Tracking Borrows with Regular Expressions

TODD NOWACKI, Mysten Labs, USA
SAM BLACKSHEAR, Mysten Labs, USA
JOHN MITCHELL, Stanford University, USA
SHAZ QADEER, Microsoft, USA
ILYA SERGEY✉*, National University of Singapore, Singapore

Safe systems languages such as Rust enforce an *ownership* discipline through types: every value has a unique owner, and the type system tracks *borrows*—references that provide temporary access to values without transferring their ownership. *Borrow checking* is a static analysis ensuring that no borrow outlives its owner and that no two mutable borrows are aliases, preventing dangling references and data races at compile time. Move, a smart contract language deployed on Sui and Aptos blockchains, adopts this model but restricts references to structured *access paths* rooted in local variables, eliminating the need for complex lifetime tracking mechanisms such as lifetime annotations. We present a novel type system for Move's borrow checker in which access paths are tracked by *regular expressions*. In this model, Brzozowski derivatives make it possible to express the reachability consequences of borrowing operations, Kleene star summarises borrow chains from function calls and loops, and the aliasing check reduces to the decidable regex emptiness. The design of the type system with regular expression-based borrow tracking extends naturally to vectors and enumeration types. The proposed design of a borrow checker has been implemented in the Move bytecode verifier for Sui blockchain, where it superseded the original borrow analyser while maintaining full backwards compatibility. We mechanised the type system in Lean with a machine-checked soundness proof and an executable algorithmic type checker tested against the production Move compiler. Notably, this 39,000-line metatheory was developed with an AI proof assistant in roughly one month, and we report on our experience of conducting this proof effort, which is among the largest AI-assisted PL metatheory mechanisations to date.

CCS Concepts: • **Theory of computation → Type theory**.

Additional Key Words and Phrases: borrow checking, type systems, mechanised metatheory, Lean

ACM Reference Format:
Todd Nowacki, Sam Blackshear, John Mitchell, Shaz Qadeer, and Ilya Sergey. 2026. Tracking Borrows with Regular Expressions. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 389 (October 2026), 27 pages. https://doi.org/10.1145/3839521

1 Introduction

Static borrow checking is the mechanism by which a programming language prevents, at compile time, dangling references, double frees, data races, and other aliasing-induced violations that lead to possibly undefined runtime behaviour. Pioneered by Rust [26, 27, 41], borrow tracking has become a defining feature of safe systems languages: every reference must be accounted for at compile time, and the compiler should be able to verify that no two mutable references to the same

*Work done while employed as a part-time research consultant at Mysten Labs.

Authors' Contact Information: Todd Nowacki, Mysten Labs, Palo Alto, USA, tmn@mystenlabs.com; Sam Blackshear, Mysten Labs, Palo Alto, USA, sam@mystenlabs.com; John Mitchell, Stanford University, Palo Alto, USA, jcm@stanford.edu; Shaz Qadeer, Microsoft, Redmond, USA, shaz.qadeer@gmail.com; Ilya Sergey, National University of Singapore, Singapore, Singapore, ilya@nus.edu.sg.

ACM 2475-1421/2026/10-ART389
https://doi.org/10.1145/3839521

memory location exist simultaneously, thus eliminating the possibility of undefined behaviours. Despite the existence of many proposals to implement static borrow checkers [2, 30, 35, 41], formalising and proving them sound remains challenging. Existing formalisations of Rust's borrow checker either target restricted subsets of the language [35, 41] or require heavyweight semantic frameworks [26, 27, 40]. Other proposals, such as type systems for fearless concurrency [30] and reachability types [2], remain academic prototypes without wide practical adoption.

Move [4] is a smart contract language deployed on Sui [5, 42] and Aptos [39]. As a production language managing financial resources on blockchains, Move implements a non-trivial borrow-checking discipline that must be both sound and practical. Like Rust, Move enforces an ownership discipline: every value has a unique owner, and references must not outlive the values they borrow. Unlike Rust, however, Move does not allow for references inside data structures: every reference is a structured *access path* [7] rooted in a local variable and descending through a sequence of field names. This restriction eliminates the need for complex lifetime tracking mechanisms—such as the explicit scope markers and lifetime annotations that Rust programmers must provide—because Move's type system can track the provenance of every reference using its path alone.

The Move borrow checker currently deployed for the Sui and Aptos blockchains [8] uses abstract interpretation over a domain of *borrow graphs* whose edges are labelled with access paths that may be *extensible* (allowing unknown suffixes), denoting all extensions of a given prefix. For instance, if a reference y borrows field f from variable x, and the checker later loses track of which sub-fields of y are borrowed, it records an extensible edge $x \xrightarrow{f.\star} y$, meaning that y may reach any location reachable from $x.f$—a safe but conservative over-approximation. The analysis operates directly on the deployed Move bytecode, propagating these graphs through the instructions, joining them at control-flow merge points, and checking safety conditions at each mutating operation. While sufficient for production use, this design conflates inference and checking and is justified only informally, making it hard to extract a clean type system—let alone a machine-checked soundness proof. For a language that governs the custody of financial assets, this is unsatisfying: a bug in the borrow checker could let an attacker exploit a dangling reference and compromise on-chain funds. Separating inference from checking and supplying a machine-checked soundness proof would deliver the assurance that no such exploitable bugs exist.

In this work, we present a novel type system for borrow checking in Move based on a conceptually simple idea: tracking reachability between references using *regular expressions over field paths*. The key insight is to label edges in a reference reachability graph with regular expressions: if there is an edge from abstract reference ρ to ρ' labelled with regex r, then r over-approximates the set of all field paths that, when traversed from the run-time location of ρ in the heap, reach the location of ρ'. This formulation turns three core borrow-checking tasks into standard regular-language operations. Extending a reference (by borrowing a field, copying, or receiving a call result) becomes *Brzozowski derivative* computation [31]: borrowing field f from a reference whose reachability is described by regex r yields a new reference with reachability $\delta(r, f)$. Summarising the effect of loops and function calls amounts to closing a path expression under the Kleene star. And the key safety check, whether writing through a mutable reference can invalidate another reference, reduces to checking that the reachability between the two references is captured by an *empty language*: if so, the references point into disjoint regions of memory. This achieves the same precision as the deployed checker's extensible paths through a simpler mechanism.

Any new type system design must remain backwards-compatible with existing programs: every program accepted by the deployed checker must also be accepted by ours. The regex-based borrow tracking extends naturally to vectors and multi-variant enumerated types (although recursive data

types are not yet supported by Move, the regex framework should be able to accommodate them without significant changes).

To gain trust in correctness of the new checker design, we provide its fully mechanised reference implementation in Lean [14] that includes relational and algorithmic formulations of the type system, the latter proven sound with respect to the former, a small-step operational semantics, and a machine-checked proof of type soundness, statically ruling out accesses to dangling references, writes that invalidate outstanding borrows, and use-after-move errors. We extensively tested our algorithmic checker written in Lean for conformance against programs drawn from the production Move compiler's own test suite. The design of our regex-based checker has been integrated into the soon-to-be-deployed Move implementation, superseding the original design [8].

Contributions. In summary, our work makes the following contributions:

- The design of a regex-based type system for borrow tracking. To the best of our knowledge, this is the first application of the regular language theory to static borrow checking (Sec. 2).
- A formal proof of type soundness with respect to a heap/stack-based small-step semantics, guaranteeing the absence of dangling references at runtime (Sec. 3).
- Extension of the borrow checker design to vectors and non-recursive enumerated types (Sec. 4).
- Full mechanisation of the borrow checker in Lean, including an executable algorithmic type checker, conformance tests against real Move programs, and decidable execution safety certificates, together with an experience report on its AI-assisted construction—one of the largest such developments to date (~39K lines of Lean in roughly one month)—analysing where the AI was effective and where human steering remained indispensable (Sec. 5).
- Integration into the production Move borrow checker, superseding the original analysis (Sec. 6).

2 Overview

We start by introducing the key ideas of our type system through a series of small but characteristic examples, building intuition before the formal description of the borrow checker in Sec. 3.

2.1 Move, Move Bytecode, and MoveIR

Move is compiled to a stack-based bytecode that is deployed on-chain and verified at load time, much like the JVM or the CLR. The borrow checker is part of this load-time verifier: it analyses each function body before execution, rejecting any Move program that could create a dangling reference or an aliasing violation. Between Move source and bytecode sits MoveIR (Move Intermediate Representation), a lower-level LLVM-style format with named local variables, structured control flow via labelled blocks and jumps, and explicit borrow/move/copy annotations. Fig. 1 shows a small example implemented in both Move and MoveIR. Several elements of the MoveIR syntax in Fig. 1 (right) deserve comment. User-defined data types are qualified with `Self` (*e.g.*, `Self.S`) to distinguish them from primitives of the language. Every use of a variable (regardless of its type) is annotated with `copy` (duplicate the reference) or `move` (consume it); there is no implicit semantics for variable access. Field access on references uses double-colon syntax (*e.g.*, `copy(s).S::f`), and control flow is structured via labelled blocks (`label b0:`) and explicit `jump`/`return` instructions.

A type system requires named variables and structured environments; working directly with the stack-based bytecode would require reconstructing both. We therefore use MoveIR as the basis for our presentation and formalisation. Since the Move compiler emits MoveIR internally, we can test our algorithmic type checker against programs drawn from the compiler's own test suite (Sec. 5).

2.2 Abstract References and Reachability

Like Rust, Move enforces an ownership discipline: every value has a unique owner, and references must not outlive the values they borrow. Unlike Rust, Move does not permit references inside data

```
1 // Move source
2 struct S { f: u64 }
3
4 fun dangling() {
5     let s = S { f: 42 };
6     let vf = &s.f;
7     s = S { f: 239 }; // error!
8 }
```

```
1 // MoveIR
2 struct S { f: u64 }
3 dangling() {
4     let s: Self.S;
5     let vf: &u64;
6   label b0:
7     s = S { f: 42 };
8     vf = &copy(s).S::f;
9     s = S { f: 239 }; // error!
10    return;
11 }
```

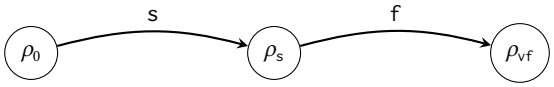

Fig. 1. Move source (left) and MoveIR (right) for a rejected program: every variable use is annotated copy (duplicate) or move (consume), and borrows are explicit. *Bottom:* the reachability graph after `vf = &copy(s).S::f`—the root ρ_0 reaches ρ_s via field `s`, and ρ_s reaches ρ_{vf} via field `f`. The graph is transitively closed, so $\text{paths}(\rho_0, \rho_{vf}) = \mathsf{s} \cdot \mathsf{f}$; diagrams show only generating edges and omit self-loops ε.

structures—every reference is a structured *access path* [7] rooted in a local variable and descending through a sequence of field names. An access path such as `s.f` names the memory location reached by starting at local variable `s` and following field `f`. This restriction eliminates the need for lifetime tracking mechanisms: the type system can track the provenance of every reference using its path.

Consider the MoveIR program in Fig. 1 (right). On line 8, the borrow `vf = &copy(s).S::f` creates a reference `vf` that points into the memory owned by `s`, at field `f`. On line 9, the assignment `s = S { f: 0 }` would overwrite the entire value of `s`, invalidating `vf` and creating a dangling reference. The type checker must reject this program. To detect such conflicts, our type checker tracks the evolution of the reachability graph between run-time memory location using *path environments*.

Abstract references and path environments. During type checking, every borrow-related instruction (creating, copying a reference, *etc.*) introduces a fresh *abstract reference* ρ. Abstract references are finitely many (one per such instruction in the program text), and they serve as symbolic names for run-time memory locations. The distinguished reference ρ_0 (the *root*) represents the ownership root from which all local variables are reached; conceptually, ρ_0 "owns" the entire stack frame.

A *path environment* Π records borrowing relationships between abstract references as a directed graph: the edge from ρ to ρ' is labelled with a regular expression $\text{paths}(\rho, \rho')$ over variables and field names that *over-approximates* concrete reachability: every concrete field path in the heap from the location of ρ to the location of ρ' is accepted by $\text{paths}(\rho, \rho')$. The graph is *transitively closed*: for every pair of references (ρ, ρ'), the edge $\text{paths}(\rho, \rho')$ already accounts for all intermediate paths through other references, so reachability between any two nodes can be read off directly without composing edges. Fig. 1 (bottom) shows the reachability graph after the borrow `vf = &copy(s).S::f`. Borrowing local variable `s` creates a fresh reference ρ_s with edge $\text{paths}(\rho_0, \rho_s) = \mathsf{s}$, connecting it to the root. The field borrow then creates ρ_{vf} with edge $\text{paths}(\rho_s, \rho_{vf}) = \mathsf{f}$. Transitive closure gives $\text{paths}(\rho_0, \rho_{vf}) = \mathsf{s} \cdot \mathsf{f}$, recording the composite path from root to ρ_{vf}.

