Mechanising a Regex-Based Borrow Checker

an Experiment in AI-Assisted Metatheory

Ilya Sergey

ilyasergey.net

FP Launchpad Kickoff, IIT Madras

Two Hats, One Talk

For the PL Theorists	For the PL Practitioners
A new take on Rust-style ownership	Quickly validating that rules match intuition
Regex-based aliasing path semantics	Finding type design bugs before users do
Brzozowski derivatives for field borrows	Iterating on rules with fast proof repair
Write safety = regex emptiness	Using AI to absorb proof maintenance cost

Outline

- **Part 1** (~20 min): The Move borrow checker: what it tracks, how it works
- **Part 2** (~15 min): AI-driven formalisation: how I built 39K lines of Lean with Claude Code
- **Part 3** (~5 min): Conclusions: vision, discussion, and what comes next

Mechanising a

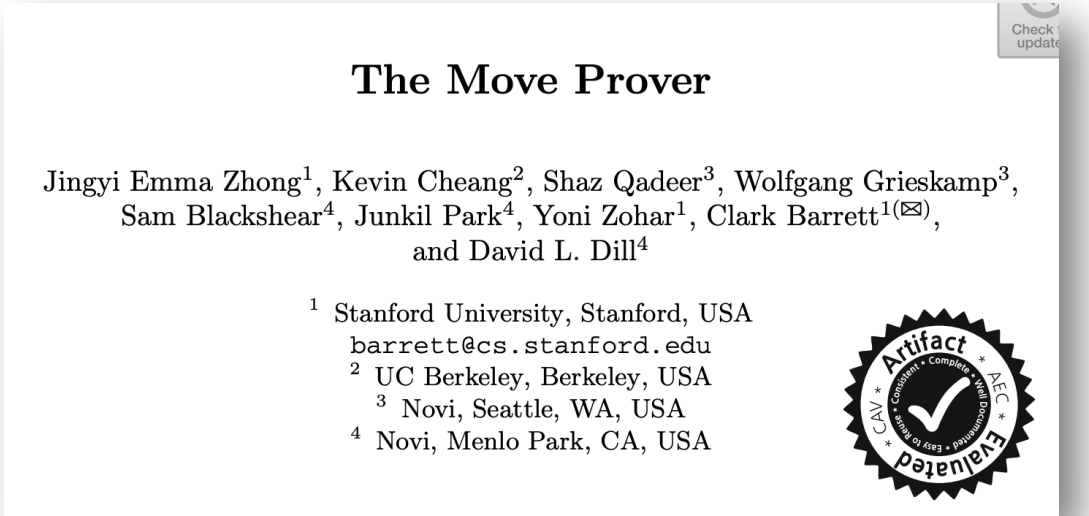
Regex-Based Borrow Checker

an Experiment in AI-Driven Metatheory

Part 1: Design of the Type System

What is Move?

- **Smart contract language** (Sui, Aptos blockchains,)
  - On Sui: 424,098 modules and 2,922,972 functions (as of 25 March 2026)
- **Rust-inspired ownership**: memory safety without garbage collection
- Simpler than Rust: **no lifetimes**, no trait system, no references inside records
- Compiled to bytecode via **MVIR** (Move Intermediate Representation)
- Borrow checker operates on MVIR: **basic blocks**, jumps, explicit temporaries



The Move Prover

Jingyi Emma Zhong[1], Kevin Cheang[2], Shaz Qadeer[3], Wolfgang Grieskamp[3], Sam Blackshear[4], Junkil Park[4], Yoni Zohar[1], Clark Barrett[1(✉)], and David L. Dill[4]

[1] Stanford University, Stanford, USA
barrett@cs.stanford.edu
[2] UC Berkeley, Berkeley, USA
[3] Novi, Seattle, WA, USA
[4] Novi, Menlo Park, CA, USA

https://www.sui.io/move

Move vs. Rust: Simpler but Still Deep

Move captures core Rust ownership:

- Move semantics (**consume-on-use**)
- Mutable/immutable borrowing with **field-level tracking**
- **Inter-procedural** borrow propagation

What Move drops:

- Lifetimes → scope-based analysis
- Trait-based generics → monomorphic IR
- References inside records → simpler memory model (and no linked lists…)

Sweet spot for formalisation: complex enough to be interesting, simple enough to be tractable

First machine-checked soundness proof for any Move-like borrow checker

Example 1: Dangling Reference (REJECTED)

Borrow a field, then overwrite the parent — creates a dangling pointer.

```
struct S has copy { f: u64 }
t(s: &mut S) {
    let f: &u64;
  label b0:
    f = &copy(s).S::f;          // borrow field f of s
    *move(s) = S { f: 0 };      // overwrite s — REJECTED!
    return;                     // f would be dangling
}
```

Example 2: Subtree Writes with Release (ACCEPTED)

Borrow disjoint fields, release one, write to it — safe because borrows don't overlap.

```
struct S { l: u64, r: u64 }
t(s: &mut S) {
    let x: &mut u64;
    let y: &mut u64;
  label b0:
    x = &mut copy(s).S::l;     // x borrows field l
    y = &mut copy(s).S::r;     // y borrows field r (disjoint!)
    _ = move(x);               // release x
    *(&mut copy(s).S::l) = 42; // write to l — safe: y borrows r, not l
    *move(y) = 0;              // write to y — safe: no outbound borrows
    return;
}
```

