AENEAS

from Rust Programs
to Pure Lambda Calculus

Son Ho (Inria),
Jonathan Protzenko (MSR)

Our experience with large-scale verification (1)

- 2016-2020: Project Everest (HACL*, EverCrypt, Signal*)
  - Low*, a shallow embedding of C in F*
  - **No separation logic:** modifies-clause theory *à la* Dafny
  - Heavy usage of **SMT-patterns** to reason about aliasing
  - Mostly **crypto primitives**: few buffers, straightforward aliasing, few branchings, arithmetic proofs
  - 300k LoCs in F*, 100k LoCs of generated C + ASM

Our experience with large-scale verification (1)

- 2016-2020: Project Everest (HACL*, EverCrypt, Signal*)
  - Low*, a shallow embedding of C in F*
  - **No separation logic:** modifies-clause theory *à la* Dafny
  - Heavy usage of **SMT-patterns** to reason about aliasing
  - Mostly **crypto primitives**: few buffers, straightforward aliasing, few branchings, arithmetic proofs
  - 300k LoCs in F*, 100k LoCs of generated C + ASM

- 2021: Noise*, a protocol compiler for 59+ protocols
  - **High-level API:** sessions, state machines, peer library (imperative maps), etc.
  - Memory safety, functional correctness, security proofs
  - 55k LoCs of F*; 5k LoCs of generated C per protocol instantiation
  - Still in Low* (!)
  - Planned to verify WireGuard VPN on top of Noise*: out of reach with current techno

Our experience with large-scale verification (2)

What we no longer want to do:

```
// Yes, we need this!!!
let step_plus_minus_eq (step : ℕ) : Lemma ((step + 1) - 1 = step) = ()
```

Our experience with large-scale verification (2)

What we no longer want to do:

```
// Yes, we need this!!!
let step_plus_minus_eq (step : ℕ) : Lemma ((step + 1) - 1 = step) = ()
```

```
 #restart-solver
-#push-options "--z3rlimit 2000 --ifuel 0"
-let mk_dstate_p_create #idc ssi initialize initiator r dvp epriv epub pid =
+#push-options "--z3rlimit 200 --ifuel 0 --using_facts_from '*,-LowStar.Monotonic.Buffer.unused_in_not_unused_in_disjoint_2'"
+let mk_dstate_p_create #idc ssi initialize initiator r0 dvp epriv epub pid =
```

Our experience with large-scale verification (2)

What we no longer want to do:

```
// Yes, we need this!!!
let step_plus_minus_eq (step : ℕ) : Lemma ((step + 1) - 1 = step) = ()
```

```
 #restart-solver
-#push-options "--z3rlimit 2000 --ifuel 0"
-let mk_dstate_p_create #idc ssi initialize initiator r dvp epriv epub pid =
+#push-options "--z3rlimit 200 --ifuel 0 --using_facts_from '*,-LowStar.Monotonic.Buffer.unused_in_not_unused_in_disjoint_2'"
+let mk_dstate_p_create #idc ssi initialize initiator r0 dvp epriv epub pid =
```

```
// Auxiliary functions
inline let mk_session_p_create_uses_e_no_alloc = ...
inline let mk_session_p_create_uses_e_no_alloc_memzero = ... // calls mk_session_p_create_uses_e_no_alloc
inline let mk_session_p_create_uses_e = ...  // mk_session_p_create_uses_e_no_alloc_memzero

let mk_session_p_create = ... // calls mk_session_p_create_uses_e
```

Our experience with large-scale verification (2)

What we no longer want to do:

```
// Yes, we need this!!!
let step_plus_minus_eq (step : ℕ) : Lemma ((step + 1) - 1 = step) = ()
```

```
 #restart-solver
-#push-options "--z3rlimit 2000 --ifuel 0"
-let mk_dstate_p_create #idc ssi initialize initiator r dvp epriv epub pid =
+#push-options "--z3rlimit 200 --ifuel 0 --using_facts_from '*,-LowStar.Monotonic.Buffer.unused_in_not_unused_in_disjoint_2'"
+let mk_dstate_p_create #idc ssi initialize initiator r0 dvp epriv epub pid =
```

```
// Auxiliary functions
inline let mk_session_p_create_uses_e_no_alloc = ...
inline let mk_session_p_create_uses_e_no_alloc_memzero = ... // calls mk_session_p_create_uses_e_no_alloc
inline let mk_session_p_create_uses_e = ...  // mk_session_p_create_uses_e_no_alloc_memzero

let mk_session_p_create = ... // calls mk_session_p_create_uses_e
```

1. Boring aliasing
2. Lack of interactivity (too much automation)
3. Non-modular (intrinsic) proofs

Our experience with large-scale verification (2)

What we no longer want to do:

```
// Yes, we need this!!!
let step_plus_minus_eq (step : ℕ) : Lemma ((step + 1) - 1 = step) = ()
```

```
 #restart-solver
-#push-options "--z3rlimit 2000 --ifuel 0"
-let mk_dstate_p_create #idc ssi initialize initiator r dvp epriv epub pid =
+#push-options "--z3rlimit 200 --ifuel 0 --using_facts_from '*,-LowStar.Monotonic.Buffer.unused_in_not_unused_in_disjoint_2'"
+let mk_dstate_p_create #idc ssi initialize initiator r0 dvp epriv epub pid =
```

```
// Auxiliary functions
inline let mk_session_p_create_uses_e_no_alloc = ...
inline let mk_session_p_create_uses_e_no_alloc_memzero = ... // calls mk_session_p_create_uses_e_no_alloc
inline let mk_session_p_create_uses_e = ...  // mk_session_p_create_uses_e_no_alloc_memzero

let mk_session_p_create = ... // calls mk_session_p_create_uses_e
```

1. Boring aliasing
2. Lack of interactivity (too much automation)
3. Non-modular (intrinsic) proofs

Our experience with large-scale verification (3)

- In HACL*, …, Noise*: actually, **memory reasoning should be straightforward**
- Using **separation logic** is overkill
- Can we exploit this *disciplined* memory usage?

