

Tracking Borrows with Regular Expressions

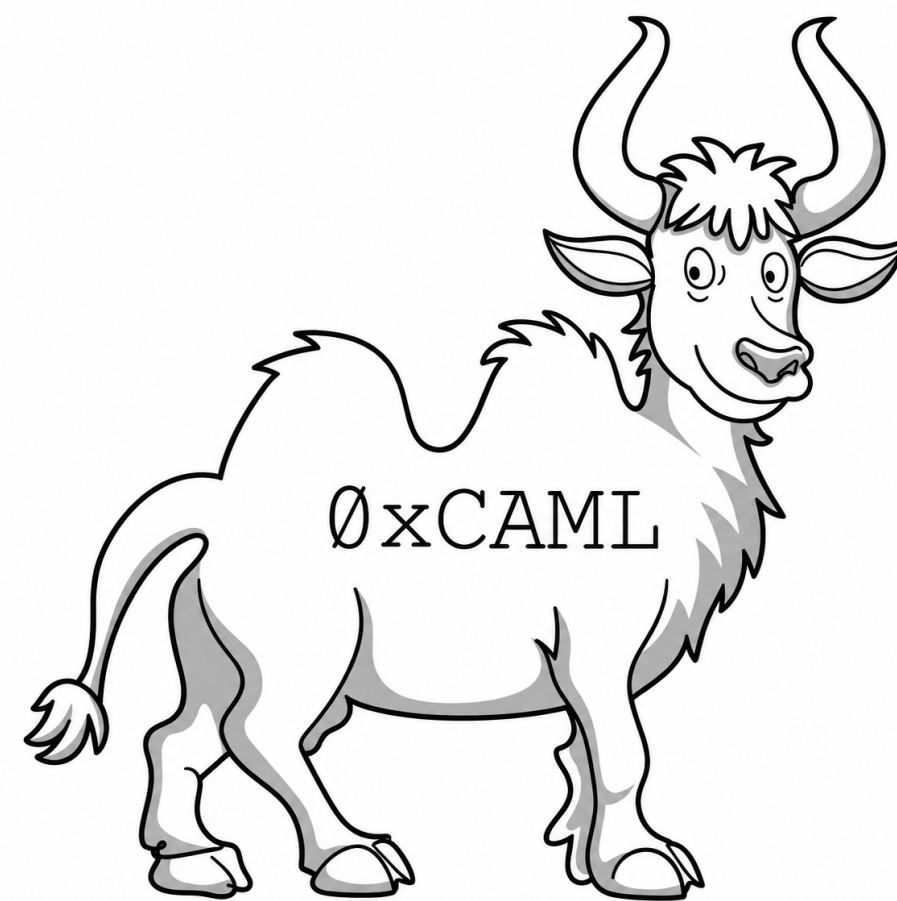
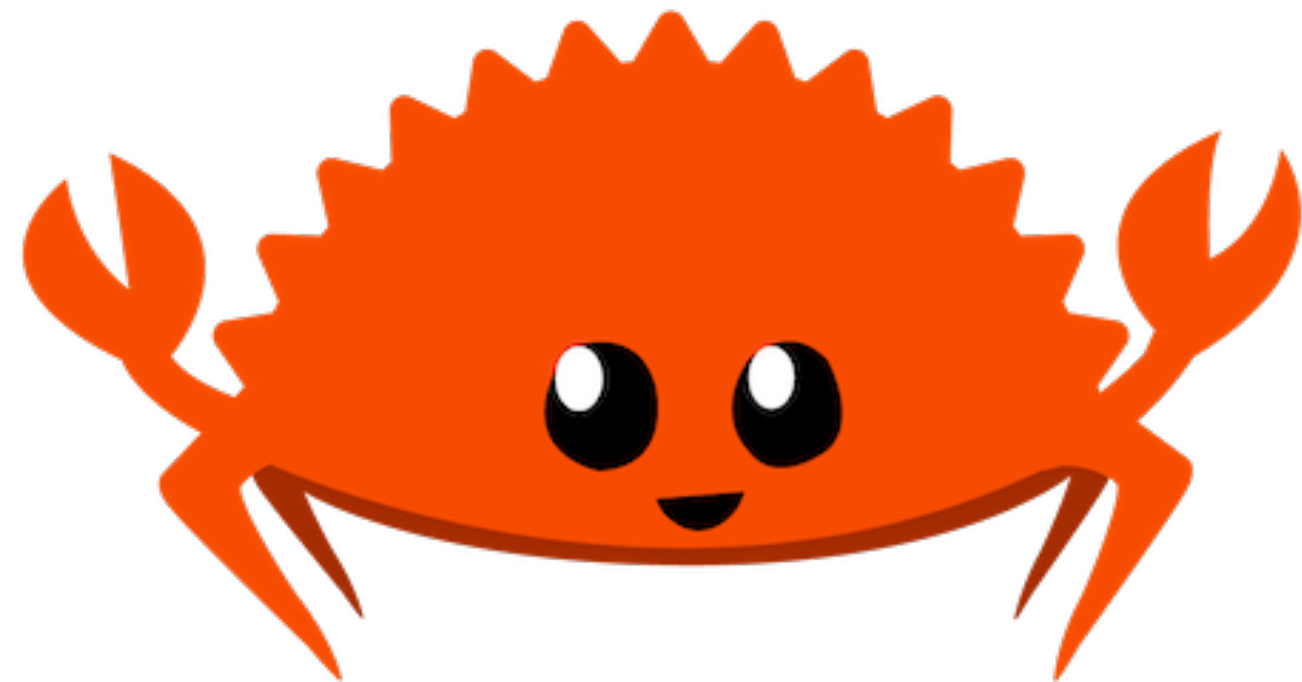
Todd Nowacki Sam Blackshear John C. Mitchell Shaz Qadeer Ilya Sergey



OOPSLA 2026

Borrows and Borrow Tracking

- *Ownership* discipline for safe memory management: every value has a unique owner
- *Borrows* — references with temporary access to values without ownership transfer
- *Borrow tracking* — statically ensuring that no borrow outlives its value's owner



Move Language



- Rust-inspired ownership: memory safety without garbage collection
- Borrow propagation across call boundaries
- Simpler than Rust: no lifetimes, no traits, no references inside records
- Compiles to LLVM-style bytecode (MVIR): basic blocks, jumps, explicit temporaries

Move: A Language With Programmable Resources

Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, Runtian Zhou*

Abstract. We present *Move*, a safe and flexible programming language for the Libra Blockchain [1][2]. *Move* is an executable bytecode language used to implement custom transactions and smart contracts. The key feature of *Move* is the ability to define custom *resource types* with semantics inspired by linear logic [3]: a resource can never be copied or implicitly discarded, only moved between program storage locations. These safety guarantees are enforced statically by *Move*'s type system. Despite these special protections, resources are ordinary program values — they can be stored in data structures, passed as arguments to procedures, and so on. First-class resources are a very general concept that programmers can use not only to implement safe digital assets but also to write correct business logic for wrapping assets and enforcing access control policies. The safety and expressivity of *Move* have enabled us to implement significant parts of the Libra protocol in *Move*, including Libra coin, transaction processing, and validator management.