2.3 Mutable Borrows and the Write Check

A mutable reference grants exclusive write access to the memory it points to. Before writing through a mutable reference tracked by abstract reference ρ, we must verify that no other tracked reference borrows into ρ's region of memory; otherwise the write would invalidate that reference, creating a dangling pointer. The check is expressed as a predicate on the path environment:

$$\text{check\_outbound}(\rho) \stackrel{\text{def}}{=} \forall \rho'.\ \mathcal{L}(\text{paths}(\rho, \rho')) \subseteq \{\varepsilon\}. \tag{1}$$

```
1  // REJECTED: write invalidates field borrow
2  struct S { f: u64 }
3  t() {
4      let s: Self.S;
5      let r: &mut Self.S;
6      let fref: &u64;
7    label b0:
8      s = S { f: 42 };
9      r = &mut s;
10     fref = &copy(r).S::f; // borrow field
11     *copy(r) = S { f: 0 }; // error!
12     return;
13 }
```

```
1  // ACCEPTED: coexisting borrows
2  t() {
3      let a: u64;
4      let rmut: &mut u64;
5      let rimm: &u64;
6    label b0:
7      a = 0;
8      rmut = &mut a;
9      rimm = &a;
10     *copy(rmut) = 0; // safe
11     _ = *copy(rimm); // safe
12     return;
13 }
```

Fig. 2. Left: a write through a mutable reference is rejected because an outstanding field borrow exists: $\mathcal{L}(\text{paths}(\rho_{\mathtt{r}}, \rho_{\mathtt{fref}})) = \mathcal{L}(\mathtt{f}) \not\subseteq \{\varepsilon\}$. Right: a mutable and an immutable borrow of the same variable coexist: the write through `rmut` is safe since the immutable borrow does not extend the mutable one.

The condition requires that, for every tracked reference ρ', the only word in the regular language defined by $\text{paths}(\rho, \rho')$ is the empty word ε. Since every reference has a trivial self-loop ($\text{paths}(\rho, \rho) = \varepsilon$), this amounts to requiring that no *non-trivial* outbound path exists from ρ.

Consider the program in Fig. 1. After the borrow `vf = &copy(s).S::f`, we have $\text{paths}(\rho_{\mathtt{s}}, \rho_{\mathtt{vf}}) = \mathtt{f}$. The assignment `s = S { f: 0 }` triggers $\text{check\_outbound}(\rho_{\mathtt{s}})$, which fails because $\mathcal{L}(\mathtt{f}) \not\subseteq \{\varepsilon\}$.

Fig. 2 (left) shows a mutable reference `r` to a struct, with a field borrow `fref = &copy(r).S::f` on line 10. The field borrow creates an abstract reference $\rho_{\mathtt{fref}}$ with edge $\text{paths}(\rho_{\mathtt{r}}, \rho_{\mathtt{fref}}) = \mathtt{f}$. When the program attempts to write through `r` on line 11, the checker evaluates $\text{check\_outbound}(\rho_{\mathtt{r}})$ as per (1), which fails because $\mathtt{f} \not\subseteq \{\varepsilon\}$: the field borrow `fref` borrows into `r`'s region of memory, and overwriting `r` would result in creating a dangling reference through `fref`.

Fig. 2 (right) shows that Move, unlike Rust, permits a mutable and an immutable borrow of the same variable to coexist. The mutable borrow on line 8 and the immutable borrow on line 9 both target `a`, yet the write through `rmut` on line 10 succeeds. Because each borrow independently links to the root (via $\text{paths}(\rho_0, \rho_{\mathtt{rmut}}) = \mathtt{a}$ and $\text{paths}(\rho_0, \rho_{\mathtt{rimm}}) = \mathtt{a}$), there is no non-trivial path from $\rho_{\mathtt{rmut}}$ to $\rho_{\mathtt{rimm}}$: $\text{paths}(\rho_{\mathtt{rmut}}, \rho_{\mathtt{rimm}}) = \emptyset$, and $\text{check\_outbound}(\rho_{\mathtt{rmut}})$ passes. This is safe because writing through `rmut` replaces the value at `a` with a new value of the same type, so any reference to that location (such as `rimm`) still points to a valid value and is never left dangling. By contrast, a *field borrow* points into the interior of a value; after overwriting the parent, the internal structure may differ (*e.g.*, a different enum variant), making the field path invalid. This is exactly what check_outbound detects—it rejects a write only when a sub-borrow extends the written reference along a non-trivial field path—and the same reasoning covers two mutable borrows of one variable, which never create non-trivial paths between each other.

```
t() {
    let a: u64;
    let rmut: &mut u64;
    let rimm: &u64;
  label b0:
    a = 0;
    rmut = &mut a;
    rimm = freeze(copy(rmut));
    *copy(rmut) = 0; // safe
    _ = *copy(rimm); // safe
    return;
}
```

Fig. 3. Freeze: converting a mutable reference to an immutable one.

We note that Move targets single-threaded execution, so coexisting mutable and immutable borrows introduce no data races. A concurrent setting would additionally require mutable borrows to be *exclusive*—no other live reference sharing a root path, $\mathcal{L}(\text{paths}(\rho_0, \rho)) \cap \mathcal{L}(\text{paths}(\rho_0, \rho')) = \emptyset$—recovering Rust's exclusive-mutability discipline [27]. We leave concurrency to future work.

```
1  struct S { l: u64, r: u64 }
2  t() {
3      let s: Self.S;
4      let x: &mut u64;
5      let y: &mut u64;
6    label b0:
7      s = S { l: 1, r: 2 };
8      x = &mut s.S::l;       // borrow l
9      y = &mut s.S::r;       // borrow r
10     *move(x) = 42; // safe: disjoint
11     *move(y) = 0;  // safe: disjoint
12     return;
13 }
```

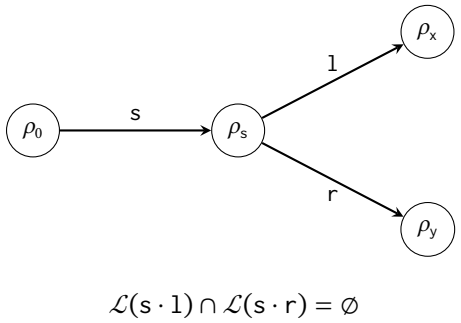

Fig. 4. Disjoint field borrows. Both writes succeed because $\mathcal{L}(\text{paths}(\rho_x, \rho_y)) = \emptyset \subseteq \{\varepsilon\}$.

Freeze. The right-hand example in Fig. 2 obtained its immutable reference `rimm` by borrowing `a` directly. In practice, it is more common to derive an immutable reference from an existing mutable one using the `freeze` operation, as shown in Fig. 3. Here, `freeze(copy(rmut))` copies the mutable reference and converts it to an immutable one. In the path environment, freeze allocates a fresh abstract reference $\rho_{\texttt{rimm}}$ connected to the original $\rho_{\texttt{rmut}}$ via an ε edge (the identity path, since both references point to the same memory). The write through `rmut` remains safe: $\mathcal{L}(\text{paths}(\rho_{\texttt{rmut}}, \rho_{\texttt{rimm}})) = \mathcal{L}(\varepsilon) \subseteq \{\varepsilon\}$, so check_outbound($\rho_{\texttt{rmut}}$) passes. Note the contrast with Fig. 2 (right), where the independent borrows yield $\mathcal{L}(\text{paths}(\rho_{\texttt{rmut}}, \rho_{\texttt{rimm}})) = \mathcal{L}(\emptyset) = \emptyset$: both $\emptyset$ and $\{\varepsilon\}$ are subsets of $\{\varepsilon\}$, so both pass the check, but for different reasons. Only a *non-trivial* field path would cause check_outbound to fail. Freeze is the standard mechanism in Move for sharing read access to a value that was initially borrowed mutably.

2.4 Field Borrows and Regular Expression Derivatives

Like Rust, Move permits multiple mutable references into the same struct, provided they borrow *disjoint* fields. Our type system tracks disjointness using regular expressions, which reduces the safety check to emptiness of language intersection. Consider an example in Fig. 4, which shows an accepted program that borrows two disjoint fields `l` and `r` of a struct `s` and writes through both.

After the two field borrows on lines 8–9, the path environment records $\text{paths}(\rho_s, \rho_x) = \texttt{l}$ and $\text{paths}(\rho_s, \rho_y) = \texttt{r}$. The two borrows are disjoint because the languages of their paths from root do not overlap: $\mathcal{L}(s \cdot l) \cap \mathcal{L}(s \cdot r) = \emptyset$. Since the root reaches ρ_x and ρ_y through non-overlapping field paths, the two references point into disjoint regions of memory and neither can reach the other: $\text{paths}(\rho_x, \rho_y) = \emptyset$. Therefore check_outbound(ρ_x) passes, and both writes succeed.

```
1  struct S { f: u64 }
2  struct P { x: Self.S }
3  t() {
4      let p: Self.P;
5      let w: &u64;
6      let u: &mut Self.S;
7    label b0:
8      p = P { x: S { f: 0 } };
9      w = &copy(&p).P::x.S::f; // deep
10     u = &mut p.P::x;  // re-borrow
11     *move(u) = S { f: 1 }; // error!
12     return;
13 }
```

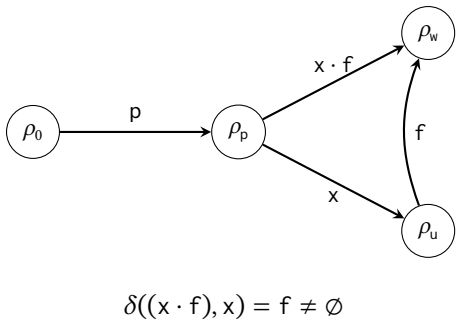

Fig. 5. Implicit aliasing via nested fields. The derivative reveals that ρ_u reaches ρ_w via field `f`.

```
1  t(cond: bool) {
2      let a: u64; let b: u64;
3      let x: &mut u64;
4      let y: &mut u64;
5    label l0:
6      a = 0; b = 0;
7      jump_if (move(cond)) l2;
8    label l1:
9      x = &mut a; y = &mut b; jump l3;
10   label l2:
11     x = &mut b; y = &mut a; jump l3;
12   label l3:
13     *move(x) = 42;
14     *move(y) = 239;
15     return;
16 }
```

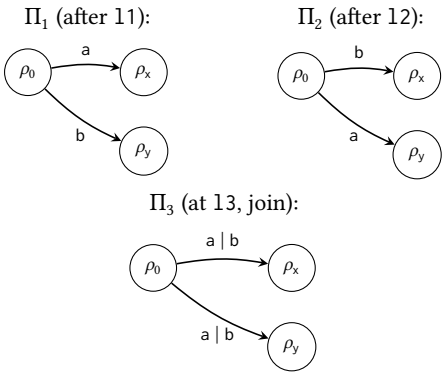

Fig. 6. Tracking borrows across control flow. Labels l1 and l2 are the two branches; l3 is their join point. The path environments Π_1 and Π_2 are unified into Π_3 via subsumption.

Brzozowski derivatives and implicit aliasing. We claimed above that $\mathsf{paths}(\rho_x, \rho_y) = \emptyset$, but how does the type checker arrive at this? When a new reference ρ' is created by borrowing a field f from an existing reference ρ, the paths from ρ' to every other reference ρ'' must be calculated. The type checker computes them via the *Brzozowski derivative* [9]: if $\mathsf{paths}(\rho, \rho'') = r$, then $\mathsf{paths}(\rho', \rho'') = \delta(r, f)$, *i.e.*, the result of "stripping" the prefix f from all words of $\mathcal{L}(r)$:

$$w \in \mathcal{L}(\delta(r, f)) \quad \Longleftrightarrow \quad f \cdot w \in \mathcal{L}(r). \tag{2}$$

In Fig. 4, the path from ρ_x to ρ_y is the derivative $\delta(\mathtt{r}, \mathtt{l}) = \emptyset$, since $\mathtt{l} \neq \mathtt{r}$. Since intersection of paths starting with disjoint fields yields the empty language, their borrows cannot alias.

The path regex derivatives become essential when nested structures create implicit aliasing. Fig. 5 shows a program with two levels of structs: a `P` containing a field `x` of type `S`, which itself has a field `f`. First, `w` borrows the deep path `p.x.f` (line 9), giving $\mathsf{paths}(\rho_p, \rho_w) = \mathtt{x} \cdot \mathtt{f}$. Then `u` reborrows `p.x`. The derivative computes the path from ρ_u to ρ_w: $\mathsf{paths}(\rho_u, \rho_w) = \delta((\mathtt{x} \cdot \mathtt{f}), \mathtt{x}) = \mathtt{f}$. The derivative strips the shared prefix `x`, revealing that ρ_u can, in fact, reach ρ_w via field `f`. The write on line 11 triggers $\mathsf{check\_outbound}(\rho_u)$, which fails because $\mathcal{L}(\mathsf{paths}(\rho_u, \rho_w)) = \mathcal{L}(\mathtt{f}) \not\subseteq \{\varepsilon\}$: overwriting the entire `S` behind `u` would invalidate the deep borrow `w`.

