**Our sponsors make OPLSS possible.**

Diamond Sponsor

**Jane Street**

https://www.janestreet.com/

---

Platinum Sponsors

National Science Foundation
WHERE DISCOVERIES BEGIN

https://www.nsf.gov/

*SIGPLAN*

https://www.sigplan.org/

---

Silver Sponsors

# Linearity, Rust and Verification

Jonathan Protzenko (Google)

with Son Ho, Guillaume Boisseau, Aymeric Fromherz, and the Aeneas collaborators

https://github.com/AeneasVerif/aeneas/

README Apache-2.0 license

Unknown author, *lapyx removing arrowhead from Aeneas* [Fresco]. Wall in Pompei, digital image from Michael Lahanis. Source

# Aeneas

Aeneas (pronunced [Eh-nee-as]) is a verification toolchain for Rust programs. It relies on a translation from Rust's MIR internal language to a pure lambda calculus. It is intended to be used in combination with Charon, which compiles Rust programs to an intermediate representation called LLBC. It currently has backends for F*, Coq, HOL4 and LEAN.

If you want to contribute or ask questions, we strongly encourage you to join the Zulip. In particular, we welcome external contributions but ask contributors to engage with us before making deep modifications of the code or implementing non trivial features, as these require in-depth design discussions and careful reviews. If you want to contribute a new feature you should either discuss it on the Zulip or open the corresponding issue on github.

# Lecture Overview

**Day 1**

- Linearity: what is it, why do we care, how does it work, how does it fail
- LLBC: a core model of Rust, with design choices, tradeoffs, and semantics

**Day 2**

- Aeneas: a borrow-checker, an operational semantics, and a functional translation
- The real world: Lean, the Aeneas tool, and a demo for proving the correctness of an unsafe function in jxl-rs

# Linearity


- Core idea: track the ***usage*** of variables according to their ***type***
- Example: a let-binding, normally, is type-checked thusly:

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : t_2}$$

- Linearity = ***usage*** of things in $\Gamma$ is restricted!

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

notice how the rules are ***declarative***

# Linearity is…

- an ***analysis*** bolted onto the type system

- which enforces ***invariants*** regarding the usage of certain types

- provides ***guarantees*** notably around ownership, and thus, aliasing

**Example 1:**

```
let x = ref 0 in
(x, x)
```

**Example 2:**

```
let x = ref 0 in
spawn (fun _ => incr x);
incr x
```

# Why substructural type systems

- Original motivation (1990): for performance
  - pure language, but `update` is in-place
- Linear vs affine: → deallocated on scope exit
  - no GC
- Linear regions
  - *en masse* memory management for deallocation

Not so much *à la mode* anymore:

- Lean4: if you use your object linearly, you get in-place updates…
- Type-checking time → Run-time

Linear types can change the world!

Philip Wadler
University of Glasgow*

The four operations on linear arrays are:

$$\begin{array}{lcl} alloc & : & ¡Arr, \\ lookup & : & Ix \to ¡Arr \to Val \otimes ¡Arr, \\ update & : & Ix \to Val \to ¡Arr \to ¡Arr, \\ dealloc & : & Val \otimes ¡Arr \to Val. \end{array}$$

**Counting Immutable Beans**

Reference Counting Optimized for Purely Functional Programming

Sebastian Ullrich
Karlsruhe Institute of Technology
Germany
sebastian.ullrich@kit.edu

Leonardo de Moura
Microsoft Research
USA
leonardo@microsoft.com

# Why substructural type systems

Enforcing usage protocols:

- imagine an `open` / `update` / `close` API for file system descriptors
- use linearity to guarantee the ***resource*** is always closed

Type state:

- generalization, with multiple states, transitions in the state diagram, and ownership tracking to ensure consistency via ruling out aliases

Never really took of on an industrial scale – I have yet to see someone worry about forgetting to close file descriptors…

# Why substructural type systems


Strong updates, i.e. change the type of an object via assignment

- If a piece of mutable data has a unique owner, it's ok to change its type
- Useful for memory reuse
- More useful for gradual initialization

But… putting sensible values as a default has no cost (see Rust, C++):

"The performance impact is negligible (less than 0.5% regression) to slightly positive (that is, some code gets *faster* by up to 1%). The code size impact is negligible (smaller than 0.5%). Compile-time regressions are negligible. Were overheads to matter for particular coding patterns, compilers would be able to obviate most of them"

**P2723R1**
**Zero-initialize objects of automatic storage duration**

Published Proposal, 2023-01-15

**This version:**
http://wg21.link/P2723

**Author:**
JF Bastien (Woven Planet)

# Substructural type systems… why?

Very strong limitations.

- Need to mix linear and non-linear: kind polymorphism?
- Do you ***really*** want to thread your linear state?
- What about things that initially start as mutable then become immutable?
- More importantly: how about things that ***truly*** cannot be linear?

# Substructural type systems… why?

Seminal example: doubly-linked list

- fundamentally breaks linearity
- comes with a ownership structure that is ***not*** linear

Generally:

- you can devise local relaxations (e.g. Mezzo, nesting, borrows, others),
- as long as the static ownership is tree-like (i.e. can be statically tracked)

Morale:

- you need an escape hatch! (more on that soon)

# Substructural type systems… why?

- Tradeoffs are not the same as in the 1990s, especially with decades of research on compiler optimization (LLVM), GC performance, etc.
- Lots of use-cases failed to materialize (file descriptor…)
- Severe limitations on non-linear usage patterns (graphs, doubly-linked lists, etc.)
- We might as well:
  - pull out a separation logic, and just prove memory safety without bothering about linearity…?
  - or even better, simply rely on run-time checks: refcounting, gc, and pull out separation logic only if we care about functional correctness

# Linearity: we ***do*** care!

**What if we…**

- take inspiration from a real-world language design
- care about low-level concepts (stack vs. heap)
- want to manually manage allocations
- use linearity for memory safety (temporal) ***and*** thread safety
- are willing to add fancy features to make it all usable
- are ok with an opt-out mechanism
- benefit from efficient compiler backends to offset potential overhead

**PART II: SECURING THE BUILDING BLOCKS OF CYBERSPACE**

To reduce the burden currently placed on end users to protect themselves from cybersecurity threats, efforts must be made to proactively eliminate entire categories of software vulnerabilities. To better understand the prevalence of these categories, software manufactures should consider publishing timely, complete, and consistent Common Vulnerability and Exposures (CVEs) data, including the Common Weakness Enumeration (CWE). Past analysis of CVE data identified memory safety bugs as one of the most pervasive classes of vulnerabilities that has plagued cyber defenders for decades.