Our experience with large-scale verification (3)

- In HACL*, …, Noise*: actually, **memory reasoning should be straightforward**
- Using **separation logic** is overkill
- Can we exploit this *disciplined* memory usage?

- What does “*disciplined*” mean? Rust answers the question!

Our experience with large-scale verification (3)

- In HACL*, …, Noise*: actually, **memory reasoning should be straightforward**
- Using **separation logic** is overkill
- Can we exploit this *disciplined* memory usage?

- What does "*disciplined*" mean? Rust answers the question!

- In C:

```
void f (int *x, int *y);
```

Dangling pointers?
Aliasing?

```
B.live h x /\
B.live h y /\
B.disjoint x y
```

Our experience with large-scale verification (3)

- In HACL*, …, Noise*: actually, **memory reasoning should be straightforward**
- Using **separation logic** is overkill
- Can we exploit this *disciplined* memory usage?

- What does "*disciplined*" mean? Rust answers the question!

- In C:

```
void f (int *x, int *y);
```

Dangling pointers?
Aliasing?

```
B.live h x /\
B.live h y /\
B.disjoint x y
```

- In Rust:

```
fn f(x : &mut i32, y : &mut i32);
```

'x' and 'y' are live and disjoint

Using Rust’s type system to simplify verification

- **Not a new idea**:
  - **Prusti** (Viper): automate the application of memory reasoning rules
  - RustHornBelt/**Creusot**: “purification” through prophecy variables
  - **Verus**: early work (frontend *à la* Dafny)

  ⇒ all target user-friendly frontends, **intrinsic proofs** and **high-automation** through SMT solvers via a *logical encoding*

Using Rust's type system to simplify verification

- **Not a new idea**:
  - **Prusti** (Viper): automate the application of memory reasoning rules
  - RustHornBelt/**Creusot**: "purification" through prophecy variables
  - **Verus**: early work (frontend *à la* Dafny)

  ⇒ all target user-friendly frontends, **intrinsic proofs** and **high-automation** through SMT solvers via a *logical encoding*

- **Our project**:
  - **Extrinsic proofs**
  - **Modularity**
  - Automated solvers are great, but we need an escape hatch: **interaction with tactics**

Using Rust's type system to simplify verification

- **Not a new idea**:
  - **Prusti** (Viper): automate the application of memory reasoning rules
  - RustHornBelt/**Creusot**: "purification" through prophecy variables
  - **Verus**: early work (frontend *à la* Dafny)

  ⇒ all target user-friendly frontends, **intrinsic proofs** and **high-automation** through SMT solvers via a *logical encoding*

- **Our project**:
  - **Extrinsic proofs**
  - **Modularity**
  - Automated solvers are great, but we need an escape hatch: **interaction with tactics**

  ⇒ translate safe Rust programs to *idiomatic* pure lambda calculus (*executable*)

  ⇒ generate code for several back-ends (Coq, F*, HOL4…)

  - Similar to: **Electrolysis** (Rust to Lean), **Heapster** (LLVM to pure spec in Coq)

Pure Translation – Returning Unknown Borrows

```rust
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32)
  -> &'a mut i32
{
    if b { x }
    else { y }
}
```

```rust
// Usage example:
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y); // (i)
*z = 2; // (ii)
// Lifetime 'a ends here - (iii)
assert!(x == 2);
assert!(y == 1);
...
```

Pure Translation – Returning Unknown Borrows

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32)
  -> &'a mut i32
{
    if b { x }
    else { y }
}
```

```
let choose (b : bool) (x : i32) (y : i32) : i32 =
  if b then x else y
```

```
// Usage example:
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y); // (i)
*z = 2; // (ii)
// Lifetime 'a ends here - (iii)
assert!(x == 2);
assert!(y == 1);
...
```

```
let x = 0 in
let y = 1 in
let z = choose true x y in // (i)
let z = 2 in // (ii)
... // (iii)
```

Pure Translation – Returning Unknown Borrows

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32)
  -> &'a mut i32
{
    if b { x }
    else { y }
}
```

```
let choose (b : bool) (x : i32) (y : i32) : i32 =
  if b then x else y
```

```
// Usage example:
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y); // (i)
*z = 2; // (ii)
// Lifetime 'a ends here - (iii)
assert!(x == 2);
assert!(y == 1);
...
```

```
let x = 0 in
let y = 1 in
let z = choose true x y in // (i)
let z = 2 in // (ii)
let (x, y) = ?? in // (iii)
...
```

Pure Translation – Returning Unknown Borrows

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32)
  -> &'a mut i32
{
    if b { x }
    else { y }
}
```

```
let choose (b : bool) (x : i32) (y : i32) : i32 =
  if b then x else y
```

```
// Usage example:
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y); // (i)
*z = 2; // (ii)
// Lifetime 'a ends here - (iii)
assert!(x == 2);
assert!(y == 1);
...
```

```
let x = 0 in
let y = 1 in
let z = choose true x y in // (i)
let z = 2 in // (ii)
let (x, y) = (z, y) in // (iii)
...
```

Pure Translation – Returning Unknown Borrows

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32)
  -> &'a mut i32
{
    if b { x }
    else { y }
}
```

```
let choose (b : bool) (x : i32) (y : i32) : i32 =
  if b then x else y
```

```
// Usage example:
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y); // (i)
*z = 2; // (ii)
// Lifetime 'a ends here - (iii)
assert!(x == 2);
assert!(y == 1);
...
```

```
let x = 0 in
let y = 1 in
let z = choose true x y in // (i)
let z = 2 in // (ii)
let (x, y) = if true then (z, y) else (x, z) in // (iii)
...
```