2.5 Type-Checking Control Flow

Fig. 6 shows a program where the two branches (line 9 and line 11) assign `x` and `y` to different variables. After the join at `l3` (line 12), the type checker cannot distinguish which branch was taken. To continue type checking, the environments from both branches must be unified into a single post-join environment via *subsumption*. A path environment Π_1 *subsumes* Π_2 (written $\Pi_1 \sqsupseteq \Pi_2$) if Π_1 is "more restrictive" than Π_2: for every pair of abstract references ρ, ρ', the language of the path in Π_2 is contained in that of Π_1. Intuitively, if a program type-checks under the more restrictive environment Π_1 (which tracks more paths between abstract references), it also type-checks under the less restrictive Π_2. At join points, the checker verifies that each branch's post-environment subsumes a common target environment, which is then used for the continuation.

In the example in Fig. 6, after `l1` the path environment Π_1 maps $\mathsf{paths}(\rho_0, \rho_x) = \mathtt{a}$ and, similarly, $\mathsf{paths}(\rho_0, \rho_y) = \mathtt{b}$; after `l2` the path environment Π_2 reverses the mapping. The target path environment Π_3 at `l3` unifies both cases: each edge becomes the *union* of the corresponding paths from the two branches, giving $\mathsf{paths}(\rho_0, \rho_x) = \mathtt{a} \mid \mathtt{b}$ and $\mathsf{paths}(\rho_0, \rho_y) = \mathtt{a} \mid \mathtt{b}$ (Fig. 6, bottom). The target Π_3 subsumes each branch environment: $\mathcal{L}(\mathtt{a}) \subseteq \mathcal{L}(\mathtt{a} \mid \mathtt{b})$ and likewise for `b`, so every

```
1  struct Point { x: u64, y: u64 }
2  struct Box { tl: Self.Point, br: Self.Point }
3  borrow(p: &mut Self.Box)
4      : &mut Self.Point {
5    label b0:
6      return &mut copy(p).Box::tl;
7  }
8  write(b: &mut Self.Box)
9      : &mut Self.Point {
10     let p: &mut Self.Point;
11     let px: &mut u64;
12     let py: &mut u64;
13   label b0:
14     p = Self.borrow(copy(b));
15     px = &mut copy(p).Point::x;
16     py = &mut copy(p).Point::y;
17     *move(px) = 5;  // safe
18     *move(py) = 8;  // safe
19     return move(p);
20 }
```

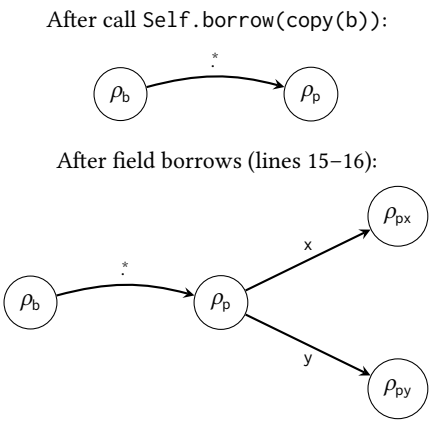

Fig. 7. Function call. The returned reference ρ_p is connected to ρ_b via $\overset{*}{.}$; both writes to the mutable field references borrowed from the call's result are safe because $\text{paths}(\rho_{px}, \rho_{py}) = \emptyset$.

path tracked by Π_1 or Π_2 is also tracked by Π_3. After the join, the writes on lines 13–15 are safe. In each branch, x and y borrow different variables, so $\text{paths}(\rho_x, \rho_y) = \emptyset$ in both Π_1 and Π_2; the union $\emptyset \mid \emptyset = \emptyset$ persists in Π_3, and $\text{check\_outbound}(\rho_x)$ passes. By symmetry, $\text{paths}(\rho_y, \rho_x) = \emptyset$ and $\text{check\_outbound}(\rho_y)$ passes as well, so the program is accepted by the borrow checker.

2.6 Tracking Borrows through Function Calls via the Kleene Star

Function calls are the most intricate aspect of the type system, because the caller cannot see the callee's body and must conservatively account for any borrow chains it might create.

Fig. 7 shows a program that borrows two disjoint fields of a struct through a function call. The function `borrow` (lines 3–7) returns a mutable reference to the `tl` field of its argument. When calling `write` (line 14), the type checker does not know exactly which field was borrowed, it only knows the callee's type signature. Therefore, at a call site, the checker conservatively extends the caller's path environment to account for borrow chains the callee may create. For each mutable input reference ρ and each returned mutable reference ρ', the inbound path is set to $\text{paths}(\rho, \rho') = \overset{*}{.}$ (dot-star), where the dot (.) matches any single field name and the Kleene star allows repetition. No outbound path from ρ' (call output) to ρ (call input) is added: adding a reverse edge would cause $\text{check\_outbound}(\rho')$ to fail, preventing the caller from writing through the returned reference. For immutable inputs and outputs, the checker additionally records outbound paths to prevent writes that would invalidate shared borrows.[1]

In the example, the returned reference p is assigned a fresh abstract reference ρ_p so that we have $\text{paths}(\rho_b, \rho_p) = \overset{*}{.}$ (Fig. 7, right). The subsequent field borrows on lines 15–16 create ρ_{px} and ρ_{py} with $\text{paths}(\rho_p, \rho_{px}) = x$ and $\text{paths}(\rho_p, \rho_{py}) = y$. Despite the imprecision of $\overset{*}{.}$, since ρ_{px} borrows field x from ρ_p, the derivative rule (2) gives $\text{paths}(\rho_{px}, \rho_{py}) = \delta(\text{paths}(\rho_p, \rho_{py}), x) = \delta(y, x) = \emptyset$, so the two borrows are disjoint and both writes on lines 17–18 are safe. The deployed Move checker [8] achieves the same precision for disjoint field borrows using specialised graph operations (factor_f, elim) that explicitly decompose and propagate borrow edges. Our regex-based

[1] Strictly speaking, the Kleene star is not necessary for our language, since MoveIR lacks recursive data types and all borrow chains therefore have bounded depth. However, $\overset{*}{.}$ provides a uniform, compact over-approximation that avoids computing type-specific depth bounds and will naturally extend to recursive data types in the future.

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

Fig. 8. MoveLight syntax. Metavariable a ranges over sites, x over variables, f over fields, L over labels.

approach subsumes these operations with a single uniform mechanism: the extend operation, where Brzozowski derivatives handle edge decomposition automatically. The $.^*$ extension conservatively covers all reachable fields, yet subsequent derivatives can still distinguish disjoint paths ($\delta(\mathsf{r}, \mathsf{l}) = \emptyset$) from overlapping ones ($\delta((\mathsf{x} \cdot \mathsf{f}), \mathsf{x}) = \mathsf{f}$).

The call protocol above is complemented by the static checks done at return sites. At a `return` statement, the type checker verifies that the returned values have the declared return types and that no returned reference has a path from the root ρ_0 (otherwise the caller would receive a dangling pointer to a local variable). Additionally, every returned mutable reference must pass check_outbound, ensuring it has no non-trivial outbound paths to non-returned references. No returned mutable reference may alias any other returned reference: the language $\mathcal{L}(\mathsf{paths}(\rho_j, \rho_i))$ must be empty (not merely $\subseteq \{\varepsilon\}$) when ρ_i is mutable; immutable returns may alias freely. In Fig. 7, the return on line 6 is safe: the returned reference borrows a field of the input parameter `p` (not a local), so it has no path from ρ_0, and check_outbound passes since it is the only returned value.

3 Formal Model

We now present MoveLight, a formal calculus that faithfully and completely captures the essence of MoveIR, together with our novel typing rules, operational semantics, an soundness argument.

3.1 MoveLight

Fig. 8 defines the grammar of MoveLight. Types are either *basic* types (`u64`, `bool`, `unit`, records) or *reference* types $\mathsf{ref}(\tau, \rho, k)$, where τ is the referent type, ρ is an abstract reference, and k is the borrowing kind (mutable or immutable). Embedding ρ directly in the type simplifies environment bookkeeping: the typing rules can read and update a reference's identity without a separate lookup, while type compatibility for reference types ignores the ρ component. Abstract references come in three kinds: ρ_0 represents the local stack frame, ρ_n is allocated at borrow time, and ρ_x is introduced by a function parameter. The type system treats these kinds differently: parameter references may outlive the callee's frame, so the return rule must ensure that no allocated reference escapes.

MoveLight is faithful to MoveIR but uses the *Administrative Normal Form* (ANF) [18]: every intermediate result is bound to a named temporary called a *site*. A site is a typed slot in the current stack frame that holds the value produced by an expression before it is consumed by a subsequent operation. Expressions produce values bound to sites; statements consume sites and update the

type environment. ANF makes the typing rules compositional: each rule transforms the environment by producing or consuming sites, avoiding the need for a separate expression-level typing judgement. Most statements carry a *continuation* s (the rest of the block); `ret`, `jump`, and `branch` are terminal and have none. Among the less standard forms: pack assembles a record from sites, unpack destructures one, and release explicitly drops a reference (consuming the site and removing its abstract reference from the path environment). Assignment $x := a$ assigns a value into a variable, while $*a_1 := a_2$ stores a value through a mutable reference. A call $g(\overline{b})$ may return multiple values, bound to a tuple of sites.

3.2 Type System

The type system is syntax-directed: each typing rule corresponds to a single syntactic form and transforms a *type environment* $\mathcal{E} = (\Gamma, \Sigma, \Pi)$. The *variable environment* $\Gamma : \mathit{Var} \to \mathit{Flag} \times T \times \mathit{Mut}$ maps each variable to a validity flag (valid or invalid after a move), a type, and a mutability annotation. The *site environment* $\Sigma : \mathit{Site} \to T$ maps each site to its type; sites are consumed after use—removing a site from Σ models the single-use consumption of an intermediate value. The *path environment* $\Pi = (\mathit{refs}, \mathrm{paths})$ tracks a set of abstract references and a function $\mathrm{paths}(\rho, \rho') : \mathit{Regex}$ that maps each pair to a regular expression over path elements (field accesses and root-to-variable links), recording the reachability between them. The image of paths uses standard regex forms: the empty language $\emptyset$, the empty word ε, single path elements $\mathtt{f}$, concatenation, union, and Kleene star. The semantics of the derivative $\delta(r, \mathtt{f})$ is as per (2). Three decidable operations are used by the typing rules: $\nu(r)$ checks whether $\mathcal{L}(r)$ is non-empty; only_matches_empty(r) decides $\mathcal{L}(r) \subseteq \{\varepsilon\}$ (the predicate underlying check_outbound); and has_nonempty_match(r) conservatively checks $\exists w \neq \varepsilon.\ w \in \mathcal{L}(r)$ (used in the return rule to detect non-trivial aliasing). Our Lean mechanisation verifies their correctness against the semantic definition of regular language membership.

A key invariant of the path environment is that it is *transitively closed*: $\mathrm{paths}(\rho, \rho')$ accounts for *all* intermediate paths through other references, so reachability between any two nodes can be read off directly. Two operations maintain this invariant. The *extension* $\mathrm{extend}(\Pi, \rho, \rho', p)$ adds a fresh reference ρ' to the tracked set with an edge $\mathrm{paths}(\rho, \rho') = p$, and restores transitive closure: for every existing reference ρ'', *inbound* edges use concatenation, $\mathrm{paths}(\rho'', \rho') = \mathrm{paths}(\rho'', \rho) \cdot p$, while *outbound* edges use the Brzozowski derivative, $\mathrm{paths}(\rho', \rho'') = \delta(\mathrm{paths}(\rho, \rho''), p)$. The derivative captures the key insight: extending ρ by path expression p means that ρ' reaches anything that ρ reaches, but with the prefix p stripped. When $p = \varepsilon$ (as in T-CopyRef and T-Freeze—*cf.* Fig. 9), the extension is symmetric: both inbound and outbound edges of ρ' copy those of ρ, making ρ' a full alias. *Node removal* $\Pi \setminus \{\rho\}$ removes ρ from the tracked set and clears all edges involving ρ. Since the graph is maintained transitively closed by extend, no reachability information is lost: any path that formerly passed through ρ is already recorded as a direct edge between the remaining references. Without node removal, references that have been consumed (by a write or release) would remain in the tracked set, causing check_outbound to reject programs that are in fact safe.

