Linearity Rust Verification — Jonathan Protzenko

Lecture 1 - *June 29, 2026*

1 Linearity and its limits

Linearity tracks the *usage* of a variable according to its *type*. An ordinary type system checks a `let`-binding by giving the *same* context Γ to both premises, so a hypothesis can be used any number of times; a linear system splits the context, $\Gamma = \Gamma_1 \uplus \Gamma_2$ (a disjoint partition), so each linear variable is used in exactly one subderivation:

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x{:}t_1 \vdash e_2 : t_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t_2} \qquad \frac{\Gamma = \Gamma_1 \uplus \Gamma_2 \quad \Gamma_1 \vdash e_1 : t_1 \quad \Gamma_2, x{:}t_1 \vdash e_2 : t_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t_2}$$

These rules are declarative. The algorithmic counterpart threads one context ($\texttt{tc} : \mathsf{env} \to \mathsf{expr} \to \mathsf{type} \times \mathsf{env}$) and deletes each variable as it is used, so a second use fails. Linearity is thus an analysis on top of the type system: it enforces usage invariants and provides guarantees around ownership, and hence aliasing. For instance, `let x = ref 0 in (x, x)` is rejected because the owned reference is used twice, and `let x = ref 0 in spawn (fun _ => incr x); incr x` is rejected because two threads would race on `x`. The same discipline covers both memory safety and thread safety.

The original substructural type systems had three main motivations, each now addressed in other ways:

- **Performance** (Wadler, 1990): in a pure language an array `update` must copy unless the array is used linearly, in which case it can be updated in place, and the exactly-once reading even makes deallocation explicit (scope-based reclamation, no garbage collector). This is now usually recovered at run time instead, by reference counting that mutates in place when the count is one (Lean 4; "Counting Immutable Beans"); the static discipline was too rigid, since it forces the programmer to track every alias by hand.

Compiled By:
Markus de Medeiros, Dongyu Wu, John Rizkalla,
Carolyn Zech, Weiyi Chen

- **Usage protocols and typestate**: an `open`/`update`/`close` API where linearity guarantees the resource is closed. This never took off in practice.
- **Strong updates**: changing a value's type by assignment when it has a unique owner. But zero-initializing costs almost nothing on today's compilers (C++ proposal P2723 measures under 0.5%).

There is also a structural limitation. Some shapes cannot be linear at all: the doubly-linked list is the standard example, since each node is reachable from both neighbours and the ownership graph is not a tree. Relaxations exist (borrows, Mezzo, nesting), but only while ownership stays tree-like and statically trackable, so a usable language always needs an escape hatch for the rest. With the 1990s performance argument also weakened by decades of work on compilers and garbage collectors, one could then argue for dropping linearity altogether: prove memory safety with a separation logic, or rely on run-time checks (reference counting, GC), and bring out heavier machinery only when functional correctness is at stake.

2 Rust: linearity in the service of memory safety

A more targeted use of linearity is for a single job, memory safety, rather than as the foundation of a whole language. A language built for this needs several things at once:

- low-level control over the stack and the heap, with manual allocation;
- linearity for both memory safety over time and thread safety;
- enough extra features to stay usable, plus an opt-out for the hard cases;
- efficient compiler backends to hide the cost.

Several real-world languages are relevant: OCaml (through OxCaml), and Rust. OxCaml has a `unique` mode (a uniqueness, or linearity, feature), stack allocation, and a type system that statically rules out data races. The course follows Rust, working backwards from existing systems: wanting a better C gave us Rust.

Rust ties linearity to types. A value with a unique owner (a `Vec`, a resource) is linear and is *moved*; a value with no such obligation (a `bool`) is unrestricted and is *copied*. Borrows then take the address of a value under one rule, *aliased XOR mutable*: a mutable borrow `&mut` gives unique, writable access and behaves linearly, while a shared borrow `&` forbids writes but may be duplicated.

