The Operational Semantics of Gradual Typing

1. Prehistory + What is Gradual Typing?

2. Core Semantics of types: Natural, Transient, Erasure

3. Attempts to compare them:
   Type Soundness,
   Complete Monitoring,
   Kafka Litmus test

Static vs Dynamic

- Types checked at compile time

- $\dfrac{\vdash e_1 : \boxed{\tau_1 \to \tau_2} \quad \vdash e_2 : \tau_1}{\vdash e_1\ e_2 : \tau_2}$

- Strong error protection in advance

- program analysis $\Rightarrow$ optimizations

- Types checked at Runtime

- $\dfrac{v_1 \neq \lambda x.e}{v_1\ v_2 \longrightarrow \boxed{\text{Err}}}$

- rapid prototyping

- flexible idioms

- Support legacy code!

Pre history

- GT has roots in interop between static and dynamic languages
- Gray et al 2005:
  Java + Scheme
- Matthews / Findler 2009:
  ML + Scheme / general framework
- interop can be understood as generating and checking contracts
  dynamic → static
  static → dynamic

Gradual Typing

- Mix static and dynamic typing in one language, boundary calculus:
  - boundary terms send static to dynamic and vice versa
  - boundary terms may reduce to checks/monitors
  - Checks and propogation in Operational Semantics lead to semantics of types

What are the core starting points for semantics of types?

- Natural - Siek/Taha 2006, Tobin-Hochstadt / Felleisen 2006
  - original core formalization
  - deep, expensive checks
- Transient - Vitousek et al 2017
  - Alternative "transient" checks
  - shallow checks, less expensive
- Erasure - Bierman et al 2014
  - Comes from industry: Typescript
  - boundaries are erased, code runs as dynamic, no checks

Common Language

$$e = \lambda x{:}\tau.e \mid \lambda x.e \mid (e, e)$$
$$\mid DS\ \tau\ e \mid SD\ \tau\ e$$
$$\mid Err \mid \ldots$$

- distinct typed and untyped lambdas
- DS means dynamic outside, static inside
- SD means static outside, dynamic inside

Common Language

$$\tau = \tau \times \tau \mid \tau \to \tau \mid \ldots$$

$$\Gamma = \cdot \mid \Gamma, x \mid \Gamma, x : \tau$$

$$\frac{\Gamma, x \vdash e}{\Gamma \vdash \lambda x.e} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau.e : \tau \to \tau'}$$

$$\frac{\Gamma \vdash e}{\Gamma \vdash SD\ \tau\ e : \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash DS\ \tau\ e}$$

- Γ contains bindings for both typed and untyped
- SD, DS convert between typed and untyped

- $\vdash e$ is "typing" for untyped
  - closed for dynamic, typing for static

Natural Semantics

- When values cross boundaries, they're either:
  - eagerly checked to have type
  - wrapped in a monitor, which lazily checks when needed
    - Similar to higher order Contracts

Natural Semantics

- What happens when value flows across boundary?

- Pairs are checked eagerly:

$$SD\ (\tau_1 \times \tau_2)\ (v_1, v_2) \longrightarrow^{V}_{S} (SD\ \tau_1\ v_1,\ SD\ \tau_2\ v_2)$$

$$DS\ (\tau_1 \times \tau_2)\ (v_1, v_2) \longrightarrow^{V}_{D} (DS\ \tau_1\ v_1,\ DS\ \tau_2\ v_2)$$