All typing rules have the form $\Lambda;\ \mathcal{E} \vdash s : \overline{T}$, where Λ is the *label environment* mapping each block label to the type environment expected at its entry (needed for `jump` and `branch`), $\mathcal{E}$ is the current type environment, s is the statement being checked, and $\overline{T}$ are the expected return types. The full judgement also carries a function environment for resolving calls; we omit it here for brevity. A function is well-typed when every block in its body is well-typed under this judgement; the top-level definitions are standard and can be found in our Lean mechanisation. We abuse the set notation and write $\Sigma \setminus \{a\}$ for the site environment with site a removed. In the rest of this subsection, we focus on the most interesting typing rules that govern borrowing, ownership transfer, and control flow. The complete collection of rules is available in our Lean mechanisation.

$$\text{T-CopyRef}\ \frac{\Gamma(x) = (\text{valid}, \text{ref}(\tau, \rho, k), \_) \qquad a \notin \text{dom}(\Sigma) \qquad \rho' \text{ fresh} \qquad \Pi' = \text{extend}(\Pi, \rho, \rho', \varepsilon) \qquad \Lambda;\ \mathcal{E}[\Sigma(a) \mapsto \text{ref}(\tau, \rho', k),\ \Pi := \Pi'] \vdash s : \overline{T}}{\Lambda;\ \mathcal{E} \vdash \text{let } a = \text{copy}(x);\ s : \overline{T}}$$

$$\text{T-Move}\ \frac{\Gamma(x) = (\text{valid}, T, \_) \qquad \text{not\_borrowed}(x, \mathcal{E}) \qquad a \notin \text{dom}(\Sigma) \qquad \Lambda;\ \mathcal{E}[\Gamma(x) := (\text{invalid}, T, \_),\ \Sigma(a) \mapsto T] \vdash s : \overline{T}}{\Lambda;\ \mathcal{E} \vdash \text{let } a = \text{move}(x);\ s : \overline{T}}$$

$$\text{T-BorrowMut}\ \frac{\Gamma(x) = (\text{valid}, \text{basic}(\tau), \mathsf{M}) \qquad a \notin \text{dom}(\Sigma) \qquad \rho \text{ fresh} \qquad \Pi' = \text{extend}(\Pi, \rho_0, \rho, x) \qquad \Lambda;\ \mathcal{E}[\Sigma(a) \mapsto \text{ref}(\tau, \rho, \mathsf{M}),\ \Pi := \Pi'] \vdash s : \overline{T}}{\Lambda;\ \mathcal{E} \vdash \text{let } a = \text{borrow}_{\mathsf{M}}(x);\ s : \overline{T}}$$

$$\text{T-BorrowField}\ \frac{\Sigma(a) = \text{ref}(\{\ldots, f : \tau', \ldots\}, \rho, k) \qquad a_f \notin \text{dom}(\Sigma) \qquad \rho_f \text{ fresh} \qquad \Pi' = \text{extend}(\Pi, \rho, \rho_f, f) \qquad \Lambda;\ \mathcal{E}[\Sigma := \Sigma \setminus \{a\},\ \Sigma(a_f) \mapsto \text{ref}(\tau', \rho_f, k),\ \Pi := \Pi'] \vdash s : \overline{T}}{\Lambda;\ \mathcal{E} \vdash \text{let } a_f = \text{borrow-field}_k(a, \tau, f);\ s : \overline{T}}$$

$$\text{T-WriteRef}\ \frac{\Sigma(a) = \text{ref}(\tau, \rho, \mathsf{M}) \qquad \Sigma(b) = \text{basic}(\tau) \qquad \text{check\_outbound}(\Pi, \rho) \qquad \Pi' = \Pi \setminus \{\rho\} \qquad \Lambda;\ \mathcal{E}[\Sigma := \Sigma \setminus \{a, b\},\ \Pi := \Pi'] \vdash s : \overline{T}}{\Lambda;\ \mathcal{E} \vdash *a := b;\ s : \overline{T}}$$

$$\text{T-Freeze}\ \frac{\Sigma(a) = \text{ref}(\tau, \rho, \mathsf{M}) \qquad a_f \notin \text{dom}(\Sigma) \qquad \rho_f \text{ fresh} \qquad \Pi' = \text{extend}(\Pi, \rho, \rho_f, \varepsilon) \qquad \Lambda;\ \mathcal{E}[\Sigma := \Sigma \setminus \{a\},\ \Sigma(a_f) \mapsto \text{ref}(\tau, \rho_f, \mathsf{I}),\ \Pi := \Pi'] \vdash s : \overline{T}}{\Lambda;\ \mathcal{E} \vdash \text{let } a_f = \text{freeze}(a);\ s : \overline{T}}$$

Fig. 9. Typing rules for borrowing and ownership transfer.

3.2.1 Borrowing and Ownership Transfer. Fig. 9 presents the rules for creating, consuming, and transferring references, which form the core of MoveLight's regex-based borrow checker. In the rules below, we write Γ, Σ, and Π for the components of the environment $\mathcal{E}$ in a rule's conclusion.

T-CopyRef duplicates a reference-typed variable x into a fresh site a. Because the copy creates a second reference to the same heap location, a fresh abstract reference ρ' is allocated and connected to the original ρ via ε (the empty path), recording that ρ' and ρ are aliases. The variable remains valid after the copy; copying a basic (non-reference) value is simpler and omitted, as it requires no path environment update. T-Move transfers ownership of x into a, which works for any type, including references. The premise $\text{not\_borrowed}(x, \mathcal{E})$ checks that no tracked reference currently borrows into x, specifically, no outbound path from ρ_0 begins with the root-to-variable path element for x. After the move, x is marked invalid so that its any subsequent use attempt is statically rejected, while the moved value becomes available through the freshly bound site a.

T-BorrowMut creates a mutable reference to variable x, which must be valid, carry a basic (non-reference) type, and be declared mutable. A fresh abstract reference ρ is allocated and the path environment is extended with an edge from ρ_0 to ρ via the root-to-variable path element for x, recording that ρ borrows x; the new site a receives the reference type. T-BorrowImm (not shown) is analogous but produces an immutable reference and does not require x to be mutable. T-BorrowField borrows a single field f from a reference site a, consuming a and producing a new site a_f with a fresh abstract reference ρ_f connected to the parent's reference ρ via field f. The Brzozowski derivative (Sec. 2.2) is implicit in extend: the transitive path from any other reference ρ' to ρ_f is computed as $\delta(\text{paths}(\rho', \rho), f)$, stripping the prefix f from the reachability language.

T-WriteRef writes the value in site b through the mutable reference in site a. The key premise is $\text{check\_outbound}(\Pi, \rho)$: for every tracked reference ρ' the language $\mathcal{L}(\text{paths}(\rho, \rho')) \subseteq \{\varepsilon\}$, which

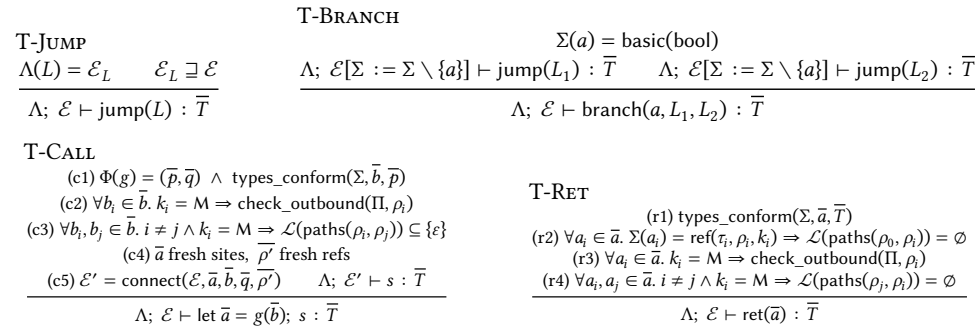

Fig. 10. Typing rules for control flow and function calls.

guarantees that no existing borrow extends beyond ρ itself (Sec. 2.3), preventing creation of dangling references. After the write, both sites are consumed and ρ is removed from Π. T-Freeze converts a mutable reference to an immutable one by allocating a fresh abstract reference ρ_f connected to the original via ε (the empty path), expressing that both references point to the same heap location; the original mutable site is consumed and replaced by the new immutable site.

3.2.2 Control Flow. Fig. 10 presents the rules for control flow: jumps, function calls, and returns. T-Jump checks that the target label's environment $\mathcal{E}_L$ subsumes the current environment $\mathcal{E}$. This is the mechanism through which branches are unified at join points (Sec. 2.5). Subsumption requires a bijective reference substitution σ that maps the abstract references in $\mathcal{E}_L$ to those in $\mathcal{E}$, preserving types and ensuring that paths in $\mathcal{E}_L$ are contained in the corresponding paths of $\mathcal{E}$:

$$\mathcal{E}_L \sqsupseteq \mathcal{E} \ \stackrel{\text{def}}{=}\ \exists\sigma.\ \forall\rho, \rho'.\ \mathcal{L}(\text{paths}(\sigma(\rho), \sigma(\rho'))) \subseteq \mathcal{L}(\text{paths}(\rho, \rho')).$$

The direction of inclusion ensures that the current environment $\mathcal{E}$ is at least as restrictive as the target $\mathcal{E}_L$: the current state tracks at least as many paths, so any aliasing conflict detected by $\mathcal{E}_L$ is also visible in $\mathcal{E}$. T-Branch consumes a boolean site and reduces the branching check to two individual T-Jump judgements, one per target label.

The typing rules for function calls and returns are where the regex-based borrow tracking using path environment enables precise interprocedural alias analysis without whole-program reasoning. Calls and returns are tightly intertwined: the conditions enforced by T-Ret at the callee side are exactly what allow T-Call to safely connect the caller's environment with the callee's outputs.

T-Call is the most involved rule in MoveLight, featuring the following premises:

(c1) The function environment Φ provides g's signature: parameter types $\overline{p}$ and return types $\overline{q}$. The input sites $\overline{b}$ must conform to the parameter types, matching basic types exactly and reference types up to abstract reference renaming.

(c2) Every mutable input must pass check_outbound: the callee may write through any mutable parameter, so the same isolation check used by T-WriteRef is enforced at the call boundary.

(c3) Every mutable input must be isolated from all other inputs: for distinct b_i, b_j where b_i is mutable, $\mathcal{L}(\text{paths}(\rho_i, \rho_j)) \subseteq \{\varepsilon\}$. This ensures that mutable arguments cannot alias other arguments, mirroring condition (r4) at the return site.

(c4) Fresh output sites $\overline{a}$ and fresh abstract references $\overline{\rho'}$ are allocated for reference-typed returns.

(c5) The connect operation builds the post-call environment $\mathcal{E}'$: it populates output sites with types from $\overline{q}$, consumes input sites $\overline{b}$, and extends the path environment to relate each returned reference ρ' to the caller's existing references. For mutable outputs, only inbound

$$\textsc{S-Copy}\ \frac{\mathcal{V}(x) = \text{Some}(\ell) \qquad \mathcal{H}(\ell) = v}{\langle a = \text{copy}(x); s,\ \mathcal{V},\ S,\ \mathcal{H}\rangle \rightsquigarrow \langle s,\ \mathcal{V},\ S[a \mapsto v],\ \mathcal{H}\rangle}$$

$$\textsc{S-Move}\ \frac{\mathcal{V}(x) = \text{Some}(\ell) \qquad \mathcal{H}(\ell) = v}{\langle a = \text{move}(x); s,\ \mathcal{V},\ S,\ \mathcal{H}\rangle \rightsquigarrow \langle s,\ \mathcal{V}[x \mapsto \text{None}],\ S[a \mapsto v],\ \mathcal{H}\rangle}$$

$$\textsc{S-Borrow}\ \frac{\mathcal{V}(x) = \text{Some}(\ell)}{\langle a = \text{borrow}_k(x); s,\ \mathcal{V},\ S,\ \mathcal{H}\rangle \rightsquigarrow \langle s,\ \mathcal{V},\ S[a \mapsto \text{ref}(\ell, [])],\ \mathcal{H}\rangle}$$

$$\textsc{S-BorrowField}\ \frac{S(a) = \text{ref}(\ell, p)}{\langle a_f = \text{borrowField}_k(a, f); s,\ \mathcal{V},\ S,\ \mathcal{H}\rangle \rightsquigarrow \langle s,\ \mathcal{V},\ S[a_f \mapsto \text{ref}(\ell, p \cdot f)],\ \mathcal{H}\rangle}$$

$$\textsc{S-ReadRef}\ \frac{S(a) = \text{ref}(\ell, p) \qquad \mathcal{H}.\text{read}(\ell, p) = v}{\langle a' = \text{readRef}(a); s,\ \mathcal{V},\ S,\ \mathcal{H}\rangle \rightsquigarrow \langle s,\ \mathcal{V},\ S[a' \mapsto v],\ \mathcal{H}\rangle}$$

$$\textsc{S-WriteRef}\ \frac{S(a) = \text{ref}(\ell, p) \qquad S(b) = v \qquad \mathcal{H}.\text{write}(\ell, p, v) = \mathcal{H}'}{\langle *a := b;\ s,\ \mathcal{V},\ S,\ \mathcal{H}\rangle \rightsquigarrow \langle s,\ \mathcal{V},\ S,\ \mathcal{H}'\rangle}$$

$$\textsc{S-Assign}\ \frac{S(a) = v \qquad \mathcal{H}.\text{alloc}(v) = (\mathcal{H}', \ell)}{\langle x := a;\ s,\ \mathcal{V},\ S,\ \mathcal{H}\rangle \rightsquigarrow \langle s,\ \mathcal{V}[x \mapsto \ell],\ S,\ \mathcal{H}'\rangle}$$

$$\textsc{S-Jump}\ \frac{\text{block}(L) = s'}{\langle \text{jump}(L),\ \mathcal{V},\ S,\ \mathcal{H}\rangle \rightsquigarrow \langle s',\ \mathcal{V},\ \emptyset,\ \mathcal{H}\rangle}$$

$$\textsc{S-Call}\ \frac{\Phi(g) = (\overline{p}, s_g) \qquad S(\overline{b}) = \overline{v} \qquad \mathcal{H}.\text{allocArgs}(\overline{p}, \overline{v}) = (\mathcal{H}', \mathcal{V}_g)}{\left\langle \overline{a} = g(\overline{b});\ s,\ \mathcal{V},\ S,\ \mathcal{H}\right\rangle \rightsquigarrow \left\langle s_g,\ \mathcal{V}_g,\ \emptyset,\ \mathcal{H}'\right\rangle\ \text{push}(\mathcal{V}, S, s, \overline{a})}$$

$$\textsc{S-Ret}\ \frac{S(\overline{a}) = \overline{v} \qquad \text{stack} = (\mathcal{V}_c, S_c, s_c, \overline{a_r}) :: \mathit{rest}}{\langle \text{ret}(\overline{a}),\ \mathcal{V},\ S,\ \mathcal{H}\rangle \rightsquigarrow \langle s_c,\ \mathcal{V}_c,\ S_c[\overline{a_r} \mapsto \overline{v}],\ \mathcal{H}\rangle}$$

Fig. 11. Selected small-step semantic rules. State components: $\mathcal{V}$ (variable store), S (site store), $\mathcal{H}$ (heap).

paths (from caller references to ρ') are added, using $.^*$ to over-approximate the callee's internal borrowing; no outbound paths are added, so the returned mutable reference is immediately writable. For immutable outputs, outbound paths are also added, reflecting that immutable references do not require exclusive access. Additionally, connect adds paths from ρ_0 to each output reference, so that the caller's stack frame now reaches the returned values.