The Key Idea: Regex-Based Path Tracking

- **Abstract references r**: *root* (ownership root), *refid n* (allocated), *param x* (parameter)
- **Path graph**: (Aref × Aref) → Regex PathElement
- Path elements: **field f** (record field), **root_to_var x** (root-to-variable link)
- Safety check before writing to reference **r**: are all outbound regexes from **r** *trivial*?
  - Intuition: *no outstanding borrows into* **r***'s structure --- safe to overwrite*
  - *This doesn't prevent aliasing, which is handled by Rust-like move semantics*
- Why regexes? **Kleene star** for unbounded paths + **Brzozowski derivatives** for field borrows

Why Regexes? Kleene Star for Unbounded Paths

- **Function calls**: callee creates unknown relationships between parameters and outputs
- At return, the caller must account for **all possible borrow chains**
- **Kleene star**: e* = ε | e | e·e | … — finitely represents infinite path sets
- After call f(mut& x, mut& y): G(r_x, r_y) might become (**field "a"** · **field "b"**)*
- **Loops**: fixpoint at loop heads — star absorbs iteration (e* · e = e*)
- Key: regexes (with star) are **closed** under all needed operations

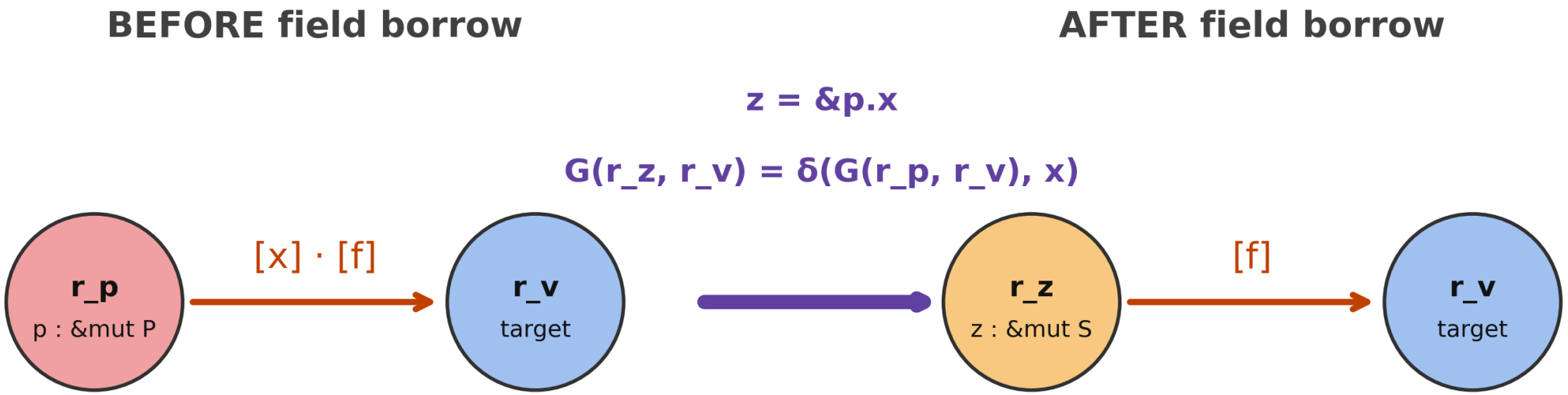
Brzozowski Derivatives

- Borrowing a field = **stripping a prefix** from all paths

  z = &s.f $\Longrightarrow$ for all v, G(z, v) = δ(G(s, v), f)

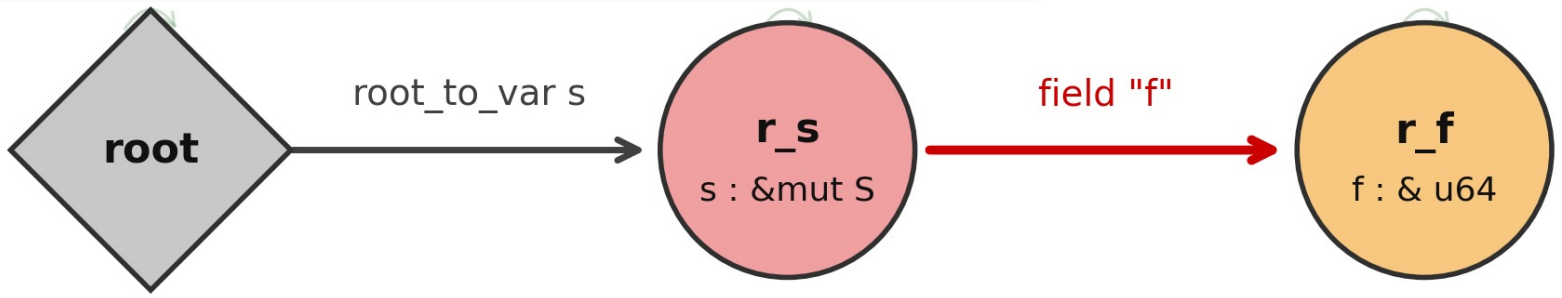
- Derivative semantics: **⟦δ(r, a)⟧(w) ⟺ ⟦r⟧(a · w)**
- $\delta(r_1 \cdot r_2, a) = \delta(r_1, a) \cdot r_2 \mid \nu(r_1) \cdot \delta(r_2, a)$
- Example:

  G(r_p, r_v) = [x]·[f] → after **z = &p.x** → **G(r_z, r_v) = δ([x]·[f], x) = [f]**



Example 1 Explained: The Blocked Write

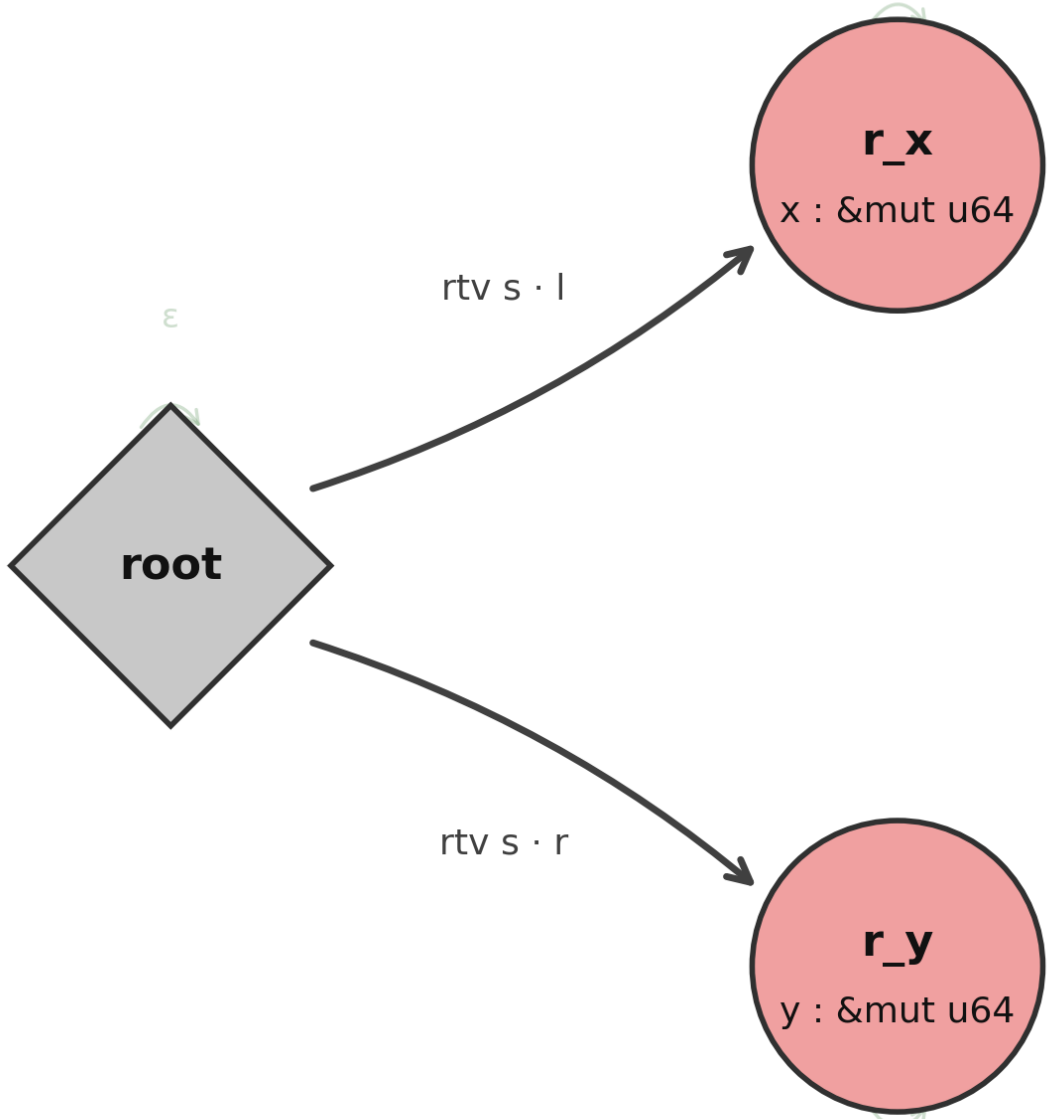
```
struct S has copy { f: u64 }
t(s: &mut S) {
    let f: &u64;
  label b0:
    f = &copy(s).S::f;        // ◀ borrow creates r_f
    *move(s) = S { f: 0 };    // ◀ write REJECTED
    return;
}
```



