Pure Borrow

Linear Haskell Meets Rust-Style Borrowing

Yusuke Matsushita Kyoto University & MPI-SWS

Sept 15, 2026 @ Bristol PLRG

About Me

- **Yusuke Matsushita　松下 祐介**
  - July 15, 1996, Osaka
- **’15–’24　Bachelor, Master, Ph.D. at UTokyo**
  - **Computer science — Programming languages**
    - Supervised by **Prof. Naoki Kobayashi**
- **’25–　Assist. Prof. at Hakubi, KyotoU**
  - Focus on **Rust**
  - **’26–　Visiting scientist** at **MPI-SWS**
    - **Germany, Prof. Derek Dreyer**

About Me

Google Scholar SIGN IN

Yusuke Matsushita

FOLLOW

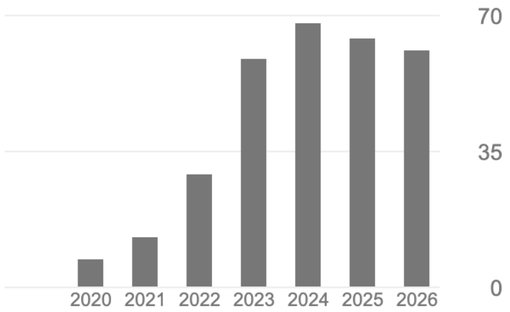
GET MY OWN PROFILE

Kyoto University
Verified email at fos.kuis.kyoto-u.ac.jp - Homepage
Computer Science Programming Languages Machine Learning

TITLE	CITED BY	YEAR
RustHorn: CHC-based Verification for Rust Programs Y Matsushita, T Tsukada, N Kobayashi ACM Transactions on Programming Languages and Systems (TOPLAS) 43 (4), 1-54	109	2021
RustHornBelt: A Semantic Foundation for Functional Verification of Rust Programs with Unsafe Code Y Matsushita, X Denis, JH Jourdan, D Dreyer Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language …	95	2022
RustHorn: CHC-Based Verification for Rust Programs Y Matsushita, T Tsukada, N Kobayashi European Symposium on Programming (ESOP), 484-514	65	2020
Nola: Later-Free Ghost State for Verifying Termination in Iris Y Matsushita, T Tsukada Proceedings of the ACM on Programming Languages 9 (PLDI)	8	2025
Borrowable Fractional Ownership Types for Verification T Nakayama, Y Matsushita, K Sakayori, R Sato, N Kobayashi International Conference on Verification, Model Checking, and Abstract …	7	2024

Cited by

	All	Since 2021
Citations	301	294
h-index	5	5
i10-index	3	3



Public access VIEW ALL

0 articles	6 articles
not available	available

About Me

PLDI Review Committee PLDI Research Papers

Benjamin Mariano
The University of Texas at Austin, Texas, USA
United States

Guido Martínez
Microsoft Research
United States

Yusuke Matsushita
Kyoto University
Japan

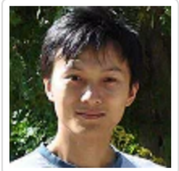
Jedidiah McClurg
Cornell University
United States

Roland Meyer
TU Braunschweig
Germany

Mae Milano
Princeton University
United States

Program Committee POPL

Yannick Zakowski
Inria
France



Yingfei Xiong
Peking University
China

Youyou Cong
Institute of Science Tokyo
Japan

Yu Feng
University of California at Santa Barbara
United States

Yu-Fang Chen
Academia Sinica
Taiwan

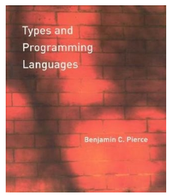
Yusuke Matsushita
Kyoto University
Japan

Up to Now

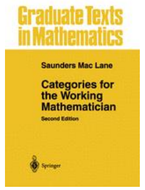
High Schooler — **Undergrad** — **Grad** — **Now**

Haskell

λ-calculus Monad



Type theory



Category theory

PL

Rust

Fast like C/C++,
Safe like Haskell?

Study PL!

RustHorn
ESOP '20 & TOPLAS '21

Rust → Functional

RustHornBelt
PLDI '22

Logic for RustHorn

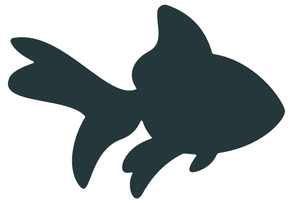
Nola
PLDI '25

Pure Borrow
PLDI '26

Haskell Meets Rust!

This Talk's Focus

Matsushita & Ishii, PLDI '26

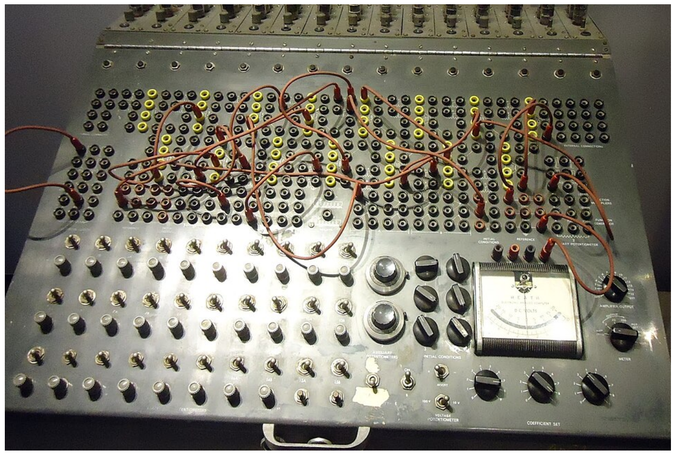
Pure Borrow 

Rust-Style Borrowing in Linear Haskell with Purity

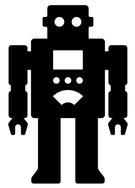
Introduction

Software

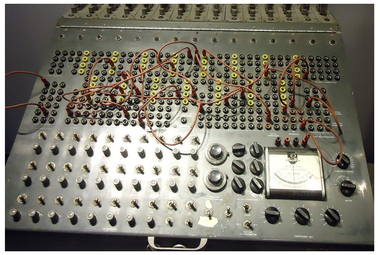
Software



Software embodies human ideals 

on finite physical machines 

Two Major Paradigms of Programming



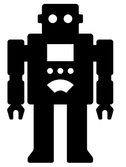
Functional

High-level abstraction

Well-behaved

Safe & Predictable

Imperative



Low-level control

Well-tuned

Performative

Functional, with Purity

Haskell '90

λ-calculus '36

Purity

Same result regardless of timing

Well-behaved, esp. *parallelism*

Purity = Referential Transparency

Super Well-behaved!

Deterministic

Same result across executions

$(t, t) \equiv \textbf{let}\ x = t\ \textbf{in}\ (x, x)$

Order-insensitive

Commutes with other computations

$(t, u) \equiv \textbf{let}\ x = u\ \textbf{in}\ (t, x)$

✔ ***Predictable***

✔ ***Equations***

Parallelism, Lazy evaluation, ...

Reasoning & Optimization

e.g. $\text{map}\ (g \circ f) \equiv \text{map}\ g \circ \text{map}\ f$

Functional, More Imperative

Linear Haskell *Bernardy+ '18*

Mutate! **Linear Types** *'90–*

Performance?

Haskell *'90*

Purity

Same result regardless of timing

λ-calculus *'36*

Well-behaved*, esp.* ***parallelism***

Linear Types

Wadler '90, Bernardy+ '18

- **Original Haskell** **Persistent** data → **Copy**

$$\text{Vec a} \to \text{Int} \to (\text{a} \to \text{a}) \to \text{Vec a}$$

`[0,1,2,…,999]` `0` `(+ 7)` `[7,1,2,…,999]` ***Pure!***

`[0,1,2,…,999]`

- **Linear Haskell** **Linear** → **Mutate** in place!

Linear arrow $\multimap$ *Argument is used exactly* ***once***

$$\underline{\text{Vec a}} \multimap \text{Int} \to (\text{a} \multimap \text{a}) \multimap \underline{\text{Vec a}}$$

Direct ownership passing

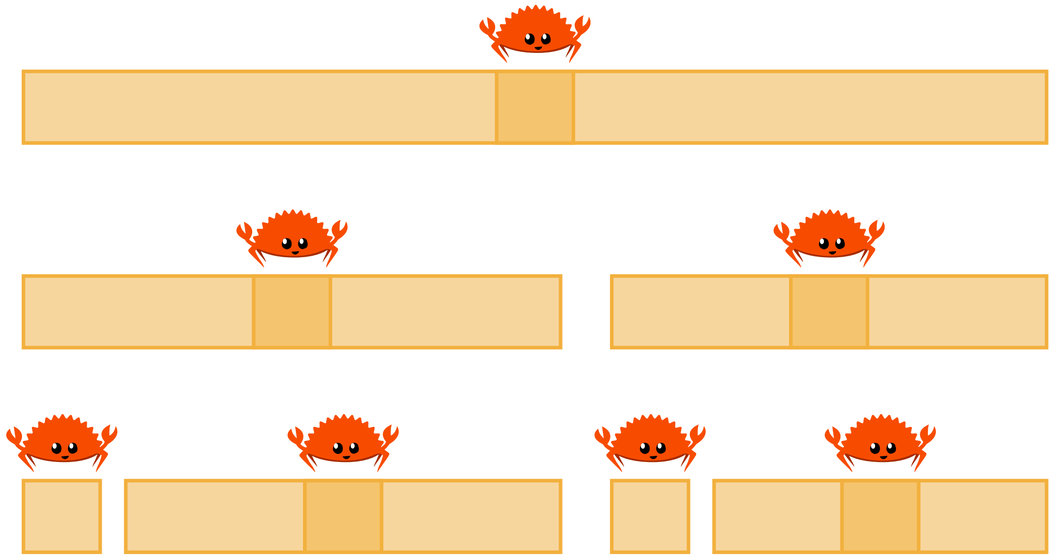
~~`[0,1,2,…,999]`~~ `0` `(+ 7)` `[7,1,2,…,999]` ***Pure!***

No aliases!