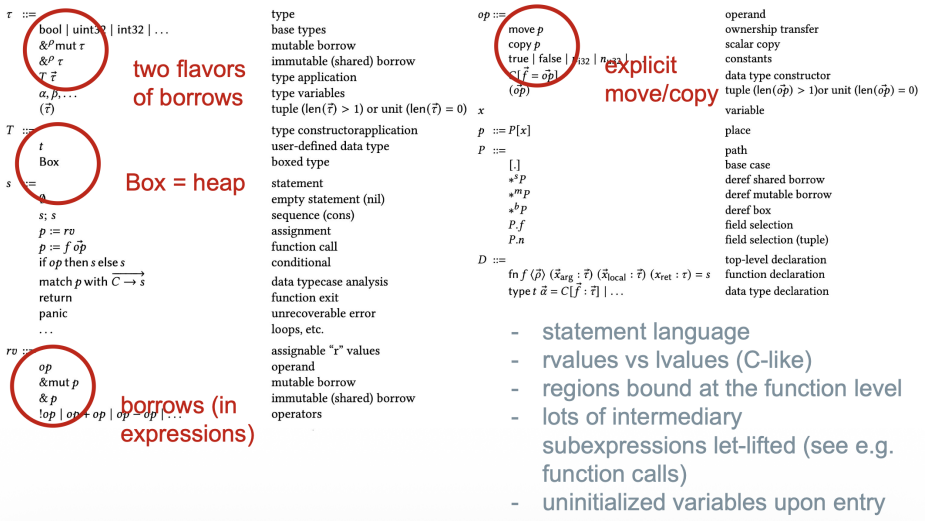
Borrows can be seen as an *implementation detail*. A shared borrow could be replaced by a copy of the immutable value with no change in meaning, only in cost. A mutable borrow can be treated

the same way if each function comes with a *backward function* recording how the borrowed data is written back when the borrow ends. If borrows are only detail, the research question is which core model of Rust, and which semantics, make best use of its ownership structure. Rust also has run-time escape hatches (`Cell`, `RefCell`, and the rest of interior mutability), not covered here for simplicity.

3 Modelling Rust without memory: LLBC

The model is **LLBC** ("Low-Level Borrow Calculus"), essentially a tidied MIR (the Rust compiler's mid-level IR), produced from `rustc` by the **Charon** tool. It keeps the content of MIR, makes moves and copies explicit, and serves as a base for analyses and translations (Aeneas papers, ICFP'22 and '24). It is a statement language with a C-style split of rvalues and places, two borrow flavours each tagged with a region, `Box` for heap values, regions bound per function, let-lifted subexpressions, and locals uninitialized on entry. On why move and copy are distinguished when they compile the same, an ongoing RFC (rust-lang/rfcs#3943) would make use-after-move undefined behaviour, which unlocks optimizations (the compiler can drop copies).

The semantics for LLBC uses *no addresses and no memory*. State is just *values*, *loan identifiers* ℓ, and an environment Ω mapping *variables* to values. A borrow and the loan it came from share an identifier ℓ, which is how aliasing is recorded; a mutable borrow holds the borrowed value, a shared borrow does not. The value grammar is:



A mutable loan $\mathsf{loan}^m \ell$ marks where a value was borrowed from, and the matching mutable borrow $\mathsf{borrow}^m \ell\, v$ carries the value away. A shared loan $\mathsf{loan}^s \{\vec{\ell}\}\, v$ keeps the value in place and records all of its outstanding shared borrows in the set $\vec{\ell}$. The symbol $\bot$ is an inaccessible value (for example,

one that has been moved out), and reserved borrows $\mathsf{borrow}^r\,\ell$ are used for two-phase borrows, covered below.