Pure Translation – Returning Unknown Borrows

```rust
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32)
  -> &'a mut i32
{
    if b { x }
    else { y }
}
```

```ocaml
let choose_fwd (b : bool) (x : i32) (y : i32) : i32 =
  if b then x else y

let choose_back (b : bool) (x : i32) (y : i32) (z : i32) : i32 * i32 =
  if b then (z, y) else (x, z)
```

```rust
// Usage example:
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y); // (i)
*z = 2; // (ii)
// Lifetime 'a ends here - (iii)
assert!(x == 2);
assert!(y == 1);
...
```

```ocaml
let x = 0 in
let y = 1 in
let z = choose_fwd true x y in // (i)
let z = 2 in // (ii)
let (x, y) = choose_back true x y z in // (iii)
assert(x == 2);
assert(y == 1);
...
```

Pure Translation – Returning Unknown Borrows

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32)
  -> &'a mut i32
{
    if b { x }
    else { y }
}
```

```
let choose_fwd (b : bool) (x : i32) (y : i32) : i32 =
  if b then x else y

let choose_back (b : bool) (x : i32) (y : i32) (z : i32) : i32 * i32 =
  if b then (z, y) else (x, z)
```

```
// Usage example:
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y); // (i)
*z = 2; // (ii)
// Lifetime 'a ends here - (iii)
assert!(x == 2);
assert!(y == 1);
...
```

```
let x = 0 in
let y = 1 in
let z = choose_fwd true x y in // (i)
let z = 2 in // (ii)
let (x, y) = choose_back true x y z in // (iii)
assert(x == 2);
assert(y == 1);
...
```

⇒ Translation by means of forward and *backward* functions

⇒ Type of those functions only depends on *signature* of Rust functions

Pure Translation – Lists & Recursion

In Rust:

```
let ls : List<T> = ...;
let x : &mut i32 = get_nth_mut(&mut ls, n);
*x = 4;
// end the borrow in x
...
```

Pure Translation – Lists & Recursion

In Rust:

```
let ls : List<T> = ...;
let x : &mut i32 = get_nth_mut(&mut ls, n);
*x = 4;
// end the borrow in x
...
```

In F*:

```
let x = get_nth_mut_fwd ls n in // read
let x = 4 in
// end the borrow in x
let ls = get_nth_mut_back ls n x in // update
...
```

Pure Translation – Lists & Recursion

In Rust:

```
let ls : List<T> = ...;
let x : &mut i32 = get_nth_mut(&mut ls, n);
*x = 4;
// end the borrow in x
...
```

In F*:

```
let x = get_nth_mut_fwd ls n in // read
let x = 4 in
// end the borrow in x
let ls = get_nth_mut_back ls n x in // update
...
```

⇒ Read then update: **idiomatic** translation

Pure Translation – PoC (function pointers)

Our translation is ***entirely*** type-based (hence modular)

In Rust:

```
fn apply_function(f: for<'a> fn(bool, &'a mut i32, &'a mut i32) -> &'a mut i32) -> i32 {
    let mut x = 0;
    let mut y = 1;
    let pz = f(true, &mut x, &mut y);
    *pz = 2;
    // 'a ends here
    return x;
}
```

Pure Translation – PoC (function pointers)

Our translation is ***entirely*** type-based (hence modular)

In Rust:

```
fn apply_function(f: for<'a> fn(bool, &'a mut i32, &'a mut i32) -> &'a mut i32) -> i32 {
    let mut x = 0;
    let mut y = 1;
    let pz = f(true, &mut x, &mut y);
    *pz = 2;
    // 'a ends here
    return x;
}
```

In F*:

```
let apply_function (f_fwd : bool -> i32 -> i32 -> i32)
                   (f_back : bool -> i32 -> i32 -> i32 -> (i32 * i32)) : i32 =
    let x = 0 in
    let y = 1 in
    let pz = f_fwd true x y in
    let pz = 2 in
    let (x, y) = f_back true x y pz in
    x
```

Pure Translation – PoC (function pointers)

Our translation is ***entirely*** type-based (hence modular)

In Rust:

```
fn apply_function(f: for<'a> fn(bool, &'a mut i32, &'a mut i32) -> &'a mut i32) -> i32 {
    let mut x = 0;
    let mut y = 1;
    let pz = f(true, &mut x, &mut y);
    *pz = 2;
    // 'a ends here
    return x;
}
```

In F*:

```
let apply_function (f_fwd : bool -> i32 -> i32 -> i32)
                   (f_back : bool -> i32 -> i32 -> i32 -> (i32 * i32)) : i32 =
    let x = 0 in
    let y = 1 in
    let pz = f_fwd true x y in
    let pz = 2 in
    let (x, y) = f_back true x y pz in
    x
```

- Useful because Rust is very higher-order (trait system)
- Same idea applies to **external dependencies** and **opaque modules**

How does that work?

What do we need?

- Know borrow graph at each point of the program
- Precisely abstract function calls

How does that work?

What do we need?

- Know borrow graph at each point of the program
- Precisely abstract function calls

3 steps:

- **Operational semantics** for borrows
- **Regions + symbolic values** to handle functions calls;
  allows to perform symbolic executions
- Derivation of a **pure translation from a symbolic execution**

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                  // (i)

let mut px1 = &mut x;           // (ii)

*px1 = 1;                       // (iii)

let mut px2 = &mut (*px1);      // (iv)

assert!(x == 1);                // (v)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                  // (i)

let mut px1 = &mut x;           // (ii)

*px1 = 1;                       // (iii)

let mut px2 = &mut (*px1);      // (iv)

assert!(x == 1);                // (v)
```

```
// (i)
x -> (0 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                  // (i)

let mut px1 = &mut x;           // (ii)

*px1 = 1;                       // (iii)

let mut px2 = &mut (*px1);      // (iv)

assert!(x == 1);                // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> (0 : u32)
px1 -> ?
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                // (i)

let mut px1 = &mut x;         // (ii)

*px1 = 1;                     // (iii)

let mut px2 = &mut (*px1);    // (iv)

assert!(x == 1);              // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> (0 : u32)
px1 -> ?
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                // (i)

let mut px1 = &mut x;         // (ii)

*px1 = 1;                     // (iii)

let mut px2 = &mut (*px1);    // (iv)

assert!(x == 1);              // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> ?
px1 -> mut_borrow .. (0 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                // (i)

let mut px1 = &mut x;         // (ii)

*px1 = 1;                     // (iii)

let mut px2 = &mut (*px1);    // (iv)

assert!(x == 1);              // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                // (i)

let mut px1 = &mut x;         // (ii)

*px1 = 1;                     // (iii)

let mut px2 = &mut (*px1);    // (iv)

assert!(x == 1);              // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                // (i)

let mut px1 = &mut x;         // (ii)

*px1 = 1;                     // (iii)

let mut px2 = &mut (*px1);    // (iv)

assert!(x == 1);              // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                  // (i)

let mut px1 = &mut x;           // (ii)

*px1 = 1;                       // (iii)

let mut px2 = &mut (*px1);      // (iv)

assert!(x == 1);                // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                 // (i)

let mut px1 = &mut x;          // (ii)

*px1 = 1;                      // (iii)

let mut px2 = &mut (*px1);     // (iv)

assert!(x == 1);               // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                // (i)

let mut px1 = &mut x;         // (ii)

*px1 = 1;                     // (iii)

let mut px2 = &mut (*px1);    // (iv)

assert!(x == 1);              // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                  // (i)

let mut px1 = &mut x;           // (ii)

*px1 = 1;                       // (iii)

let mut px2 = &mut (*px1);      // (iv)

assert!(x == 1);                // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                 // (i)

let mut px1 = &mut x;          // (ii)

*px1 = 1;                      // (iii)

let mut px2 = &mut (*px1);     // (iv)

assert!(x == 1);               // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)

// (iv)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)
px2 -> ?
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                // (i)

let mut px1 = &mut x;         // (ii)

*px1 = 1;                     // (iii)

let mut px2 = &mut (*px1);    // (iv)

assert!(x == 1);              // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)

// (iv)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)
px2 -> ?
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                // (i)

let mut px1 = &mut x;         // (ii)

*px1 = 1;                     // (iii)

let mut px2 = &mut (*px1);    // (iv)

assert!(x == 1);              // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)

// (iv)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)
px2 -> ?
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;               // (i)

let mut px1 = &mut x;        // (ii)

*px1 = 1;                    // (iii)

let mut px2 = &mut (*px1);   // (iv)

assert!(x == 1);             // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)

// (iv)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                // (i)

let mut px1 = &mut x;         // (ii)

*px1 = 1;                     // (iii)

let mut px2 = &mut (*px1);    // (iv)

assert!(x == 1);              // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)

// (iv)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                // (i)

let mut px1 = &mut x;         // (ii)

*px1 = 1;                     // (iii)

let mut px2 = &mut (*px1);    // (iv)

assert!(x == 1);              // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)

// (iv)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)

// (v)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                 // (i)

let mut px1 = &mut x;          // (ii)

*px1 = 1;                      // (iii)

let mut px2 = &mut (*px1);     // (iv)

assert!(x == 1);               // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)

// (iv)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)

// (v)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                 // (i)

let mut px1 = &mut x;          // (ii)

*px1 = 1;                      // (iii)

let mut px2 = &mut (*px1);     // (iv)

assert!(x == 1);               // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)

// (iv)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)

// (v)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                // (i)

let mut px1 = &mut x;         // (ii)

*px1 = 1;                     // (iii)

let mut px2 = &mut (*px1);    // (iv)

assert!(x == 1);              // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)

// (iv)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)

// (v)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                // (i)

let mut px1 = &mut x;         // (ii)

*px1 = 1;                     // (iii)

let mut px2 = &mut (*px1);    // (iv)

assert!(x == 1);              // (v)
```

```
// (i)
x -> (0 : u32)
// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)
// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)
// (iv)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)
// (v)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                // (i)

let mut px1 = &mut x;         // (ii)

*px1 = 1;                     // (iii)

let mut px2 = &mut (*px1);    // (iv)

assert!(x == 1);              // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)

// (iv)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)

// (v)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                // (i)

let mut px1 = &mut x;         // (ii)

*px1 = 1;                     // (iii)

let mut px2 = &mut (*px1);    // (iv)

assert!(x == 1);              // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)

// (iv)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)

// (v)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;               // (i)

let mut px1 = &mut x;        // (ii)

*px1 = 1;                    // (iii)

let mut px2 = &mut (*px1);   // (iv)

assert!(x == 1);             // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)

// (iv)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)

// (v)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)
px2 -> ⊥
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                  // (i)

let mut px1 = &mut x;           // (ii)

*px1 = 1;                       // (iii)

let mut px2 = &mut (*px1);      // (iv)

assert!(x == 1);                // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)

// (iv)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)

// (v)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)
px2 -> ⊥
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                // (i)

let mut px1 = &mut x;         // (ii)

*px1 = 1;                     // (iii)

let mut px2 = &mut (*px1);    // (iv)

assert!(x == 1);              // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)

// (iv)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)

// (v)
x   -> (1 : u32)
px1 -> ⊥
px2 -> ⊥
```