# Linearity: we ***do*** care!

**What if we…**

- take inspiration from a real-world language design → **OCaml**
- care about low-level concepts (stack vs. heap)
- want to manually manage allocations
- use linearity for memory safety (temporal) ***and*** thread safety
- are willing to add fancy features to make it all usable
- are ok with an opt-out mechanism
- benefit from efficient compiler backends to offset potential overhead


The `unique` mode designates values that have only a single reference pointing to them. If an operation takes a `unique` argument, it will consume the only reference to the value. For example, an externally allocated data structure might support a `free` operation:

```
val free : t @ unique -> unit
```

Though type inference will infer where stack allocation is possible, it is often wise to annotate places where you wish to enforce locality and stack allocation. This can be done by labeling an allocation as a `stack_` allocation:

```
let x2 = stack_ { foo; bar } in
...
```

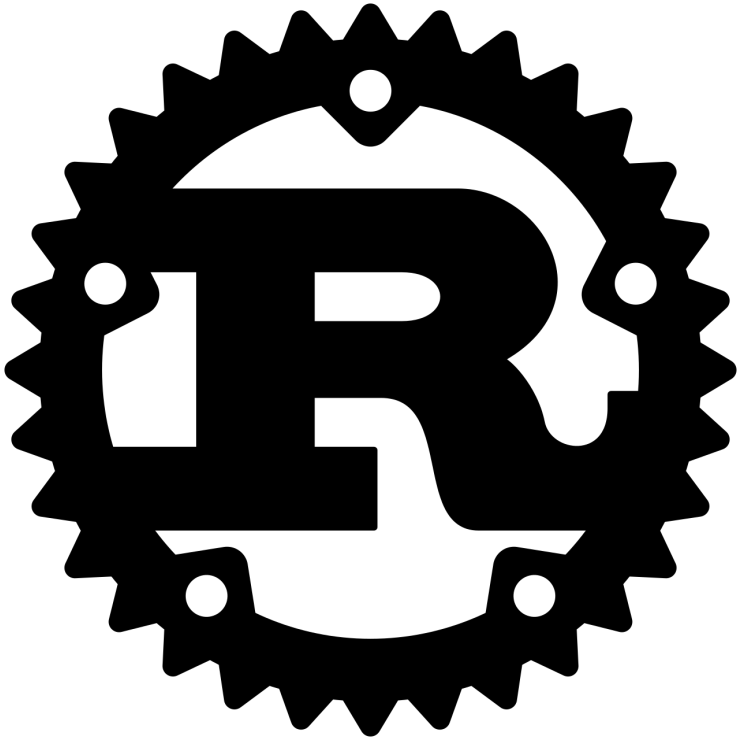
### Fearless concurrency

Writing correct concurrent programs is notoriously difficult. OxCaml includes additions to the type system to statically rule out data races.

# Linearity: we ***do*** care!

**What if we…**

- take inspiration from a real-world language design → **C**
- care about low-level concepts (stack vs. heap)
- want to manually manage allocations
- use linearity for memory safety (temporal) ***and*** thread safety
- are willing to add fancy features to make it all usable
- are ok with an opt-out mechanism
- benefit from efficient compiler backends to offset potential overhead



# Aside (personal opinion)

Lots of interesting research at the intersection of

- real-world language design – we want people to use our ***stuff***
- PL techniques to improve the state of the world

Working ***backwards*** from existing systems and seeing what we can do about them ***is*** feasible – wanting a better C gave us Rust.

Now, let's take Rust and see how we can improve upon memory safety (hint: verification). But first, Rust in a nutshell, and some semantics.

# Rust in a nutshell (simplified!)


Types of variable determine their linearity.

- Some things have a unique owner (e.g. Vec, a resource), and are handled linearly – these are ***move***d around.
- Some other things do not (e.g. bool), and be freely ***copy***'d around – these are unrestricted.

Borrows allow taking the address of an object and observe the "***aliased XOR mutable***" discipline

- mutable borrows permit mutation, and convey unique ownership (linear)
- shared borrows disallow mutation, but allow shared ownership (unrestricted)

# Rust in a nutshell: an observation

Shared borrows are an ***implementation*** detail: from the perspective of ***semantics***, you might as well pass a copy and not even think about the memory (and you can do so! it's just not efficient).

We will see later (functional translation) that mutable borrows can also be seen as an implementation detail, provided we have ***backward functions***.

Research question: what is an interesting core model of Rust, and what is a program semantics that leverage the strong ownership structure and borrowing-centric design of Rust?