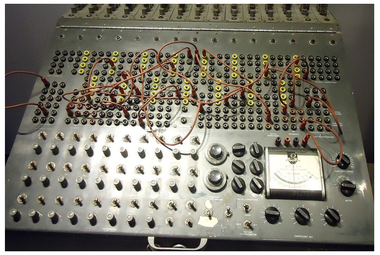
Challenge

Separate mutations to disjoint subregions

E.g., **Parallel quicksort**



Two Major Paradigms of Programming



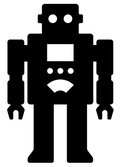
Functional

High-level abstraction

Well-behaved

Safe & Predictable

Imperative

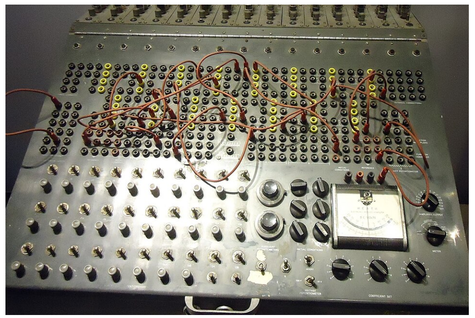


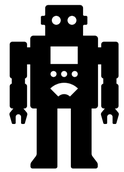
Low-level control

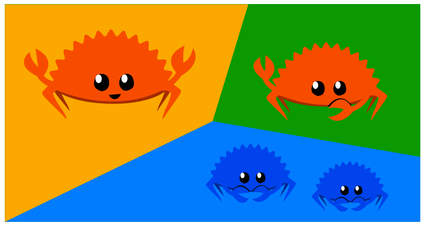
Well-tuned

Performative

Imperative, More Functional







Ownership & Region Types *'94–*

Well-behaved

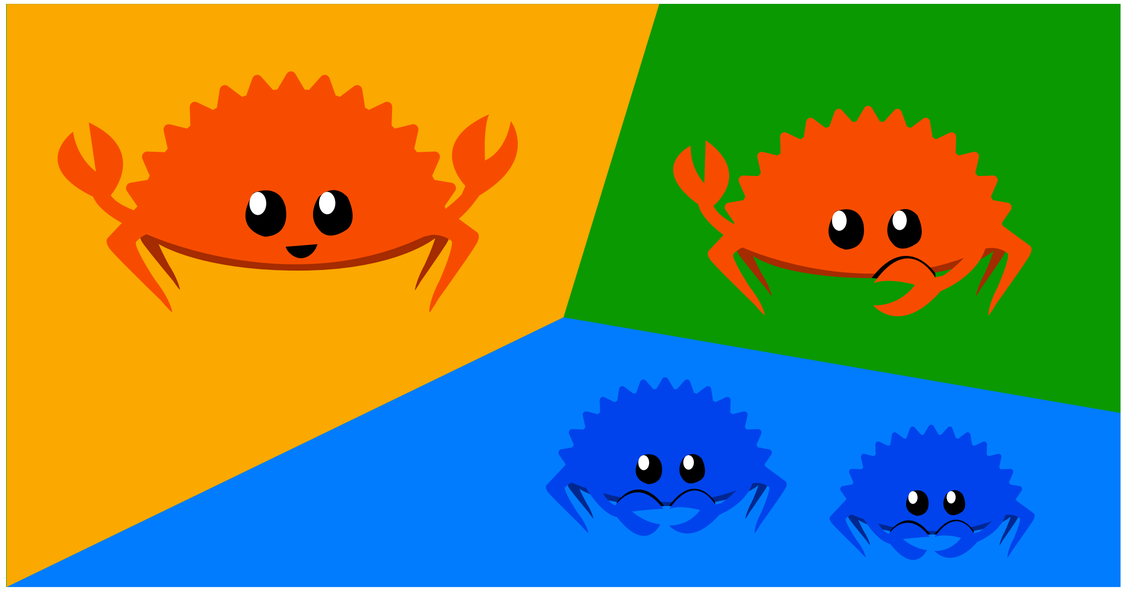
Memory corruption, Race condition, …

'72

'85

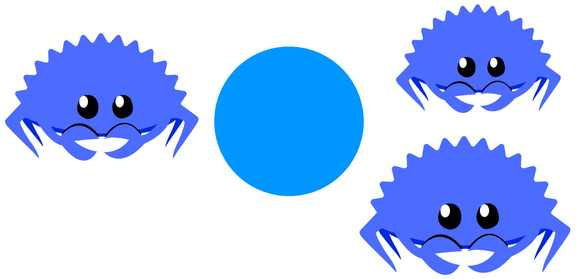
Ownership

Separate Mutable State



XOR

Aliasing



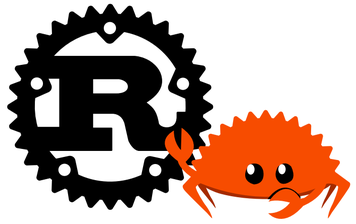
→ No dangerous interference between computations