Operational Semantics for Borrows - Mutable

```
let mut x = 0;                // (i)

let mut px1 = &mut x;         // (ii)

*px1 = 1;                     // (iii)

let mut px2 = &mut (*px1);    // (iv)

assert!(x == 1);              // (v)
```

```
// (i)
x -> (0 : u32)

// (ii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (0 : u32)

// (iii)
x   -> mut_loan l0
px1 -> mut_borrow l0 (1 : u32)

// (iv)
x   -> mut_loan l0
px1 -> mut_borrow l0 (mut_loan l1)
px2 -> mut_borrow l1 (1 : u32)

// (v)
x   -> (1 : u32)
px1 -> ⊥
px2 -> ⊥
```

- **Lazy** semantics of borrows
- Semantic analysis of borrows (Rust borrow checker is syntactic)

Operational Semantics for Borrows - Shared

```
let x = 0;       // (i)

let px1 = &x;    // (ii)

let px2 = &x;    // (iii)
```

Operational Semantics for Borrows - Shared

```
let x = 0;       // (i)

let px1 = &x;    // (ii)

let px2 = &x;    // (iii)
```

```
// (i)
x -> (0 : i32)
```

Operational Semantics for Borrows - Shared

```
let x = 0;       // (i)

let px1 = &x;    // (ii)

let px2 = &x;    // (iii)
```

```
// (i)
x -> (0 : i32)


// (ii)
x   -> shared_loan {l0} (0 : i32)
px1 -> shared_borrow l0
```

Operational Semantics for Borrows - Shared

```
let x = 0;       // (i)

let px1 = &x;    // (ii)

let px2 = &x;    // (iii)
```

```
// (i)
x -> (0 : i32)


// (ii)
x   -> shared_loan {l0} (0 : i32)
px1 -> shared_borrow l0


// (iii)
x   -> shared_loan {l0, l1} (0 : i32)
px1 -> shared_borrow l0
px2 -> shared_borrow l1
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> mut_borrow l0 (0 : i32)
py -> mut_borrow l1 (1 : i32)
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> mut_borrow l0 (0 : i32)
py -> mut_borrow l1 (1 : i32)
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> mut_borrow l0 (0 : i32)
py -> mut_borrow l1 (1 : i32)
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥ // mut_borrow l0 (0 : i32)
py -> mut_borrow l1 (1 : i32)
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥ // mut_borrow l0 (0 : i32)
py -> ⊥ // mut_borrow l1 (1 : i32)
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥ // mut_borrow l0 (0 : i32)
py -> ⊥ // mut_borrow l1 (1 : i32)
z  -> mut_borrow l2 (s0 : i32)
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥ // mut_borrow l0 (0 : i32)
py -> ⊥ // mut_borrow l1 (1 : i32)
z  -> mut_borrow l2 (s0 : i32)
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> mut_borrow l2 (s0 : i32)
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    mut_loan l2
}
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> mut_borrow l2 (s0 : i32)
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    mut_loan l2
}
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> mut_borrow l2 (s0 : i32)
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    mut_loan l2
}
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> mut_borrow l2 (s0 : i32)
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    mut_loan l2
}
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> mut_borrow l2 (s1 : i32)
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    mut_loan l2
}
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> mut_borrow l2 (s1 : i32)
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    mut_loan l2
}
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> mut_borrow l2 (s1 : i32)
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    mut_loan l2
}
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> mut_borrow l2 (s1 : i32)
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    mut_loan l2
}
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> ⊥
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    (s1 : i32)
}
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> ⊥
r0 {
     mut_borrow l0 (0 : i32)
     mut_borrow l1 (1 : i32)
     (s1 : i32)
}
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> ⊥
r0 {
     mut_borrow l0 (s2 : i32)
     mut_borrow l1 (s3 : i32)
     (s1 : i32)
}
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> (s2 : i32)
y  -> (s3 : i32)
px -> ⊥
py -> ⊥
z  -> ⊥
r0 {
    ⊥ // gave back: s2
    ⊥ // gave back: s3
    (s1 : i32)
}
```

Abstracting function calls

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> (s2 : i32)
y  -> (s3 : i32)
px -> ⊥
py -> ⊥
z  -> ⊥
r0 {
    ⊥ // gave back: s2
    ⊥ // gave back: s3
    (s1 : i32)
}
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
```

```
// Translation
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> mut_borrow l0 (0 : i32)
py -> mut_borrow l1 (1 : i32)
```

```
// Translation
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> mut_borrow l0 (0 : i32)
py -> mut_borrow l1 (1 : i32)
```

```
// Translation
.. <-- choose_fwd .... . .;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> mut_borrow l0 (0 : i32)
py -> mut_borrow l1 (1 : i32)
```

```
// Translation
.. <-- choose_fwd true . .;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥ // mut_borrow l0 (0 : i32)
py -> mut_borrow l1 (1 : i32)
```

```
// Translation
.. <-- choose_fwd true 0 .;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥ // mut_borrow l0 (0 : i32)
py -> ⊥ // mut_borrow l1 (1 : i32)
```

```
// Translation
.. <-- choose_fwd true 0 1;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥ // mut_borrow l0 (0 : i32)
py -> ⊥ // mut_borrow l1 (1 : i32)
z  -> mut_borrow l2 (s0 : i32)
```