✗ check_outbound(r_s) FAILS

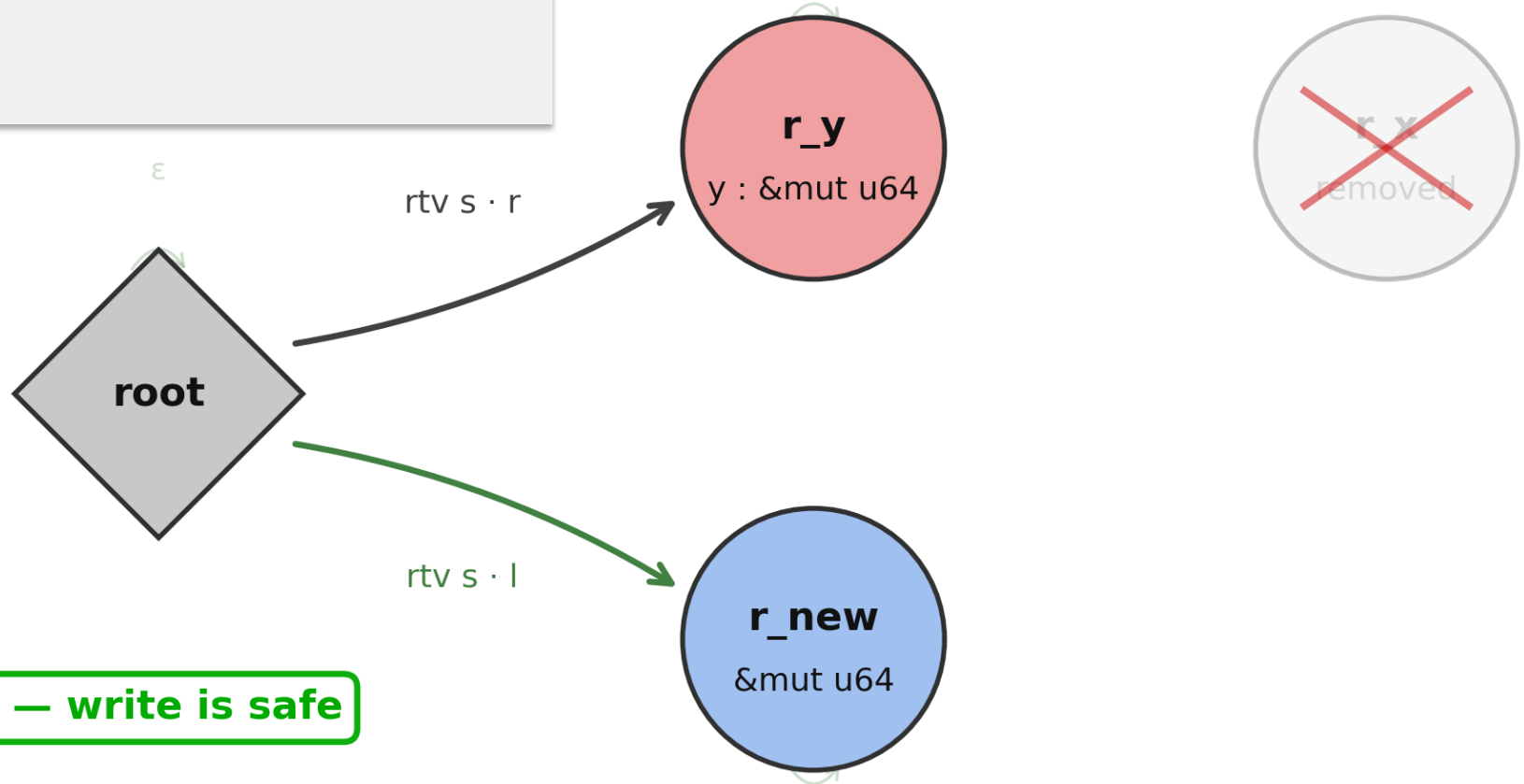
Example 2 Explained: Disjoint Borrows

```
struct S { l: u64, r: u64 }
t(s: &mut S) {
    let x: &mut u64;
    let y: &mut u64;
  label b0:
    x = &mut copy(s).S::l; // ◀
    y = &mut copy(s).S::r; // ◀
    ...
}
```



Example 2 Explained: Release and Write

```
...
x = &mut copy(s).S::l;
y = &mut copy(s).S::r;
_ = move(x);                    // ◀ release x
*(&mut copy(s).S::l) = 42; // ◀ write — SAFE
*move(y) = 0;
return;
```