Imperative, More Functional with Borrows

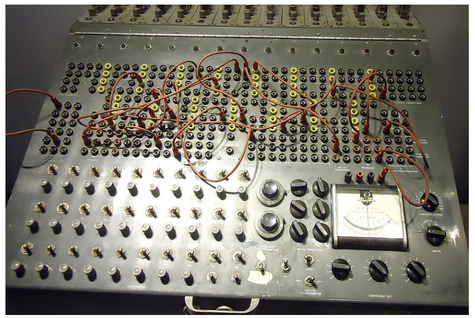
Borrow **Non-local!**

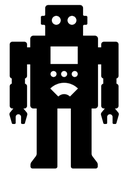
Separate ownership by time

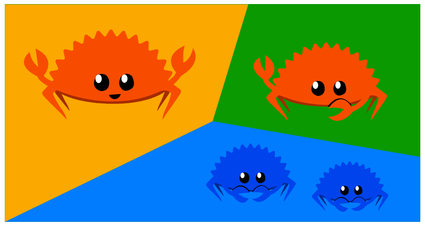
Mutation w/o ownership passing



Rust '15







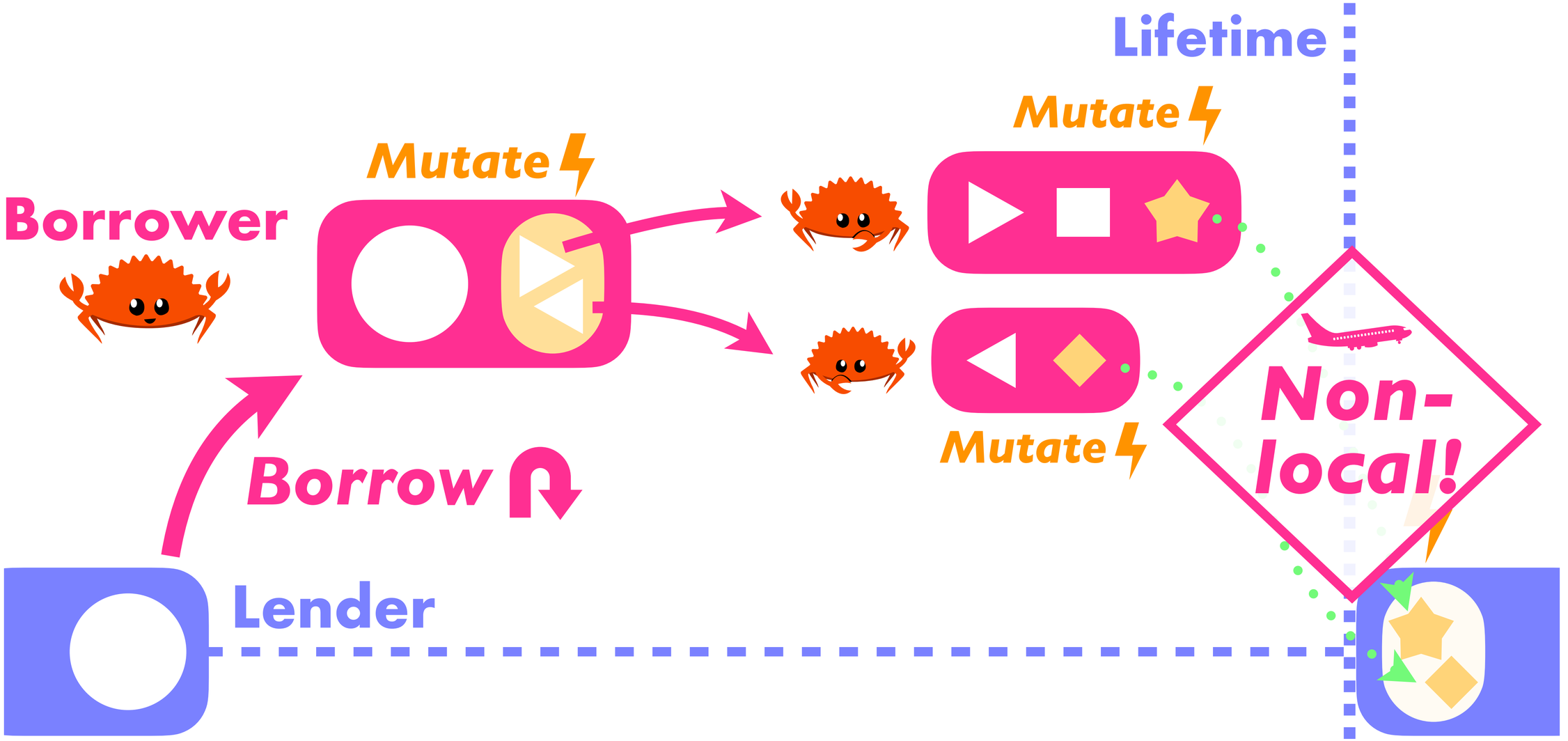
Ownership & Region Types '94–

Well-behaved

Memory corruption, Race condition, …

'72 '85

Rust-style Borrow

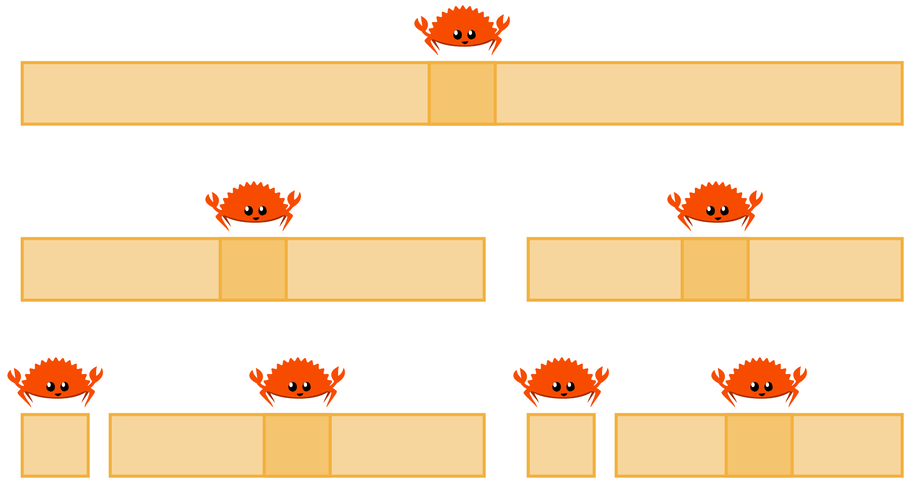


Borrowing Solved! But, ...

✔ Borrow

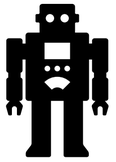
Separate mutations to disjoint subregions

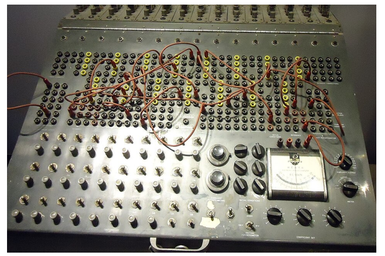
E.g., **Parallel** quicksort



Purity?

Challenge





Purity

Same result regardless of timing

Borrow **Non-local!**

Separate ownership by time

Mutation w/o ownership passing

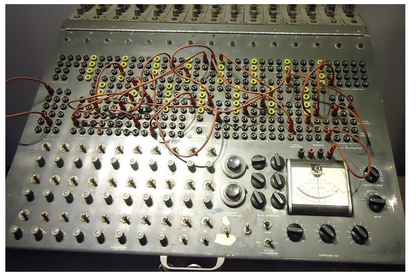
Linear Haskell

Rust

Our Work

Matsushita & Ishii, PLDI '26

Pure Borrow



Rust-Style Borrowing in Linear Haskell with Purity

High-Level Comparison

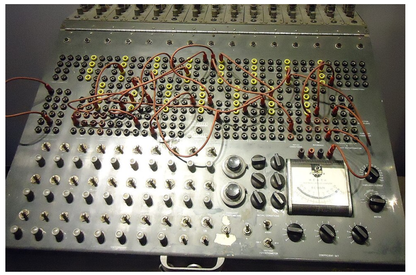
	Pure 	Mutate	Borrow	Parallel	Leak-free
Pure Only	●	○		●	
ST Monad	●	●	○	○	○
Rust	○	●	●	●	○
Basic Linear Haskell	●	●	○	●	●
Pure Borrow	●	●	●	●	●

Our Work

Matsushita & Ishii, PLDI '26

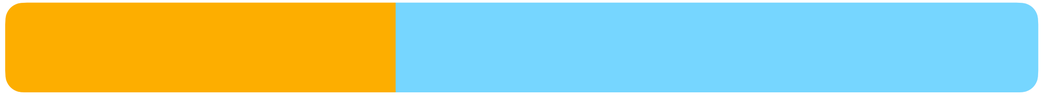
Pure Borrow

Rust-Style Borrowing in Linear Haskell with Purity



- ✦ **Pure & Mutate & Parallelize** *Parallel quicksort*
- ✦ **Just APIs** *Rich features of Haskell*
- ✦ **Metatheory: Why Pure** WIP *Histories for borrows*

This Talk



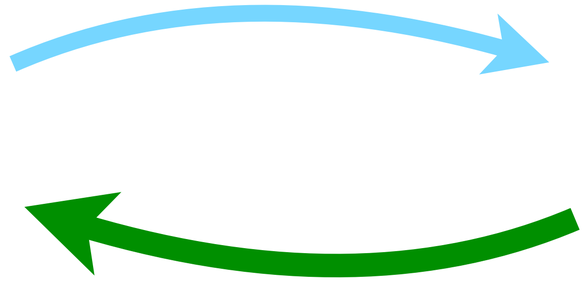
25/66

- ✦ Overview
- ✦ Case Study
- ✦ Metatheory
- ✦ Related Work
- ✦ Conclusion

Overview

Preliminary **Linearity in Linear Haskell**

```
data Ur a where
  Ur :: a → Ur a
```



Unrestricted ω

Default

Linear 1

Opt-in

```
lin :: (Lin ⊸ Ur a) ⊸ Ur a
```

cf. runST :: (∀ s. ST s a) → a

```
class Lin
```

Linearity witness *Spiwack+ '22*

Used arbitrarily finite times *But never in* Ur

Preliminary **Mutable State in Linear Haskell**

Vector **data** Vec a

newVec :: Lin ⊸ [a] ⊸ Vec a

freeVec :: Vec a ⊸ [a]

class Movable a **where**
move :: a ⊸ Ur a

Before Pure Borrow

modifyAt′ :: Vec a ⊸
Int → (a ⊸ a) ⊸ Vec a

readAt′ :: Movable a ⇒
Vec a ⊸ Int → (Ur a, Vec a)

Direct ownership passing!

Preliminary **Example of Basic Linear Haskell**

```
example :: Ur (Int, [Int])
example = lin do°
  v ← newVec [0,1,2]
  v ← modifyAt' v 0 (+ 3); v ← modifyAt' v 2 (+ 5)
  v ← modifyAt' v 0 (× 4)
  (Ur n, v) ← readAt' 0 v
  move (n, freeVec v)
```

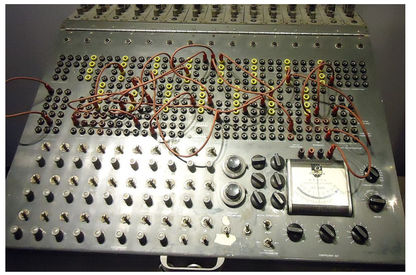
```
do° p ← t; c ≡
  let !p = t in do° c
```