Move Language

- Rust-inspired ownership: memory safety without garbage collection
- Borrow propagation across call boundaries
- Simpler than Rust: no lifetimes, no traits, no references inside records
- Compiles to LLVM-style bytecode (MVIR): basic blocks, jumps, explicit temporaries

This work: a new simple approach for Borrow Tracking in Move

Move: A Language With Programmable Resources

Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, Runtian Zhou*

Abstract. We present *Move*, a safe and flexible programming language for the Libra Blockchain [1][2]. *Move* is an executable bytecode language used to implement custom transactions and smart contracts. *Move* enforces safety through a type system that guarantees linear memory usage and prevents copying of program values to arbitrary storage locations. These safety guarantees are enforced statically by *Move*'s type system. Despite these special protections, resources are ordinary program values — they can be stored in data structures, passed as arguments to procedures, and so on. First-class resources are a very general concept that programmers can use not only to implement safe digital assets but also to write correct business logic for wrapping assets and enforcing access control policies. The safety and expressivity of *Move* have enabled us to implement significant parts of the Libra protocol in *Move*, including Libra coin, transaction processing, and validator management.

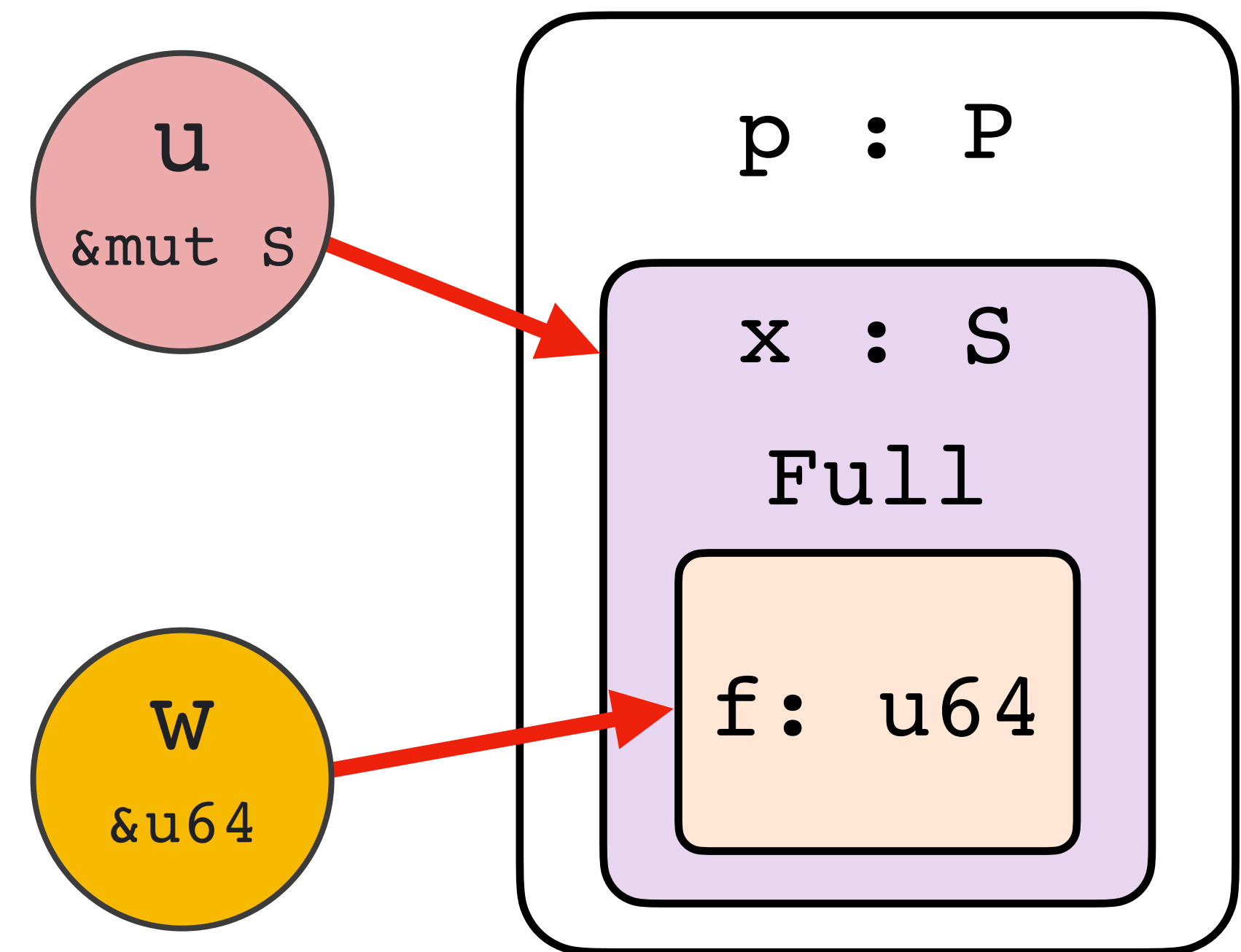
Borrow Tracking in Move



```
enum S { Full {f: u64}, Empty }  
struct P { x: S }
```

```
// Initially p.x = S::Full {f: 0}
```

```
w = &p.x.f;  
u = &mut p.x;  
*u = S::Empty;  
read(w);
```