```
// Translation
s0 <-- choose_fwd true 0 1;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥ // mut_borrow l0 (0 : i32)
py -> ⊥ // mut_borrow l1 (1 : i32)
z  -> mut_borrow l2 (s0 : i32)
```

```
// Translation
s0 <-- choose_fwd true 0 1;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> mut_borrow l2 (s0 : i32)
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    mut_loan l2
}
```

```
// Translation
s0 <-- choose_fwd true 0 1;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> mut_borrow l2 (s0 : i32)
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    mut_loan l2
}
```

```
// Translation
s0 <-- choose_fwd true 0 1;
.. <-- i32_add .. .;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> mut_borrow l2 (s0 : i32)
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    mut_loan l2
}
```

```
// Translation
s0 <-- choose_fwd true 0 1;
.. <-- i32_add s0 .;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> mut_borrow l2 (s0 : i32)
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    mut_loan l2
}
```

```
// Translation
s0 <-- choose_fwd true 0 1;
.. <-- i32_add s0 2;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> mut_borrow l2 (s1 : i32)
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    mut_loan l2
}
```

```
// Translation
s0 <-- choose_fwd true 0 1;
s1 <-- i32_add s0 2;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> mut_borrow l2 (s1 : i32)
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    mut_loan l2
}
```

```
// Translation
s0 <-- choose_fwd true 0 1;
s1 <-- i32_add s0 2;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> mut_borrow l2 (s1 : i32)
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    mut_loan l2
}
```

```
// Translation
s0 <-- choose_fwd true 0 1;
s1 <-- i32_add s0 2;
........ <-- choose_back true 0 1 ..;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> mut_borrow l2 (s1 : i32)
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    mut_loan l2
}
```

```
// Translation
s0 <-- choose_fwd true 0 1;
s1 <-- i32_add s0 2;
........ <-- choose_back true 0 1 ..;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> ⊥
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    (s1 : i32)
}
```

```
// Translation
s0 <-- choose_fwd true 0 1;
s1 <-- i32_add s0 2;
........ <-- choose_back true 0 1 s1;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> ⊥
r0 {
    mut_borrow l0 (0 : i32)
    mut_borrow l1 (1 : i32)
    (s1 : i32)
}
```

```
// Translation
s0 <-- choose_fwd true 0 1;
s1 <-- i32_add s0 2;
........ <-- choose_back true 0 1 s1;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> mut_loan l0
y  -> mut_loan l1
px -> ⊥
py -> ⊥
z  -> ⊥
r0 {
    mut_borrow l0 (s2 : i32)
    mut_borrow l1 (s3 : i32)
    (s1 : i32)
}
```

```
// Translation
s0 <-- choose_fwd true 0 1;
s1 <-- i32_add s0 2;
........ <-- choose_back true 0 1 s1;
[.]
```

Abstracting function calls + Translation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

```
// Env
x  -> (s2 : i32)
y  -> (s3 : i32)
px -> ⊥
py -> ⊥
z  -> ⊥
r0 {
    ⊥ // gave back: s2
    ⊥ // gave back: s3
    (s1 : i32)
}
```

```
// Translation
s0 <-- choose_fwd true 0 1;
s1 <-- i32_add s0 2;
(s2, s3) <-- choose_back true 0 1 s1;
[.]
```

Abstracting function calls + Translation

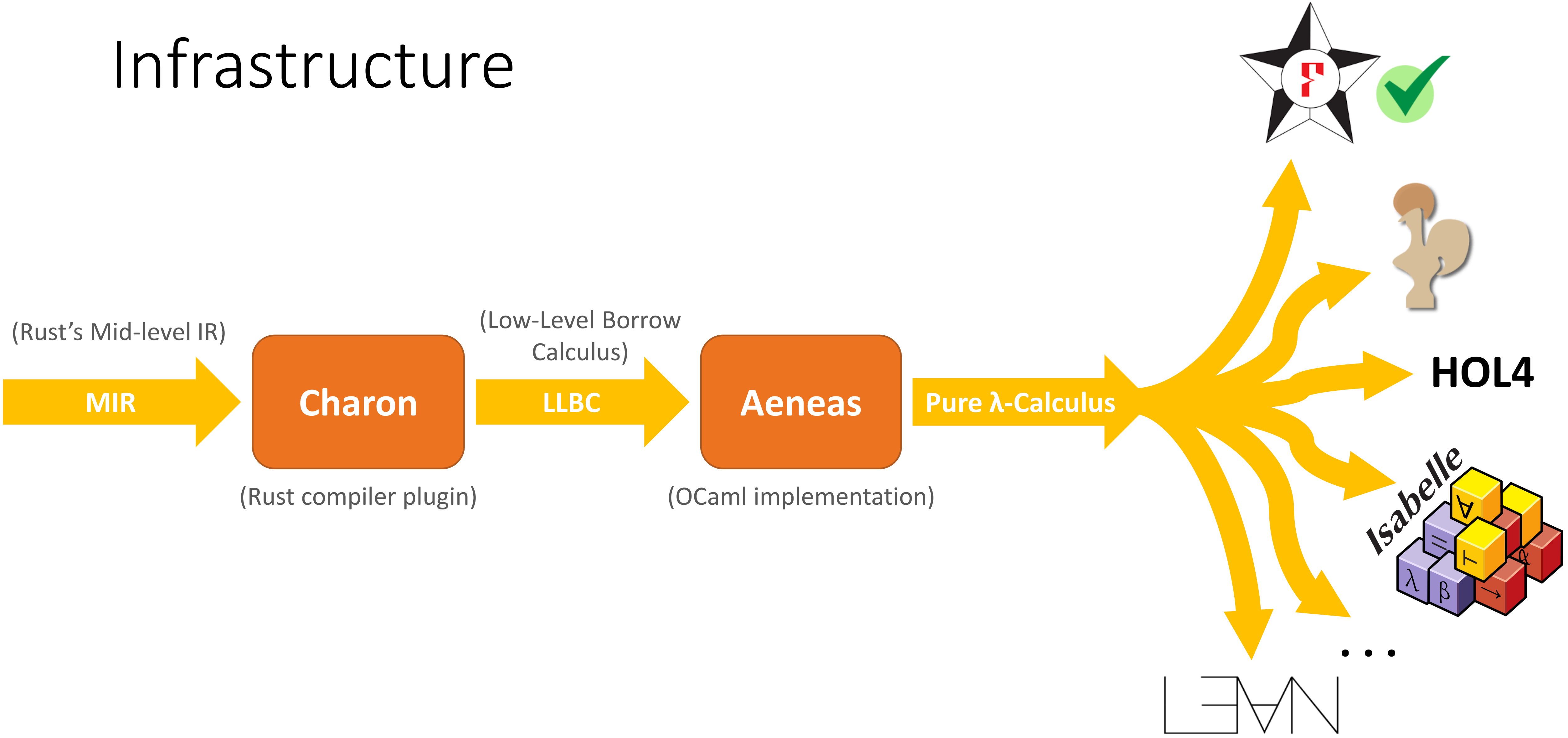
```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32
```