Direct ownership passing!

```
Ur (12, [12, 1, 7])
```

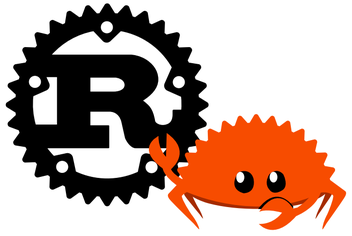
Our Work

Matsushita & Ishii, PLDI '26

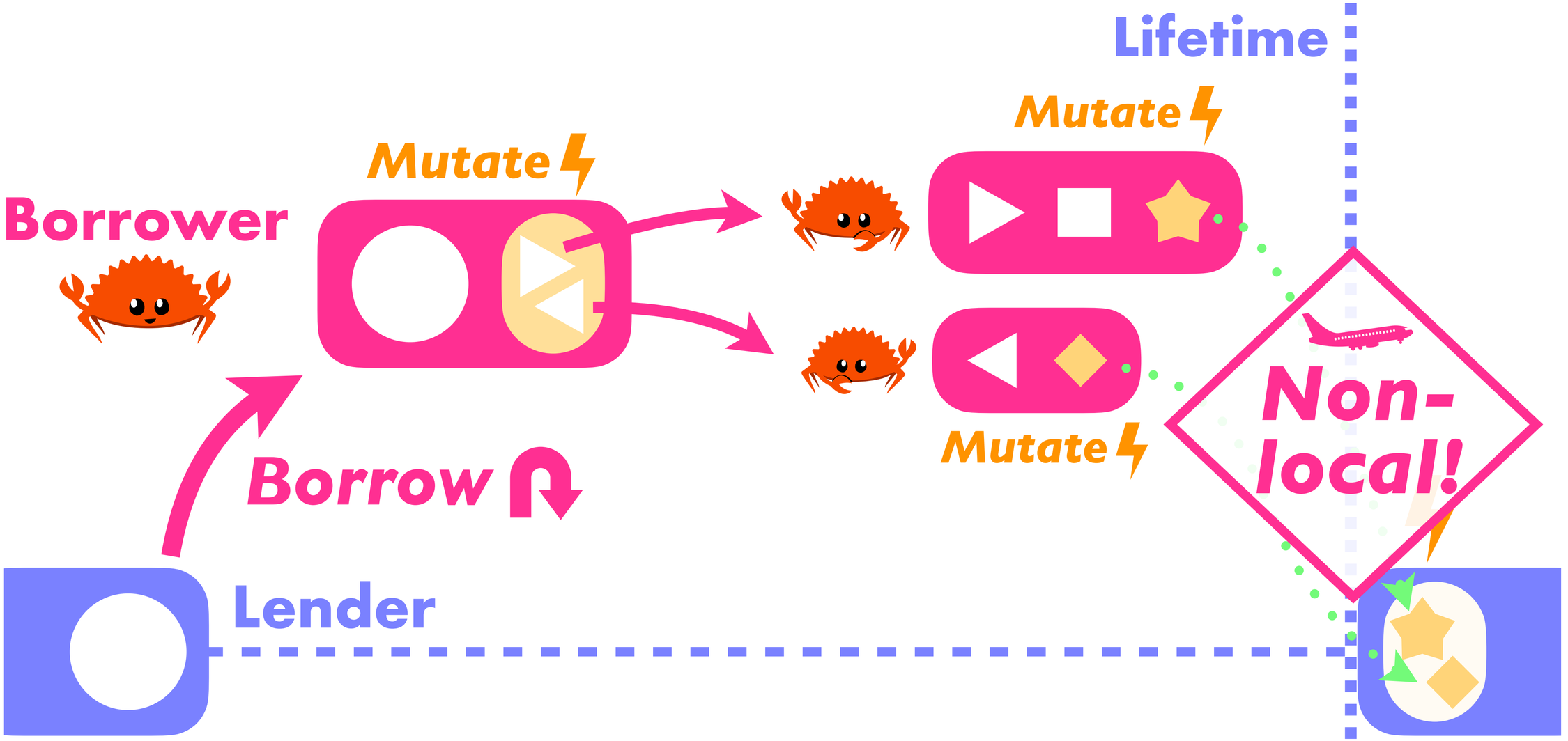


Pure Borrow

Rust-Style Borrowing in Linear Haskell with Purity



Recap **Rust-Style Borrow**

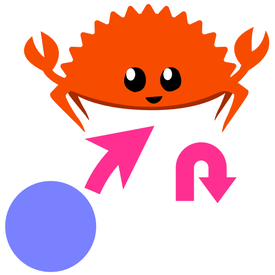


Core APIs

Mutable borrower **Lender**

Borrow $\text{borrow} :: \text{Lin} \multimap a \multimap (\text{Mut}^{\alpha}\ a, \text{Lend}^{\alpha}\ a)$

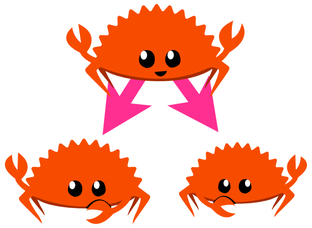
newtype of a



Mutate $\text{modifyAt} :: \text{Mut}^{\alpha}\ (\text{Vec}\ a) \multimap \text{Int} \to (a \multimap a) \multimap \text{BO}^{\alpha}\ (\text{Mut}^{\alpha}\ (\text{Vec}\ a))$

Monad

Split $\text{splitAt} :: \text{Mut}^{\alpha}\ (\text{Vec}\ a) \multimap \text{Int} \to (\text{Mut}^{\alpha}\ (\text{Vec}\ a), \text{Mut}^{\alpha}\ (\text{Vec}\ a))$



α has ended

Reclaim $\text{reclaim} :: \text{End}^{\alpha} \Rightarrow \text{Lend}^{\alpha}\ a \multimap a$

BO Monad

$\mathrm{BO}^{\alpha}\ \mathrm{a}$ ***Computation permitted only before α ends***

Borrower accesses $<$ ***α ends*** $\leq$ ***Lender reclaim***

BO^{α} End^{α}

Pure $\mathrm{runBO} :: \mathrm{Lin} \multimap (\forall\, \alpha.\ \mathrm{BO}^{\alpha}\ (\mathrm{End}^{\alpha} \Rightarrow \mathrm{a})) \multimap \mathrm{a}$

Rank-2 poly

cf. $\mathrm{runST} :: (\forall\, \mathrm{s.}\ \mathrm{ST}^{\mathrm{s}}\ \mathrm{a}) \to \mathrm{a}$

Linearity ensures commutativity

Parallel $\mathrm{parBO} :: \mathrm{BO}^{\alpha}\ \mathrm{a} \multimap \mathrm{BO}^{\alpha}\ \mathrm{b} \multimap \mathrm{BO}^{\alpha}\ (\mathrm{a}, \mathrm{b})$

cf. ST is not parallelizable

Example

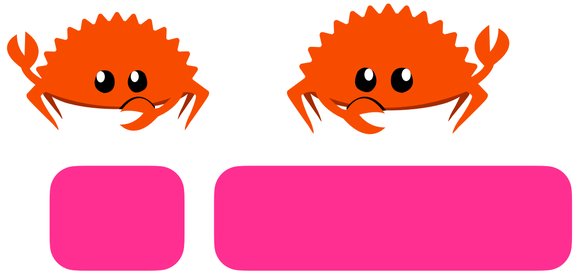
```
example :: Ur [Int]
example = lin do°
  v ← newVec [0, 1, 2]
  runBO do
    let !(mv, lv) = borrow v
    mut_example mv
    pure $ move $
      freeVec $ reclaim lv
                Ur [12, 1, 7]
```

```
mut_example ::
      Mut^α (Vec Int) ⊸ BO^α ()
mut_example mv = do
  mv ← modifyAt mv 0 (+ 3)
  mv ← modifyAt mv 2 (+ 5)
  mv ← modifyAt mv 0 (× 4)
  pure $ consume mv

class Consumable a where
  consume :: a ⊸ ()
```

Example with Parallelization

```
example :: Ur [Int]
example = lin do°
  v ← newVec [0, 1, 2]
  runBO do
    let !(mv, lv) = borrow v
    mut_example mv
    pure $ move $
      freeVec $ reclaim lv
                  Ur [12, 1, 7]
```

```
mut_example mv =
  let !(mv₀, mv₁) = splitAt mv 1 in
  consume <$> parBO
    (do mv₀ ← modifyAt mv₀ 0 (+ 3)
        modifyAt mv₀ 0 (× 4))
    (modifyAt mv₁ 1 (+ 5))
```

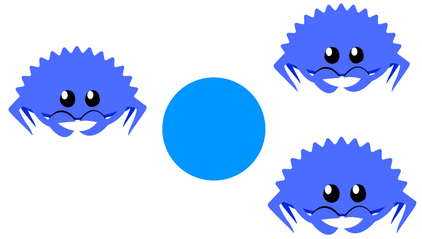


Parallelize!

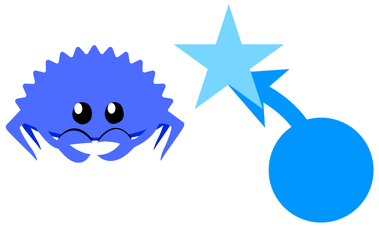
Share

Shared borrower

Share share :: Mut^{α} a ⊸ Ur (Share^{α} a)

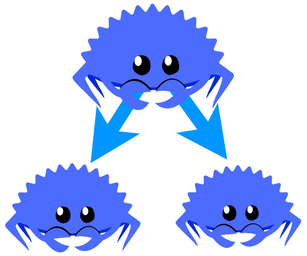


Read copyAt :: Copyable a ⇒
Share^{α} (Vec a) → Int → BO^{α} (Ur a)



class Copyable a **where**
copy :: Share^{α} a → a

Split splitAt :: Share^{α} (Vec a) ⊸
Int → (Share^{α} (Vec a), Share^{α} (Vec a))



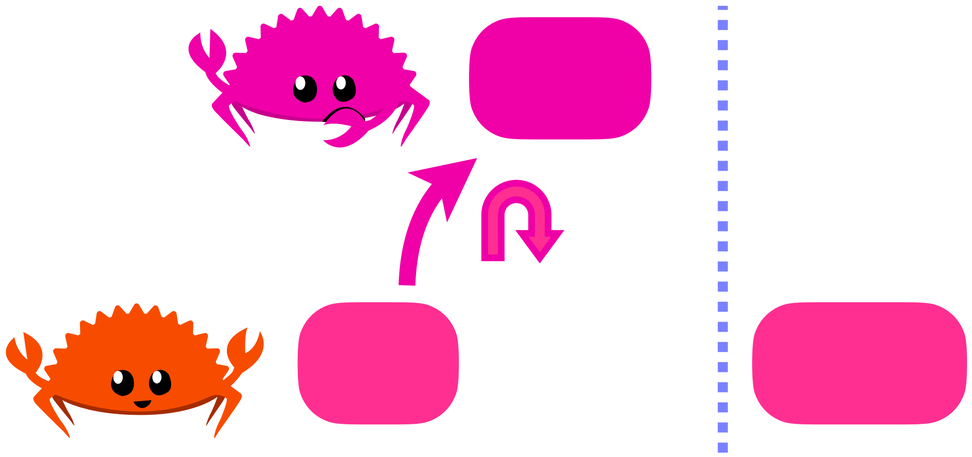
Example

```
example :: Ur (Int, [Int])
example = lin do°
  v ← newVec [0, 1, 2]
  runBO do
    let !(mv, lv) = borrow v
    n ← mut_example mv
    pure $ move $
      (n, freeVec $ reclaim lv)

    Ur (12, [12, 1, 7])
```

```
mut_example :: Mut^α (Vec Int) ⊸ BO^α Int
mut_example mv = do
  let !(mv₀, mv₁) = splitAt mv 1
  (Ur n, mv₁) ← parBO    Can't keep it?
    (do mv₀ ← modifyAt mv₀ 0 (+ 3)
        modifyAt mv₀ 0 (× 4)
        let !(Ur sv) = share mv₀
        copyAt sv 0)
    (modifyAt mv₁ 1 (+ 5))
  let () = consume mv₁; pure n
```