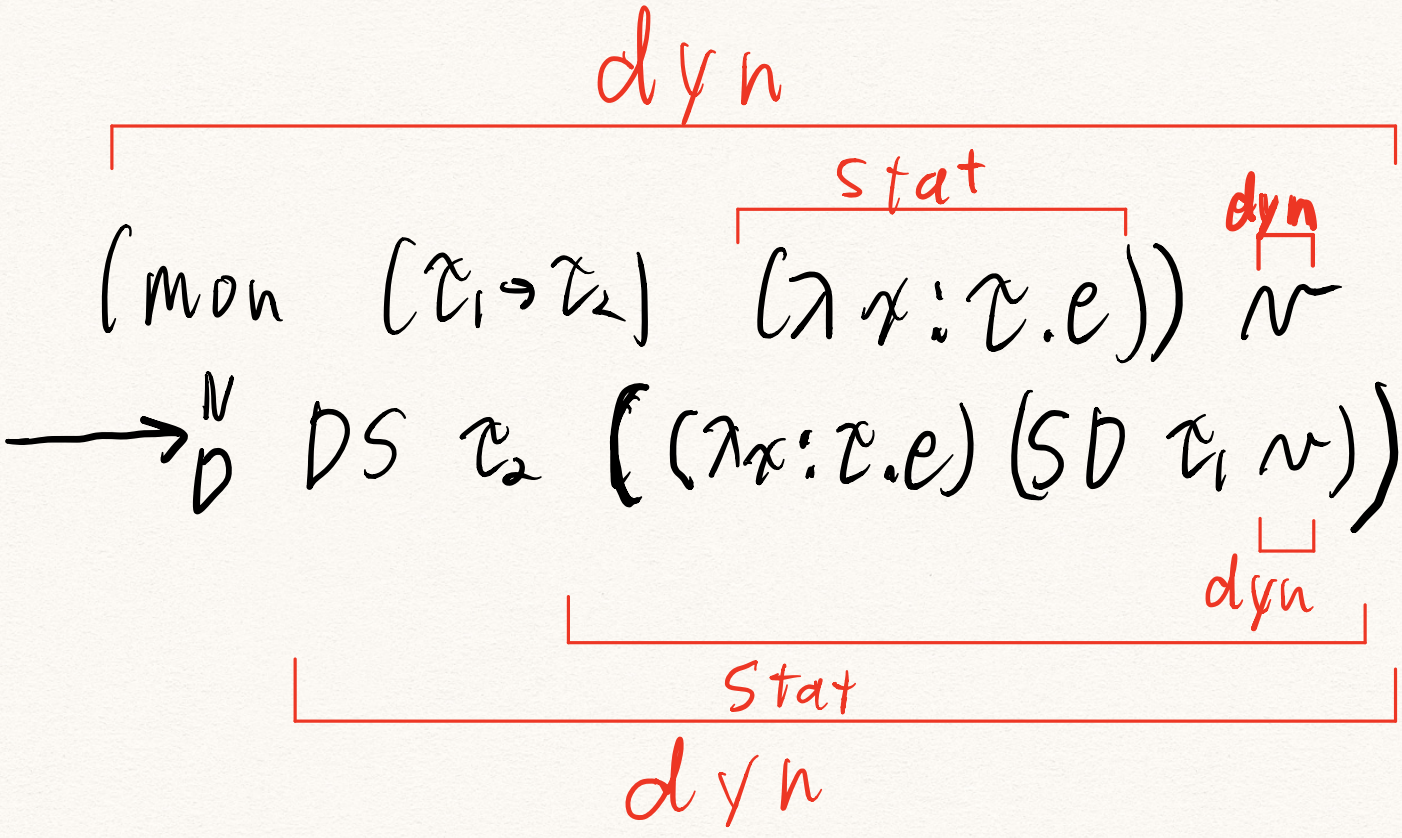
- functions acquire a monitor which lazily checks type

$$e ::= \ldots \mid mon\ (\tau_1 \to \tau_2)\ e$$

$$SD\ (\tau_1 \to \tau_2)\ (\lambda x.e) \longrightarrow^{N}_{S} mon\ (\tau_1 \to \tau_2)\ (\lambda x.e)$$

$$DS\ (\tau_1 \to \tau_2)\ (\lambda x{:}\tau.e) \longrightarrow^{N}_{D} mon\ (\tau_1 \to \tau_2)\ (\lambda x{:}\tau.e)$$

- how do monitors work?



- argument flows to static

- result of application flows to dynamic

- When do boundaries error?