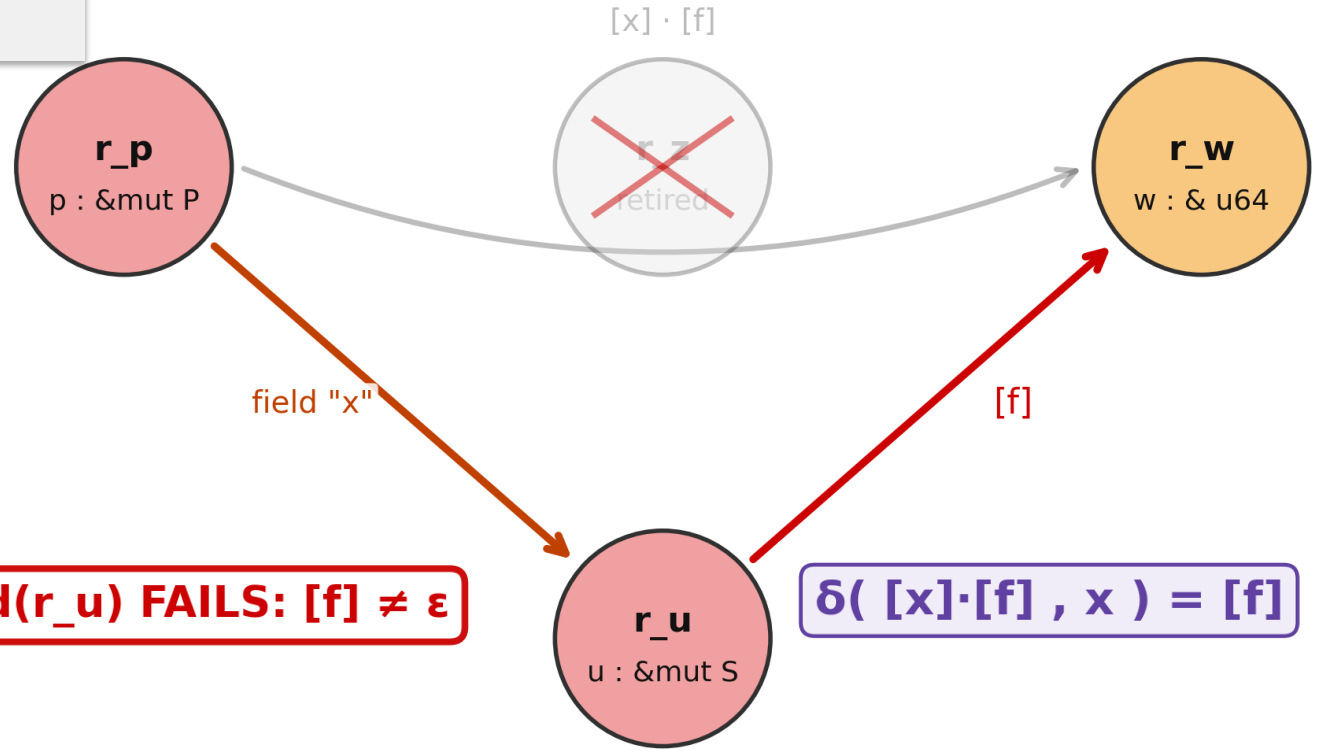


✓ check_outbound(r_new) PASSES — write is safe

Derivative in Action: Non-Trivial Path Stripping

```
struct P { x: S }
struct S { f: u64 }

p : &mut P
z = &copy(p).P::x     // borrow p.x
w = &copy(z).S::f     // borrow z.f = p.x.f
_ = move(z);          // retire z
u = &mut copy(p).P::x // re-borrow p.x
*move(u) = S {f: 0};  // ◀ BLOCKED!
```



Write Typing Rule

```
Γ(a) = ref(τ, r, mut)
Γ(b) = basic(τ)
∀ v ∈ G.refs.  G(r, v) ⊆ {ε}      ← KEY CHECK
Δ; ⟨Γ \ {a,b},  G⟩ ⊢ cont : ρ
──────────────────────────────────────
Δ; ⟨Γ, G⟩ ⊢ *a := b; cont : ρ
```

Γ = siteEnv, G = pathEnv, Δ = label-associated environment, ρ = retTypes, gc = remove old reachability

Intuition: G(r, v) ⊆ {ε} means "no real paths exist from r to v".

Self-loops G(r, r) = ε always hold, so we allow the empty word.

Any non-empty word means someone borrows into r’s structure — writing would create a dangling reference.

Mutable Field Borrow Rule

```
Γ(a) = ref(τ, r_s, mut)
τ = record { …, f : τ', … }
af ∉ dom(Γ)
r_f is fresh in G
G' = update_with_extension(r_f, r_s, [f], G)
Δ; ⟨Γ[af ↦ ref(τ', r_f, mut)] \ {a},  G'⟩ ⊢ cont : ρ
──────────────────────────────────────────────────
Δ; ⟨Γ, G⟩ ⊢ let af = &mut a.f; cont : ρ
```

update_with_extension(r_f, r_s, [f], G):

- Outbound: G′(r_f, v) = δ(G(r_s, v), f) — **Brzozowski derivative** strips the field prefix
- Inbound: G′(u, r_f) = G(u, r_s) · [f] — extend parent's inbound with the field step
- Self-loop: G′(r_f, r_f) = ε — trivial identity
- Parent a is consumed — must re-borrow to access other fields

Call Rule (Simplified)

```
lookup(funEnv, f) = (params, rets)
Γ ⊢ bs : params                              (inputs conform)
as, outRefs fresh in Γ, G                    (fresh outputs)
Γ' = populate(Γ, as, rets, outRefs)          (bind outputs)
mut inputs isolated                          (no mut aliasing)
Δ; ⟨Γ' \ bs, connect(G, as, bs) ⟩ ⊢ cont : ρ
──────────────────────────────────────────────────────────
Δ; ⟨Γ, G⟩ ⊢ as = f(bs); cont : ρ
```

connect(Γ′, as, bs) wires the path graph:

- Immutable outputs ← (.)* **all inputs** (unknown aliasing from callee)
- Mutable outputs ← (.)* **mutable inputs** (inbound only — no outbound paths)
- G(m_out, root) = ε (restores aliasing detection for later borrows)
- Immutable outputs ← (.)* **each other** (may alias through callee)

The Type Soundness Statement (in Lean 4)

```
theorem type_soundness
    (htyped : typecheck_fun f lenv)          -- function is well-typed
    (hfunEnv : ∀ callee ∈ funEnv,            -- all callees are safe
               FunTypeSafe callee funEnv)
    (ha : SoundnessAssumptions ...)          -- runtime config is valid
    (e: Error) (hna : ¬e.isAcceptable) :     -- error is preventable

    ∀ n, run n (initState f ...) ≠ .error e
```