# Rust in a nutshell: escape hatches

Rust has escape hatches in the form of run-time borrow checking

Cell<T>: barebones mutability via shared references (owns the value)

RefCell<T>: run-time borrow checking, gives & and &mut

… and more (see “interior mutability”). ***Not covering those today, for simplicity.***

# Signposting time


- LLBC, or how to model a low-level (mostly) imperative language cleanly
  - types, places (“lvalues”), operands, rvalues… and borrows
- operational semantics in the presence of linearity: moves & copies
- why these unusual “borrow-centric” semantics work and capture the essence of Rust
  - a mix of high- and low-level
- symbolic semantics – what if we don’t know the value of variables?
  - region abstractions, a key technical device
- functional translation – how to map the imperative format to a pure lambda calculus

# LLBC: a cleaned-up version of MIR

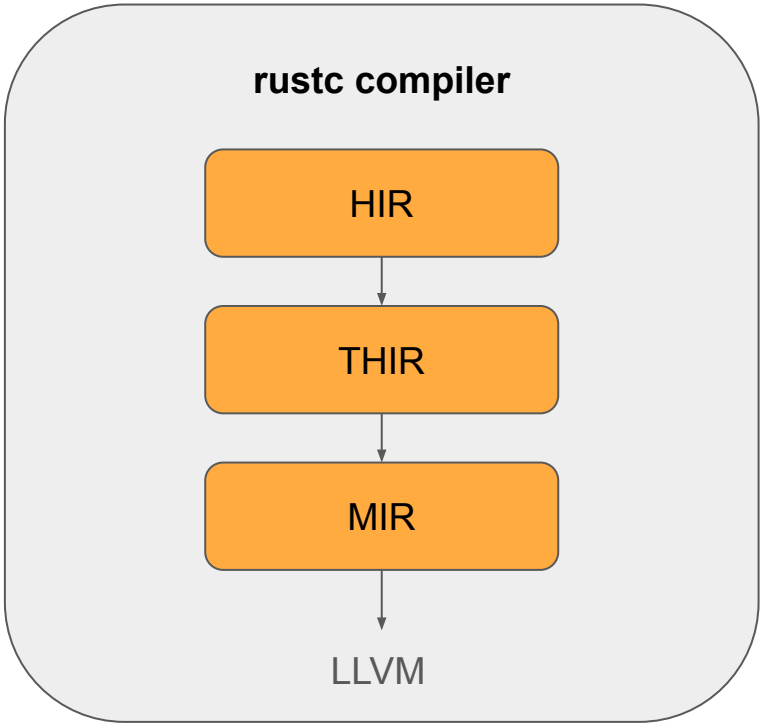
DEMO!

“Low-Level Borrow Calculus”

From an engineering perspective:

- hook into the Rust compiler
- handle all the painful infrastructure
- reconstruct additional information (traits)
- reconstruct control-flow (CFG/AST)
- offer a cleanup-view with Rust/OCaml libraries for consumption

Implemented in the ***Charon*** tool (demo!)



# LLBC: a cleaned-up version of MIR

“Low-Level Borrow Calculus”

From a theoretical perspective:

- captures the essence of MIR
- forms a semantic foundation upon which to build analyses / rules / translations
- moves & copies are explicit

Described in the Aeneas papers (ICFP’22 and ‘24), and used by others (e.g. RaRust, Soteria, Eurydice, etc.).

| | | |
|---|---|---|
| $\tau$ ::= | | type |
| | bool \| uint32 \| int32 \| ... | base types |
| | $\&^{\rho}$ mut $\tau$ | mutable borrow |
| | $\&^{\rho}$ $\tau$ | immutable (shared) borrow |
| | $T\ \vec{\tau}$ | type application |
| | $\alpha, \beta, \ldots$ | type variables |
| | $(\vec{\tau})$ | tuple (len($\vec{\tau}$) > 1) or unit (len($\vec{\tau}$) = 0) |
| $T$ ::= | | type constructorapplication |
| | $t$ | user-defined data type |
| | Box | boxed type |
| $s$ ::= | | statement |
| | $\varnothing$ | empty statement (nil) |
| | $s; s$ | sequence (cons) |
| | $p := rv$ | assignment |
| | $p := f\ \vec{op}$ | function call |
| | if $op$ then $s$ else $s$ | conditional |
| | match $p$ with $\overrightarrow{C \rightarrow s}$ | data typecase analysis |
| | return | function exit |
| | panic | unrecoverable error |
| | ... | loops, etc. |
| $rv$ ::= | | assignable "r" values |
| | $op$ | operand |
| | &mut $p$ | mutable borrow |
| | & $p$ | immutable (shared) borrow |
| | $!op \mid op + op \mid op - op \mid \ldots$ | operators |

two flavors of borrows

Box = heap

borrows (in expressions)

| | | |
|---|---|---|
| $op$ ::= | | operand |
| | move $p$ | ownership transfer |
| | copy $p$ | scalar copy |
| | true \| false \| $n_{i32}$ \| $n_{u32}$ \| | constants |
| | $C[\vec{f} = \vec{op}]$ | data type constructor |
| | $(\vec{op})$ | tuple (len($\vec{op}$) > 1)or unit (len($\vec{op}$) = 0) |
| $x$ | | variable |
| $p$ ::= $P[x]$ | | place |
| $P$ ::= | | path |
| | [.] | base case |
| | $*^{s} P$ | deref shared borrow |
| | $*^{m} P$ | deref mutable borrow |
| | $*^{b} P$ | deref box |
| | $P.f$ | field selection |
| | $P.n$ | field selection (tuple) |
| $D$ ::= | | top-level declaration |
| | fn $f\ \langle\vec{\rho}\rangle\ (\vec{x}_{\text{arg}} : \vec{\tau})\ (\vec{x}_{\text{local}} : \vec{\tau})\ (x_{\text{ret}} : \tau) = s$ | function declaration |
| | type $t\ \vec{\alpha} = C[\vec{f} : \vec{\tau}] \mid \ldots$ | data type declaration |

explicit move/copy

- statement language
- rvalues vs lvalues (C-like)
- regions bound at the function level
- lots of intermediary subexpressions let-lifted (see e.g. function calls)
- uninitialized variables upon entry

# Side-node: why moves when it’s all compiled the same?

- Ongoing RFC to add UB to use a variable after it’s moved in MIR
- Unlocks more optimizations
- See https://github.com/rust-lang/rfcs/pull/3943: “This RFC … makes accessing memory after a move undefined behavior, [and] introduces a new MIR optimization pass which exploits these guarantees to eliminate copies between locals when it is safe to do so.”