Reborrow

Reborrow



$$\text{reborrowing} :: \text{Mut}^{\alpha}\ a \multimap$$

$$(\forall\, \beta.\ \text{Mut}^{\beta \wedge \alpha}\ a \multimap \text{BO}^{\beta \wedge \alpha'}\ r) \multimap$$

$$\text{BO}^{\alpha'}\ (r, \underline{\text{Mut}^{\alpha}\ a})$$

Lifetime intersection

Example

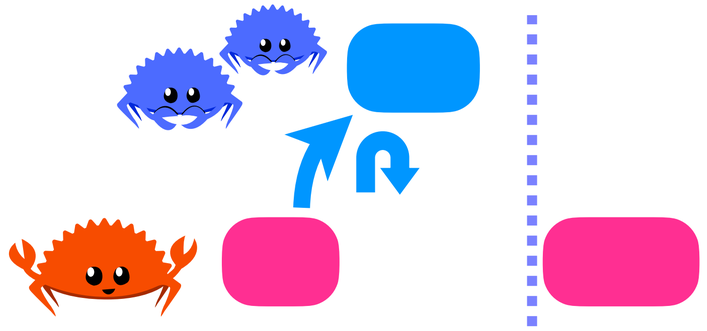
```
example :: Ur (Int, [Int])
example = lin do°
  v ← newVec [0, 1, 2]
  runBO do
    let !(mv, lv) = borrow v
    n ← mut_example mv
    pure $ move $
      (n, freeVec $ reclaim lv)

    Ur (12, [12, 1, 7])
```

```
mut_example :: Mut^α (Vec Int) ⊸ BO^α Int
mut_example mv = do
  ((), mv) ← reborrowing mv λmv →
    let !(mv₀, mv₁) = splitAt mv 1
    consume <$> parBO
      (do mv₀ ← modifyAt mv₀ 0 (+ 3)
          modifyAt mv₀ 0 (× 4))
      (modifyAt mv₁ 1 (+ 5))
  let !(Ur sv) = share mv; copyAt sv 0
```

Temporary Sharing

Temporarily share

$$\begin{aligned}&\text{sharing} :: \text{Mut}^{\alpha}\ a \multimap \\ &\quad (\forall\, \beta.\ \text{Share}^{\beta \wedge \alpha}\ a \to \text{BO}^{\beta \wedge \alpha'}\ r) \multimap \\ &\quad \text{BO}^{\alpha'}\ (r, \text{Mut}^{\alpha}\ a)\end{aligned}$$



```
sharing m k =
  reborrowing m λm →
    let !(Ur s) = share m in k s
```

Read from Mut

$$\begin{aligned}&\text{copyAtMut} :: (\text{Copyable}\ a, \beta \leq \alpha) \Rightarrow \\ &\quad \text{Mut}^{\alpha}\ (\text{Vector}\ a) \to \text{Int} \to \\ &\quad \text{BO}^{\beta}\ (\text{Ur}\ a, \text{Mut}^{\alpha}\ (\text{Vector}\ a))\end{aligned}$$

```
copyAtMut i mv =
  sharing mv $ copyAt i
```

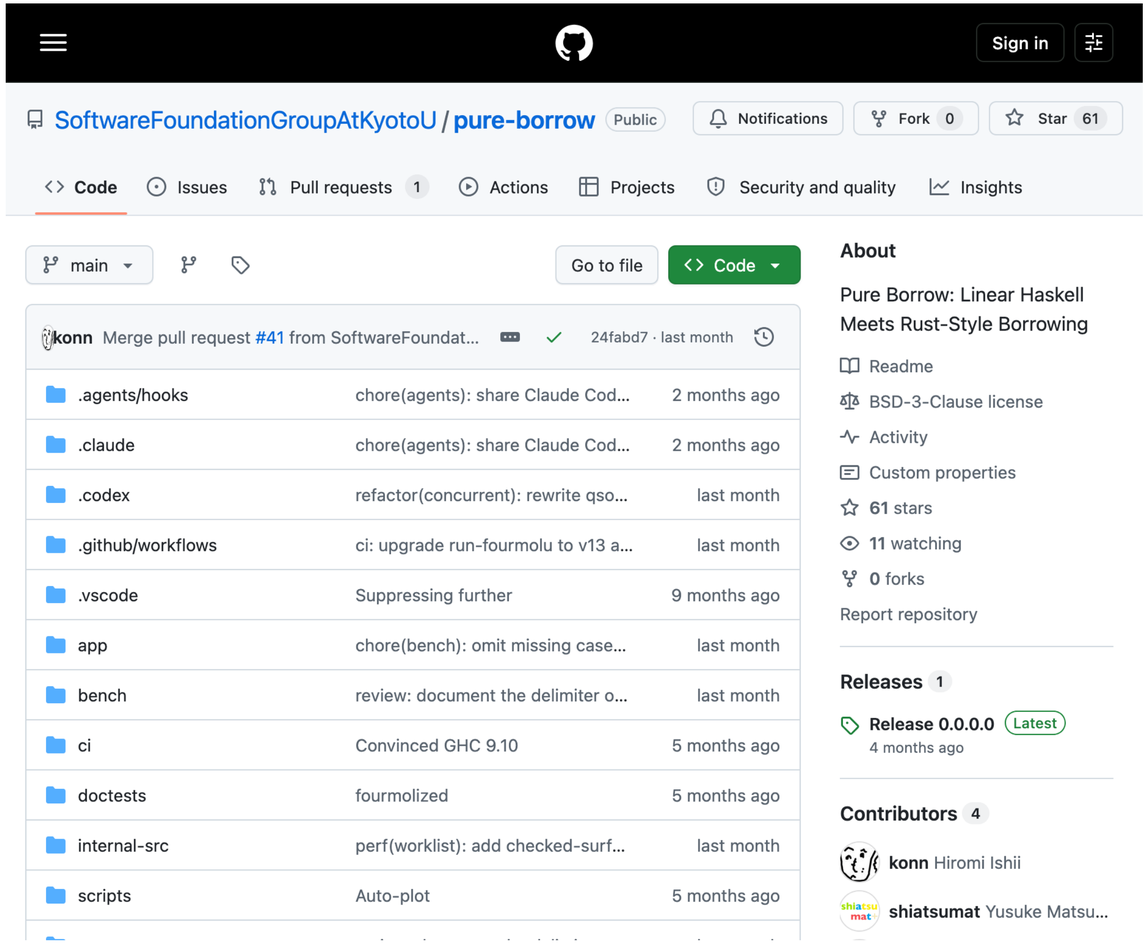
Remark **Internal Implementation**

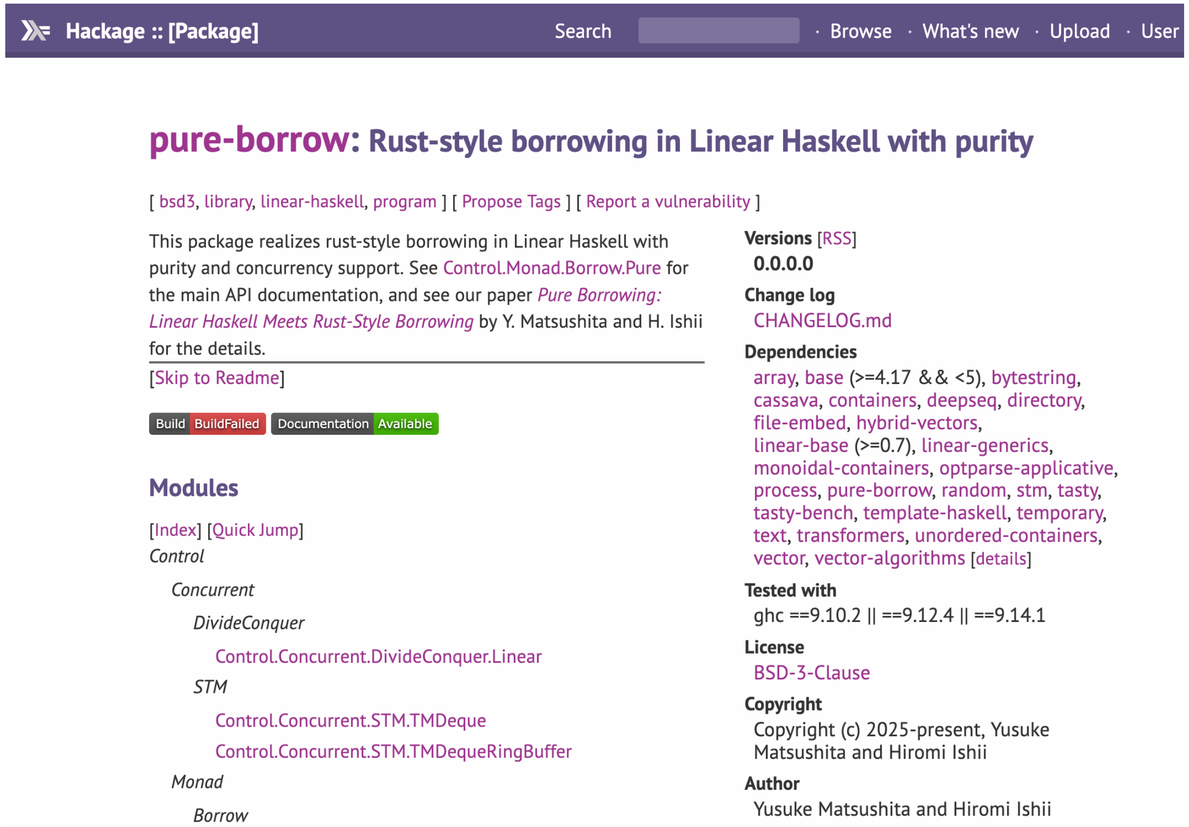
Just unsafe coercions → Efficient

$$\begin{aligned} \text{borrow} &= \text{unsafeCoerce}\ \lambda a \to (a, a) \\ \text{reclaim} &= \text{unsafeCoerce}\ \lambda a \to a \\ \text{share} &= \text{unsafeCoerce}\ \lambda a \to \text{Ur}\ a \end{aligned}$$

Case Study

Pure Borrow is Already There!

