Linearity Rust Verification — Jonathan Protzenko

Lecture 2 - *June 30, 2026*

1 Symbolic Semantics

Go from *concrete semantics* to *symbolic semantics* where we don't know the concrete values of variables. We do *flow-sensitive refinement* by:

- Adding *constructor types*.
- Strong-updates with fresh variables.

1.1 Running Example

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

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

`'a` is a lifetime indicator to indicate that the return type is related to `x` and `y`. This function:

- Returns a mutable reference
- Need to reason about the caller

As long as `z` is live, you do not have ownership of `x` and `y` because their mutability is borrowed (by `z`).

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

1.1.1 Lean Translation

```
def choose (b : Bool) (x : T) (y: T): Result (T x (T -> (T x T))) := do
    if b then let back := fun x1 => (x1, y)
        ok (x, back)
    else let back := fun y1 => (x, y1)
        ok (y, back)

def main : Result Unit := do
    let (z, choose_back) <- choose true 0#i32 0#i32
    let z1 <- z + 1#i32
    let (x, _) := choose_back z1
    massert (x = 1#i32)
```

`choose` returns a callback that *tells the caller how to restore the values of* `x` *and* `y` after `z` is released. This is the essence of the *backward function* we build up to in the functional translation.

1.2 Symbolic Example

```
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
```

Symbolic evaluation refines values on demand under match branches (here, `Option` splits into `None` and `Some (σ : i32)`), while tracking loans and borrows symbolically rather than concrete memory.

1.3 Function Example

A *lifetime* becomes a *region*. There is a loss of precision in the *borrow graph*:

- A region owns borrows and loans,

- without specifying the relationships between them,
- capturing unknown aliasing relationships between the inputs and outputs of a function.

Our analysis must *remain modular*: we use *region abstractions* noted $A(\rho)$.

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

Borrow regions are captured but we lose information about exactly what got borrowed (e.g. we keep track that `z` is related to `x` and `y` but we don't know which one it is). Evaluation proceeds step by step, consuming the region abstraction $A(\rho)$ as borrows and loans are handed out and reclaimed.

1.4 Technical Ingredients

- Extend values with *symbolic values*, denoted σ
- Extend types with *constructor types*, denoted $C(v_1, \ldots)$
- Add rules to symbolically refine values under match branches
- 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 returned to the caller

2 Functional Translation Rules

How do we get to a functional translation from here? Two high-level intuitions:

- Our symbolic semantics already explain what happens for any program value — so we *generate code accordingly*.
- A function call means we *lose knowledge*. In the symbolic semantics this imprecision (about aliasing) is carried by the *region abstraction*; in the functional translation it is carried by a *backward / forward function* pair (imprecision about the runtime behavior of the code).

Each borrowing function is translated into a *forward* function (computing the result) and one or more *backward* functions (telling the caller how the borrowed inputs are updated once the returned reference is released). For `choose`:

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

At the call site the forward and backward functions are threaded together:

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

$$
\begin{aligned}
p &\mapsto (\text{loan}^m\ \ell_1, \text{loan}^m\ \ell_2)\\
px &\mapsto \bot\\
py &\mapsto \bot\\
pz &\mapsto \bot\\
px_g &\mapsto \text{borrow}^m\ \ell_1\ (\sigma_6 : \text{u32})\\
py_g &\mapsto \text{borrow}^m\ \ell_2\ (\sigma_7 : \text{u32})\\
&\text{borrow}^m\ \ell_2\ (\sigma_2 : \text{u32}),\\
&\text{loan}^m\ \ell_3,\\
&\}
\end{aligned}
$$

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

2.1 Brief Glimpse of the Rules

ıce

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

$$
\textsc{Pure-Mut-Borrow}\quad
\frac{\Omega \vdash v \updownarrow e}{\Omega \vdash \text{borrow}^m\ \ell\ v \updownarrow e}
\qquad
\textsc{Pure-Shared-Borrow}\quad
\frac{\text{loan}^s\ \{\vec{\ell}\}\ v \in \Omega \qquad \ell \in \vec{\ell} \qquad \Omega \vdash v \updownarrow e}{\Omega \vdash \text{borrow}^s\ \ell \updownarrow e}
$$

A few things to notice:

- *Borrows are values.*
- *Projectors* are a gadget to deal with multiple regions at once.
- The translation *skips return.*

Where is the additional complexity?

- *Multiple regions*: the general 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 σ_x vs. σ_{ret} business).
- Constructing terms and going from a statement/expression to just an expression language (context with holes).
- Allowing reorganizations at key points.
- ...

2.2 What's Missing

- *Reconciling symbolic environments*: to avoid exponential explosion with sequences of conditionals, we need a *merge* operation.
- *Loop handling*: also needs a merge operation, to compute a *fixpoint* for the symbolic environment (similar to Abstract Interpretation's widening).

These are described in a later paper.

2.3 Live Demo

```
fn optimized_mul(x: &mut u32) {
    *x <<= 1;
}

fn main() {
    let mut x = 1u32;
    optimized_mul(&mut x);
    optimized_mul(&mut x);
    assert!(x == 4u32);
}
```

We draw a boundary around the code to avoid analysing all of the standard library, which also includes unsafe Rust code; we *assert* the Rust standard library's implementation at the boundary.

Once we have the Lean translation we can verify properties about it. Since the translation abstracts away everything other than values, we don't have to think about low-level Rust pointers/references.

3 Connecting Aeneas to Lean

3.1 Basics (pure Lean standard library)

Starting with no Aeneas-specific libraries — just the 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.

This is leveraged by the `hax` tool, whose design is to: use the Aeneas translation, rely on annotations *within* the Rust source (not on the side), but use `mvcgen` (standard) rather than the Aeneas proving apparatus.

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/`

3.2 The Aeneas proof backend

Real-world example: proving the correctness of a `get_unchecked` inside an *unsafe* block (from the `jxl-rs` codebase at Google). We look at:

- Aeneas-style annotations,
- Aeneas' stepping tactic,
- `#decompose` and other goodies.

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

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`
- Example proof for a backward function: `https://github.com/AeneasVerif/aeneas/blob/main/tests/lean/Tut`

3.3 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.

At scale, ex-colleagues at Microsoft verified six cryptographic algorithms using Aeneas; further applications include BAIF (x25519) and the Ethereum Foundation (PQ polynomial commitments), and beyond cryptography.

3.4 Takeaways

- Linearity is good, especially with escape hatches and low-level control.
- Rust occupies a practical spot in the design space, and is a prime target for PL researchers.
- Aeneas: verify code as-is *and* interactive proof *and* a custom Lean backend.
- Theory: loans-and-borrows, concrete & symbolic semantics, functional translation.
- Tool: Charon + Aeneas, LLBC, proof automation, custom tactics.