Borrow Tracking in Move

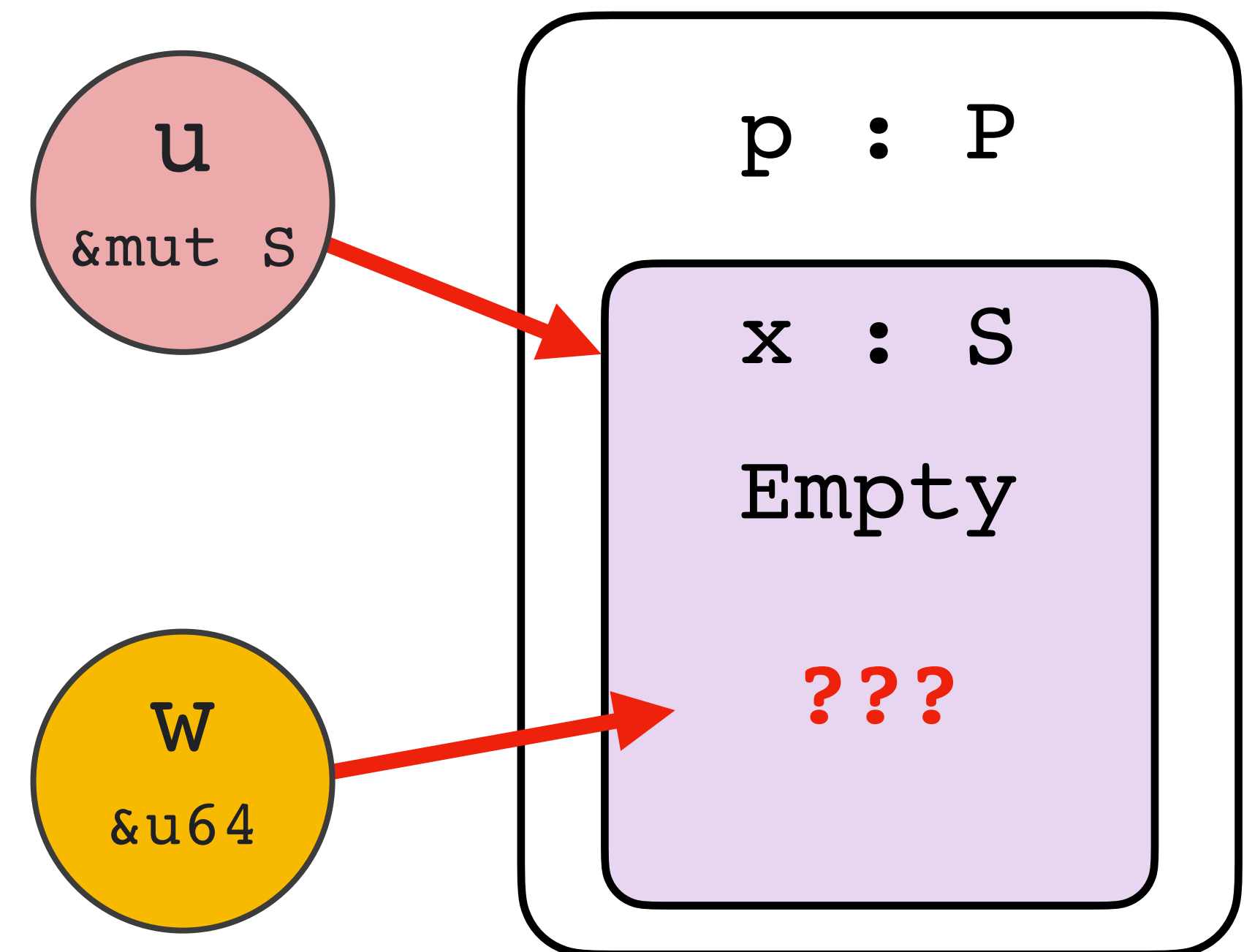
```
enum S { Full {f: u64}, Empty }  
struct P { x: S }
```

```
// Initially p.x = S::Full {f: 0}
```

```
w = &p.x.f;  
u = &mut p.x;
```

```
*u = S::Empty;
```

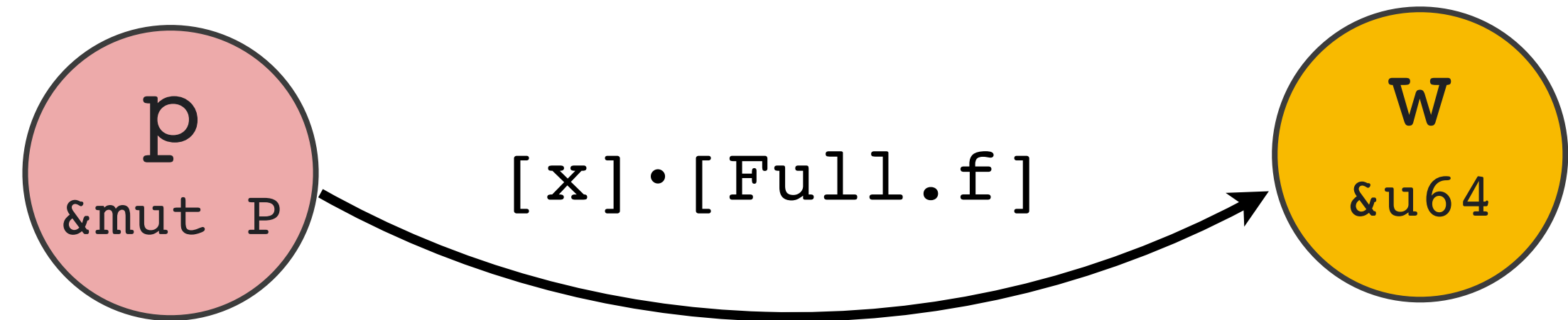
```
read(w);
```



Reference Paths as Regexes



```
enum S { Full {f: u64}, Empty }  
struct P { x: S }
```



```
w = &p.x.f;  
u = &mut p.x;
```