T-Ret enforces four safety conditions on the returned sites $\overline{a}$:

(r1) Types of $\overline{a}$ match the declared return signature $\overline{T}$.

(r2) No returned reference has a path from ρ_0: such a path would indicate borrowing a local variable, leading to a dangling reference after the function returns.

(r3) Returned mutable references pass check_outbound, so the caller can write through them.

(r4) No returned mutable reference may alias any other returned reference: for distinct a_i, a_j where a_i is mutable, the path language $\mathcal{L}(\text{paths}(\rho_j, \rho_i))$ must be empty (not merely $\subseteq \{\varepsilon\}$). Immutable returns may alias freely.

Notice the interplay between the two rules: the return-side conditions (r2)–(r4) establish invariants that the call-side connect (c5) relies on. Because the callee guarantees that returned mutable references are outbound-safe and non-aliasing, connect can add inbound paths for them without introducing conflicts in the caller's path environment.

3.3 Operational Semantics

A MoveLight machine state $\langle s, \mathcal{V}, S, \mathcal{H}\rangle$ consists of the current statement s, a variable store $\mathcal{V} : \mathit{Var} \to \mathit{Loc}_\perp$ mapping variables to optional heap locations, a site store $S : \mathit{Site} \to \mathit{Value}$ holding intermediate values, and a heap $\mathcal{H} : \mathit{Loc} \to \mathit{Value}$. A call stack of saved frames supports function calls and returns. The heap provides three operations: $\mathcal{H}(\ell)$ retrieves the value at location ℓ; $\mathcal{H}.\text{read}(\ell, p)$ navigates a field path p within the value at ℓ, returning a sub-value; $\mathcal{H}.\text{write}(\ell, p, v)$ replaces that sub-value in place. $\mathcal{H}.\text{alloc}(v)$ allocates a fresh location and returns the updated heap together with the new location; $\mathcal{H}.\text{allocArgs}$ extends this to allocate a list of parameter values.

Fig. 11 presents selected transition rules; we highlight several. S-Move copies the value from x's heap location into site a and sets $\mathcal{V}(x)$ to None, making any subsequent access to x a runtime error—the run-time counterpart of the typing rule's invalid marker. S-WriteRef is the most delicate: it reads the reference in a to obtain a location-path pair (ℓ, p), then uses $\mathcal{H}$.write to overwrite the sub-value at path p within $\mathcal{H}(\ell)$; the typing rule T-WriteRef (Fig. 9) guarantees that old and new values share the same type, so the heap remains well-formed. S-Call pushes the current frame (including the continuation s and result sites $\overline{a}$) onto the stack via push, allocates the callee's parameters on the heap, and jumps to the callee's entry block with a fresh site store; S-Ret pops the stack and binds returned values to the caller's result sites.

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

Fig. 12. Classification of runtime errors.

The figure shows only successful transitions; when a premise fails, *e.g.*, $\mathcal{V}(x) = \text{None}$ in S-Copy or S-Borrow, or $\mathcal{H}$.read(ℓ, p) returning nothing in S-ReadRef—the machine transitions to one of the error states classified in Fig. 12. Our mechanisation uses a standard fuel-based semantics [1, 38]: the step function consumes one unit of fuel per transition and returns outOfFuel when the supply is exhausted. The rules in Fig. 11 elide this counter for clarity; the soundness theorem (Theorem 3.1) quantifies over all fuel values, ensuring that no finite execution reaches a preventable error.

When a safety violation is detected at runtime, the machine transitions to an error state. Fig. 12 classifies the runtime errors; those in the *preventable* section are precisely those excluded by our type soundness theorem. The distinction between preventable and acceptable errors is a design choice: division by zero, explicit aborts, and running out of gas (execution fuel) are runtime-dependent and cannot be statically excluded without significantly restricting the language or making type system too complicated. The presented type system guarantees that, given well-formed inputs, no well-typed program reaches any of the preventable error states.

3.4 Soundness

The type system guarantees that a well-typed program never reaches a preventable error state (Fig. 12). Among the preventable errors, the most interesting are `danglingRef` (accessing a freed heap location, prevented by the path environment's tracking of borrow lifetimes), `uninitializedVar` (use-after-move, prevented by the invalid marker in Γ), and `typeMismatch` (writing a value of the wrong type through a reference, prevented by the check_outbound check in T-WriteRef). The remaining preventable errors—uninitialised sites, arity mismatches, unknown functions and labels, invalid field accesses—are excluded by straightforward syntactic checks in the typing rules.

Theorem 3.1 (Type Soundness). *If function f is well-typed, all callees satisfy their type signatures, and the initial configuration satisfies the soundness assumptions, then for every preventable error e and every number of steps n, executing f for n steps does not produce e.*

The proof follows the standard progress/preservation technique [43]. Progress is straightforward: given a well-typed state, the machine can either take a step or has reached a terminal ret, and any step it takes does not produce a preventable error. Preservation is the actual challenging part: it proceeds by case analysis on the typing rule that typed the current statement (41 cases in total in our formalisation) and maintains a *well-typed state invariant* connecting the type environment to the concrete machine state via a *reference map* $\mu : \mathit{Aref} \to \mathit{Loc}$ that translates abstract references to heap locations. The invariant comprises 29 clauses, grouped as follows:

- *Variable and site validity*: each valid variable in the environment Γ corresponds to a well-typed value at its heap location; the same holds true for each site in Σ.
- *Reference tracking*: every abstract reference in $\Pi.refs$ maps via the maintained reference map μ to a live heap location; μ is injective on tracked references.
- *Path–heap coherence*: if $\mathsf{paths}(\rho, \rho')$ matches a field sequence p, then navigating along p from $\mu(\rho)$ in the heap reaches $\mu(\rho')$—the key clause linking abstract paths to concrete memory.
- *Heap typing*: every heap value has the type assigned by μ.
- *Borrow safety*: at any point, mutable references have no non-trivial outbound paths, which is the key to establish the write isolation outlined Sec. 2.3.
- *Structural*: new references are fresh, self-loop-only for ρ_0, no paths to/from removed references.

Two auxiliary lemmas are essential for preservation. The first is *weakening*, used at control-flow joins (Sec. 2.5) and function calls (Sec. 2.6). Recall that subsumption $\mathcal{E}_1 \sqsupseteq \mathcal{E}_2$ is defined for type environments (Sec. 3.2.2), but primarily concerns path environments: there must exist a bijective reference substitution σ such that $\mathcal{L}(\mathsf{paths}(\sigma(\rho), \sigma(\rho'))) \subseteq \mathcal{L}(\mathsf{paths}(\rho, \rho'))$ for all pairs ρ, ρ'.

Lemma 3.2 (Weakening). *If* $\mathcal{E}_1 \sqsupseteq \mathcal{E}_2$ *and* $\Lambda;\ \mathcal{E}_1 \vdash s : \overline{T}$, *then* $\Lambda;\ \mathcal{E}_2 \vdash s : \overline{T}$.

When branches merge, the checker types the continuation under a target environment $\mathcal{E}_L$ and verifies that each branch's post-environment $\mathcal{E}$ satisfies $\mathcal{E}_L \sqsupseteq \mathcal{E}$; weakening then guarantees that the continuation, already well-typed under $\mathcal{E}_L$, remains well-typed under $\mathcal{E}$. The proof threads σ through every environment component, verifying that variable types, site types, and path regexes are all preserved under the substitution. The second lemma concerns writes through references:

Lemma 3.3 (Regex Path Transfer). *If values* v_1 *and* v_2 *have the same type* τ *and navigating the field sequence* p *through* v_1 *succeeds (*i.e.*, each field lookup along* p *returns a sub-value), then navigating* p *through* v_2 *also succeeds.*

Path transfer is required by the T-WriteRef preservation case: after S-WriteRef replaces a sub-value at a heap location, all existing regex paths through that location must remain valid. The proof proceeds by induction on the type τ: since v_1 and v_2 share the same type, they have identical field structure at every level, so every field path readable in one is readable in the other. The non-trivial aspect is that regex-summarised paths in the path environment encode *sets* of concrete field paths; the proof must show that for every word accepted by a path regex, the corresponding traversal succeeds in the new heap. This requires a case split on each regex constructor and, in the concatenation case, the fact that if $f \cdot w \in \mathcal{L}(r_1 \cdot r_2)$ then $w \in \mathcal{L}(\delta(r_1, f) \cdot r_2)$, *i.e.*, the derivative strips the field already traversed, reducing the problem to a shorter path.

3.5 Decidability and Worst-Case Complexity

The type system is also *efficiently decidable*. The algorithmic checker (Sec. 5.2) is a single syntax-directed forward pass, which performes no search or backtracking, and every regex operation it uses is polynomial in the size of the regexes involved: the Brzozowski derivative, the inclusion $\mathcal{L}(r) \subseteq \{\varepsilon\}$ underlying check_outbound, and the disjointness test $\mathcal{L}(r_1) \cap \mathcal{L}(r_2) = \emptyset$ at calls and returns. Crucially, none of these is the PSPACE-complete language-equivalence (or universality) problem; emptiness, emptiness-of-intersection, and inclusion in the singleton $\{\varepsilon\}$ are all polynomial. In practice, our regexes also stay small: as MoveIR has no recursive types, every path language is finite (*cf.* the footnote in Sec. 2.6), so a function with r references induces an $r \times r$ table of bounded regexes, and checking is *modular*, since each call is verified against the callee's signature alone, the cost of checking never accumulates across the call graph.

```
1  t() {
2      let v: vector<u64>;
3      let e: &mut u64;
4    label b0:
5      v = vec_pack_1<u64>(0);
6      e = vec_mut_borrow<u64>(
7            &mut v, 0);
8      *move(e) = 42;
9      return;
10 }
```

```
1  t() {
2      let v: vector<u64>;
3      let e1: &u64;
4      let e2: &mut u64;
5    label b0:
6      v = vec_pack_0<u64>();
7      vec_push_back<u64>(&mut v, 0);
8      vec_push_back<u64>(&mut v, 1);
9      e1 = vec_imm_borrow<u64>(&v, 0);
10     e2 = vec_mut_borrow<u64>(
11           &mut v, 1); // error!
12     return;
13 }
```

Fig. 13. Vector examples in MoveIR. Left: borrowing an element and writing through it (accepted). Right: two simultaneous borrows on the same vector (rejected at line 11).

4 Extensions

We extend the core type system with vectors and variant types, demonstrating that our approach accommodates these features with modest additions to the rules and the soundness argument.

4.1 Vectors

Move provides built-in vectors with eight primitive operations covering construction and destruction (pack, unpack), querying length, immutable and mutable element borrowing, appending and removing elements (push-back, pop-back), and swapping two elements. Fig. 13 (left) shows a program that borrows a vector element mutably and writes through the resulting reference. Vector elements are modelled as numeric field indices: borrowing element i of a vector reference creates a fresh abstract reference connected to the vector's abstract reference via a single path element vecElem. Since all elements share the same path element, two borrows of distinct indices are *not* distinguished by the type system; this is a conservative but sound approximation that matches the behaviour of the deployed Move checker. Fig. 13 (right) is thus rejected: after borrowing element 0 immutably, the mutable borrow of element 1 triggers check_outbound(Π, ρ_v), which fails because $\mathcal{L}(\mathsf{paths}(\rho_v, \rho_{e1})) = \mathcal{L}(\mathsf{vecElem}) \not\subseteq \{\varepsilon\}$, *i.e.*, the vector already has an outstanding borrow.