- When the type doesn't match the term

SD Nat -2 $\rightarrow^{N}_{S}$ Err

-2 & Nat

SD (Nat x Nat) (-2, -2)

$\rightarrow^{N}_{S}$ (SD Nat -2, SD Nat -2)

$\rightarrow^{N}_{S}$ Err

- When do monitors error?

$(\text{mon}\ (\text{Nat} \to \text{Nat})\ (\lambda x{:}\text{Nat}.\, x))\ -2$

$\xrightarrow[0]{N}$ DS Nat $((\lambda x{:}\text{Nat}.\, x)\ (\text{SD Nat}\ -2))$

$\xrightarrow[0]{N}$ Err

- Because of monitor, argument flows into static and errors

Transient Semantics

- When values cross boundaries, and when values are consumed, first order properties are checked
- translate transient terms before running:

$$\Gamma \vdash e : \tau \rightsquigarrow e'$$

  - We check the result type of applications and projections, ie

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \rightsquigarrow e'}{\Gamma \vdash \mathrm{fst}\ e : \tau_1 \rightsquigarrow \mathrm{check}\ \tau_1\ (\mathrm{fst}\ e)}$$

$e ::= \ldots \mid \text{check } \tau\ e$

- check τ v succeeds iff the shape of τ and v match, ie
  - $\tau = \tau_1 \times \tau_2$, $v = (v_1, v_2)$
  - $\tau = \tau_1 \to \tau_2$, $v = \lambda x{:}\tau.\, e$

- When values cross boundaries:

  $S\ D\ \tau\ v \longrightarrow_S^T \text{check } \tau\ v$

  $D\ S\ \tau\ v \longrightarrow_D^T v$

— When values are consumed:

$$(\lambda x{:}\tau.e)\ v \longrightarrow_S^T e[v/x]$$

iff the shape of τ and v match, otherwise

$$\longrightarrow_S^T Err$$

SD Nat -2 $\rightarrow_S^T$ Err

-2 & Nat

SD (Nat x Nat) (-2, ~2)

$\rightarrow_S^T$ check (Nat x Nat) (~2, -2)

$\rightarrow_S^T$ (-2, -2)

Check succeeds !!

⊢ fst (SD (Nat x Nat) (-2, -2)) : Int

$\rightarrow_S^{T^*}$ check Int (fst (-2, -2))

$\rightarrow_S^{T^*}$ check Int -2 $\rightarrow_S^T$ -2

misleading !

$$\text{fst}\ (DS\ (\text{Nat} \times \text{Nat})\ (SD\ (\text{Nat} \times \text{Nat})\ (-2, -2)))$$

$$\xrightarrow{T^*_S}\ \text{fst}\ (DS\ (\text{Nat} \times \text{Nat})\ (-2, -2))$$

$$\xrightarrow{T_D}\ \text{fst}\ (-2, -2)$$

$$\xrightarrow{T_D}\ -2$$

misleading!

Erasure Semantics

- When values cross a boundary, check nothing!

$$SD \; \tau \; v \xrightarrow{E}_{S} v$$

$$DS \; \tau \; v \xrightarrow{E}_{D} v$$

- Lose all guarantees about the structure of terms beyond dynamic

$SD\ \underline{Nat}\ {-2} \xrightarrow{E}_{S} {-2}$

this can be any type!

$SD\ (Nat \times Nat)\ (-2, -2)$

$\xrightarrow{T}_{S}\ (-2, -2)$

$fst\ \Big(DS\ (Nat \times Nat)\ \big(SD\ (Nat \times Nat)\ (-2, -2)\big)\Big)$

$\xrightarrow{E}_{S}\ fst\ \big(DS\ (Nat \times Nat)\ (-2, -2)\big)$

$\xrightarrow{E}_{D}\ fst\ (-2, -2)$

$\xrightarrow{E}_{D}\ -2$

Summary

- Natural Semantics
  - deep checks / monitors
- Transient Semantics
  - shallow/shape checks
- Erasure Semantics
  - No checks!