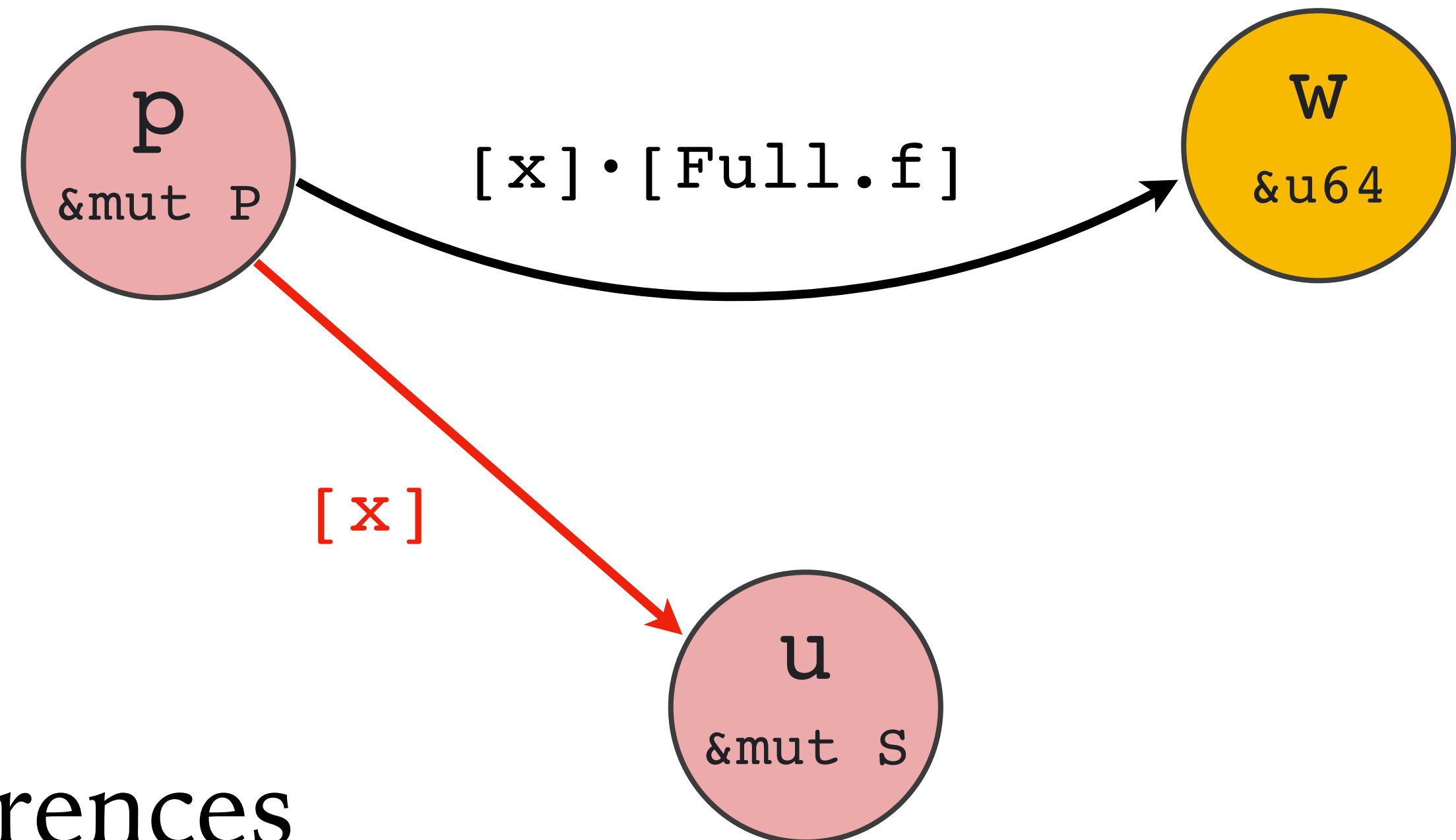
Recall: no references within data structures

An edge is a **path** between references

Reference Paths as Regexes

```
enum S { Full {f: u64}, Empty }  
struct P { x: S }
```

```
w = &p.x.f;  
u = &mut p.x;
```



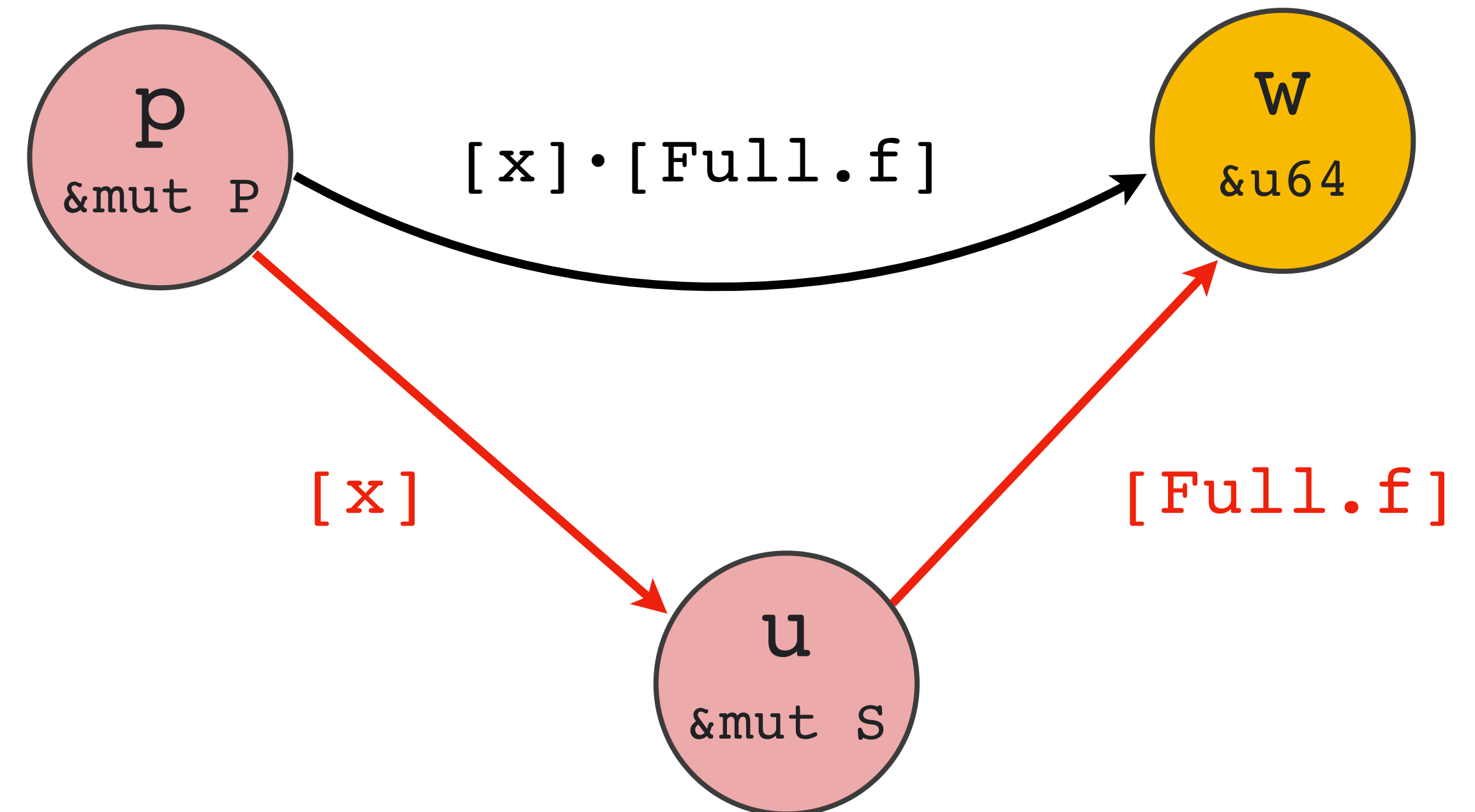
An edge is a **path** between references

Field Borrows are Regex Derivatives

Taking a derivative = “stripping the prefix”

$$\text{path}(u, w) = \delta([x \cdot \text{Full.f}], x) = [\text{Full.f}]$$

```
w = &p.x.f;  
u = &mut p.x;
```



Write Safety as Regex Emptiness Check

self-aliases are ok

Idea: check for outgoing paths: $\forall v, \mathcal{L}(\text{paths}(u, v)) \subseteq \{\epsilon\}$

```
w = &p.x.f;  
u = &mut p.x;  
*u = S::Empty;  
read(w);
```