Fig. 14 shows two characteristic rules. T-VecMutBorrow is analogous to T-BorrowField (Fig. 9): it consumes the vector reference and the index, produces a fresh reference to the element, and extends the path environment via the same extend operation used for record fields—the only difference is that the path element is vecElem instead of a field name. T-VecPushBack requires check_outbound because push-back may reallocate the vector's internal storage, invalidating any

$$\textsc{T-VecMutBorrow}\ \frac{\begin{array}{c}\Sigma(a) = \mathsf{ref}(\mathsf{vec}(\tau), \rho, \mathsf{M}) \qquad \Sigma(b) = \mathsf{basic}(\mathsf{u64}) \\ a_e \notin \mathsf{dom}\,(\Sigma) \quad \rho_e \text{ fresh} \quad \Pi' = \mathsf{extend}(\Pi, \rho, \rho_e, \mathsf{vecElem}) \\ \Lambda;\ \mathcal{E}[\Sigma := \Sigma \setminus \{a, b\},\ \Sigma(a_e) \mapsto \mathsf{ref}(\tau, \rho_e, \mathsf{M}),\ \Pi := \Pi'] \vdash s : \overline{T}\end{array}}{\Lambda;\ \mathcal{E} \vdash \mathsf{let}\ a_e = \mathsf{vec\_mut\_borrow}(a, b);\ s : \overline{T}}$$

$$\textsc{T-VecPushBack}\ \frac{\begin{array}{c}\Sigma(a) = \mathsf{ref}(\mathsf{vec}(\tau), \rho, \mathsf{M}) \qquad \Sigma(b) = \mathsf{basic}(\tau) \qquad \mathsf{check\_outbound}(\Pi, \rho) \\ \Pi' = \Pi \setminus \{\rho\} \qquad \Lambda;\ \mathcal{E}[\Sigma := \Sigma \setminus \{a, b\},\ \Pi := \Pi'] \vdash s : \overline{T}\end{array}}{\Lambda;\ \mathcal{E} \vdash \mathsf{vec\_push\_back}(a, b);\ s : \overline{T}}$$

Fig. 14. Selected vector typing rules. T-VecMutBorrow creates a mutable borrow of a vector element via extend with path vecElem. T-VecPushBack requires check_outbound because push may reallocate, invalidating element borrows; it then removes all element references from the path environment.

```
1  enum X {
2    One { x: u64 },
3    Two { x: u64, y: u64 },
4  }
5  h(e: &mut Self.X) {
6      let vx: &mut u64;
7      let vy: &mut u64;
8    label b0:
9      &mut X.Two { x: vx, y: vy } = copy(e);
10     *move(vx) = 1;
11     &mut X.One { x: vx } = move(e);
12     *move(vy) = 3;
13     *move(vx) = 2;
14     return;
15 }
```

```
1  enum X {
2    One { x: u64 },
3    Two { x: u64, y: u64 },
4  }
5  k(e: &mut Self.X) {
6      let vx: &u64;
7      let vy: &u64;
8    label b0:
9      &X.Two { x: vx, y: vy } =
10             freeze(copy(e));
11     _ = move(vx);
12     *move(e) = X.One { x: 0 }; // error!
13     abort *move(vy);
14 }
```

Fig. 15. Enum examples in MoveIR. Left: sequential mutable unpacks (accepted). Right: write a different variant while holding an outstanding borrow (rejected at line 12).

existing element borrow. After verifying safety, it removes all references reachable from the vector reference, reflecting the fact that element pointers may have been invalidated.

The vector extension adds no new difficulties to the soundness proof. Vectors reuse the same path environment operations as record fields; the only new ingredient is the vecElem path element. The path transfer Lemma 3.3 extends to vectors because two vectors of the same type have identical element types so the same structural argument applies at each index.

4.2 Multi-Variant Enumerated Types

Move supports non-recursive variant types (enums), with each constructor carrying its own set of named fields. Three operations are responsible for manipulating enum values: `pack_variant` (construct a value), `unpack_variant` (destructure a value, checking the constructor at runtime), and `variant_switch` (dispatch on the constructor, analogous to `match`). Fig. 15 shows two examples. The left program mutably unpacks an enum reference twice, each time accessing the fields of the matching variant; this is safe because the second unpack via `move(e)` consumes the enum reference, so the borrows do not overlap. The right program is rejected: after immutably unpacking via `freeze(copy(e))`, the write through `e` would change the active variant from `Two` to `One`, and the borrow `vy` would then point at field `Two.y`, which no longer exists in the active variant. The type system detects this via check_outbound(Π, ρ_e), which fails because paths(ρ_e, ρ_{vy}) is non-trivial.

Fig. 16 presents two representative typing rules. An *enum environment* Δ maps enum type identifiers to their variant definitions (constructors with associated field names and types). T-PackVariant looks up variant V's field types in Δ, verifies that the provided sites carry values of the correct types, consumes those sites, and produces a fresh site holding a value of the enum type. T-VariantSwitch dispatches on the active variant of the scrutinee: it consumes the scrutinee site and checks that each branch's label environment subsumes the current environment minus the scrutinee; the same subsumption mechanism used by T-Jump, applied once per arm.

At the semantic level, the central challenge is the runtime representation of enum values. Different variants may carry different fields, but the path transfer Lemma 3.3 requires that when T-WriteRef replaces a sub-value, all existing field paths through that heap location remain navigable. For records this holds trivially, since two values of the same record type have identical field structure, but for enums a naïve representation, where only the active variant's fields are present, would break path transfer. One might try variant-aware paths (annotating each path element with the expected variant) or a record-of-optionals encoding, but, in our experience, both approaches cascade into substantial changes throughout the invariant maintenance for the preservation proof.

$$\textsc{T-PackVariant}\ \frac{\Delta(id) = \{\ldots, V \mapsto \overline{f : \tau}, \ldots\} \qquad \forall (f_i, a_i) \in \overline{f \mapsto a}.\ \Sigma(a_i) = \mathsf{basic}(\tau_i) \\ a_v \notin \mathsf{dom}\,(\Sigma) \qquad \Lambda;\ \mathcal{E}[\Sigma := \Sigma \setminus \overline{a},\ \Sigma(a_v) \mapsto \mathsf{basic}(\mathsf{enum}(id))] \vdash s : \overline{T}}{\Lambda;\ \mathcal{E} \vdash \mathsf{let}\ a_v = \mathsf{pack}(V, \overline{f \mapsto a});\ s : \overline{T}}$$

$$\textsc{T-VariantSwitch}\ \frac{\Sigma(a) = \mathsf{basic}(\mathsf{enum}(id)) \qquad \Delta(id) = \{V_1, \ldots, V_k\} \\ \forall j.\ \Lambda(L_j) = \mathcal{E}_{L_j} \quad \mathcal{E}_{L_j} \sqsupseteq \mathcal{E}[\Sigma := \Sigma \setminus a]}{\Lambda;\ \mathcal{E} \vdash \mathsf{variant\_switch}(a, \overline{V \mapsto L}) : \overline{T}}$$

Fig. 16. Selected enum typing rules.

Our solution is a *flat encoding*: a value of enum type with variants $V_1, \ldots, V_k$, where variant V_j carries fields $f_1, \ldots, f_{n_j}$, is represented as a tagged record carrying the active variant name and *qualified* field names $\{V_j.f_m \mid 1 \le j \le k,\ 1 \le m \le n_j\}$. Components of inactive variants are filled with default values of the appropriate types. The consequence is that two values of the same enum type *always* have identical field structure, regardless of the active variant, making path transfer hold unconditionally. The flat encoding is adequate for execution: packing a variant populates the active fields and fills inactive ones with defaults; unpacking inspects a tag and reads the active fields, and so does variant switching. This representation overhead is confined to the program state and operational semantics: the typing rules are oblivious to the encoding.

Thanks to the flat encoding, Lemma 3.3 holds for all types, including enums: the preservation case for T-WriteRef requires no case split on whether the written type contains an enum. The well-typed state invariant for preservation proof is expanded with six new clauses (from 29 to 35 total): consistency of the runtime and static enum environments, uniqueness of enum names, variant names, field names within each variant, uniqueness of qualified field names across variants, and well-typedness of default values used to populate inactive variant fields. All six are structural well-formedness conditions on Δ; because the enum environment is fixed throughout execution, they are established once in the initial state and transferred unchanged at every preservation step.

5 AI-Assisted Mechanisation

The Lean mechanisation of the entire formal development described in this paper—definitions, operational semantics, typing rules, soundness proofs, algorithmic checker, parser, translator from MoveIR to MoveLight, and test suite—was produced by a single researcher working with an AI coding assistant (Claude Code, Opus 4.5 and 4.6 models). The design of MoveLight and the typing rules were human-encoded, following many discussions with the designers of the production Move borrow checker; the rest of the artefacts, including the runtime semantics and the soundness theorems, were AI-generated and scrutinised by the human (more on this in Sec. 5.2). Importantly, *no* pen-and-paper proof preceded it: from paper-level typing rules only, the entire metatheory—semantics, invariant, and soundness proofs—was developed *Lean-first* and only later transcribed into this paper. To the best of our knowledge, this is one of the largest PL metatheory developments in Lean, comparable in scale to the Cedar language formalisation (~70K LOC) [13], and, perhaps, the largest AI-assisted PL metatheory development to date. The active development phase took approximately one month (*cf.* Sec. 5.1). We report this experiment as a contribution in its own right: such a development would traditionally take a person-year, and we find it important to highlight, which parts current AI tools can tackle comfortably and which still demand steering by an expert. A longer, more systematic experience report is available as a companion blog post [37].

5.1 Overview of the Lean Development

The artefact comprises approximately 39,000 non-blank, non-comment lines of Lean code (267 commits, zero axioms or `sorry` declarations). Table 1 gives the breakdown by component. The soundness proofs dominate the code base (~23K LOC, 59%). The preservation theorem has 153 lemmas covering 41 cases (one per typing rule), each re-establishing all 35 fields of the well-typed state invariant. The proof of the weakening lemma 3.2 accounts for ~7K LOC: joining two type environments at a control-flow merge requires a reference substitution that must be threaded through every component of the environment. The relational and algorithmic typing rules total ~6K LOC, roughly split between the declarative specification and the executable Boolean checker, plus ~2.5K LOC for soundness and completeness proofs between the two representations. Data structures include the regex implementation (matching, derivatives, emptiness, subset), associative maps with verified lookup, and the path environment operations (extension, node removal, transitive closure).

Table 1. Lines of code by component.

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**

Fig. 17 shows the daily commit intensity during the active development phase, which spanned 27 working days. The algorithmic type checking and its soundness *w.r.t.* the relational one were formalised in two days. Infrastructure (parser, macros, test framework) took four days. The soundness proofs took 13 days, with peaks at the weakening proofs (22 commits in a single day) and the call rule (17 commits). The extension of the type system for vectors (Sec. 4.1) was added in one day (~3K LOC); the multi-variant enum extension (Sec. 4.2) required five days (~5K LOC).

5.2 Gaining Trust in Formalisation through Testing

Even though mechanisation eliminates the need to trust proofs, when metatheory definitions and soundness statements are almost entirely AI-generated, extra care is needed to ensure the right thing is being proved. In our case, "the right thing" has three components: (a) the type system must match the behaviour of the actual Move borrow checker; (b) the semantics must match the semantics of the code produced by the Move compiler; and (c) the type soundness statement must not make assumptions that render it vacuously true (*e.g.*, by ruling out any possible input state).

Type system conformance. To address the point (a), we developed an algorithmic version of the type checker for MoveLight and proved its soundness with respect to the relational specification:

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

That is, whenever the algorithmic (*i.e.*, decidable) Boolean check accepts a program, the relational typing judgement holds. We then used the algorithmic checker on 156 MoveIR programs drawn from the Move compiler's own test suite, ensuring that accepted and rejected programs match the production implementation. To bridge the syntactic gap between the actual MoveIR verifier and our formalisation, we implemented a parser and translator (~1.3K LOC) that converts MoveIR text into MoveLight ASTs. Parsing correctness is validated by 37 alpha-equivalence tests comparing output of the translation against hand-written reference ASTs, which are written using custom Lean macros that mirror the human-readable MoveLight concrete syntax used in this paper.

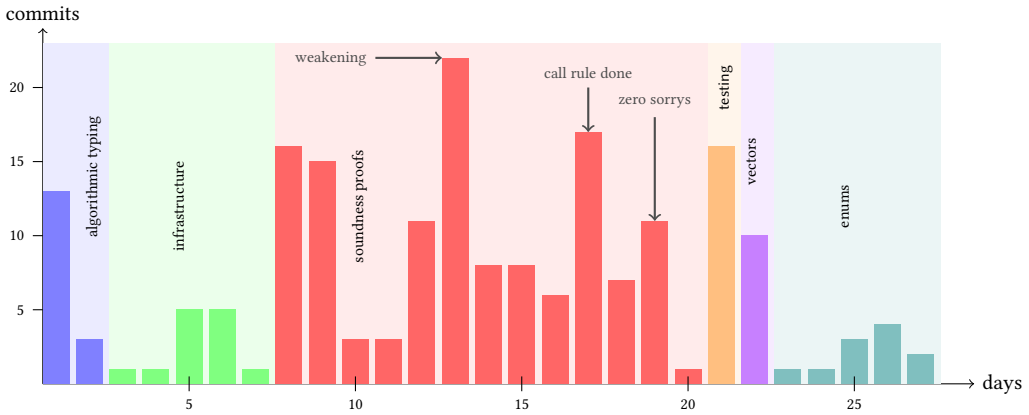

Fig. 17. Daily commit intensity during the active development phase (27 days). The soundness proof push (days 8–20) dominates, with peaks at the weakening proofs (day 13) and the call rule preservation (day 17).