# Defining reduction without memory or addresses

- loan identifiers
- a map from ***variables*** to values
- mutable borrows hold the value (unique owner)
- shared borrows do not (inconsequential design choice)

| | | | |
|---|---|---|---|
| $v$ | ::= | | value |
| | | $\text{true} \mid \text{false} \mid n_{\text{i32}} \mid n_{\text{u32}} \mid \ldots$ | booleans and integers |
| | | $\text{borrow}^m\ \ell\ v$ | mutable borrow |
| | | $\text{borrow}^s\ \ell$ | shared borrow |
| | | $\text{borrow}^r\ \ell$ | reserved borrow |
| | | $\text{loan}^m\ \ell$ | mutable loan |
| | | $\text{loan}^s\ \{\vec{\ell}\}\ v$ | shared loan |
| | | $\bot$ | inaccessible value |
| | | $C\ [\vec{f} = \vec{v}]$ | data type constructor |
| | | $(\vec{v})$ | tuple constructor ($\text{len}(\vec{v}) > 1$) |
| $r$ | ::= | | result |
| | | panic | unrecoverable error state |
| | | return $v$ | successful, possibly-early exit |
| | | $\ldots$ | loop states, etc. |
| $\ell$ | | | loan identifier |
| $\Omega$ | ::= | | evaluation context |
| | | $\emptyset$ | empty |
| | | $x \mapsto v, \Omega$ | new mapping |

# Examples of operational semantics: mutable borrows

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

- tracks statically-known aliasing relationships
- one central location for the value (as opposed to multiple)
- no notion of addresses or memory
- intentionally restrictive: cannot model ***unsafe*** code (no cycles)

# Examples of operational semantics: shared borrows

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

- (still) tracks statically-known aliasing relationships
- borrow does not own the value (remains with its original owner)
- we care about precise tracking to know whether full ownership of `x` has been regained (the set is empty)

# Commentary on this style of semantics

- A mix of low-level and high-level
  - there is still a notion of pointer (borrow)
  - but no concrete addresses, bytes, layout, offsets, or anything – it's all values + borrow machinery
- This calls for reorganization rules – the ability to "fold" (to regain ownership of x) and unfold (to justify borrowing)
- This is a ***semantic*** view of borrowing – there is no notion of lifetime, and these rules follow control-flow; almost like a static analysis for points-to (more on that later!)
- We are ***lazy*** compared to the Rust type-checker, because we fold on demand

# Commentary on this style of semantics (2)

- Powerful enough to subsume the Rust borrow-checker
- Because we are flow-sensitive, we accept some examples rejected by the Rust borrow-checker
- We can even explain tricky examples:

```
1  let mut x = 0;          // x ↦ 0
2  let mut px = &mut x;    // x ↦ loan^m ℓ,   px ↦ borrow^m ℓ 0
3  px = &mut (*px);        // x ↦ loan^m ℓ,   px_old ↦ borrow^m ℓ (loan^m ℓ'),   px ↦ borrow^m ℓ '0
4                          // x ↦ loan^m ℓ,   px_old ↦ borrow^m ℓ 0,             px ↦ ⊥
5  assert!(x==0);          // x ↦ 0,          px_old ↦ ⊥,                        px ↦ ⊥
```

# Selected reduction rules (1)

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

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

# Selected reduction rules (2)

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

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

# A different view of borrow-checking

- Rather than a system of equations (Polonius NLL), we maintain precise knowledge about ***state*** (what is loaned out, what is not) and ***aliasing*** (via the unique loan identifiers) – in a flow-sensitive fashion (environments)
  - E-Shared-Or-Reserved-Borrow: augment the loan-set
- Enforcing well-formedness is done via side-conditions
  - E-Mut-Borrow: cannot borrow something already loaned out, or moved-out; cannot borrow through a shared borrow (would allow shared & mutable references to co-exist)
  - E-Shared-Or-Reserved-Borrow: cannot borrow something ***mutably*** loaned out or moved-out (borrowing something loaned-out with sharing is ok)
  - E-Move: cannot move a loaned-out value, i.e., cannot traverse pointers to move things you don't own; cannot move something moved already!
  - E-Copy: cannot copy something ***mutably*** loaned out, or ***mutably*** borrowed (this would duplicate ownership!!)
- Captures earlier intuitions (e.g. naming the old value in the E-Assign rule)

# Some gadgetry to support these rules

- Ghost updates to reorganize the environment to update loan-sets, or materialize borrows
- Read/write auxiliary judgments that allow fetching/updating in paths
- Free reorganizations of the environment that may be performed at any time (declarative), but in practice are done lazily (algorithmic)
- An auxiliary judgment “no outer loans”: overwriting something in an assignment that has outstanding loans would create dangling pointers

# Auxiliary judgment: no outer loans

- Judgment defined by omission
- Not a style I like a lot, but works well here
- The \ominus is just shorthand notation because the LaTeX looks bad otherwise :-)

$$\textsc{A-Shorthand}\ \frac{\ominus\ v_p}{v_p \text{ has no outer loans}} \qquad \textsc{A-Scalar}\ \frac{v = \text{true or false or } n_{\text{i32}} \text{ or } n_{\text{u32}} \text{ or } \ldots}{\ominus\ v} \qquad \textsc{A-Tuple}\ \frac{\ominus\ v_i}{\ominus\ (\vec{v})} \qquad \textsc{A-Constructor}\ \frac{\ominus\ v_i}{\ominus\ C[\vec{f} = \vec{v}]}$$

$$\textsc{A-Borrow-M}\ \frac{}{\ominus\ \text{borrow}^m\ \ell\ v} \qquad \textsc{A-Borrow-R-S}\ \frac{}{\ominus\ \text{borrow}^{r,s}\ \ell} \qquad \textsc{A-Bot}\ \frac{}{\ominus\ \bot}$$

# Auxiliary judgment: read

READ

$$\frac{p = P[x] \qquad x \mapsto v_x \in \Omega \qquad \Omega \vdash P(v_x) \Rightarrow v}{\Omega(p) \Rightarrow v}$$