Borrow Tracking across Calls

```
fn borrow_point(b: &mut Box): &mut Point;
```

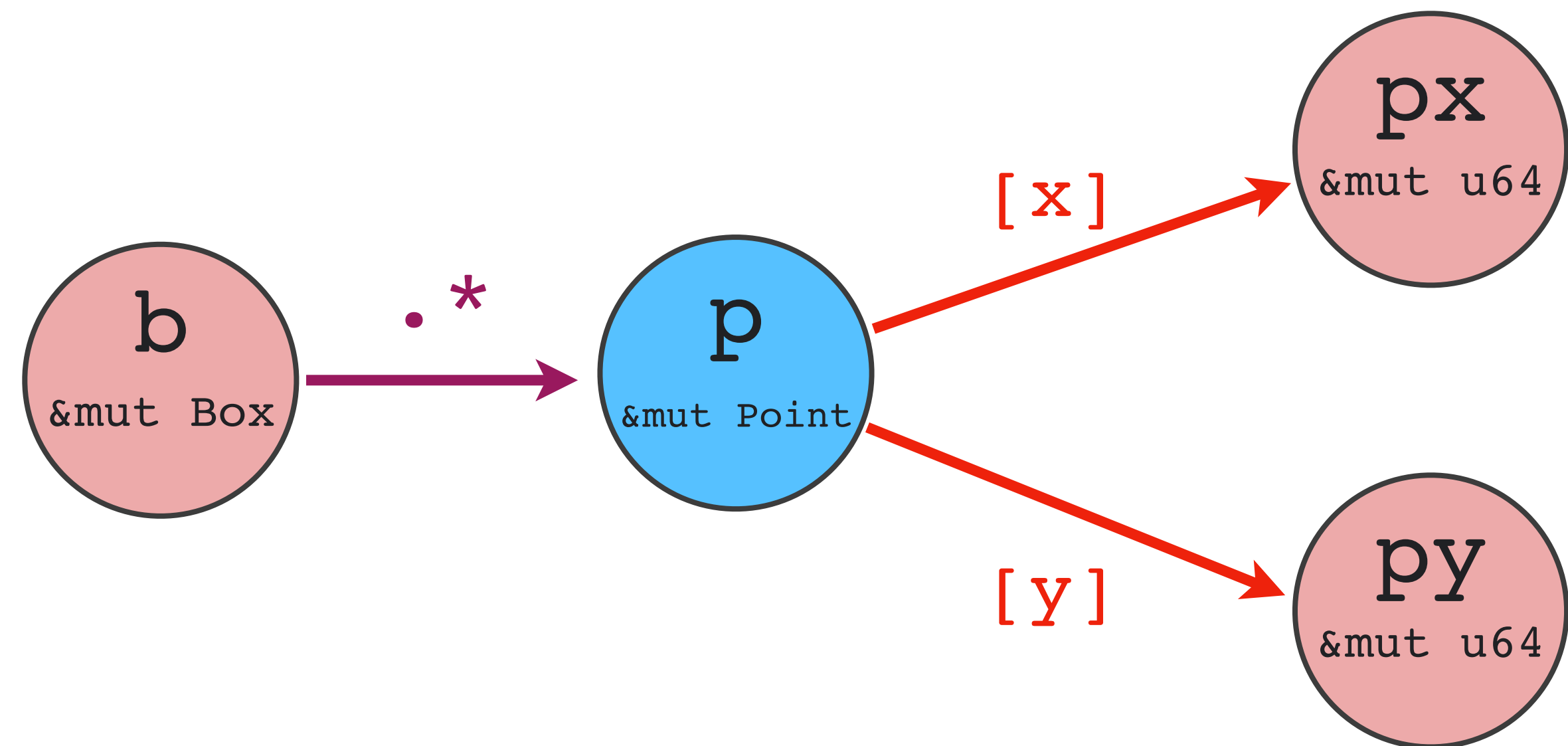
```
p = borrow_point(b);
```

```
px = &mut p.x;
```

```
py = &mut p.y;
```

```
*px = 5;
```

```
*py = 8;
```