Runtime conformance. To address the trust point (b), we performed lightweight conformance testing of the operational semantics, checking on 31 programs that their outcome (error or successful termination) is consistent with the outcome of the corresponding Move programs. We did not perform instruction-level conformance testing (*e.g.* comparing intermediate machine states) due to time constraints, so we leave deeper runtime conformance testing to future work.

Non-vacuity of assumptions. The type soundness theorem takes a function definition, a label environment, an enum environment, a function environment, a heap, and a list of argument values:

```
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
```

The premises require that the function type-checks (`htyped`), every function it calls is type-safe (`hfunEnv`), and the input satisfies well-formedness conditions (`ha`). The conclusion states that for any *non-acceptable* error `e` and fuel bound `n`, execution never produces that error. `SoundnessAssumptions` is a Lean record that bundles 23 preconditions: well-formedness of label environments, type compatibility of arguments with parameters, distinctness of parameter references, heap well-formedness, completeness of label-to-block mappings, and structural invariants on the enum environment. To guard against vacuously false assumptions, we developed a decidable checker that collapses all 23 preconditions plus the type-checking judgement into a single Boolean test:

```
theorem type_soundness_dec (f : FunDef) (lenvDec : LabelEnvDec) ...
  (hdec : SoundnessAssumptions.checkDecidable f lenvDec enumEnv funEnv fte heap args = true)
  (e : RuntimeError) (hna : ¬e.isAcceptable) : ∀ n, run n (initState f funEnv args) ≠ .error e
```

For each of the 31 runtime test programs, we instantiate `type_soundness_dec` and discharge is precondition via Lean evaluator, producing a machine-checked certificate that the specific run cannot produce any preventable error. This is *per-execution safety*: each certificate is independently verifiable by Lean's kernel, ensuring that the assumptions are satisfiable for concrete inputs.

5.3 What Has Not Been Mechanised

Our current formalisation omits several features of MoveIR: generics (the borrow checker operates on monomorphised code, so this is a matter of adding a monomorphisation pass), abilities such as

`copy` and `drop` (orthogonal to borrow tracking), global storage operations, and references within records (Move specification currently forbids these, but there are plans to add them in the future).

A more substantial omission is the abstract interpreter that infers label environments. In the production Move compiler, a forward dataflow analysis computes the type environment at each basic block entry by iterating to a fixpoint; this is how the compiler determines what borrows are live at each program point. Our formalisation assumes these environments are given and checks them, following the translation-validation architecture: a bug in the inference engine cannot compromise soundness, it can only cause a valid program to be rejected, never an unsafe program to be accepted. For our test suite, label environments were produced with the help of AI and manually validated. Automating this step (*e.g.*, by translating environments inferred by the Move compiler to Lean counterparts) would not be difficult conceptually, but we have not carried out this exercise.

5.4 Lessons from the AI-Assisted Formalisation

We summarise the most noteworthy observations from our experiment, as they may inform future efforts at AI-assisted PL metatheory mechanisation. The AI effectively handled proofs of routine cases in the preservation proof, in which the environment update is a straightforward passthrough. It also generated boilerplate definitions, infrastructure lemmas (*e.g.*, about lists) with high reliability. Where the AI performed best was proof repair: propagating changes across a large proof landscape after a definition or invariant was modified [20, 36]. The vector extension (Sec. 4.1) took only one day despite changing 30+ files because the modifications were uniform: adding a new path element `vecElem` and a new case to the value/type compatibility relation, then threading them through existing lemmas, so the AI could apply the same pattern repeatedly without errors.

The AI frequently selected wrong proof strategies for novel statements. Two examples stand out. First, when proving preservation for T-WriteRef, the key obligation is to show that writing through a mutable reference does not invalidate any other live reference's type. The AI attempted to show that all co-existing references are connected by paths in the path environment, which is false for references created independently. The correct proof instead uses the path transfer lemma 3.3: since the old and new values at the written location share the same type, all existing field paths remain navigable, regardless of how the references were created. Second, for the enum encoding (Sec. 4.2), the AI proposed encoding enum values as nested option types, which does not solve the path transfer problem; the human designed the flat encoding that makes the path transfer lemma hold unconditionally. In both cases, the failure mode was the same: attempting to prove a false statement rather than recognising the need for a different problem decomposition.

Three issues in the typing rules were discovered through failed proof obligations, not through testing or code review. The first two were in the formalisation of the typing rules: (1) arguments to the path extension operation were reversed in the assignment rule, producing nonsensical paths; (2) the call rule applied the conservative path extension with swapped arguments and had an overly restrictive output check. The third was in the type system design: the return writability check examined all tracked references instead of only live sites, causing false rejections. This issue did not affect any deployed programs but was confirmed and fixed by the Move team.

The most effective practice for improving AI productivity was aggressive lemma extraction: the human prover was constantly forcing AI to decompose proof obligations into named lemmas that the AI could then tackle independently. The weakening proof, for instance, was broken into 20 separate lemmas, each handling one typing rule case; the preservation proof followed the same pattern. This had double benefits: it kept individual proof obligations within the AI's context window, and it produced a modular proof structure that was easier for the human to audit. Even with the structure fixed, the AI needed repeated correction on *directional* obligations: applying weakening (Theorem 3.2) hinges on the direction of subsumption ($\mathcal{E}_1 \sqsupseteq \mathcal{E}_2$), and the AI consistently

inverted the two environments, so the human had to fix these steps by hand. Another useful strategy for gaining understanding of a proof was to ask the AI to explain its approach bottom-up: starting from the sub-goals and working towards the top-level statement. This made it easy to identify dead-end attempts early, before significant effort was invested in filling in details.

Workflow and the role of human guidance. Interaction was in natural language at the level of *goals and proof plans*, not individual tactics. Representative prompts were "*decompose the preservation statement into one named lemma per statement kind*" and "*this obligation looks false for independently created references—give a counterexample or name the missing invariant clause*". The human prover rarely wrote any Lean proof script directly; the effort went into reading proof skeletons, accepting or rejecting decompositions, and supplying the few design decisions the AI could not find—predominantly concerning runtime representations (*e.g.*, the flat enum encoding) and invariant clauses. Prompt engineering mattered far less than keeping a firm grip on the proof architecture and decomposing obligations into independent pieces.

5.5 Future Outlook

Quick iteration was the key to the speed of our mechanised metatheory prototyping: in our experience, it is much more important to maintain tight control over the proof structure than to be able to one-shot a theorem. Different design choices, such as the flat encoding for enums versus the alternatives discussed in Sec. 4.2, can drastically affect proof complexity, and the AI does not always pick the optimal one. The human prover's primary role was steering: confirming the selection of invariants, choosing representations, and cutting off unproductive proof attempts early.

The main bulk of the effort was iterating on the preservation proof, discovering a sufficiently strong well-typed state invariant. Notably, the *majority* of its 35 clauses were AI-inferred: on a failing case the AI proposed candidate strengthenings, and the human decided whether to keep, rephrase, or reject each. This process—proposing candidate invariants, attempting the proof, discovering missing clauses, and strengthening the invariant—is reminiscent of invariant inference in verified distributed protocols [22, 29, 46]. In both settings, the challenge is not writing the proof itself but finding the right inductive invariant; the AI accelerates exploration of the invariant space while the human prover steers away from dead ends.

We conjecture that mechanising PL metatheory with current AI tools is no longer a Herculean person-year effort: for an expert who keeps the proof structure in check, the cost drops by an order of magnitude or more. Designing interesting type systems and correct semantics is still far from automated as of early 2026; but for standard formulations, we believe auto-formalisation of metatheory for realistic languages is within a year's reach.

6 Borrow Checker Implementation in Sui Blockchain

The regex-based borrow checker has been fully implemented in a branch of the Sui blockchain client [5], but has not yet been deployed to production at the time of this writing. We validated the implementation by running it on all packages published on the Sui blockchain: 292,141 packages comprising 424,098 modules and 2,922,972 functions. Every package accepted by the deployed checker is also accepted by ours. Since rejected bytecode is never published on-chain, we do not have direct access to programs that the deployed checker rejects. We therefore cannot quantify how many real-world programs would move from rejected to accepted. However, the new checker is *strictly more expressive*, and more importantly, it provides a more consistent programming model: the deployed checker requires borrows of the same memory location to be issued through the "correct" reference path depending on which reference currently owns the location. The regex-based checker eliminates this path-sensitivity, making the borrow discipline more intuitive.

The regex-based checker is on average 2.2× slower than the deployed checker. However, the mean verification time per function (over 2,922,972 functions) is approximately 30 µs on an Apple M1 Max, so the slowdown is not significant in practice. Additionally, the existing metering system that bounds verification time for overly large or complex programs (to prevent denial-of-service attacks) has been adapted for the new checker, mitigating any concern about increased verification cost. We also note that the deployed checker has been optimised over nearly seven years of production use, whereas the regex-based implementation has seen little optimisation effort to date.

Performance on rejected programs. Because rejected bytecode is never published on-chain, our corpus consists only of programs the deployed checker *accepts*, so we cannot measure verification time on *ill-typed* programs. We do not, however, expect rejection to be costlier: the valid/invalid asymmetry of *search-based* verifiers (where acceptance needs one derivation while rejection may try many) does not arise here, as our checker is syntax-directed and deterministic—a single forward pass of polynomial-time regex checks (Sec. 3.5) that *short-circuits* at the first failing check.

7 Related Work

Ownership type systems. Our proposal is a spiritual successor of the work on ownership types: similarly, it addresses the problem of statically controlling aliasing and mutation in the presence of references. Classical ownership types [10–12] enforce encapsulation in object-oriented languages via ownership hierarchies. Our approach differs fundamentally: rather than enforcing the owners-as-dominator invariant, we track location *reachability*, which provides more fine-grained control over field-level aliasing without requiring an ownership hierarchy. Universe types [16] attach ownership annotations to references to stratify the heap into layers with controlled cross-layer access; by contrast, our system requires no annotations beyond the borrow kind (mutable or immutable), since the path environment tracks aliasing relationships automatically. Place Capability Graphs (PCGs) [21] associate each memory place in a Rust program with a set of capabilities encoding what operations are permitted on it. Like our path environment, PCGs capture fine-grained aliasing relationships at the level of individual places, but the two approaches differ in purpose: PCGs are designed as a reusable tool-building framework for analysing real-world Rust code, while our regex-based approach targets a different language (Move) and provides a machine-checked type soundness guarantee. The work by Montenegro et al. [31] is the most relevant to ours; it uses regular languages for detection of sharing in functional languages. In their approach, regexes describe sharing patterns in heap structures; we adapt this idea to borrow tracking in an imperative setting with explicit borrow/move/release operations. A key difference is that their analysis is whole-program, while our type system is modular (checking one function at a time with type signatures for callees). We also employ derivatives as a core typing mechanism, reducing the central safety check to regex emptiness rather than performing explicit graph traversals.

Access paths and regular languages in heap analysis. Representing aliasing with regular structures over access paths has a long history. Deutsch [15] abstracts alias relationships as *right-regular equivalence relations* over access paths, and the survey of Kanvar and Khedker [28] catalogues heap abstractions, including summarising access paths as regular expressions over field names rooted at a variable. These works use regular languages as a *summarisation* device for may-alias and points-to analyses; we instead apply regexes to *borrow checking*, a type-system safety property, yielding a modular, machine-checked type-soundness theorem rather than an alias-analysis result.

Static semantics for borrow checking. RustBelt [27] proposes a core calculus for Rust, including unsafe blocks and interior mutability, embedded into Iris [25] framework in the Rocq proof assistant. The RustBelt approach for modelling ownership and borrowing relies on step-indexed

logical relations, matching the greater complexity of Rust's memory model. The soundness of our system is established syntactically (via progress and preservation) and specialised to Move's simpler setting, where references cannot reside inside data structures; this simplicity allows us to use standard invariant-based techniques rather than step-indexed reasoning. Oxide [41] is a calculus that formalises a subset of Rust's ownership discipline using *places* with lifetime constraints; its soundness proof is paper-and-pencil. We believe, our regex-based paths subsume places and additionally support unbounded borrow chains; our proofs are fully mechanised. Stacked Borrows [26] and Tree Borrows [40] provide operational aliasing models for Rust; both verify *compiler assumptions* about pointer aliasing, which is complementary to our goal of verifying the soundness of the *type checker* itself. Unlike these operational models, our type system provides a static guarantee that well-typed programs cannot produce dangling references or aliasing violations. LLBC, the intermediate language of the Aeneas verification toolchain [24], is shown to be capable of acting as a sound borrow checker for Rust [23]. LLBC is mechanised in F*. However, the borrow checking in LLBC uses symbolic execution rather than a type system and does not employ regular expressions. Our type-system-based approach is more natural for Move's setting and directly yields a decidable algorithmic checker. Finally, Featherweight Rust [35] is a lightweight formalism for reference lifetimes and borrowing in Rust with a pen-and-paper proof. The type system of Featherweight Rust covers a smaller fragment than ours: no field-level borrowing, no vectors or enums.