R-BOX

$$\frac{\Omega \vdash P(v_P) \Rightarrow v}{\Omega \vdash (*^b P)(\mathsf{Box}\, v_P) \Rightarrow v}$$

R-MUT-BORROW

$$\frac{\Omega \vdash P(v_P) \Rightarrow v}{\Omega \vdash (*^m P)(\mathsf{borrow}^m\ \ell\ v_P) \Rightarrow v}$$

R-SHARED-BORROW

$$\frac{\mathsf{loan}^s\ \{\ell \cup \_\}\ v_P \in \Omega \qquad \Omega \vdash P(v_P) \Rightarrow v}{\Omega \vdash (*^s P)(\mathsf{borrow}^s\ \ell) \Rightarrow v}$$

R-FIELD

$$\frac{\Omega \vdash P(v_P) \Rightarrow v}{\Omega \vdash (P.f)(C\ [\overrightarrow{f = v_P}]) \Rightarrow v}$$

R-BASE

$$\frac{}{\Omega \vdash [.](v) \Rightarrow v}$$

R-SHARED-LOAN

$$\frac{P \neq [.] \qquad \Omega \vdash P(v_P) \Rightarrow v}{\Omega \vdash P(\mathsf{loan}^s\ \{\_\}\ v_P) \Rightarrow v}$$

# Auxiliary judgment: write

WRITE

$$\frac{x \mapsto v_x \in \Omega \qquad p = P[x] \qquad \Omega \vdash P(v_x) \leftarrow v \Rightarrow v'_x \qquad \Omega' = \Omega[x \mapsto v'_x]}{\Omega(p) \leftarrow v \Rightarrow \Omega'}$$

W-BOX

$$\frac{\Omega \vdash P(v_P) \leftarrow v \Rightarrow v'_P}{\Omega \vdash (*^b P)(\mathrm{Box}\, v_P) \leftarrow v \Rightarrow \mathrm{Box}\, v'_P}$$

W-MUT-BORROW

$$\frac{\Omega \vdash P(v_P) \leftarrow v \Rightarrow v'_P}{\Omega \vdash (*^m P)(\mathrm{borrow}^m\, \ell\, v_P) \leftarrow v \Rightarrow \mathrm{borrow}^m\, \ell\, v'_P}$$

W-BASE

$$\frac{}{\Omega \vdash [.](v_P) \leftarrow v \Rightarrow v}$$

W-FIELD

$$\frac{f = f_i \qquad \Omega \vdash P(v_{P,i}) \leftarrow v \Rightarrow v'_{P,i} \qquad v'_{P,j} = v_{P,j} \text{ for } j \neq i}{\Omega \vdash (P.f)(C[\overrightarrow{f = v_P}]) \leftarrow v \Rightarrow C[\overrightarrow{f = v'_P}]}$$

# Auxiliary judgment: reorganizations

NOT-BORROWED

$$\frac{\nexists V', V''.\ V[\cdot] = V'[\text{borrow}^m\ \_\ (V''[\cdot])] \quad \nexists V', V''.\ V[\cdot] = V'[\text{loan}^s\ \{\_\}\ (V''[\cdot])]}{\text{not\_borrowed\_value}\ V}$$

END-SHARED-OR-RESERVED-1

$$\frac{\text{not\_borrowed\_value}\ V}{\begin{array}{l}\Omega[x_1 \mapsto V[\text{borrow}^{r,s}\ \ell], x_2 \mapsto V'[\text{loan}^s\ \{\ell\}\ v]] \hookrightarrow \\ \Omega[x_1 \mapsto V[\bot], \qquad\qquad x_2 \mapsto V'[v]]\end{array}}$$

NOT-SHARED

$$\frac{\nexists V', V''.\ V[\cdot] = V'[\text{loan}^s\ \{\_\}\ (V''[\cdot])]}{\text{not\_shared\_value}\ V}$$

END-SHARED-OR-RESERVED-2

$$\frac{\text{not\_borrowed\_value}\ V \qquad \ell \notin \vec{\ell}}{\begin{array}{l}\Omega[x_1 \mapsto V[\text{borrow}^{r,s}\ \ell], x_2 \mapsto V'[\text{loan}^s\ \{\ell \cup \vec{\ell}\}\ v]] \hookrightarrow \\ \Omega[x_1 \mapsto V[\bot], \qquad\qquad x_2 \mapsto V'[\text{loan}^s\ \{\vec{\ell}\}\ v]]\end{array}}$$

END-MUT

$$\frac{\{\text{loan}, \text{borrow}^r\} \notin v \qquad \text{not\_borrowed\_value}\ V}{\begin{array}{l}\Omega[x_1 \mapsto V[\text{borrow}^m\ \ell\ v], x_2 \mapsto V'[\text{loan}^m\ \ell]] \hookrightarrow \\ \Omega[x_1 \mapsto V[\bot], \qquad\qquad x_2 \mapsto V'[v]]\end{array}}$$

ACTIVATE-RESERVED

$$\frac{\{\text{loan}, \text{borrow}^r\} \notin v \qquad \text{not\_shared\_value}\ V'}{\begin{array}{l}\Omega[x_1 \mapsto V[\text{borrow}^r\ \ell], \quad x_2 \mapsto V'[\text{loan}^s\ \{\ell\}\ v]] \hookrightarrow \\ \Omega[x_1 \mapsto V[\text{borrow}^m\ \ell\ v], x_2 \mapsto V'[\text{loan}^m\ \ell]]\end{array}}$$

# A word about ***two-phase*** borrows

## Checking two-phase borrows

Two-phase borrows are treated as if they were mutable borrows with the following exceptions:

1. At every location in the MIR we check if any two-phase borrows are activated at this location. If a live two phase borrow is activated at a location, then we check that there are no borrows that conflict with the two-phase borrow.
2. At the reservation point we error if there are conflicting live *mutable* borrows. And lint if there are any conflicting shared borrows.
3. Between the reservation and the activation point, the two-phase borrow acts as a shared borrow. We determine (in `is_active`) if we're at such a point by using the `Dominators` for the MIR graph.
4. After the activation point, the two-phase borrow acts as a mutable borrow.