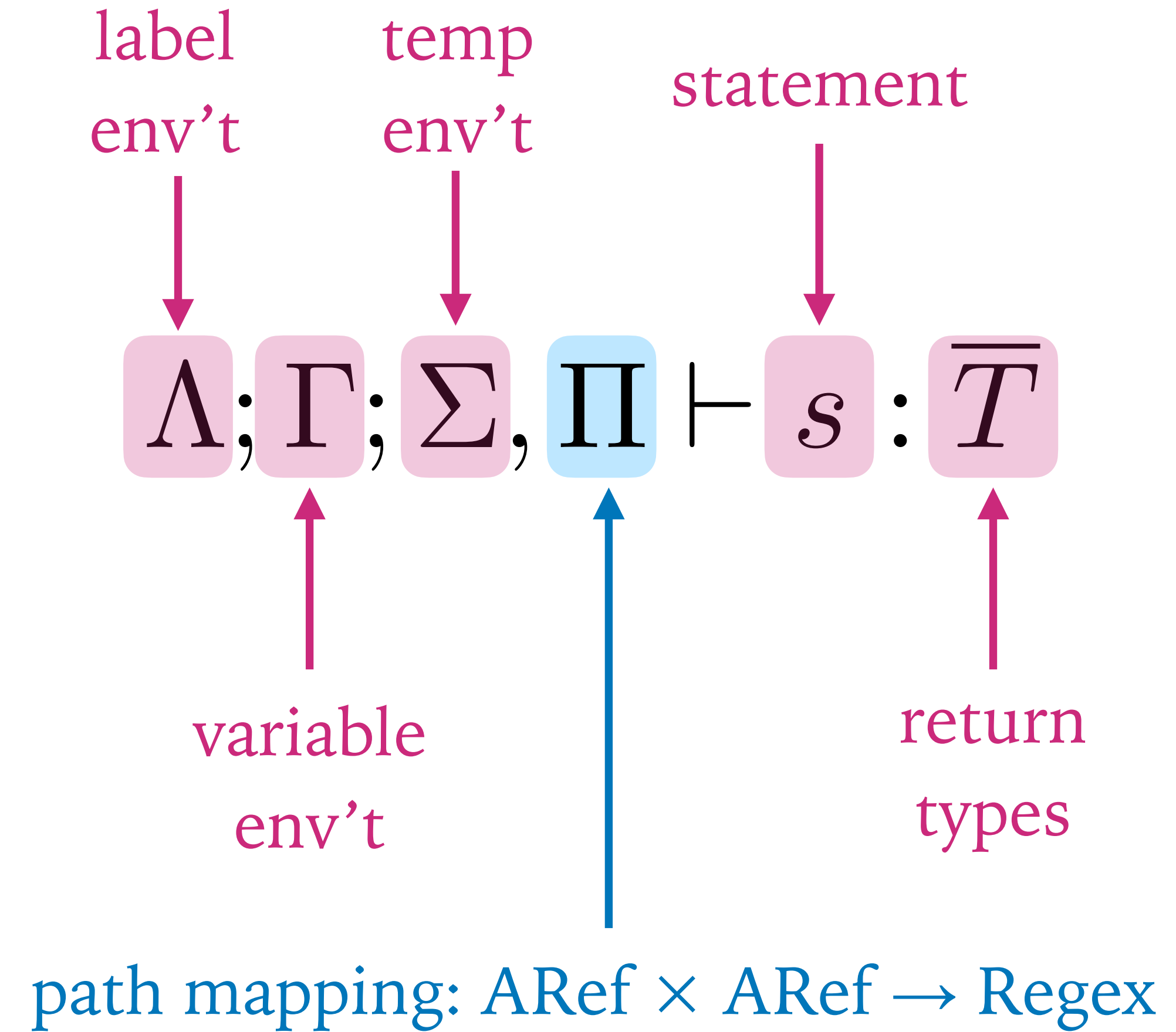


Writes to disjoint paths are acceptable!

Metatheory: MoveLight

Types	$\tau ::= \text{u64} \mid \text{bool} \mid \text{unit} \mid \{ \overline{f : \tau} \}$ $T ::= \text{basic}(\tau) \mid \text{ref}(\tau, \rho, k)$ $k ::= \text{I} \mid \text{M}$	(basic types) (move types) (borrow kind)
Abstract references	$\rho ::= \rho_0 \mid \rho_n \mid \rho_x$	(root, allocated, parameter)
Expressions	$e ::= \text{copy}(x) \mid \text{move}(x) \mid \text{borrow}_k(x) \mid n \mid b$ $\mid \text{borrow-field}_k(a, \tau, f) \mid \text{read-ref}(a)$ $\mid \text{freeze}(a) \mid \text{pack}(id, \overline{f \mapsto a})$ $\mid a_1 \oplus a_2$	(variable use, literals) (field borrow, deref) (freeze, struct pack) (binary operation)
Statements	$s ::= \text{let } a = e; s \mid \overline{f \mapsto a} = \text{unpack}(b); s$ $\mid \overline{a} = g(\overline{b}); s \mid x := a; s$ $\mid *a_1 := a_2; s \mid \text{release}(a); s$ $\mid \text{ret}(\overline{a}) \mid \text{jump}(L) \mid \text{branch}(a, L_1, L_2) \mid \text{abort}(a)$	(bind, unpack) (call, assign) (write-ref, release) (terminal)

formalization-friendly language between Move and MVIR



Type Soundness

$$\Lambda; \Gamma; \Sigma, \Pi \vdash s : \bar{T}$$

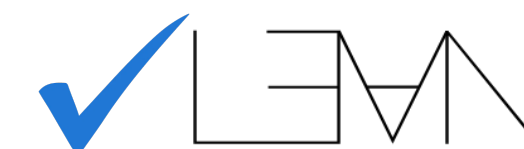
Preventable errors	Description
danglingRef	read/write through freed reference
uninitializedVar	access to moved/uninitialised variable
uninitializedSite	access to uninitialised site
typeMismatch	runtime type mismatch
arityMismatch	wrong number of arguments
unknownFunction	call to undefined function
unknownLabel	jump to undefined label
invalidFieldAccess	field not present in record
Acceptable errors	
divisionByZero	division by zero
outOfFuel	execution step limit exceeded
aborted	explicit abort

runtime error classification

Theorem (Type Soundness): If function f is well-typed, all its callees satisfy their type signatures, then no execution of f from a well-formed initial state produces a preventable error.

```

theorem type_soundness (f : FunDef) (lenv : LabelEnv) (enumEnv : EnumEnv)
  (funEnv : AssocMap Id FunDef) (args : List Value) (heap : Heap)
  (htyped : typecheck_fun f lenv enumEnv) (hfunEnv : ∀ fname fdef, ...)
  (ha : SoundnessAssumptions f lenv enumEnv funEnv heap args)
  (e : RuntimeError) (hna : ¬e.isAcceptable) :
  ∀ n, run n (initState f funEnv args) ≠ .error e
  
```



Type Soundness at Scale

```
fn check_fun(f: &FunDef, lenv: &LabelEnv, enum_env: &EnumEnv) -> bool {...}
```

differentially tested
on 156 litmus tests

```
def check_fun f lenv enumEnv : Bool := ...
```

```
theorem check_fun_sound :  $\forall$  f lenv enumEnv,  
  check_fun f lenv enumEnv = true  $\rightarrow$   
  typecheck_fun f lenv enumEnv
```

sound executable
type checking

```
theorem type_soundness (f : FunDef) (lenv : LabelEnv) (enumEnv : EnumEnv)  
  (funEnv : AssocMap Id FunDef) (args : List Value) (heap : Heap)  
  (htyped : typecheck_fun f lenv enumEnv) (hfunEnv :  $\forall$  fname fdef, ...)  
  (ha : SoundnessAssumptions f lenv enumEnv funEnv heap args)  
  (e : RuntimeError) (hna :  $\neg$ e.isAcceptable) :  
   $\forall$  n, run n (initState f funEnv args)  $\neq$  .error e
```

Key Take-Aways

- **Regex operations** give natural treatment for borrow tracking in types
- The approach works in a **production language**: Move
- **Sound executable type checking** is a way to typing adequacy at scale
- The POPLMark Challenge is Dead

github.com/ilyasergey/lean-move

Key Take-Aways

- Regex operations give natural treatment for borrow tracking in types
- The approach works in a production language: Move
- Sound executable type checking is a way to typing adequacy at scale
- The POPLMark Challenge is Dead

github.com/ilyasergey/lean-move

Mechanized Metatheory for the Masses: The POPLMARK Challenge

Brian E. Aydemir¹, Aaron Bohannon¹, Matthew Fairbairn², J. Nathan Foster¹,
Benjamin C. Pierce¹, Peter Sewell², Dimitrios Vytiniotis¹, Geoffrey
Washburn¹, Stephanie Weirich¹, and Steve Zdancewic¹