Two examples show how this works. Nesting mutable borrows, the value moves outward to the innermost borrow while loans are left behind to mark each source:

```
let mut x = 0;        // x ↦ 0
let mut px = &mut x; // x ↦ loan^m ℓ,   px ↦ borrow^m ℓ 0
let ppx = &mut px;   // x ↦ loan^m ℓ,   px ↦ loan^m ℓ',     ppx ↦ borrow^m ℓ' (borrow^m ℓ 0)
```

With shared borrows the value stays with its owner, and the loan-set grows as borrows are added:

```
let x = (0, 1);     // x ↦ (0, 1)
let px1 = &x;       // x ↦ loan^s {ℓ} (0, 1),          px1 ↦ borrow^s ℓ
let px2 = &x;       // x ↦ loan^s {ℓ ∪ ℓ'} (0, 1),    px1 ↦ borrow^s ℓ,   px2 ↦ borrow^s ℓ'
let y = copy x.0;   // x ↦ loan^s {ℓ ∪ ℓ'} (0, 1),    px1 ↦ borrow^s ℓ,   px2 ↦ borrow^s ℓ',   y ↦ 0
let z = &(*px1.0); // x ↦ loan^s {ℓ ∪ ℓ'} ((loan^s {ℓ''} 0), 1), px1 ↦ ..., px2 ↦ ..., y ↦ 0, z ↦ borrow^s ℓ''
```

Tracking the loan-sets precisely shows exactly when full ownership of `x` is regained, namely when its loan-set is empty; only then can `x` be mutated or moved again.

This is a mix of low- and high-level. There is still a notion of pointer, the borrow, but no concrete addresses, bytes, layout, or offsets; everything is values plus the borrow bookkeeping. Because of this, the system needs *reorganization* rules that *fold* the environment (collapsing a loan/borrow pair back into a plain value to regain ownership) and *unfold* it (splitting a value to justify a new borrow). This is a *semantic* view of borrowing: there is no syntactic notion of lifetime, the rules follow control-flow much like a points-to analysis, and the system is *lazy* compared with the Rust type-checker, since it folds only when needed. The model is intentionally restrictive: it cannot describe `unsafe` code or cyclic structures. It is still strong enough to subsume the Rust borrow-checker, and because it is flow-sensitive, it accepts some examples the Rust borrow-checker rejects.

4 Selected reduction rules

Write $\Omega \vdash e \rightsquigarrow v \dashv \Omega'$ for "in context Ω, the expression e reduces to value v and gives the updated context Ω'." Two auxiliary judgments handle paths: $\Omega(p) \Rightarrow v$ *reads* the value at place p, and $\Omega(p) \leftarrow v \Rightarrow \Omega'$ *writes* a value there.

A few of these read as follows:

- E-Mut-Borrow reads the value at p, checks it is not already loaned, moved out, or reserved, picks a fresh ℓ, and writes a mutable loan in its place while the value moves into the new borrow. The side-condition $*^s \notin p$ blocks borrowing through a shared borrow.

- E-Move reads the value and writes $\bot$ in its place. It refuses to move through a dereference, or to move something already loaned or moved.
- E-Assign evaluates the rvalue, checks the old contents of p have "no outer loans" (overwriting a value that still has loans would create dangling pointers), writes the new value, and saves the old contents under a fresh variable x_{old}, so that any borrows inside them are remembered and can be folded back later.

E-Mut-Borrow

$$\frac{\Omega(p) \Rightarrow v \qquad \{\bot, \text{loan}, \text{borrow}^r\} \notin v \qquad *^s \notin p \qquad \ell \text{ fresh} \qquad \Omega(p) \leftarrow \text{loan}^m\, \ell \Rightarrow \Omega'}{\Omega \vdash \&\text{mut}\, p \rightsquigarrow \text{borrow}^m\, \ell\, v \dashv \Omega'}$$

E-Shared-Or-Reserved-Borrow

$$\frac{\Omega(p) \Rightarrow v \qquad \{\bot, \text{loan}^m, \text{borrow}^r\} \notin v \qquad \ell \text{ fresh} \qquad \Omega' = \Omega[p \mapsto v'] \qquad v' = \begin{cases} \text{loan}^s\, \{\vec{\ell} \cup \ell\}\, v'' & \text{if } v = \text{loan}^s\, \{\vec{\ell}\}\, v'' \\ \text{loan}^s\, \{\ell\}\, v & \text{otherwise} \end{cases}}{\Omega \vdash \& p \rightsquigarrow \text{borrow}^{r,s}\, \ell \dashv \Omega'}$$

