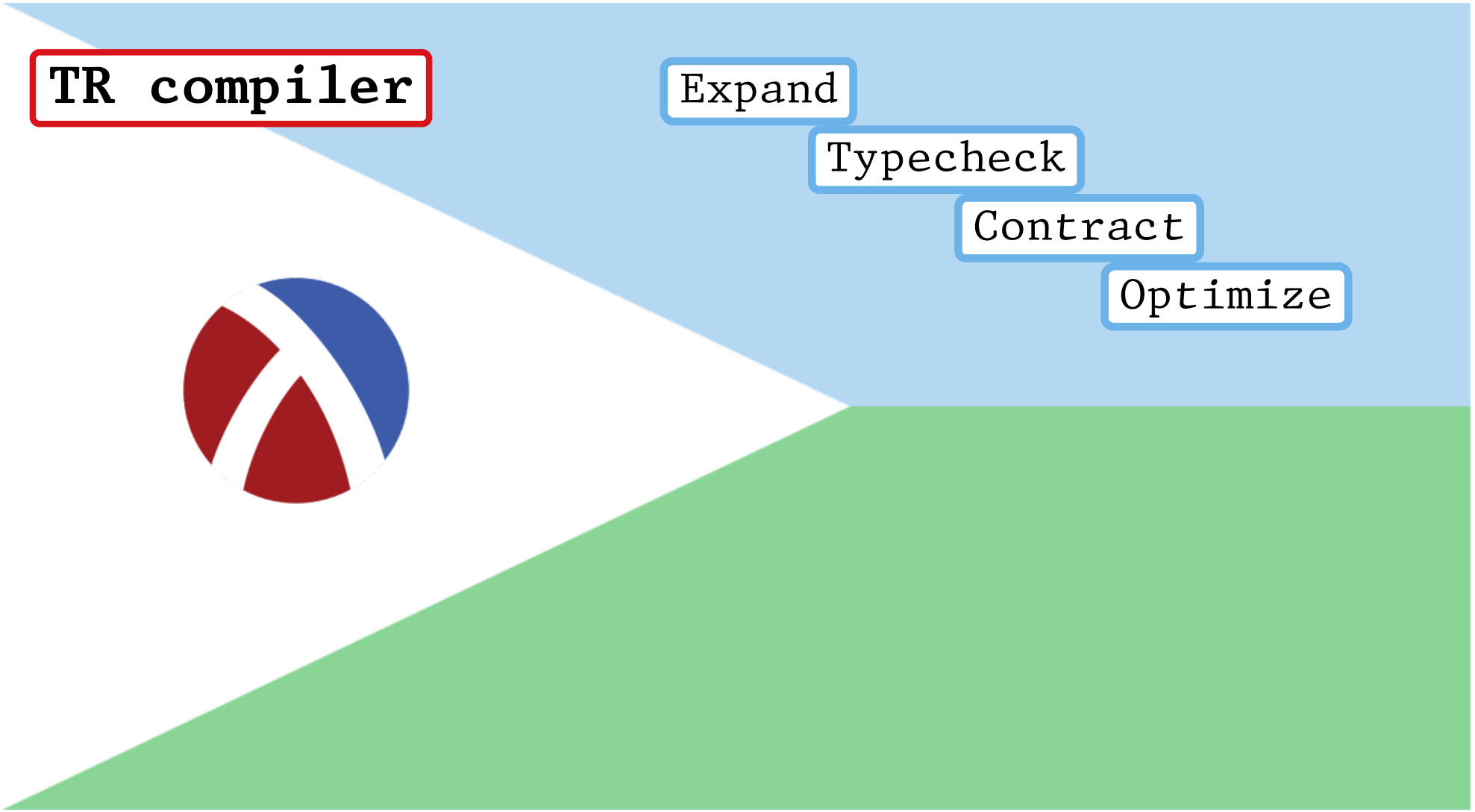
Example: Str -> Str

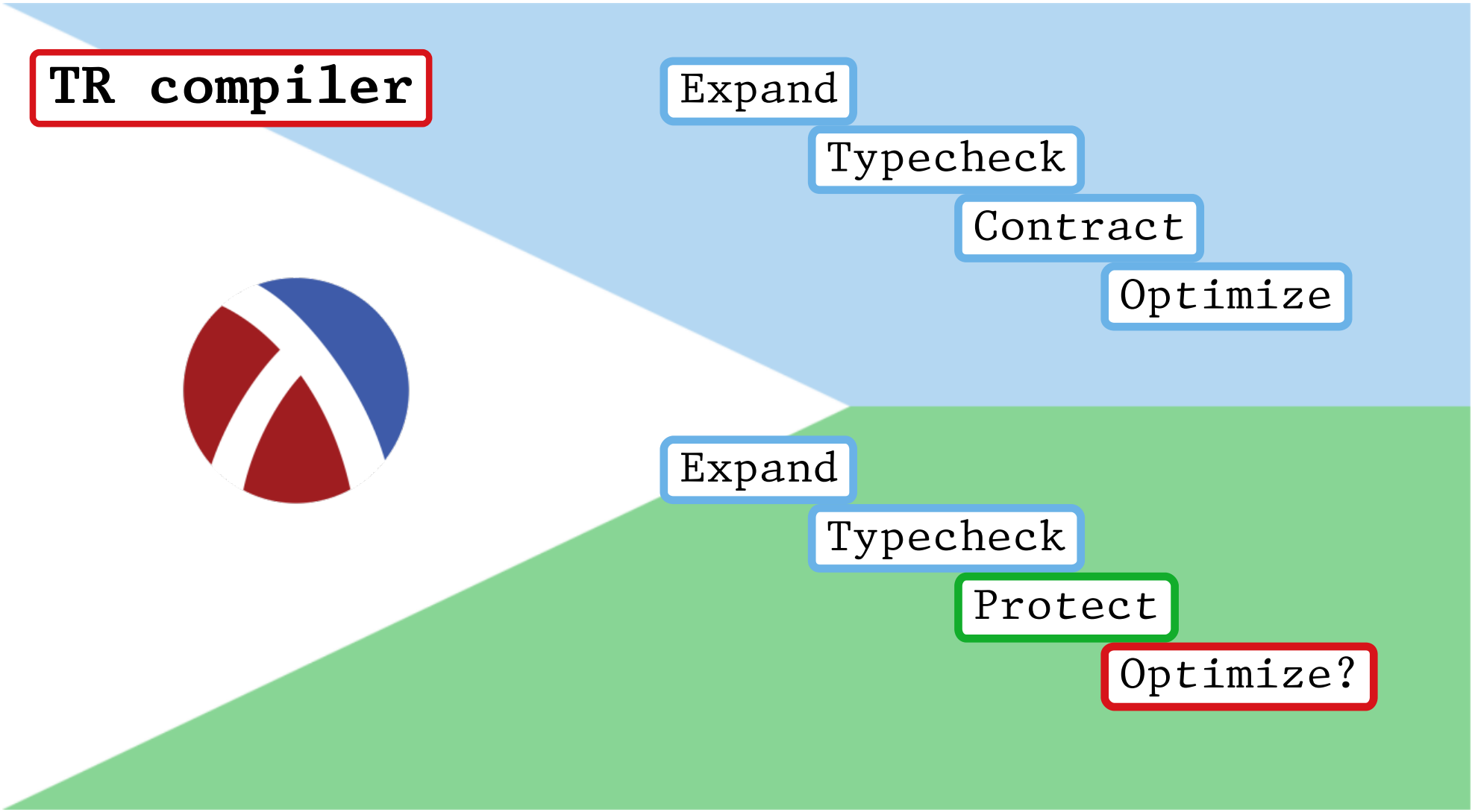


check function
& wrap

check function
& protect calls

Questions on Natural and Transient?


TR compiler
Expand
Typecheck
Contract
Optimize


TR compiler
Expand
Typecheck
Contract
Optimize
Expand
Typecheck
Protect
Optimize?

"I'm curious whether any of TR's optimizations are in fact unsound for transient. Have you checked?"