This is super low-level and operational! 😱

# A word about ***two-phase*** borrows

- Intuition
  - starts as a shared borrow, but as restricted as a mutable borrow
  - can be promoted to a mutable borrow when the time comes
- Reserved borrow: models the first state where the borrow cannot be copied, dereferenced or written into
- Capture all the restrictions above in our rules

Commentary:

- This is another way of working around the lack of borrow polymorphism
  - Complements reborrowing
- Need to call an API that requires a shared borrow, but the borrow really ought to be mutable? Two-phase borrow

See https://rustc-dev-guide.rust-lang.org/borrow-check/two-phase-borrows.html

# Final comments on day 1

- Stating rules in a declarative manner allows focusing on the essence of what the system permits or disallows
- This is much more ***semantic*** view than an operational description either in terms of dominators, or syntactic lifetimes
- Leaves room for implementation strategies, which are also worthy of formalizing!
- We've learned about linearity, how applying it to a real language pays off, a few strategic design choices
  - How Rust designed its core language
  - How we (team Aeneas!) can give it some semantics that are closer to PL tradition

# Final comments on day 1

- This is not the only way to think about Rust! Compare & contrast: RustBelt, RustHornBelt, Oxide, Verus, and many more
- Different tools for different levels of abstraction and different stances on the right approach
- More commentary on related work at the end of the lecture tomorrow

**Our sponsors make OPLSS possible.**

Diamond Sponsor

**Jane Street**

https://www.janestreet.com/

---

Platinum Sponsors

National Science Foundation
WHERE DISCOVERIES BEGIN

https://www.nsf.gov/

*SIGPLAN*

https://www.sigplan.org/

---

Silver Sponsors

# Linearity, Rust and Verification

Jonathan Protzenko (Google)

with Son Ho, Guillaume Boisseau, Aymeric Fromherz, and the Aeneas collaborators

# Previously, on Rust and Linearity

- Linearity: gone “mainstream”, for the niche feature of providing memory safety for low-level languages
  - No use-after frees, safe concurrency, etc.
- Chief example: Rust, a C-like language with linearity for memory safety
  - Takes the form of borrows, implementing a mutable XOR aliased discipline
  - Several escape hatches to make sure the programmer isn’t stuck
- Rust is so structured that we can give it an operational semantics that…
  - Does not talk about pointers or memory
  - But instead uses a notion of loans to track who borrowed who
- Linearity: a way to make reasoning about programs easier
  - In this context: for verification, thanks to the mutable XOR aliased discipline
  - Mutable borrow: no one else can touch it, and if I touch it, no one else is impacted
  - Supports ***modular reasoning***

# Quick plan for today's lesson

- Switch from ***concrete semantics*** to ***symbolic semantics***, i.e. "we don't know concrete values of variables"
- Extend ***symbolic semantics*** with ***functional translation rules***
- Overview of Aeneas-as-a-tool, including its ***Lean backend***
- Three live demos:
  - how to setup verification using only Lean's library
  - example on lists, backwards functions
  - commentary of a ***real-world example*** from Google, verified with Aeneas

# Our running example for today


Live demo:

```
fn choose<'a, T>(b: bool, x: &'a mut T, y: &'a mut T) -> &'a mut T {
    if b { return x; } else { return y; } }
```

Rust guarantees:

- **disjointness:** x and y are not aliased
- **ownership transfer:** for the duration of `choose`, no one else owns x or y
- **unicity:** modifications through x or y leave other pieces of mutable data unaffected

Why is this a good example?

- **tricky:** returns a mutable reference
- **also:** need to reason about the caller

# Our running example for today (2)

Functional translation of this?

```
4   fn test_choose() {
5       let mut x = 0i32; let mut y = 0i32;
6       let z = choose(true, &mut x, &mut y);
7       *z = *z + 1;
8       assert!(*z == 1);
9       assert!(x == 1); assert!(y == 0); }
```

what happens here?

# Switching to symbolic semantics: an example

- Concrete values are not known
- Flow-sensitive refinement
  - Add constructor types
  - In essence, strong-update with fresh variables
  - Notice how having each value held once symbolically makes refinements easier
    → this is where linearity comes in handy!

```
1  fn f(mut o: Option<i32>) {   // o ↦ (σ : Option i32)
2    let po = &mut o;           // o ↦ loan^m ℓ;   po ↦ borrow^m ℓ (σ : Option i32)
3
4    match *po {
5      None => {                // o ↦ loan^m ℓ;   po ↦ borrow^m ℓ (σ : None)
6        panic!() }
7
8      Some => {                // o ↦ loan^m ℓ;   po ↦ borrow^m ℓ (Some (σ' : i32))
9        let r =
10         &mut (*po).Some.0;   // o ↦ loan^m ℓ;   po ↦ borrow^m ℓ (Some loan^m ℓ');   o_ref ↦ borrow^m ℓ' (σ' : i32)
11       *r = 1; }};}           // o ↦ loan^m ℓ;   po ↦ borrow^m ℓ (Some loan^m ℓ');   o_ref ↦ borrow^m ℓ' 1
```

# Switching to symbolic semantics: function example

- Lifetime? Region!
- A loss of precision in the ***borrow graph***
  - A region owns borrows and loans
  - Without specifying the relationships between them
  - Captures unknown aliasing relationships between inputs and outputs of a function
- Our analysis must ***remain*** modular
  - We use “region abstractions” noted A(ρ)

```
4   fn test_choose() {
5       let mut x = 0i32; let mut y = 0i32;
6       let z = choose(true, &mut x, &mut y);
7       *z = *z + 1;
8       assert!(*z == 1);
9       assert!(x == 1); assert!(y == 0); }
```