¹ Department of Computer and Information Science, University of Pennsylvania

² Computer Laboratory, University of Cambridge

Subversion Revision: 171

Document generated on: May 11, 2005 at 15:53

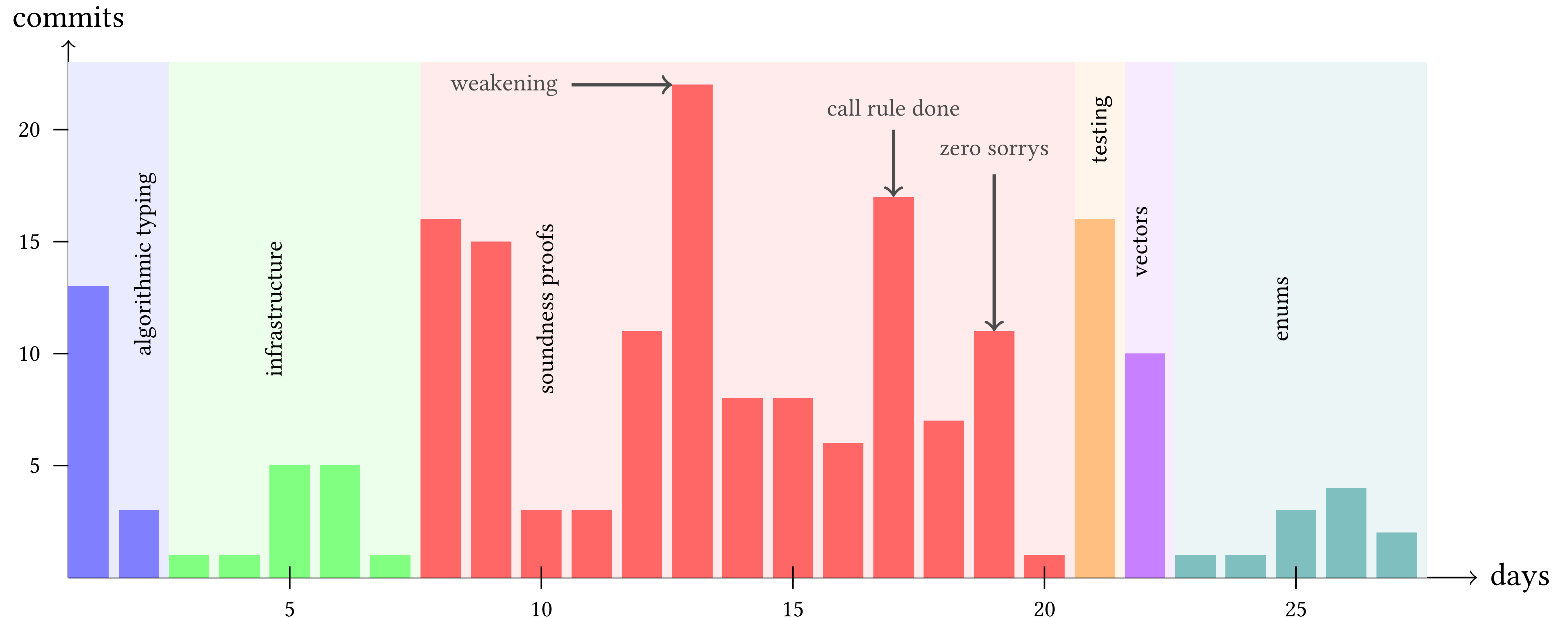
Abstract. How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine-checked proofs?

Abstract. How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine-checked proofs?

We're already here.

Proof Effort for This Paper

39,000 lines of Lean, 267 commits, 27 active working days in Feb 2026*



Timeline since February 2026

proofsandintuitions.net

Verifying Move Borrow Checker in Lean: an Experiment in AI-Assisted PL Metatheory

by Ilya Sergey · March 18, 2026 · 32 min read

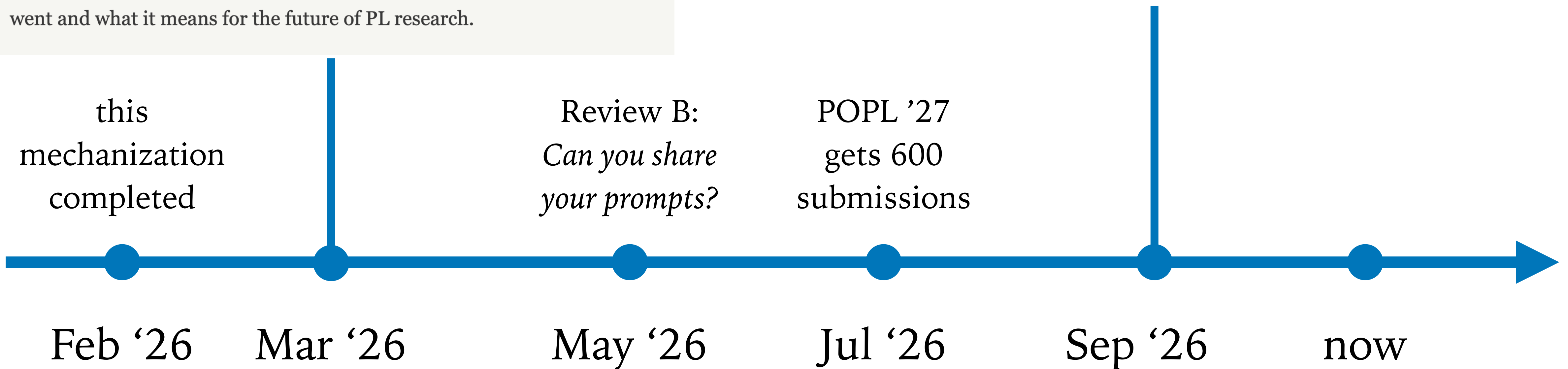
I formalised and proved the correctness of **Move**'s new borrow checker in Lean: 39,000 lines of mechanised metatheory, produced in under a month with the help of an AI coding assistant. This post tells the story of how it went and what it means for the future of PL research.

Extending concurrent separation logic to the hardware level to verify the xv6 OS kernel on RISC-V with AI agents