Sam Tobin-Hochstadt

Standard Example

```
(define (f (n : Flonum) (m : Flonum))
  (fl+ n m))
```

```
(define (f (n : Flonum) (m : Flonum))
  (unsafe-fl+ n m))
```

Standard Example

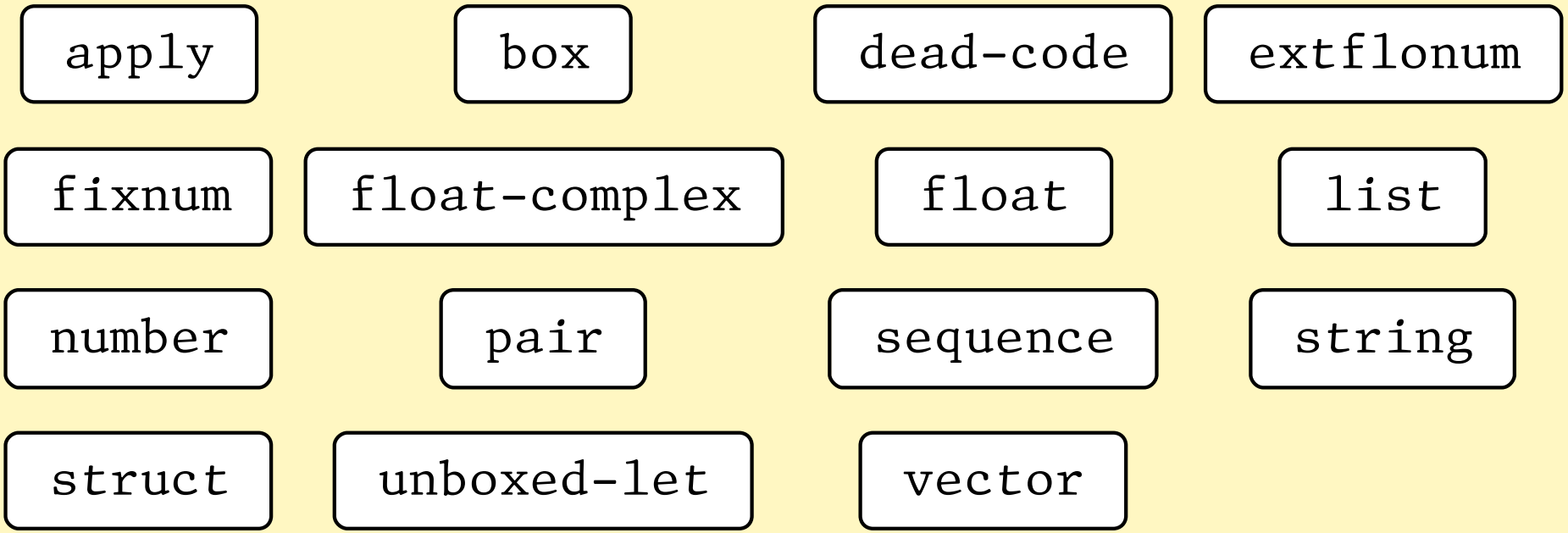
```
(define (f (n : Flonum) (m : Flonum))
  (fl+ n m))
```

```
(define (f (n : Flonum) (m : Flonum))
  (unsafe-fl+ n m))
```