# Switching to symbolic semantics: function example (2)

```
1  let mut x = 0;   let mut y = 0;
2  let px = &mut x; let py = &mut y; // x ↦ loan^m ℓ_x, y ↦ loan^m ℓ_y, px ↦ borrow^m ℓ_x 0, py ↦ borrow^m ℓ_y 0
3  let pz = choose(true, move px, move py);
4  // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,   pz ↦ borrow^m ℓ_r (σ : uint32),
5  // A(ρ) { borrow^m ℓ_x 0,   borrow^m ℓ_y 0,   loan^m ℓ_r   }
6  *pz = *pz + 1;
7  // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,   pz ↦ borrow^m ℓ_r (σ' : uint32),          step 0
8  // A(ρ) { borrow^m ℓ_x 0,   borrow^m ℓ_y 0,   loan^m ℓ_r   }
9  // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,   pz ↦ ⊥,                                  step 1
10 // A(ρ) { borrow^m ℓ_x 0,   borrow^m ℓ_y 0,   σ'   }
11 // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,   pz ↦ ⊥,                                  step 2
12 // px' ↦ borrow^m ℓ_x σ_x,   py' ↦ borrow^m ℓ_y σ_y
13 // x ↦ σ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,   pz ↦ ⊥,   px' ↦ ⊥,   py' ↦ borrow^m ℓ_y σ_y   step 3
14 assert!(x == 1);
```

# Technical ingredients

- Extend values with symbolic values, denoted **σ**
- Extend types with constructors types, denoted **C (v1, …)**
- Rules to symbolically refine values under match branches
  - note: MIR has a match on the enum tag, but Charon reconstructs this – another reason to use LLBC
  - note: the rules are as expected – skipping them here
- Extend the reorganization of the environment, to express:
  - On-demand refinement of values, e.g. **$\sigma: (\tau_1 * \tau_2)$** can become **$(\sigma_1, \sigma_2): (\tau_1 * \tau_2)$** at any time
  - Borrows can be moved into region abstractions (function arguments)
  - Region abstractions can hand out loans (return value borrows ***from*** the region)
  - The earlier loan/borrow termination rule can happen with region loans
  - Regions with no outstanding loans can be terminated, and region borrows are returned to the caller

# How to get to a functional translation from here?

Some high-level intuitions:

- Our symbolic semantics already explain what happens for any program value: ***generate code accordingly***!
- Function call means we ***lose knowledge***
  - In the symbolic semantics: region abstraction, which carries imprecision about the aliasing relationships
  - In the functional translation: backward / forward function, which carries imprecision about the runtime behavior of the code

# Functional translation example: call-site

$$
\begin{aligned}
p &\mapsto (\text{loan}^m\ \ell_1, \text{loan}^m\ \ell_2)\\
px &\mapsto \bot\\
py &\mapsto \bot\\
pz &\mapsto \bot\\
px_g &\mapsto \text{borrow}^m\ \ell_1\ (\sigma_6 : \text{u32})\\
py_g &\mapsto \text{borrow}^m\ \ell_2\ (\sigma_7 : \text{u32})
\end{aligned}
$$

$$
\begin{aligned}
&\text{borrow}^m\ \ell_2\ (\sigma_2 : \text{u32}),\\
&\text{loan}^m\ \ell_3,\\
&\}
\end{aligned}
$$

```
1  let call_choose_fwd (s0 : (u32 & u32)) : u32 =
2    let (s1, s2) = s0 in
3    s3 <-- choose_fwd true s1 s2;
4    let s4 = s3 in
5    s5 <-- u32_add s4 1;
6    (s6, s7) <-- choose_back true s1 s2 s5;
7    [.]
```

# Functional translation example: backward function

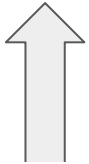
$A_{\text{input}}(\alpha)\{$ _ $\sigma_{\text{ret}},$ $\sigma_y,$ $\}$

$b \mapsto \text{true}$

$x \mapsto \bot$

$y \mapsto \bot$

```
1  let choose_back
2    (t : Type) (b : bool) (x : t) (y : t)
3    (ret : t) : result (t & t) =
4    if b then [.]
5    else [.]
```



```
1  let choose_back (t : Type) (b : bool) (x : t) (y : t) (ret : t) : result (t & t) =
2    if b then Return (ret, y)
3    else [.]
```

# A brief glimpse of the rules

- Borrows are values
- Projectors?
  - Gadget to deal with multiple regions at once
- Skipping return

T-Fun-Backward

$$\frac{A(\rho_i) = \left\{\overrightarrow{\mathrm{proj}_{\mathrm{l}}\ (\sigma : \tau)}\right\} \qquad \vec{\sigma}\ \text{fresh} \qquad \emptyset, \Omega \vdash s \uparrow^{\rho} e \qquad \Omega = \overrightarrow{A(\rho)}, \overrightarrow{x \mapsto \mathrm{proj}_{\mathrm{out}}\ \sigma}, \vec{x}_{\mathrm{local}} \mapsto \vec{\perp}, x_{\mathrm{ret}} \mapsto \perp}{\begin{array}{l}\mathrm{fn}\ f\ \langle\vec{\rho}\rangle\ (\vec{x} : \vec{\tau})\ (\vec{x}_{\mathrm{local}} : \vec{\tau}_{\mathrm{local}})\ (x_{\mathrm{ret}} : \tau_{\mathrm{ret}}) = s \uparrow^{\rho} \\ \mathrm{def}\ f_{\mathrm{back}(\rho)} = \lambda(\vec{\sigma} : \vec{\tau})(\sigma_{\mathrm{ret}} : \mathrm{proj}_{\rho}\ \tau_{\mathrm{ret}}).e\end{array}}$$

Pure-Mut-Borrow

$$\frac{\Omega \vdash v \updownarrow e}{\Omega \vdash \mathrm{borrow}^{m}\ \ell\ v \updownarrow e}$$

Pure-Shared-Borrow

$$\frac{\mathrm{loan}^{s}\ \{\vec{\ell}\}\ v \in \Omega \qquad \ell \in \vec{\ell} \qquad \Omega \vdash v \updownarrow e}{\Omega \vdash \mathrm{borrow}^{s}\ \ell \updownarrow e}$$

# Where is the additional complexity?

- Lots of machinery to deal with
  - Multiple regions: generic concept of distributing loans and borrows into the right region abstractions – we now track which region a borrow belongs to
  - Return value of backward functions: a tuple for all the arguments, where references to the return value point to the potentially-modified, caller-provided “new” return value
    - This is the **σx** vs **σret** business earlier
- Constructing terms and going from statement/expression to just an expression language (context with holes)
- Allowing reorganizations at key points
- etc.