E-Move

$$\frac{\Omega(p) \Rightarrow v \qquad \{\bot, \text{loan}, \text{borrow}^r\} \notin v \qquad \{*^m, *^s\} \notin p \qquad \Omega(p) \leftarrow \bot \Rightarrow \Omega'}{\Omega \vdash \text{move}\, p \rightsquigarrow v \dashv \Omega'}$$

E-Copy

$$\frac{\Omega(p) \Rightarrow v \qquad \{\bot, \text{loan}^m, \text{borrow}^{m,r}\} \notin v \qquad \Omega \vdash \text{copy}\, v \Rightarrow v' \dashv \Omega'}{\Omega \vdash \text{copy}\, p \rightsquigarrow v' \dashv \Omega'}$$

E-Constructor

$$\frac{\Omega_i \vdash op_i \rightsquigarrow v_i \dashv \Omega_{i+1}}{\Omega_0 \vdash C[\overrightarrow{f = op}] \rightsquigarrow C[\overrightarrow{f = v}] \dashv \Omega_n}$$

E-Return

$$\frac{\Omega_i \vdash x_{\text{local},i} := \bot \rightsquigarrow () \dashv \Omega_{i+1} \qquad \Omega_n(x_{\text{ret}}) \Rightarrow v \qquad \{\bot, \text{loan}, \text{borrow}^r\} \notin v}{\Omega_0 \vdash \text{return} \rightsquigarrow \text{return}\, v \dashv \Omega_n}$$

E-IfThenElse-T

$$\frac{\Omega \vdash op \rightsquigarrow \text{true} \dashv \Omega' \qquad \Omega' \vdash s_1 \rightsquigarrow r \dashv \Omega''}{\Omega \vdash \text{if } op \text{ then } s_1 \text{ else } s_2 \rightsquigarrow r \dashv \Omega''}$$

E-Assign

$$\frac{\Omega \vdash rv \rightsquigarrow v \dashv \Omega' \qquad \Omega'(p) \Rightarrow v_p \qquad v_p \text{ has no outer loans} \qquad x_{\text{old}} \text{ fresh} \qquad \Omega'(p) \leftarrow v \Rightarrow \Omega'' \qquad \Omega''' = \Omega''[x_{\text{old}} \mapsto v_p]}{\Omega \vdash p := rv \rightsquigarrow () \dashv \Omega'''}$$

This gives a different view of borrow-checking. Rather than a system of equations (as in Polonius/NLL), the semantics keeps precise, flow-sensitive knowledge of *state* (what is loaned and what is not) and *aliasing* (through the loan identifiers), and it enforces well-formedness with the side-conditions of each rule. You cannot borrow or move a value that is loaned out or already moved; you cannot borrow through a shared borrow, which would let shared and mutable references exist at once; and you cannot copy a value that is mutably loaned or borrowed, which would duplicate unique ownership. Several auxiliary judgments support this: read and write along paths, a "no outer loans" judgment defined by omission, and reorganization rules that end borrows and fold ownership back (for example, ending a mutable borrow returns its value to the loan site). The reorganizations are available at any time in the declarative reading, but in practice are applied lazily, only when a rule's side-conditions require them.

One feature uses the reserved borrow from the value grammar. A two-phase borrow starts as a

shared borrow but is as restricted as a mutable one, and can be *promoted* to a real mutable borrow later. The reserved borrow $\mathsf{borrow}^r\,\ell$ in the grammar models the first phase, where the borrow cannot be copied, dereferenced, or written to; a reorganization rule later activates it, turning the reserved borrow and its shared loan into a mutable borrow and a mutable loan. Like reborrowing, two-phase borrows exist to work around Rust's lack of *borrow polymorphism*: when you have to call an API that asks for a shared borrow even though the borrow should be mutable, a two-phase borrow lets the same borrow start shared and become mutable.