SAFE for Transient

Optimizations

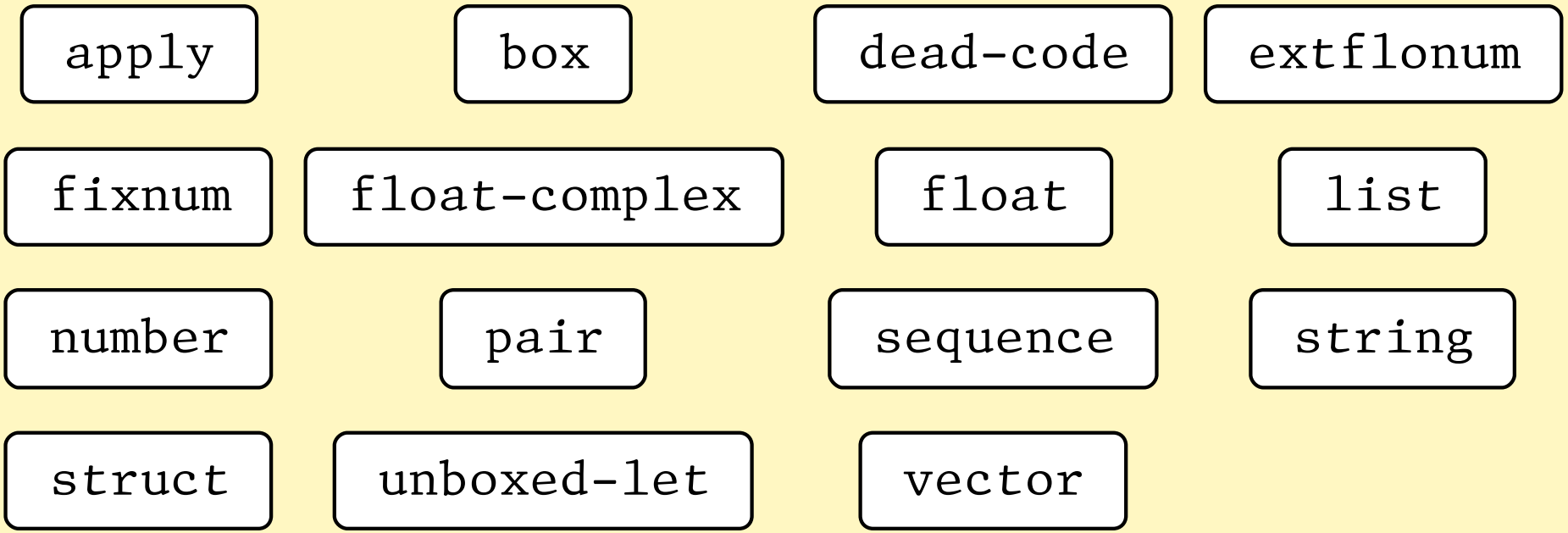
Optimizations



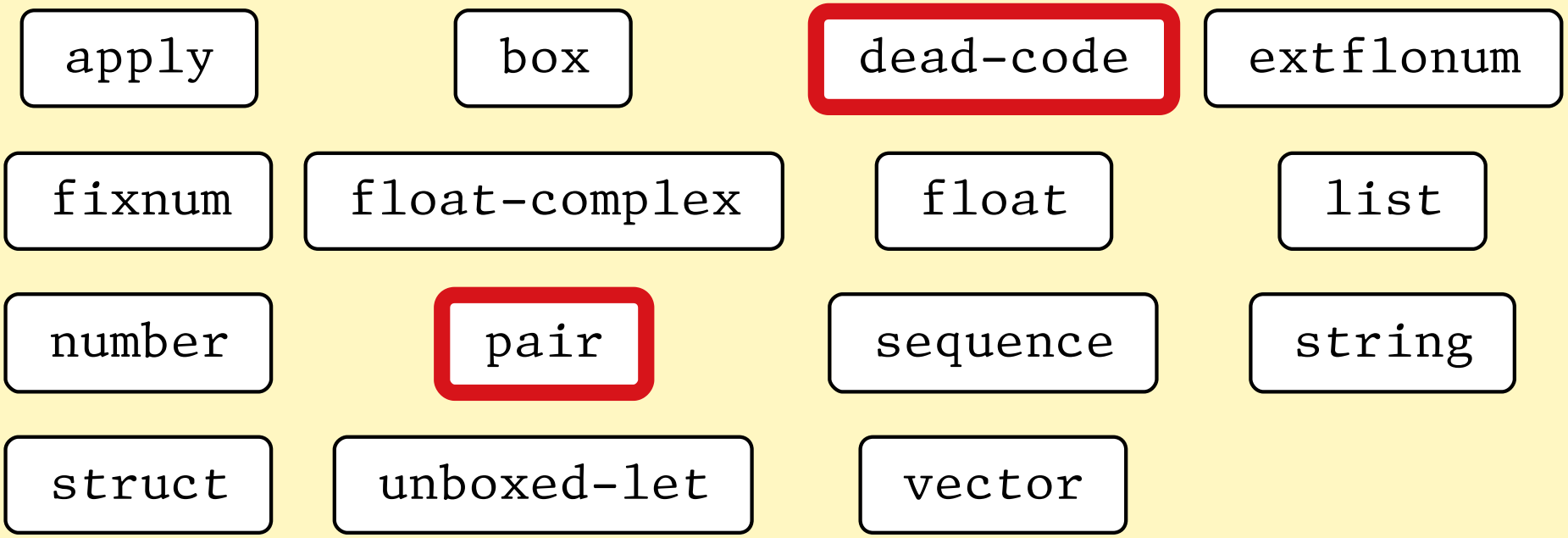
Q. Do any rely on full types?

T vs. ⌊T⌋

Optimizations



Optimizations



`dead-code` = unsafe for Transient

```
(: g (-> Str Str))
(define g