```
// Code
let mut x = 0;
let mut y = 1;
let px = &mut x;
let py = &mut y;
let z = choose(true, move px, move py);
*z = *z + 2;
assert!(x == 2);
assert!(y == 1);
```

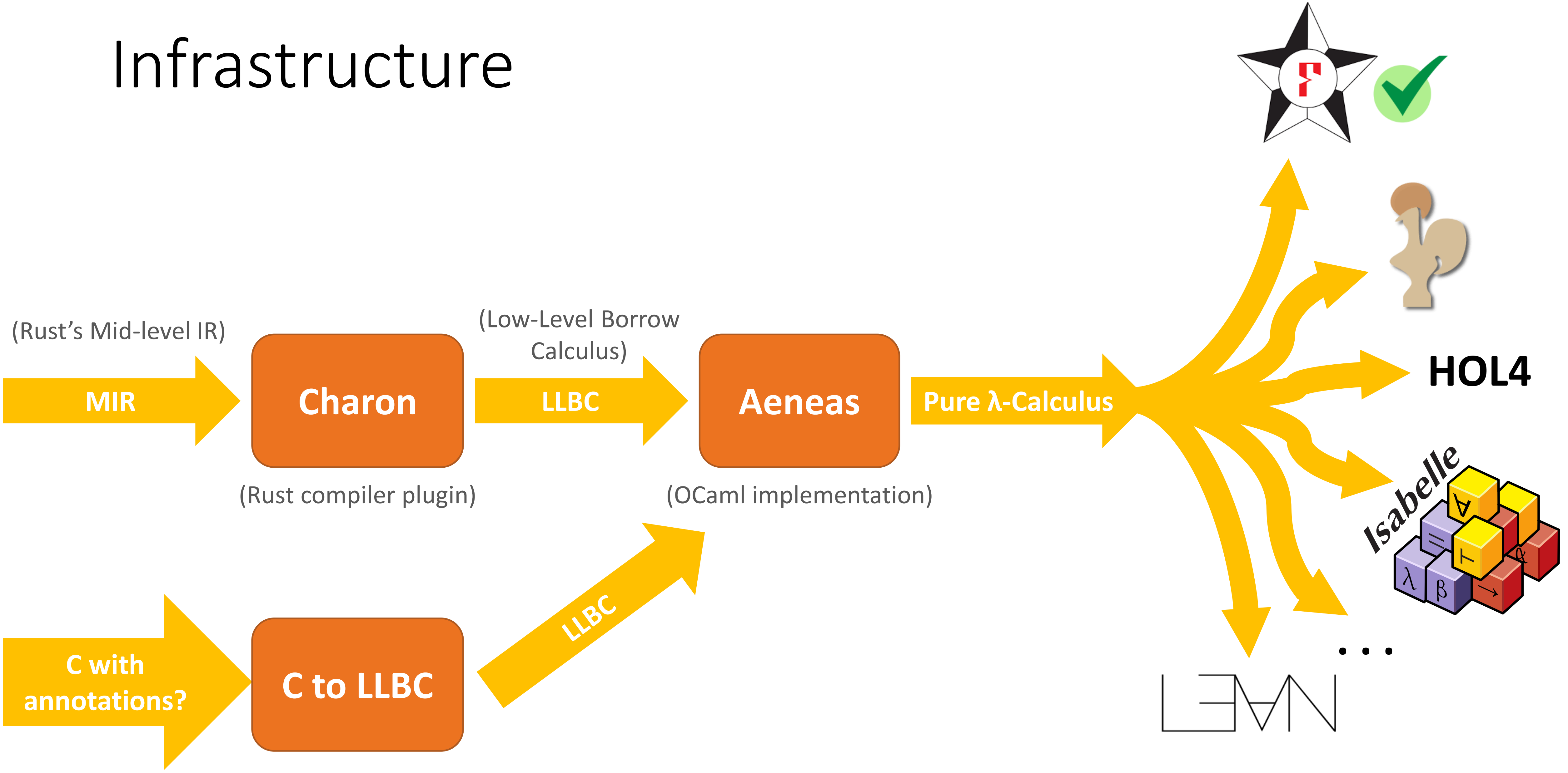
```
// Env
x  -> (s2 : i32)
y  -> (s3 : i32)
px -> ⊥
py -> ⊥
z  -> ⊥
r0 {
    ⊥ // gave back: s2
    ⊥ // gave back: s3
    (s1 : i32)
}
```