Comparing Gradually Typed Languages

- Formal Properties
  - Type Soundness - Proven in Natural/Transient papers
  - Complete Monitoring - Recent property that distinguishes Natural
- Katka Litmus test
  - examples that demonstrate distinctions

Type Soundness

- Greenman / Felleisen 2018:
  Spectrum of Type Soundness

- Natural checks deeply / monitors, so when run to value, we get full type:

  $SD\ (\tau_1 \times \tau_2)\ (e_1, e_2) \longrightarrow_S^{N*} v$

  $\Longrightarrow \vdash v : \tau_1 \times \tau_2$

- Transient checks shallow, so when run to value, we get same shape:

  $SD\ (\tau_1 \times \tau_2)\ (e_1, e_2) \longrightarrow_S^{T*} v$

  $\Longrightarrow \vdash v : \mathrm{Pair}$

- Erasure only guarantees lack of stuck states

  so $(\tau_1 \times \tau_2)\ (e_1, e_2) \longrightarrow_S^{T*} v$

  $\Longrightarrow$ v could be anything

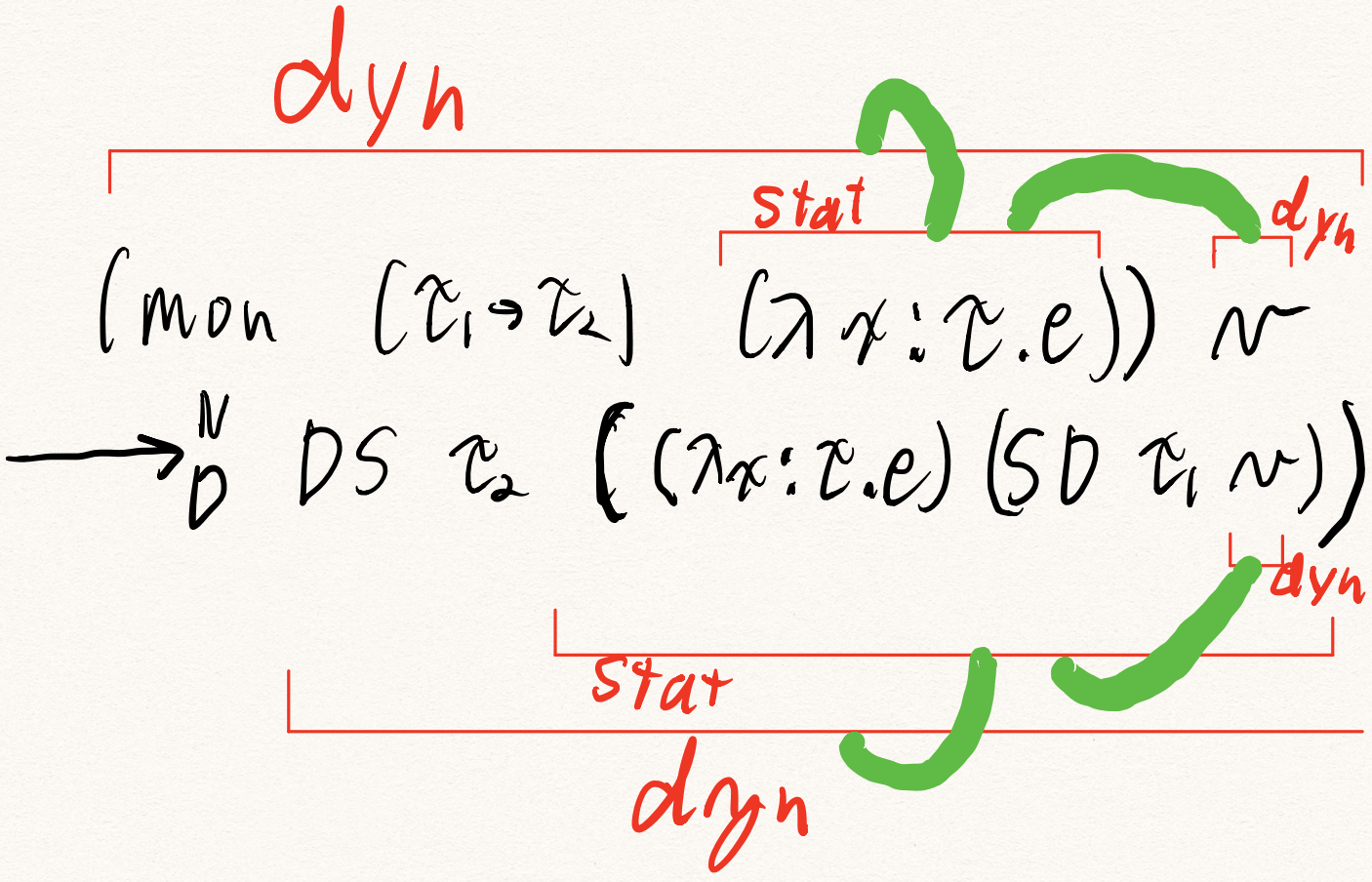
- If you avoid all interaction with untyped, Erasure and Transient have full type soundness
  - Kind of Trivial