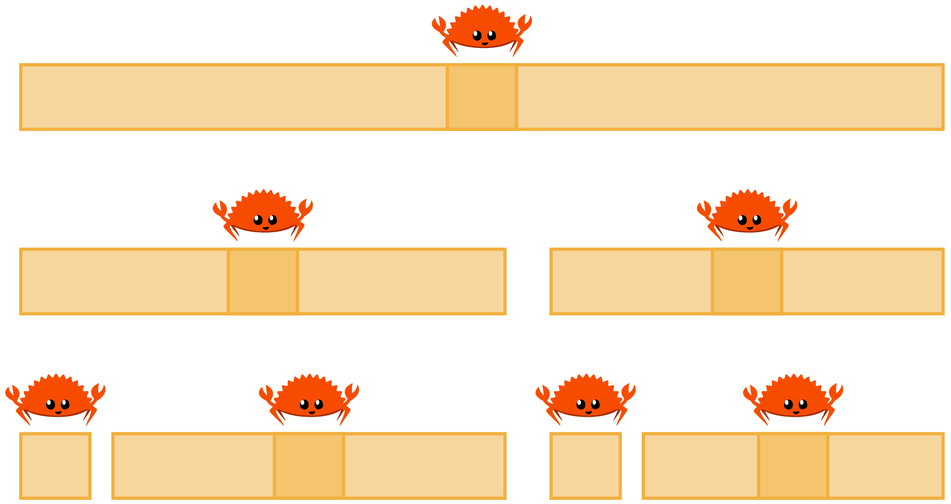




Pure Borrow works on stable GHC

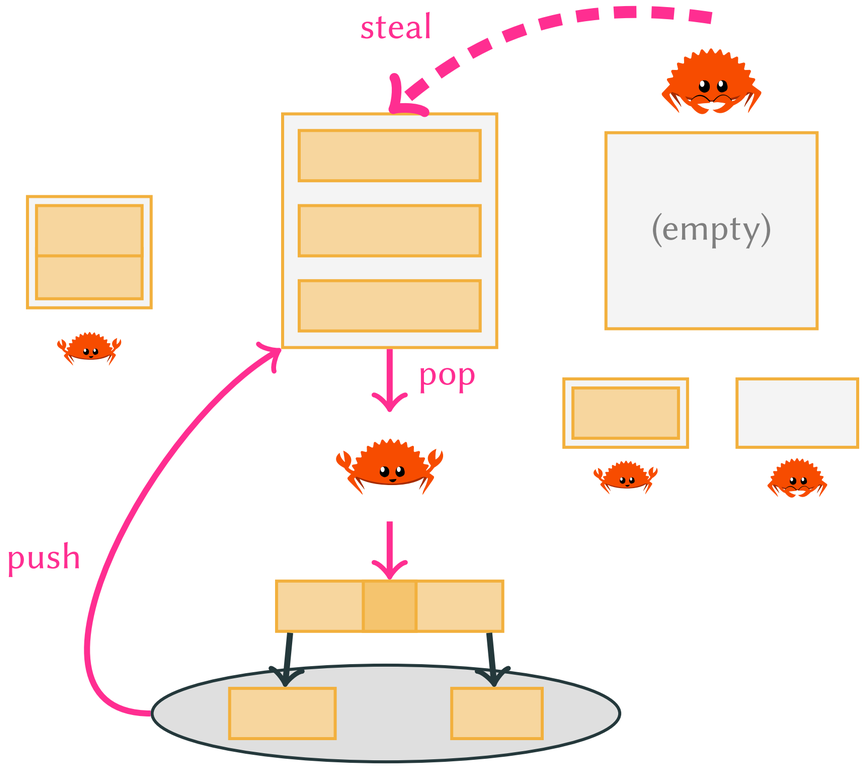
Parallel Quicksort

Pure* + *Mutate* + *Parallel



```
qsort :: Mut^α (Vec Int) ⊸ BO^α ()
qsort v = let (Ur n, v) = size v in
  if n ≤ 1 then pure (consume v) else do
    (Ur a, v) ← copyAt v (n // 2)
    (i, v) ← pivotSwap v a
    let (lo, hi) = splitAt v i
    consume <$> parBO (qsort lo) (qsort hi)
```

Advanced: Work-Stealing Scheduler

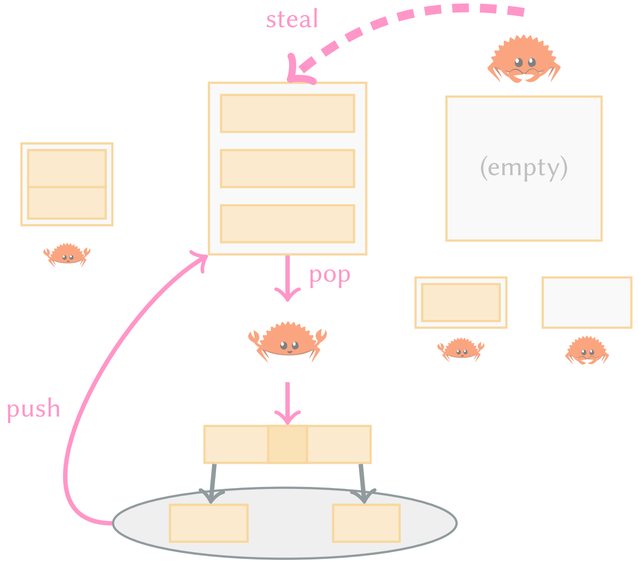


$$\textbf{data}\ \mathrm{DivideConquer}^{\alpha}\ \mathrm{a} = \mathrm{DivideConquer}\ (\forall\,\beta.\ \beta \le \alpha \Rightarrow \mathrm{Mut}^{\beta}\ \mathrm{a} \multimap \mathrm{BO}^{\beta}\ (\mathrm{Res}^{\beta}\ \mathrm{a}))$$

$$\textbf{data}\ \mathrm{Res}^{\beta}\ \mathrm{a} = \mathrm{Done} \mid \mathrm{Cont}\ [\mathrm{Mut}^{\beta}\ \mathrm{a}]$$

$$\mathrm{divideAndConquer} :: \beta \le \alpha \Rightarrow \mathrm{Int} \to \mathrm{DivideConquer}^{\alpha}\ \mathrm{a} \to \mathrm{Mut}^{\alpha}\ \mathrm{a} \multimap \mathrm{BO}^{\beta}\ (\mathrm{Mut}^{\alpha}\ \mathrm{a})$$