Type systems for reachability tracking. Reachability types [2] provide a calculus that tracks reachability via qualifier sets for higher-order functions and closures. These qualifiers denote *set membership* (*e.g.*, "this closure may reach variable x"), while our regexes denote *structured field paths* ("this reference reaches location ℓ via fields $f_1 \cdot f_2$"). Our regex approach is more precise for record and field access patterns (it can distinguish borrows of different fields, which qualifier sets cannot) but is currently limited to first-order programs. Milano et al. [30] propose tempered domination, a type discipline that partitions the heap into regions and guarantees data-race freedom by ensuring that mutable access to a region is exclusive. Their setting targets concurrent regions, while ours addresses sequential borrow checking; both share the goal of preventing aliasing violations without lifetime annotations. Their type system is mostly static but relies on a runtime primitive (`if disconnected`) that dynamically checks whether two object subgraphs are disjoint. Our approach is entirely static. Gao and Parreaux [19] use Boolean-algebraic types for invalidation safety, where subtyping lattice operations detect when a reference may be invalidated by a mutation. Like our system, they address the problem of safely using references in the presence of mutation, but via a different mechanism. Their system does not support field-level borrowing.

Prior formalisations of Move. The work by Blackshear et al. [8] describes the deployed Move borrow checker. Their formulation uses abstract interpretation over borrow graphs with wildcard-labelled edges; the soundness argument is paper-and-pencil. We provide both a formal type system and a machine-checked proof. Our regex-based approach achieves the same precision through a simpler, uniform mechanism: a single extend operation with Brzozowski derivatives replaces the five specialised graph operations used by the deployed checker, yielding a type system that is easier to formalise and prove sound. Blackshear et al. [6] formalise Move bytecode and prove *resource safety* (a linear-type discipline for conservation of on-chain assets), which is orthogonal to borrow checking and does not model the reference discipline we verify. Patrignani and Blackshear [34] study robust safety for Move, *i.e.*, ensuring that well-typed modules remain safe when composed with arbitrary (potentially adversarial) code. Their contribution is orthogonal to ours: they address open-world compositionality, while we verify the borrow checker's core type system.

Move Prover [47] is a tool for automated formal verification of Move smart contracts based on the Boogie intermediate verification language [3]. Dill et al. [17] describe an improved version of

Move Prover whose key idea is an *alias-free memory model*: references are eliminated by a source-to-source transformation that converts mutable borrows into read-update-write cycles over owned values. The transformation relies on the guarantees of the borrow checker but does not prove those guarantees; our work fills exactly this gap by providing a machine-checked soundness proof for the borrow-checking type system itself. Neither of the two Move Prover papers addresses the soundness of Move's borrow checker.

AI-assisted metatheory. Paraskevopoulou [33] reports on using Claude Code to mechanise the ANF transformation correctness for the CertiCoq compiler in Rocq (~7,800 lines, ~96 hours). A key difference is that their proof follows an existing template (the CPS transformation proof); our formalisation tackles the design of a novel type system, discovering its soundness argument from scratch. Xu and Odersky [45] present an agentic proof automation case study, mechanising type soundness of System Capless [44] in Lean (~14K LOC) using LLMs. Our work is complementary: they focus on quantitative analysis of agent success/failure patterns; we describe our experience of using AI to automate proofs in a larger (~39K LOC) and more complex formalisation.

8 Conclusion

We have presented a novel regex-based type system for borrow checking in Move, a language with a large and growing user base on production blockchains. The key idea—encoding reference reachability as regular expressions over field paths—provides a principled and extensible type system in which checking field-level borrow disjointness reduces to deciding regex emptiness. Our regex-based type system is fully mechanised in Lean proof assistant. In our future work, we are planning to extend our regex-based approach to recursive data types and to references stored inside data structures, by using the Kleene star to capture reasoning about unbounded field paths.

Finally, we believe that our experience with AI-assisted formalisation of the presented type system, described in Sec. 5, suggests that LLM-based proof tools have a potential to fundamentally accelerate the pace of foundational research in programming language theory [37].

Acknowledgments

We are grateful to Michael D. George and Cameron Swords for their inputs at the early stages of this work. In particular, Michael has suggested to formulate the effects of reference extensions in terms of Brzozowski derivatives. We also thank the reviewers of OOPSLA'26 for their feedback.

Data Availability

The software artefact for this paper is publicly available [32], with its sources located at:

https://github.com/ilyasergey/lean-move

It contains the source code and build scripts for the Lean-embedded formalisation of MoveLight and the type system, the executable type checker, conformance tests, and the machine-checked soundness proof. The artefact also includes the AI configuration file (`CLAUDE.md`) and a set of representative prompts from the development. Finally, it provides the benchmarking harness used to corroborate the empirical claims of Sec. 6, which runs the regex-based checker over all packages published on the Sui blockchain, reproducing the reported backwards compatibility and per-function verification times.

References

[1] Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *POPL*. ACM, 666–679. https://doi.org/10.1145/3009837.3009866

[2] Yuyan Bao, Guannan Wei, Oliver Bracevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–32. https://doi.org/10.1145/3485516

[3] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO (LNCS, Vol. 4111)*. Springer, 364–387. https://doi.org/10.1007/11804192_17

[4] Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. 2019. Move: A Language With Programmable Resources. https://developers.diem.com/papers/diem-move-a-language-with-programmable-resources/2019-06-18.pdf

[5] Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, Brandon Williams, and Lu Zhang. 2024. Sui Lutris: A Blockchain Combining Broadcast and Consensus. In *CCS*. ACM, 2606–2620. https://doi.org/10.1145/3658644.3670286

[6] Sam Blackshear, David L. Dill, Shaz Qadeer, Clark W. Barrett, John C. Mitchell, Oded Padon, and Yoni Zohar. 2020. Resources: A Safe Language Abstraction for Money. *CoRR* abs/2004.05106 (2020). https://doi.org/10.48550/ARXIV.2004.05106

[7] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: compositional static race detection. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 144:1–144:28. https://doi.org/10.1145/3276514

[8] Sam Blackshear, John C. Mitchell, Todd Nowacki, and Shaz Qadeer. 2022. The Move Borrow Checker. *CoRR* abs/2205.05181 (2022). https://doi.org/10.48550/ARXIV.2205.05181

[9] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494. https://doi.org/10.1145/321239.321249

[10] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. LNCS, Vol. 7850. Springer, 15–58. https://doi.org/10.1007/978-3-642-36946-9_3

[11] David G. Clarke and Sophia Drossopoulou. 2002. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*. ACM, 292–310. https://doi.org/10.1145/582419.582447

[12] David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*. ACM, 48–64. https://doi.org/10.1145/286936.286947

[13] Joseph W. Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, Eleftherios Ioannidis, John H. Kastner, Anwar Mamat, Darin McAdams, Matt McCutchen, Neha Rungta, Emina Torlak, and Andrew Wells. 2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 670–697. https://doi.org/10.1145/3649835

[14] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *CADE (LNCS, Vol. 12699)*. Springer, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

[15] Alain Deutsch. 1992. A Storeless Model of Aliasing and Its Abstractions Using Finite Representations of Right-Regular Equivalence Relations. In *ICCL*. IEEE Computer Society, 2–13. https://doi.org/10.1109/ICCL.1992.185463

[16] Werner Dietl and Peter Müller. 2005. Universes: Lightweight Ownership for JML. *J. Object Technol.* 4, 8 (2005), 5–32. https://doi.org/10.5381/JOT.2005.4.8.A1

[17] David L. Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Jingyi Emma Zhong. 2022. Fast and Reliable Formal Verification of Smart Contracts with the Move Prover. In *TACAS (LNCS, Vol. 13243)*. Springer, 183–200. https://doi.org/10.1007/978-3-030-99524-9_10

[18] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *PLDI*. ACM, 237–247. https://doi.org/10.1145/155090.155113

[19] Cunyuan Gao and Lionel Parreaux. 2025. A Lightweight Type-and-Effect System for Invalidation Safety: Tracking Permanent and Temporary Invalidation with Constraint-Based Subtype Inference. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 2623–2653. https://doi.org/10.1145/3763144

[20] Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. 2023. Mostly Automated Proof Repair for Verified Libraries. *Proc. ACM Program. Lang.* 7, PLDI (2023), 25–49. https://doi.org/10.1145/3591221

[21] Zachary Grannan, Aurel Bílý, Jonás Fiala, Jasper Geer, Markus de Medeiros, Peter Müller, and Alexander J. Summers. 2025. Place Capability Graphs: A General-Purpose Model of Rust's Ownership and Borrowing Guarantees. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 2002–2029. https://doi.org/10.1145/3763122

[22] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. 2021. Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. In *NSDI*. USENIX Association, 115–131. https://www.usenix.org/conference/nsdi21/presentation/hance

[23] Son Ho, Aymeric Fromherz, and Jonathan Protzenko. 2024. Sound Borrow-Checking for Rust via Symbolic Semantics. *Proc. ACM Program. Lang.* 8, ICFP (2024), 426–454. https://doi.org/10.1145/3674640

[24] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP (2022), 711–741. https://doi.org/10.1145/3547647
[25] The Iris Project. 2024. The Iris 4.3 Reference. https://iris-project.org/ Online; last accessed 10 March 2026.
[26] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked borrows: an aliasing model for Rust. *Proc. ACM Program. Lang.* 4, POPL (2020), 41:1–41:32. https://doi.org/10.1145/3371109
[27] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154
[28] Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. *ACM Comput. Surv.* 49, 2 (2016), 29:1–29:47. https://doi.org/10.1145/2931098
[29] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. 2019. I4: incremental inference of inductive invariants for verification of distributed protocols. In *SOSP*. ACM, 370–384. https://doi.org/10.1145/3341301.3359651
[30] Mae Milano, Joshua Turcotti, and Andrew C. Myers. 2022. A flexible type system for fearless concurrency. In *PLDI*. ACM, 458–473. https://doi.org/10.1145/3519939.3523443
[31] Manuel Montenegro, Ricardo Peña, and Clara Segura. 2015. Shape analysis in a functional language by using regular languages. *Sci. Comput. Program.* 111 (2015), 51–78. https://doi.org/10.1016/J.SCICO.2014.12.006
[32] Todd Nowacki, Sam Blackshear, John Mitchell, Shaz Qadeer, and Ilya Sergey. 2026. LeanMove: Artefact for "Tracking Borrows with Regular Expressions" (OOPSLA 2026). https://doi.org/10.5281/zenodo.21850968
[33] Zoe Paraskevopoulou. 2026. Machine-Generated, Machine-Checked Proofs for a Verified Compiler (Experience Report). *CoRR* abs/2602.20082 (2026). https://doi.org/10.48550/arXiv.2602.20082
[34] Marco Patrignani and Sam Blackshear. 2023. Robust Safety for Move. In *CSF*. IEEE, 308–323. https://doi.org/10.1109/CSF57540.2023.00045
[35] David J. Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Trans. Program. Lang. Syst.* 43, 1 (2021), 3:1–3:73. https://doi.org/10.1145/3443420
[36] Talia Ringer. 2021. *Proof Repair*. Ph. D. Dissertation. University of Washington, USA. https://hdl.handle.net/1773/47429
[37] Ilya Sergey. 2026. Verifying Move Borrow Checker in Lean: an Experiment in AI-Assisted PL Metatheory. Blog post. https://proofsandintuitions.net/2026/03/18/move-borrow-checker-lean/.
[38] Jeremy G. Siek. 2013. Type Safety in Three Easy Lemmas. Blog post. https://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html.
[39] The Aptos Team. 2022. The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure. https://aptosfoundation.org/whitepaper.
[40] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. 2025. Tree Borrows. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1019–1042. https://doi.org/10.1145/3735592
[41] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *CoRR* abs/1903.00982 (2019). https://doi.org/10.48550/arXiv.1903.00982
[42] Adam Welc and Sam Blackshear. 2023. Sui Move: Modern Blockchain Programming with Objects. In *SPLASH Companion*. ACM, 53–55. https://doi.org/10.1145/3618305.3623605
[43] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. https://doi.org/10.1006/INCO.1994.1093
[44] Yichen Xu, Oliver Bracevac, Cao Nguyen Pham, and Martin Odersky. 2025. What's in the Box: Ergonomic and Expressive Capture Tracking over Generic Data Structures. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 1726–1753. https://doi.org/10.1145/3763112
[45] Yichen Xu and Martin Odersky. 2026. Agentic Proof Automation: A Case Study. *CoRR* abs/2601.03768 (2026). https://doi.org/10.48550/ARXIV.2601.03768
[46] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2022. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *OSDI*. USENIX Association, 485–501. https://www.usenix.org/conference/osdi22/presentation/yao
[47] Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark W. Barrett, and David L. Dill. 2020. The Move Prover. In *CAV (LNCS, Vol. 12224)*. Springer, 137–150. https://doi.org/10.1007/978-3-030-53288-8_7

Received 2026-03-16; accepted 2026-08-06