Complete Monitors for Gradual Types

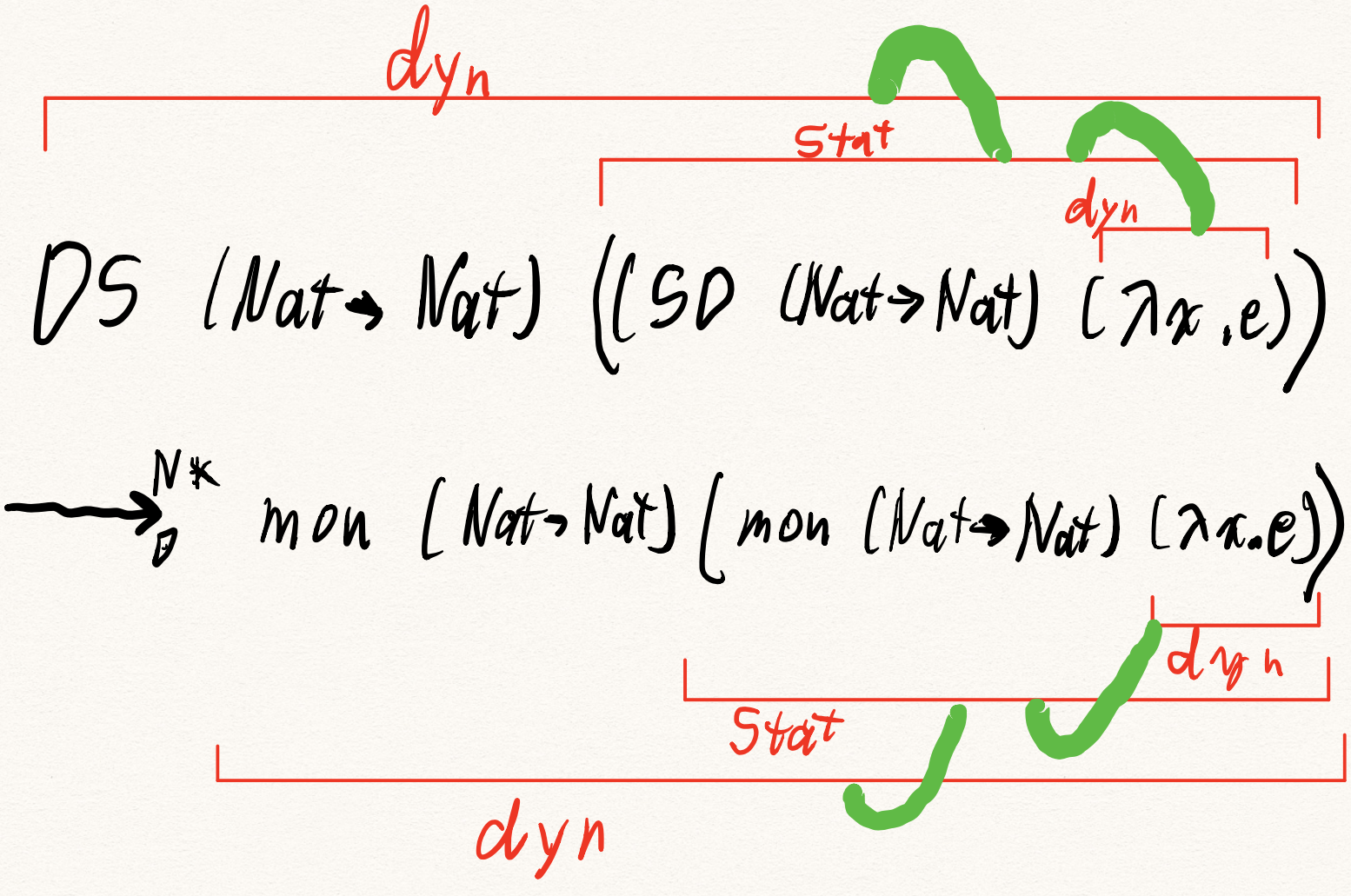
- Greenman et al 2019

- In contracts:
  - Correct blame: blamed component supplied the contract violating value
  - Complete Monitoring: Correct Blame and all communication channels are monitored

- In GT:
  - Every channel of communication between components is monitored



- Natural is Complete Monitor
  - eager checks / delayed-checking monitors are always performed / inserted


dyn
Stat
dyn
DS (Nat→Nat) ((SD (Nat→Nat) (λx.e))
N*
D
mon (Nat→Nat) (mon (Nat→Nat) (λx.e))
dyn
Stat
dyn

- Transient is not Complete Monitor

DS (Nat→ Nat) ((SD (Nat→Nat) (λx.e))

$\longrightarrow_D^{N*}$ (λx.e)

dyn

!!!

- Transient "forgets" it went through Static boundary

- Future Work on Complete Monitoring for GT

  - Formal Statement is complicated, uses ownership labels

  - label free statement likely exists

Kafka: Gradual typing for Objects

- Chung et al 2019
- Present Natural, Transient, and Erasure in one Source syntax
- Compile to one target
- Run examples with different semantics, see what is produced
  - 3 Litmus tests

```
Class A {
  m (x:A):A {this}}

Class I {
  n (x:I):I {this}}

Class T {
  s (x:I):T {this}
  t (x;*):* {this.s(x)}}
```

```
new T(). t(new A())
```

— Errors in Natural and Transient

- A and I are unrelated wrt subtyping

```
Class A {