Rules out 8 of 11 runtime errors:

Preventable	Acceptable
danglingRef, uninitializedVar, uninitializedSite	divisionByZero
unknownLabel, unknownFunction, arityMismatch	outOfFuel
invalidFieldAccess, typeMismatch	aborted

Why Formalising This Is Non-Trivial

- **WellTypedState**: 35-conjunct *preservation invariant* connecting abstract types to concrete heap
  - Value-type correspondence, path-heap coherence, reference liveness, uniqueness
- **Weakening/subsumption**: one of the largest proofs
  - At control-flow joins, code that type-checks under more restrictions must type-check under fewer
  - Thread substitution σ (renaming abstract references between branches) through every rule
- **Regex reasoning**: Prove Brzozowski derivatives faithfully track concrete heap reachability
- **Function calls**: construct callee's WellTypedState, push frame to stack, prove StackSafe

```
type_soundness
  └── safe_run  (fuel induction)
      └── safe_step  (one step)
          ├── preservation  (25 cases × 35 conjuncts)
          └── progress  (8 error types → contradiction)
```

Mechanising a Regex-Based Borrow Checker:

an Experiment in

AI-Assisted Metatheory

Part 2: AI-Driven Auto-Formalisation

Formalisation as a Design Tool

A type system must be:

- **Usable** — programmers can actually write code that type-checks
- **Not overly restrictive** — don't reject valid programs
- **Backwards-compatible** — redesign of existing type system;
  previously accepted programs must not be rejected
- **Sound** — don't accept unsafe programs

These goals push back against each other:

- Relax a rule → re-check soundness → fix proof
- Find a false rejection → change the rule → re-prove weakening
- Discover a bug via proof failure → fix the rule → re-prove everything downstream

The Algorithmic Type Checker

- **The problem:** Relational rules (Prop) are great for proofs but you can't run them
  - Rule doesn't match the reference implementation?
    You find out after days of failed proofs
  - Need a faster feedback loop: test rules on concrete programs immediately

The solution: Two parallel type systems

- **Relational** (687 lines): typecheck_stmt : ... → Prop
- **Algorithmic** (645 lines): check_stmt_dec : ... → Bool

```
theorem check_stmt_sound : ∀ lenvDec env stmt retTypes,
  check_stmt lenvDec env stmt retTypes = true →
  typecheck_stmt lenvDec.toLabelEnv env stmt retTypes
```

Soundness is proved against the relational specification — not the algorithm.

Zero-Cost Conformance Testing

Once the algorithmic checker is proved sound:
conformance testing against real compiler tests at zero cost.

```
theorem myprog_no_error : ∀ n loc,  -- type checking myprog
    run n (initState myprog ...) ≠ .error (.danglingRef loc) :=
  type_soundness_dec ... (by rfl)  -- ← entire proof!
```

- Tests parsed from real MVIR (Move bytecode verifier IR) test files
- Programs the compiler accepts → our checker accepts → safety certified
- Bottleneck: **proof of soundness** of algorithmic typing wrt. relational one

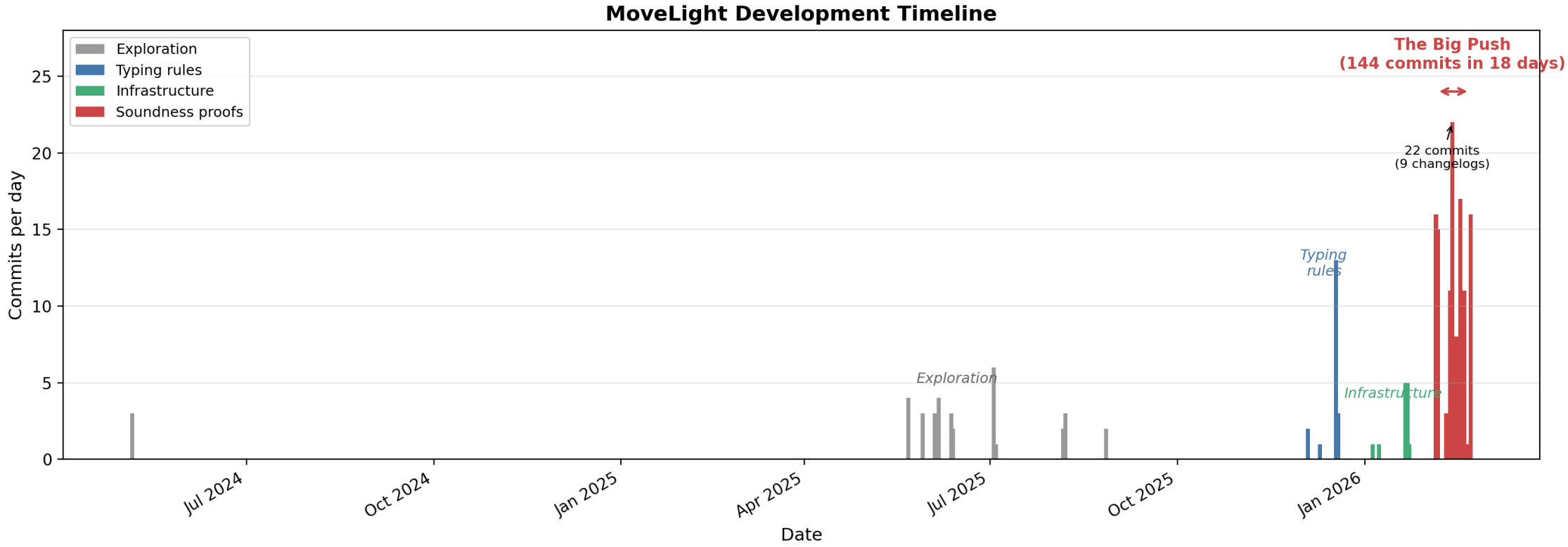
Without AI: each iteration costs days.
With AI: minutes.