  (case-lambda
   [(x) x]
   [(x y) y]))
```

```
(define g
  (case-lambda
   [(x) x]
   [(x y) (void)]))
```

Problem: untyped code can call `(g 0 1)`

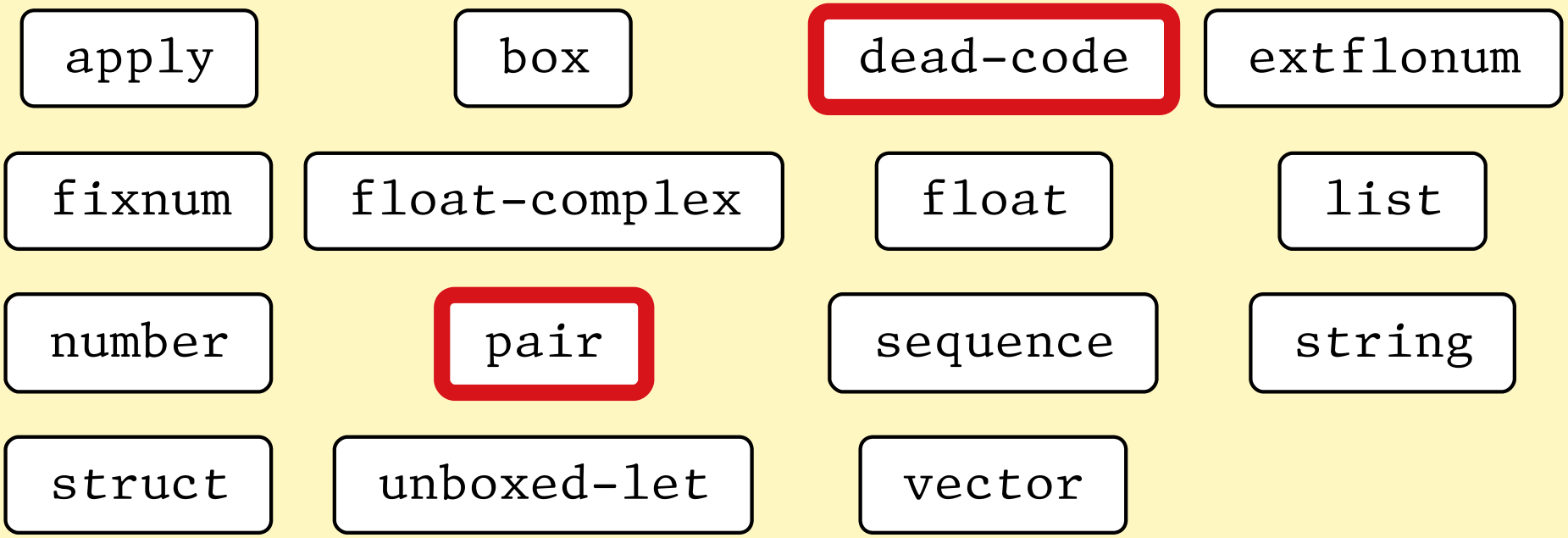
`pair` = unsound for Transient

```
(: x (Pairof (Pairof Nat Int) Str))
(cdar x)
```