Parallel Quicksort by Work Stealing

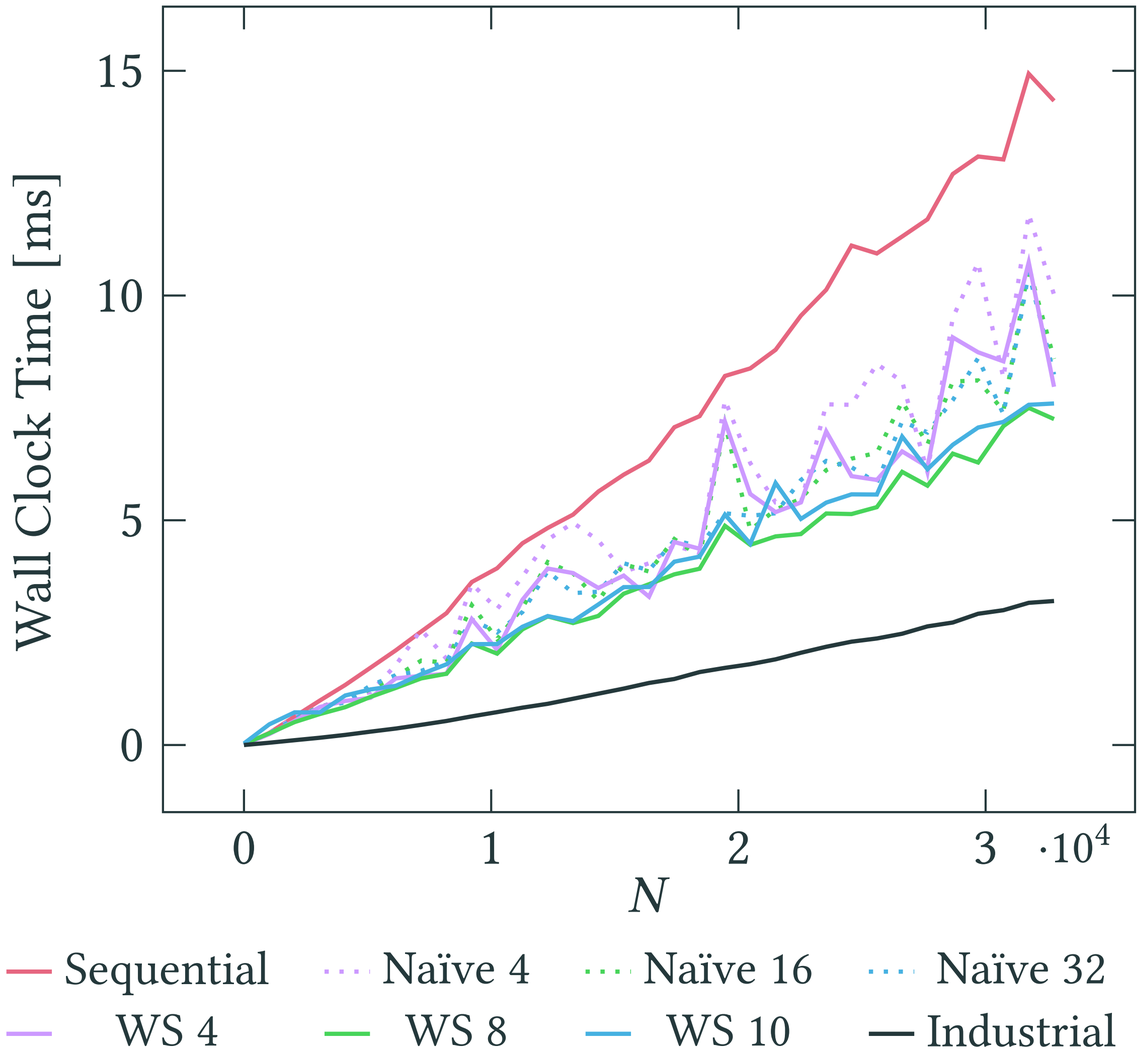



```
data DivideConquer^α a = DivideConquer
        (∀ β. β ≤ α ⇒ Mut^β a ⊸ BO^β (Res^β a))

data Res^β a = Done | Cont [Mut^β a]

divideAndConquer :: β ≤ α ⇒
    Int → DivideConquer^α a →
    Mut^α a ⊸ BO^β (Mut^α a)
```

```
qsortDC :: β ≤ α ⇒ Int →
    Mut^α (Vec Int) ⊸ BO^β (Mut^α (Vector Int))
qsortDC nwork = divideAndConquer nwork $
    DivideConquer λv → case size v of
        (Ur n, v)
            | n ≤ 1 → let () = consume v in pure Done
            | otherwise →
                (Ur a, v) ← copyAtMut v (n // 2)
                (lo, hi) ← pivotSwap v a; pure $ Cont [lo; hi]
```

No thread joining needed → Efficient

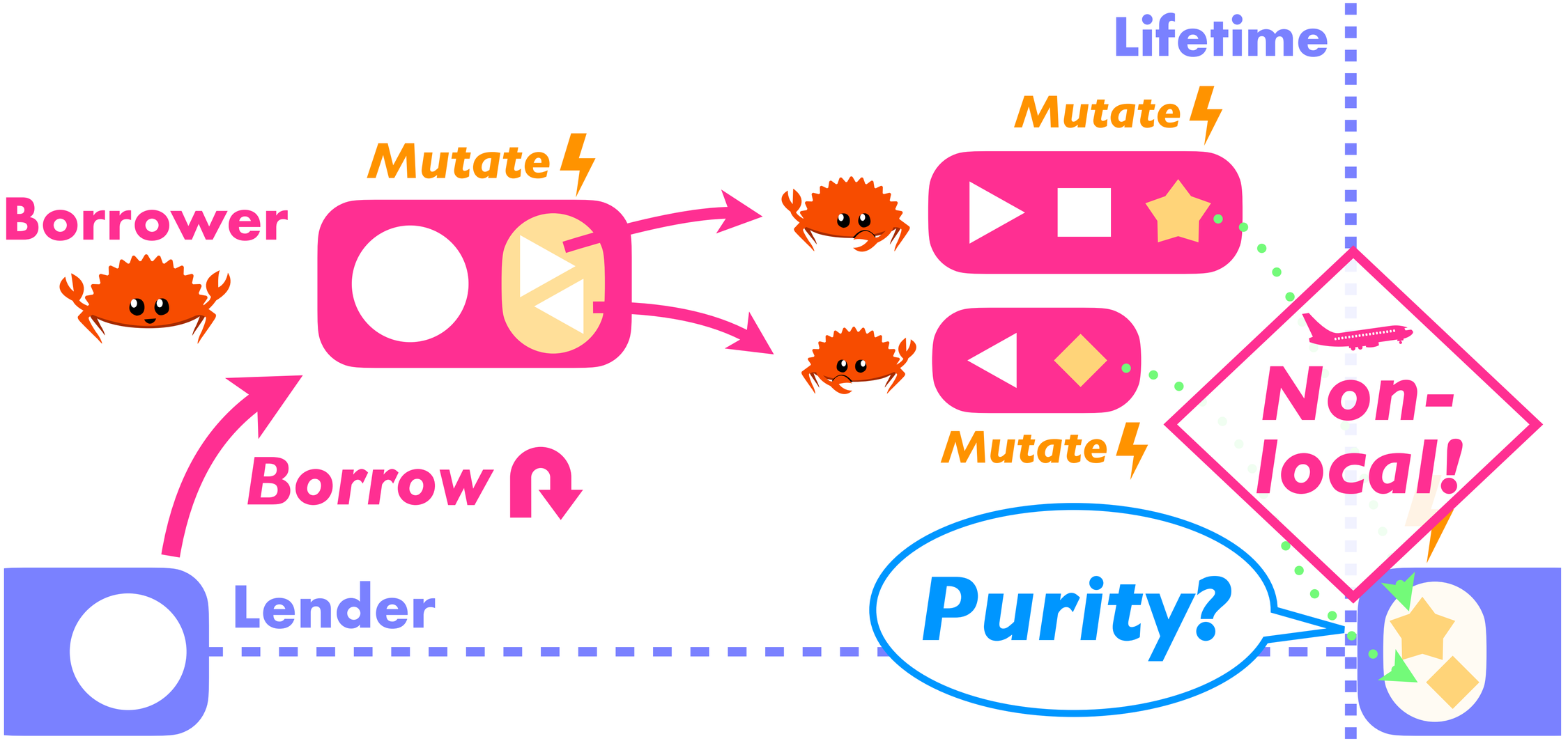
Benchmark Results



Successfully parallelized

Metatheory

Challenge: Purity of Borrowing



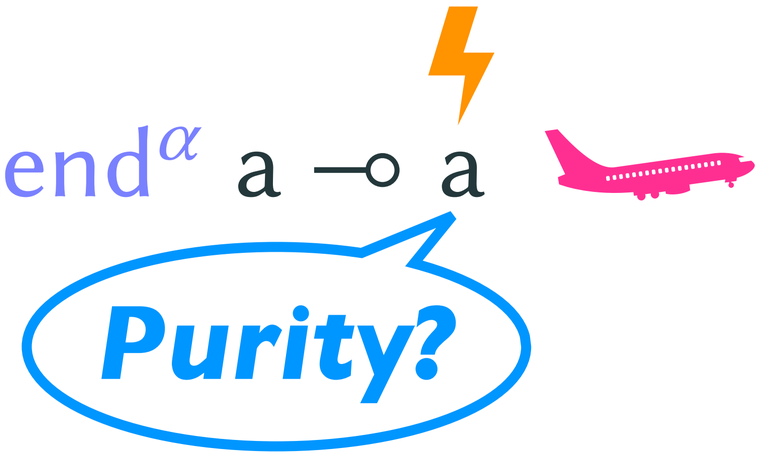
Challenge: Purity of Borrowing

Borrow $\quad \text{borrow} :: \text{Lin} \multimap a \multimap (\text{Mut}^{\alpha}\ a, \text{Lend}^{\alpha}\ a)$

Mutate $\quad \text{modifyAt} :: \text{Mut}^{\alpha}\ (\text{Vec}\ a) \multimap \text{Int} \to (a \multimap a) \multimap \text{BO}^{\alpha}\ (\text{Mut}^{\alpha}\ (\text{Vec}\ a))$

Split $\quad \text{splitAt} :: \text{Mut}^{\alpha}\ (\text{Vec}\ a) \multimap \text{Int} \to (\text{Mut}^{\alpha}\ (\text{Vec}\ a), \text{Mut}^{\alpha}\ (\text{Vec}\ a))$

Reclaim $\quad \text{reclaim} :: \text{End}^{\alpha} \Rightarrow \text{Lend}^{\alpha}\ a \multimap a$



Idea: Histories

- Manage the global history of mutations to borrows for each lifetime
- BO monad: State monad with the history state
- Lifetime end class: Owns frozen history
- reclaim: Restore the updated value from the history

Our Formalization & Metatheory

- ✦ Core calculus + Linear type system for Pure Borrow
- ✦ Two operational semantics
  - ▸ Mutative, with memories & Denotational, based on histories
- ✦ Theorem: Denotational op. sem. is confluent
  - ▸ Full purity: Future work, due to fresh borrow id generation
- ✦ WIP: Mutative & denotational opsems bisimulate?
  - ▸ For well-typed programs & configurations