  m (x:A):A  {this}}
Class Q {
  n (x:Q):Q  {this}}
class I {
  m (x:Q):I  {this}}
Class T {
  s (x:I):T  {this}
  t (x;*):*  {this.s(x)}}
```

```
new T(). t(new A())
```

- doesn't error in any
- works because both I and A have method m, despite Q and A clash

```
Class C {
  a (x:C):C { x }}
Class D {
  b (x:D): D { x }}
Class E {
  a ( x:D):D {x}}
Class F {
  m (x:E): E  {x}
  n (x:*): *  {this.m(x)}}
```

```
new F().n (new C())
        .a (new C())
```

- Only fails in Natural, Transient forgets E

Open-World Soundness

- Property found in Transient paper
  1. Compile Gradual Language to Dynamic
  2. Then wrap compiled term with dynamic context
  3. If term has bad behavior, it's because dynamic context
- Immediate consequence of type soundness with boundary calculus

- Vitousek et al Transient has restrictive syntax

- doesn't help compare Transient with Natural

The Gradual Guarantee

- also found in Transient paper
- if you replace dynamic types with more static types, it either produces same result or fails a check
- Doesn't help compare these core semantics
  - Trivially true here
- Gets interesting with advanced features like polymorphism
  - New / Ahmed 2018

Summary

- 3 semantics:
  - Natural - deep
  - Transient - shallow
  - Erasure - None
- 3 comparisons
  - Type Soundness
  - Complete Monitoring
  - Katka Litmus tests