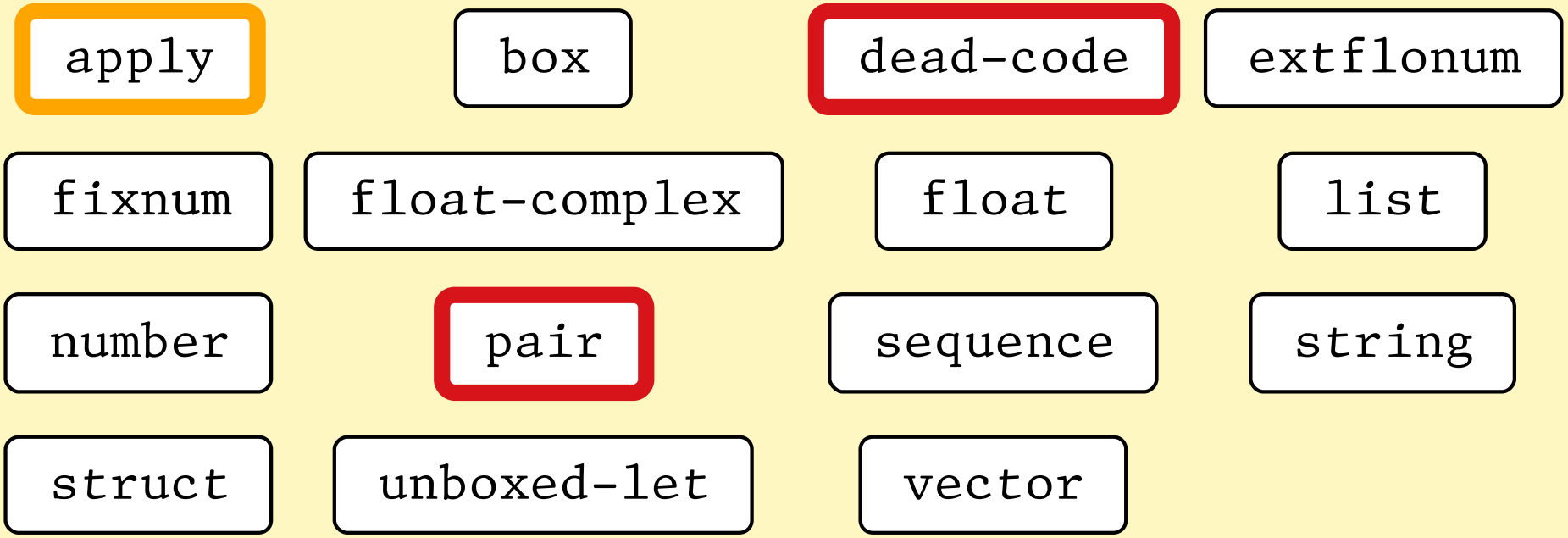
```
(unsafe-cdr (unsafe-car x))
```

Problem: no guarantee `(car x)` is a pair

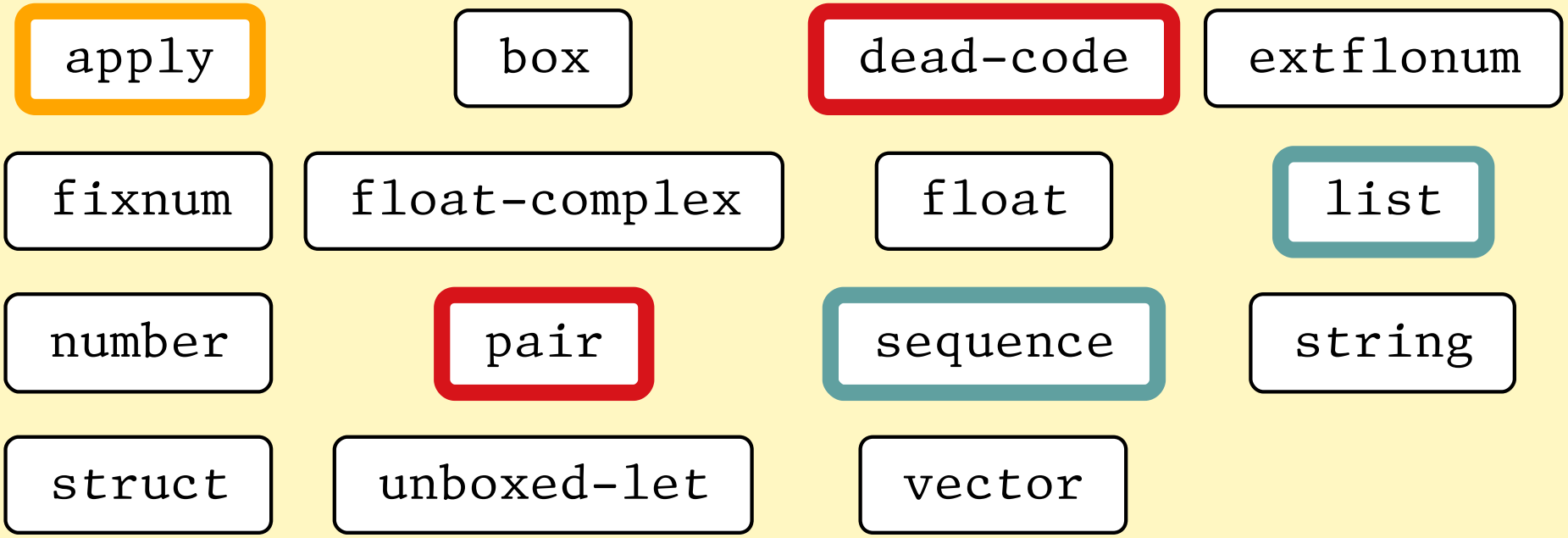
Optimizations



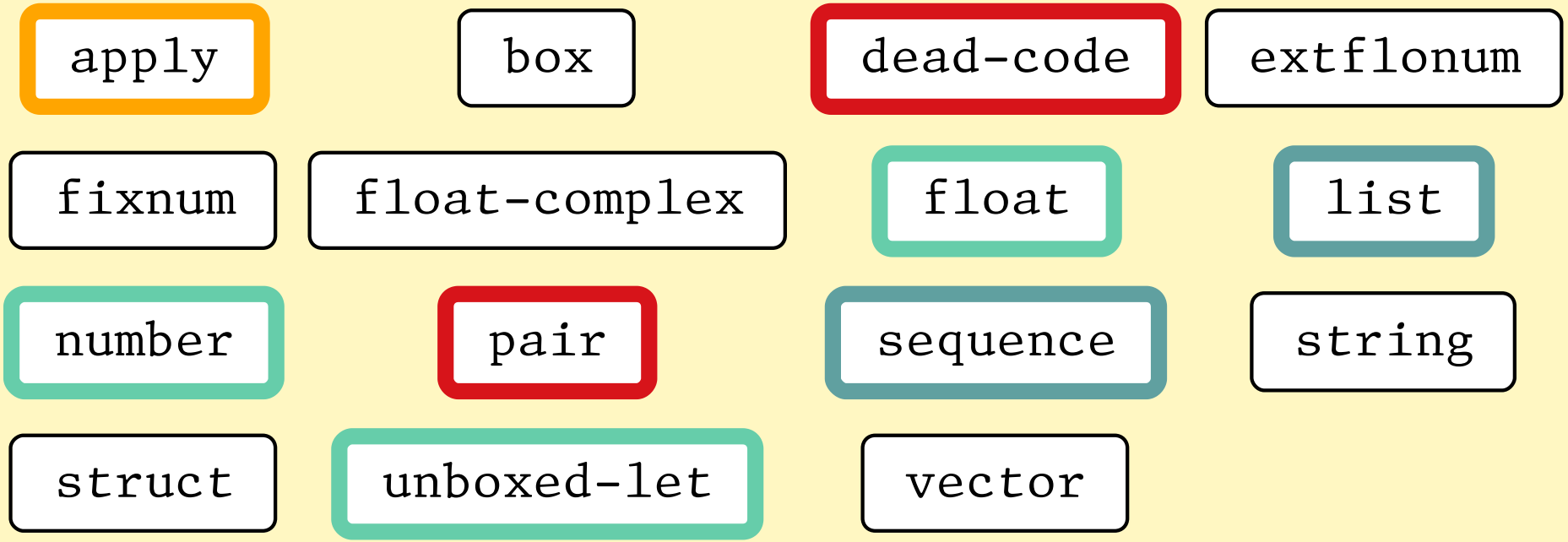
Optimizations

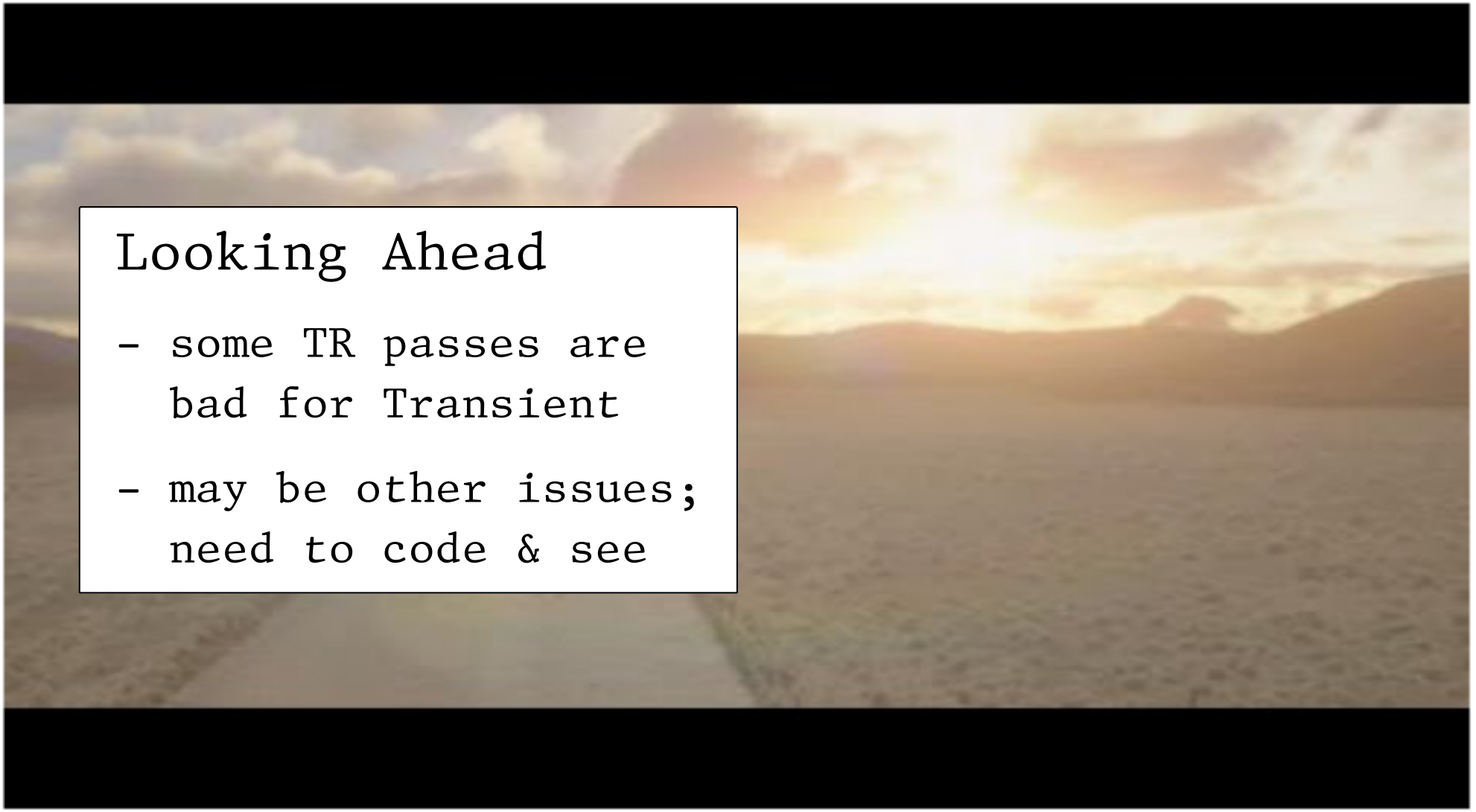


Optimizations

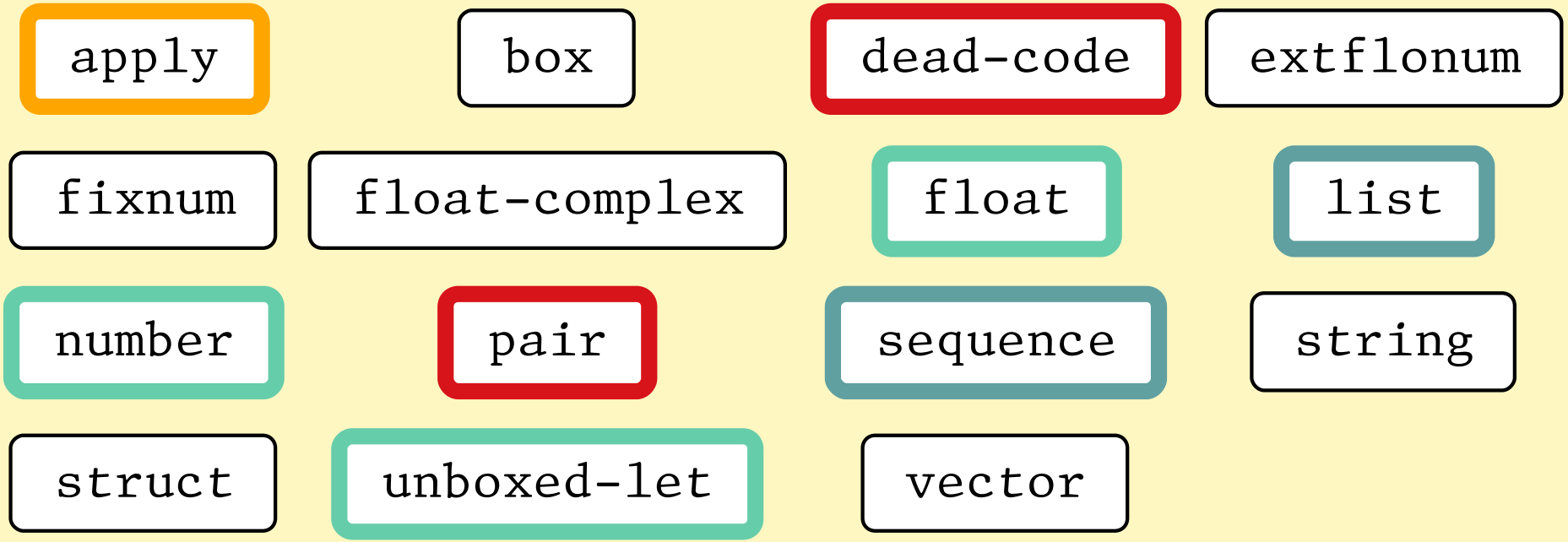


Optimizations




Looking Ahead