The Vision: AI as a Proof Engineer

- Traditional PL formalisation: months to years, multi-person teams
- This project: **one researcher + Claude Code (Opus 4.5/4.6)**, **~4 weeks** (Feb 5–28, 2026)
- Total: **39,000 lines** of Lean 4, **zero sorrys**, 267 commits
- The division of labour:
  - **Human**: design decisions, type system rules, steering prompts, bug identification
  - **AI** (Claude Code): Lean definitions, proofs, tests, refactoring, documentation

Timeline and Effort

Phase	Dates	Commits	Focus
Exploration	May 2024 – Nov 2025	39	Definitions, experiments
Typing rules	Dec 4–18, 2025	19	Rules from design discussions
Infrastructure	Jan 5–23, 2026	13	Parser, macros, test framework
Soundness proofs	Feb 5–22, 2026	144	The Big Push



Overall Codebase

Component	LOC
Language, regex, semantics	2,700
Typing rules (rel. + alg.)	5,810
Preservation	10,270
Weakening	7,230
Other soundness proofs	5,530
Data structures, utilities	1,270
MVIR parser, translator	1,280
Tests	6,100
Total	**~39,000**

Steering the AI: Actual Prompts

- **Excerpt 1** — **Architectural** decision (Weakening.lean):

  "Don’t add funEnv into subsumption. Just pass it to the weakening and ensure it’s the same for env and envL.”

- **Excerpt 2** — **Strategy** for control-flow joins:

  "Use env (current execution env) for the new WellTypedState, NOT envL. more paths = more restriction."

- **Excerpt 3** — Invariant **placement**:

  "don’t add paths_from to WellFormedEnv, it cascades. Add it to WellTypedState instead."

What Went Wrong: Bug Discovery Through Proofs

Bug 1 — Assignment rule (Feb 6): reversed update_with_extension arguments

```
-- WRONG:  update_with_extension .root r [.root_to_var x]
-- FIXED:  update_with_extension r .root [.root_to_var x]
```

Discovered when AI couldn't prove WellFormedState preservation.

- **Bug 2 — Call rule** (Feb 17): extend_with_star arguments swapped + overly restrictive check
- **Bug 3 — Return writability** (Feb 22): checked against all pathEnv.refs instead of live sites

Lesson: Proofs are the ultimate bug finder

Testing the Design: Parser and Algorithmic Checker

Ensure conformance with the reference type checker implementation.

Level 1: Parser + Alpha-Equivalence

- Parse actual Move bytecode verifier tests (MVIR) → translate to MoveLight
- Compare against hand-written ASTs using structural **alpha-equivalence**
- 13 parser tests; handles site renumbering, refid permutation, label renaming

Level 2: Algorithmic Type Checker

- Boolean check_fun_dec proven sound w.r.t. relational rules
- 17 hand-written litmus tests (6 to be accepted, 11 to be rejected)
- 156 expressivity tests **from the real Move compiler test suite** (96 to be accepted, 60 to be rejected)

Testing the Design: Runtime Conformance

Ensure conformance with the compiler.

Level 3: Semantic Conformance

- 12 expressivity tests + 17 litmus tests, checking outcomes (shallow)
- Artificially designed failed examples (dangling pointers etc)