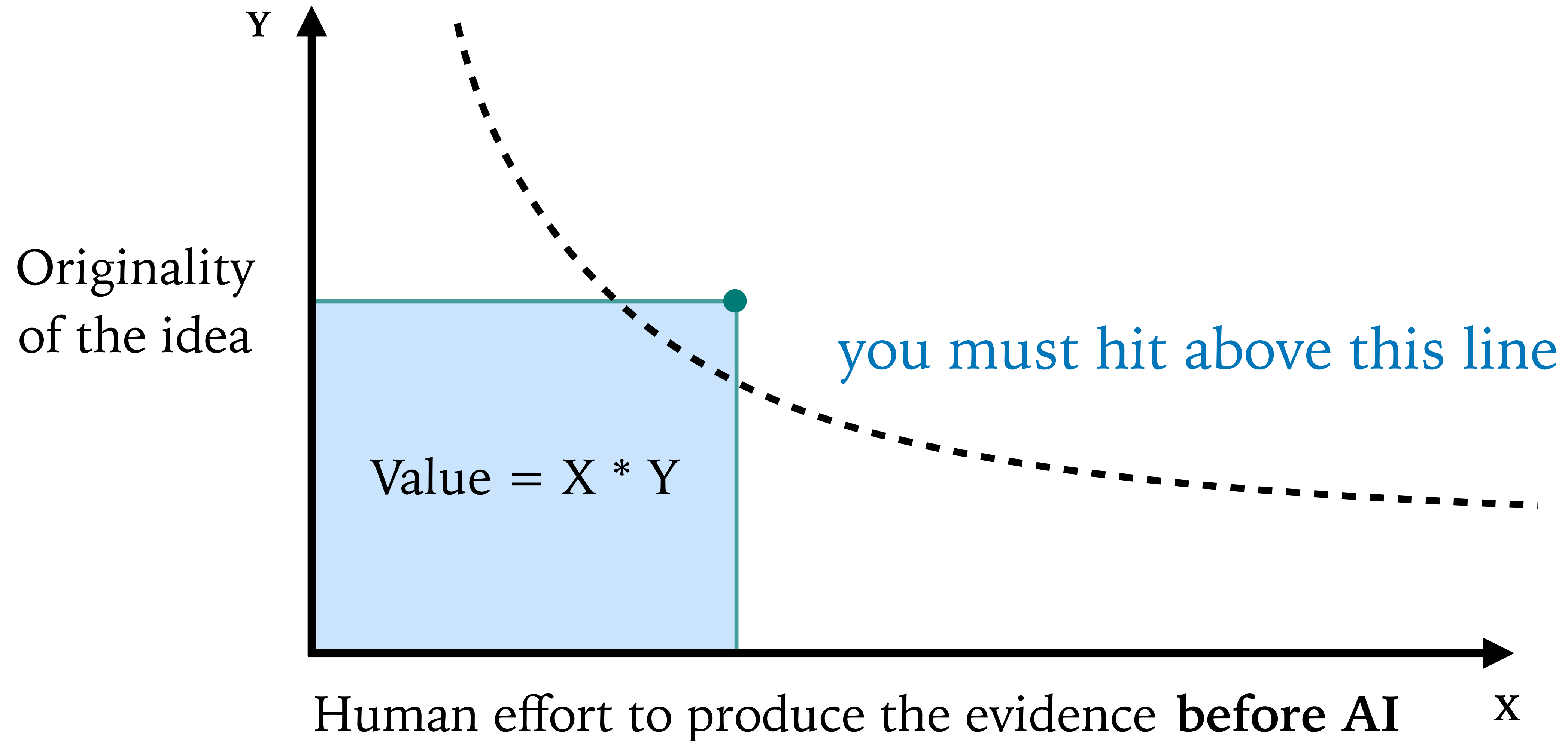
M. Frans Kaashoek and Nickolai Zeldovich
MIT CSAIL

September 20, 2026 – revision v2*

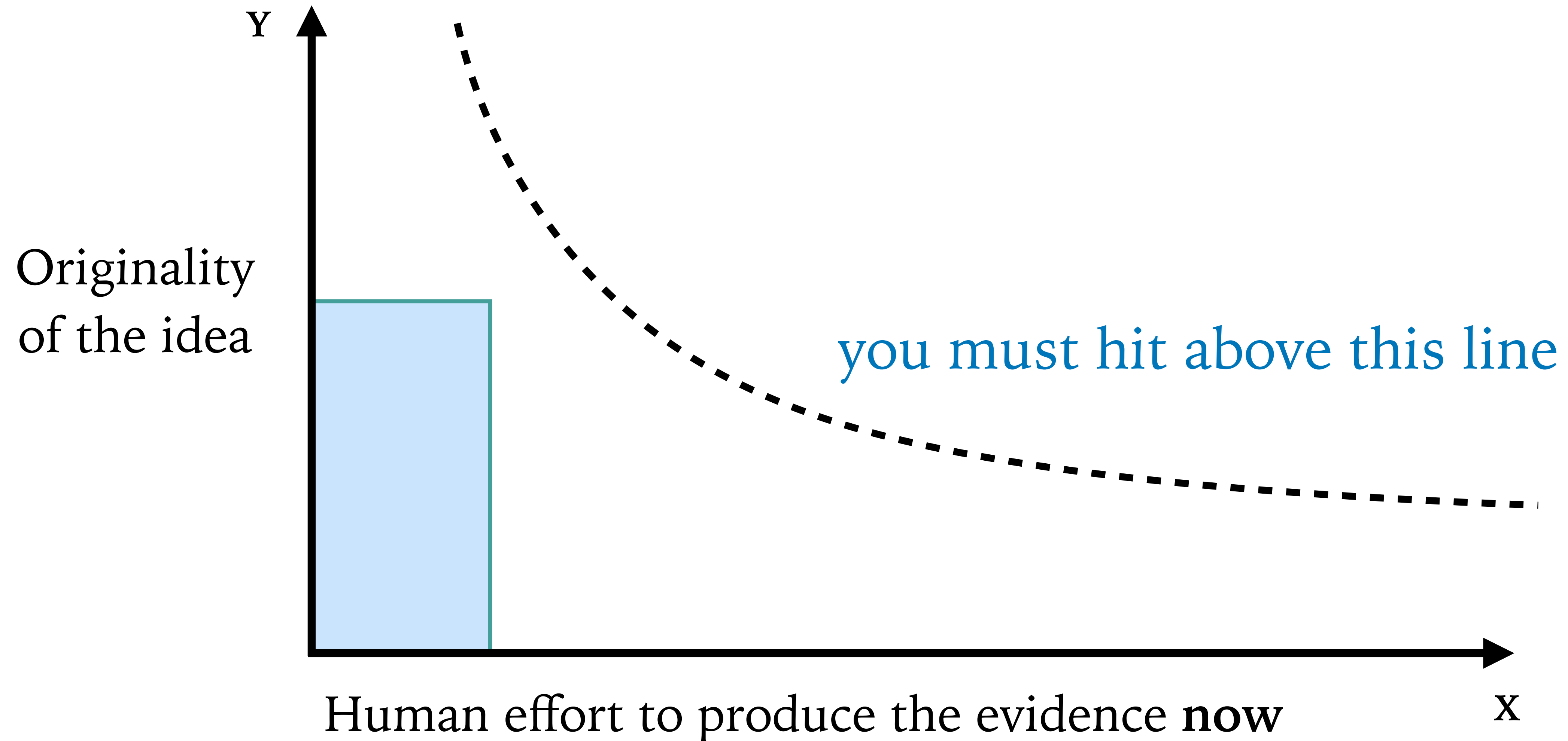
“we verify the xv6 OS kernel (6,593 lines of C and assembly code)”



Research Effort vs Contribution Value in PL



Research Effort vs Contribution Value in PL



When evidence becomes cheaper, what should a strong PL paper contribute?

- **Moonshot-like evaluation:** it's time to be ambitious!
- **Bolder ideas,** whose impact is amplified by AI, not just enabled by it.