- some TR passes are bad for Transient
- may be other issues; need to code & see

Optimizations



`apply` = safe but risky for Transient

```
(: h (-> Str Str))
(: xs (Listof Str))
(apply + (map h xs))
```

```
(+ (h (unsafe-car xs)) (h (unsafe-car (unsafe-cdr xs))) ...)
```

Caution: `h` must check inputs

`list` `sequence` = force choice for ⌊T⌋

```
(: xs (List Str Str))
(list-ref xs 1)
```

```
(unsafe-list-ref xs 1)
```

Note: ⌊**List Str Str**⌋ needs more than a tag check

`number` = ⌊T⌋ is more than a tag check

Natural

Exact-Nonnegative-Integer

Nonpositive-Inexact-Real

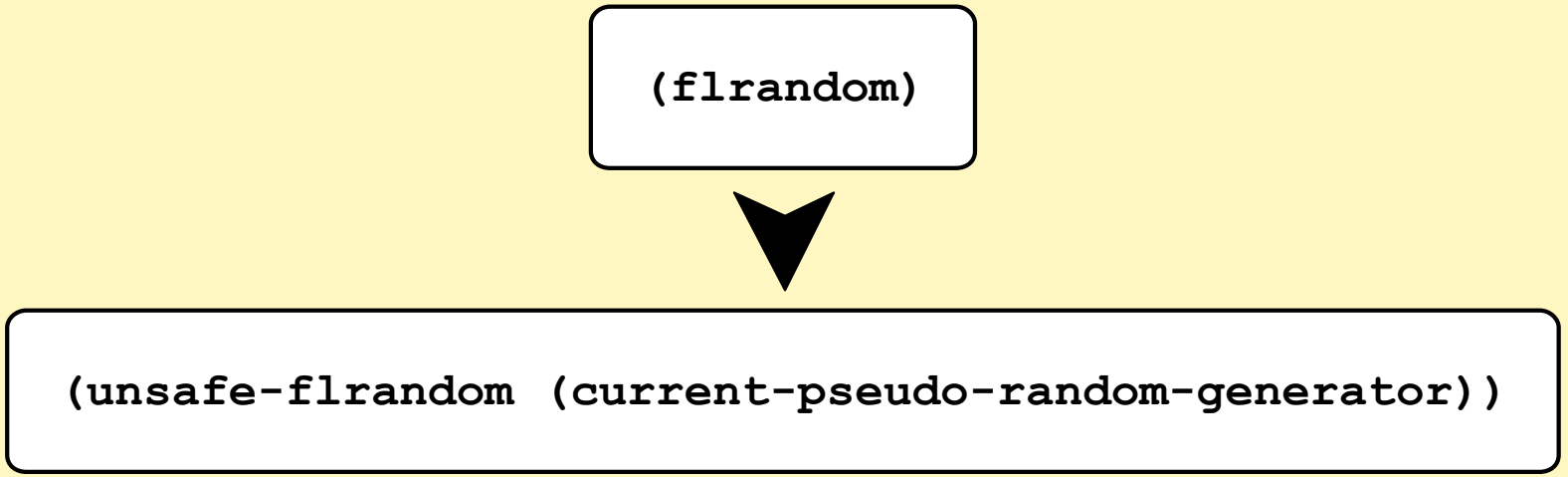
ExtFlonum-Negative-Zero

`unboxed-let` = safe with escape analysis

```
(: f (-> Float-Complex Any))
(define (f n)
  ....)
```

```
(define (f n-real n-imag)
  ....)
```

`float` = false alarm



Ok because the PRNG parameter checks inputs

A Brief History

- first commit in 2010 by Sam T-H (1e6aaf)

- appeared in PADL 2012, OOPSLA 2012, and St-Amour's dissertation

- contributors: Eric Dobson, Ryan Culpepper, Asumu Takikawa, Spencer Florence, Ben Greenman, Andrew Kent, and Matthew Flatt