Testing the Design : Decidable Type Soundness

Level 4: Decidable Type Soundness

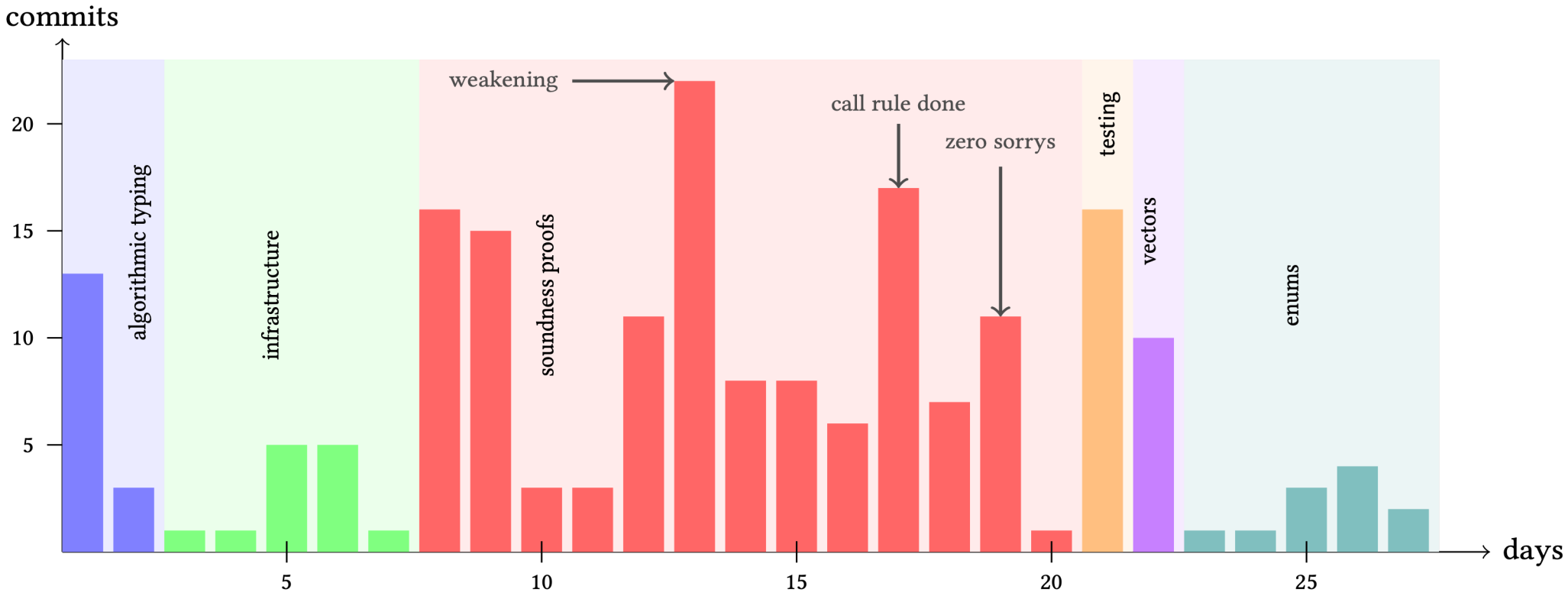
Type soundness has assumptions on the input state (heap, args) — they might not be satisfiable. Decidable checking verifies them on concrete inputs.

```
theorem type_soundness_dec
    (f : FunDef) (lenvDec : LabelEnvDec)
    (funEnv : AssocMap Id FunDef) (fte : FunTypingEnv)
    (args : List Value) (heap : Heap)
    (hdec : checkDecidable f lenvDec          -- all assumptions
              funEnv fte heap args = true)  -- reduced to one Bool
    (hna : ¬e.isAcceptable) :
    ∀ n, run n (initState f ...) ≠ .error e
```

- 96 runtime certificates: proof is **by native_decide**
- End-to-end: parse MVIR → type-check → execute → certify — all in **lake build**

What's Different: Speed and Scale

- **Speed**: Months → weeks (~35K lines in 18 intensive days)
- **Scale**: AI produces and maintains 6,000-line proofs that would be tedious for humans
- **Proof as a debugger**: Bugs found through failed proofs, not testing



Why did it work so well

- Proofs by *progress and preservation* are well understood --- lots of learning data!
- Non-trivial *supervision*: high-level proof strategy has been approved by human
- Lots of real-world "*litmus tests*" from Move compiler --- to validate the design quickly

- Might not work as good less conventional proofs
  - Logical relations (as in Iris)
  - Confluence proofs for non-standard semantics (e.g., Verse language, ICFP'23)

Useful patterns for AI-aided PL formalisation

- **Test** every definition and theorem, if possible!
  - eliminates false assumptions

- Make sure you **understand** the overall structure of the development
  - AI is pretty good at producing concise summaries --- easy to identify wrong paths

- When seems like proof is not progressing, as AI to **explain** what it's trying to prove
  - You *must* understand the proof strategy to "snap it out" of the loop

Lots of opportunity for meta-framework builders!

The Future: Auto-Formalising Programming Languages

- AI can now serve as a "proof engineer" for PL meta-theory
- What changes:
  - Bottleneck shifts: writing proofs → **designing invariants**
  - Formalisation becomes **design validation** (3 bugs found through proofs!)
  - 39K lines in 3 weeks — previously unthinkable for one researcher
- **Vision**: given language spec + operational semantics →
  - AI generates typing rules, executable checker, soundness proof, tests
- **Open challenges**:
  - Synthesis of state invariants for progress/preservation
  - Scaling to full languages (generics, modules, effects)
  - Reducing human steering (~30% of effort currently)

Take Away

Regex-based borrow checking:

- Tracking aliasing as **regular expressions** over field paths
- **Brzozowski derivatives** for expressing borrowing into fields
- Safety checks reduce to regex **emptiness**: $G(r, v) \subseteq \{\varepsilon\}$
- **Kleene star** for unbounded aliasing from loops and function returns

AI-driven formalisation:

- **One person + Claude Code**: 39,000 lines of Lean 4, zero sorrys, ~4 weeks
- Human: type system design, proof architecture, bug diagnosis
- AI: proof engineering, case analysis, helper lemmas, testing infrastructure
- **Formalisation as a design tool**: 3 bugs found through failed proofs, not testing

Further Reading

Proofs and Intuitions

A blog about mathematics, computing, formal verification, and the ideas behind them

---

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

by Ilya Sergey · March 18, 2026 · 32 min read lean move types verification ai

proofsandintuitions.net

Thanks!