Reference **Core Calculus Syntax**

Variable $\quad Var \ni x, y, z \qquad$ Data constructor $\quad C$

$$
\begin{aligned}
\text{Operator} \quad op \ ::= \ & (+)\ 2 \mid \cdots \mid (\leq)\ 2 \mid \cdots \\
& \mid \mathsf{par}\ 2 \mid \mathsf{consume}\ 1 \mid \mathsf{move}\ 1 \\
& \mid \mathsf{lin}\ 1 \mid \mathsf{withLin}\ 1 \mid \mathsf{newRef}\ 2 \mid \mathsf{freeRef}\ 1 \\
& \mid \mathsf{newLifetime}\ 2 \mid \mathsf{endLifetime}\ 1 \mid \mathsf{execBO}\ 2 \\
& \mid \mathsf{borrow}\ 2 \mid \mathsf{share}\ 1 \mid \mathsf{copy}\ 1 \mid \mathsf{joinMut}\ 1 \mid \mathsf{reclaim}\ 2
\end{aligned}
$$

$$
\begin{aligned}
\text{Monad constructor} \quad mo \ ::= \ & \mathsf{pure}\ 1 \mid (\gg\!=)\ 2 \mid \mathsf{sexecBO}\ 2 \\
& \mid \mathsf{parBO}\ 2 \mid \mathsf{deref}\ 1 \mid \mathsf{updateRef}\ 2
\end{aligned}
$$

$$
\begin{aligned}
\text{Term} \quad Term \ni t, u \ ::= \ & x \mid \mathbf{let}\ \overline{x = t}\ \mathbf{in}\ u \mid \lambda x \to t \mid t\ u \mid \mathbf{seq}\ x\ t \mid n \\
& \mid C\ \bar{t} \mid \mathbf{case}\ t\ \mathbf{of}\ \{\overline{C\ \bar{x} \to u}\} \mid op\ \bar{t} \mid mo\ \bar{t}
\end{aligned}
$$

Reference **Core Calculus Type Syntax**

Lifetime variable α, β Lifetime id ι

Lifetime $\alpha, \beta ::= \mathrm{al}\,\iota \mid \mathrm{static} \mid \alpha \wedge \beta \mid \alpha$

Multiplicity variable π, μ Multiplicity $\pi, \mu ::= 1 \mid \omega \mid \prod \overline{\pi}$

Type variable X, Y Variable $x ::= \alpha \mid \iota \mid \pi \mid X$

Type constructor F Borrower constructor $B ::= \mathrm{Mut} \mid \mathrm{Share}$

Type $\mathit{Type} \ni T, U ::= \forall x.\, T \mid X \mid T \to_\pi U \mid F\,\bar{T} \mid \mathsf{Int} \mid \mathsf{Lin}$
$\mid \mathsf{Ref}\,T \mid \mathrm{Now}^\alpha \mid \mathrm{End}^\alpha \mid B^\alpha\,T \mid \mathrm{Lend}^\alpha\,T \mid \mathrm{BO}^\alpha\,T$

$T \multimap U \triangleq T \to_1 U$ $T \to U \triangleq T \to_\omega U$

Data type declaration $\mathbf{data}\ F\,\bar{X}\ \mathbf{where}\ \overline{C :: \overline{T \to_\pi F\,\bar{X}}}$

Typing context $\Gamma, \Delta ::= \overline{x ::_\pi T}$

Related Work

Recap **High-Level Comparison**

	Pure 	Mutate	Borrow	Parallel	Leak-free
Pure Only	●	○		●	
ST Monad	●	●	○	○	○
Rust	○	●	●	●	○
Basic Linear Haskell	●	●	○	●	●
Pure Borrow	●	●	●	●	●

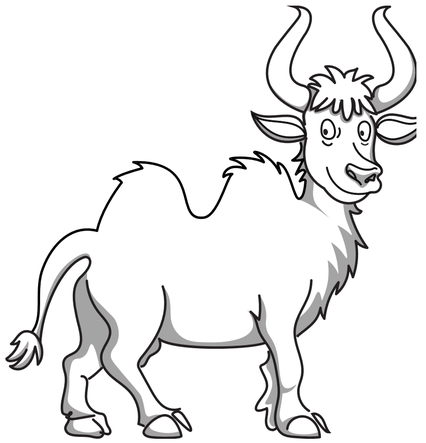
Comparison with Monads

	Pure	Mutate	Borrow	Scope	Parallel	Leak-free
IO Monad *Peyton Jones+ '93*	○	●	○	○	●	●
ST Monad *Launchbury+ '94*	●	●	○	○	○	○
Monadic Regions *Fluet+ '04*	○	●	○	●	●	◐
RIO Monad *Bernardy+ '18*	○	●	○	○	●	●
Pure Borrow	●	●	●	●	●	●

OxCaml

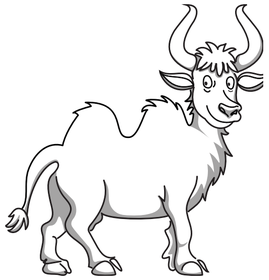
Lorenzen+ ’24, Georges+ ’25

- OCaml + Safe low-level control
  - Ownership, memory-usage, thread safety, …
- Lightweight mode annotation
  - Easy to add to legacy OCaml code
  - Coarse lifetime modes: global vs local
- No strict purity



Linearity vs Uniqueness

Marshall+ '22



Linearity many ≤ once

Uniqueness unique ≤ aliased

Comonad

Monad

Non-linearity Ur

Comonad

Ur a ⊸ a

Ur a ⊸ Ur (Ur a)

Time-delimited aliasing Mut^{α} & Lend^{α}

Monad-like

a ⊸ (Mut^{α} a, Lend^{α} a)

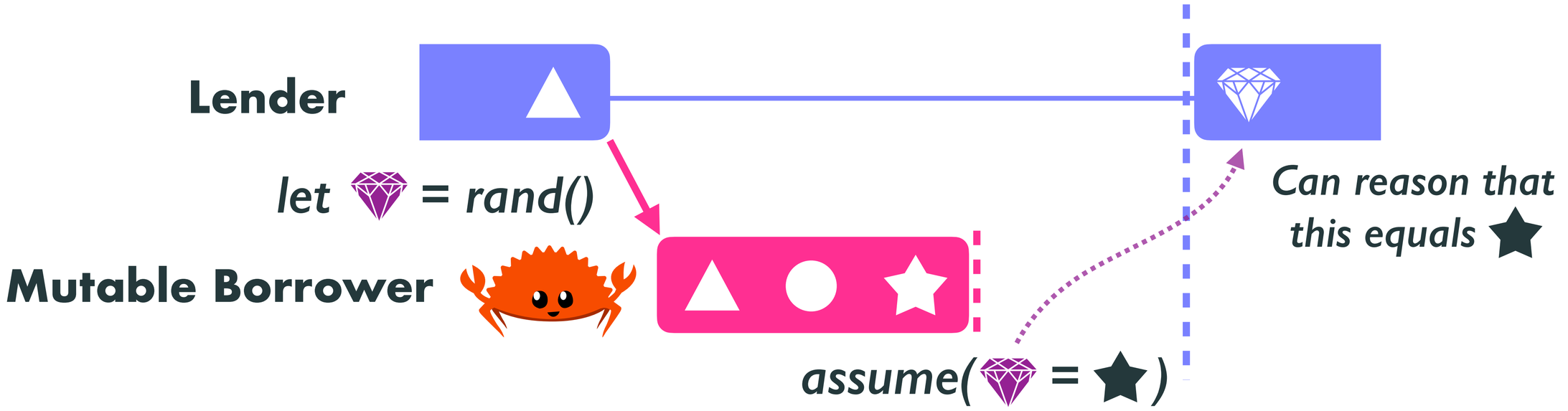
Mut^{α} (Mut^{β} a) ⊸ $\mathrm{Mut}^{\alpha \wedge \beta}$ a

RustHorn

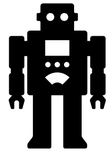
Matsushita+ '20 &'21

- Verify Rust programs as functional code
- Model mutable borrows by prophecy
  - Prophecy [Abadi+ '88]: Fetches info about the future

Final result of the borrower's mutation



Example of RustHorn'' Translation

Rust 

Functional 

```
fn take_max<α>
     (ma: &α mut i32, mb: &α mut i32)
      -> &α mut i32 {
  if *ma >= *mb { ma } else { mb }
}

fn test(mut a: i32, mut b: i32) {
  *take_max(&mut a, &mut b) += 7;
  assert!((a - b).abs() >= 7);
}
```

```
let take_max (a, a') (b, b') =
  if a >= b then
       (assume(b' = b); (a, a'))
  else (assume(a' = a); (b, b'))

let test a b =
  let a' = rand() in let b' = rand() in
  let (r, r') =
    take_max (a, a') (b, b') in
  assume(r' = r + 7);
  assert(abs (a' - b') >= 7);
```

Prophecies vs Histories

Prophecies 

- Concise & local
  - Ideal for **verification**
- Non-determinism
- Precise resolution needed
  - Otherwise **over-approximate**

Histories 

- Rather bulky & global
- Deterministic
- Robust
  - Ideal for the **purity** proof

Conclusion

Summary: Pure Borrow

- **Linear Haskell + Rust-style borrowing**
  - Pure & Mutate w/o ownership passing & Parallelize
  - BO monad for lifetime constraints
- Case study: Parallel quicksort
- Metatheory: Purity by history-based model

Future Work

- **Logical relations** like RustBelt [Jung+ '18] ?
  - Using separation logic [O'Hearn+ '01] such as **Iris** [Jung+ '15] ?
- **Denotational semantics?**
  - Fresh **borrow id** generation, as in ν**-calculus** [Pitts+ '93]
  - **Purity** & **Equational theory**?
  - Prove **soundness**? Using **logical relations**?
- **BO monad transformer?**

Thank you!