# What’s missing?

- Reconciling symbolic environments, to avoid exponential explosion with sequences of conditionals, a merge operation
- Loop handling – also needs a merge operation to compute a fixpoint for symbolic environment

Described in a later paper.

Let’s look at how to actually verify programs.

# Connecting Aeneas to Lean (1)


Basics without Aeneas-specific libraries (pure Lean standard library):

- mvcgen for monadic program verification
- how to decorate functions with theorems and Hoare triples
- how to model the behavior of an existing library function

No Lean pre-requisites – I will explain the syntax live.

# Connecting Aeneas to Lean (1)

Resources:

- mvcgen tutorial: https://lean-lang.org/doc/tutorials/4.32.0-rc1//mvcgen/
- mvcgen doc: https://lean-lang.org/doc/reference/latest/The--mvcgen--tactic/#mvcgen-tactic

This is leveraged by the `hax` tool. Design:

- use Aeneas translation
- rely on annotations ***within*** the Rust source (not on the side!)
- but use mvcgen (standard), ***not*** the Aeneas proving apparatus

https://github.com/cryspen/hax/blob/main/examples/lean_barrett/src/lib.rs#L49

# Connecting Aeneas to Lean (2)


The Aeneas proof backend. Seeing the Aeneas proof apparatus:

- list example, take a pointer into the list, mutate, give it up
- two styles: functional & imperative
- proofs of correctness

Standard example form the tutorial. See https://github.com/AeneasVerif/aeneas/blob/main/tests/lean/Tutorial/Solutions.lean for more ways to prove this lemma.

# Connecting Aeneas to Lean (3)


The Aeneas proof backend. Real-world example: proving the correctness of a `get_unchecked` in an unsafe block. We will look at:

- architecture of a large proof like this
- identifying helpers
- automation: annotating lemmas with attributes
- AI-based proof generation

All from the point of view of an Aeneas user – no discussion of Aeneas internals.

# Connecting Aeneas to Lean (2)

More resources:

- Aeneas tutorial: https://github.com/AeneasVerif/aeneas/blob/main/tests/lean/Tutorial/Exercises.lean
- jxl-rs blog example: https://jonathan.protzenko.fr/2026/05/05/jxl-rs.html
- #decompose: https://github.com/AeneasVerif/aeneas/blob/3b85a8f304e41c219c058752f14c1d6c88349e4f/backends/lean/Aeneas/Command/Decompose.lean#L22

# Aeneas at scale

- My ex-colleagues at Microsoft have verified ***six*** cryptographic algorithms using Aeneas
- More applications: BAIF (x25519), Ethereum Foundation (PQ polynomial commitments), and more beyond cryptography

The Aeneas philosophy:

- verify code ***AS WRITTEN*** by the programmer
- do not argue for modifications; do not require exotic syntax, annotations, or libraries
- leverage Lean for extensible automation and meta-programming facilities

# Comments on the benefits of using Lean

Simply an experience report.

- `grind` is excellent
- top-notch relationship with the Lean FRO (thanks!) who have supported all of our efforts
- metaprogramming in the TacticM and SymM monads is great: “elaborator reflection”, or “Lean written in Lean”

# Aeneas, looking forward

- Separation logic for talking about unsafe code
  - Open question: how fancy does it need to be?
- Separation logic for concurrency
  - Same question
- Connection with inline ASM
  - Plenty of use-cases where ASM must remain (codecs, occasional crypto)
- Improved automation with upstream tactics
- Proof stability
  - grind has the same problems as SMT?
  - at least one can debug

# The only AI slide of this talk

This is an opinion, based on my experience…

- LLMs are not a substitute for learning how ***frameworks*** and ***semantics*** work
- LLMs are not a substitute for learning how ***proofs*** are ***architectured***
- LLMs work well in a few selected situations
  - evaluation function well-specified (test vectors)
  - small proofs at the leaves (even then!... check the result)
  - enhanced lemma search
  - proofs already in the training set

For now (might change in a week): automates some tedious work, but doesn’t relieve you of understanding and designing semantics and proofs.

# The only (other) AI slide of this talk

- Kraken, an x64 semantics written in Lean
- Different style of semantics – no WPs, the evaluator ***is*** the VC generator
- Using omnisemantics (see TOPLAS 2022)

We tried tokenmaxxing

- BAD BAD BAD for semantics
- BAD BAD BAD for libraries

Was it useful?

- Yes, for things we knew how to do (but lacked time to execute)
- With substantial post-AI cleanup (this is stuff we have to maintain)

# Recap — phew!

- Linearity good, especially with escape hatches and low-level control
- Rust occupies a practical spot in the design space → success
- For PL researchers, Rust is a prime target
  - lots of code being rewritten (opportunity!)
  - PL-friendly community
  - language design is semantics and verification-friendly
- Aeneas: verify code as-is ***AND*** interactive proof ***AND*** custom Lean backend
- Theory: loans-and-borrows, concrete & symbolic semantics, functional translation
  - More on the radar
- Tool: Charon + Aeneas, LLBC, proof automation, custom tactics, etc.
  - More applications on the horizon

# Ask me anything!

Or follow up: protz@google.com – ISE Formal