```
// Translation
s0 <-- choose_fwd true 0 1;
s1 <-- i32_add s0 2;
(s2, s3) <-- choose_back true 0 1 s1;
[.]
```

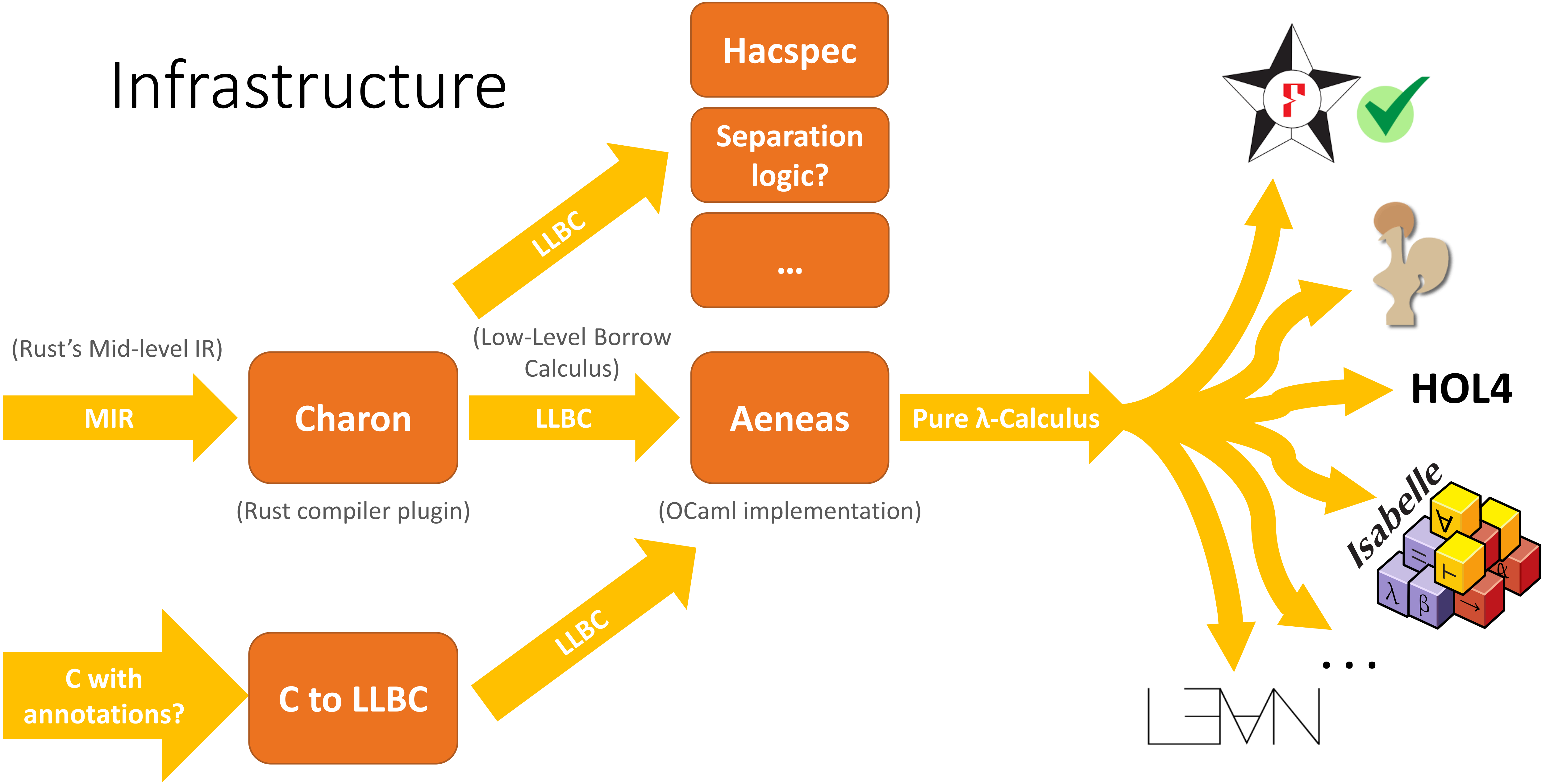
Infrastructure



Infrastructure



Infrastructure



Next steps

- Add at least one truly interactive backend (Coq, HOL4 , Lean)
- Extend supported subset (loops, traits…)
- Soundness proofs
- Ramping up verification:
  - B-epsilon-tree: ongoing
  - WireGuard VPN? MLS?

Conclusion: Aeneas, in a nutshell

- **New point in the design space**
  - functional, modular, type-based translation
  - no annotations in source program
  - translation to existing proof frameworks for *extrinsic proofs* (now: F*, ongoing : Coq, HOL4…)
- **Aim for verification in the large**
  - no “tour de force” on ninja unsafe code
  - rather, properties about large applications written in safe Rust
- **A user-friendly semantics for borrows**
  - lazy semantics, semantical analysis
  - yields a functional, executable translation, hence a program proof pipeline

Conclusion: Aeneas, in a nutshell

- **New point in the design space**
  - functional, modular, type-based translation
  - no annotations in source program
  - translation to existing proof frameworks for *extrinsic proofs* (now: F*, ongoing : Coq, HOL4…)
- **Aim for verification in the large**
  - no “tour de force” on ninja unsafe code
  - rather, properties about large applications written in safe Rust
- **A user-friendly semantics for borrows**
  - lazy semantics, semantical analysis
  - yields a functional, executable translation, hence a program proof pipeline

Questions ?

Separation Logic Interpretation

```
fn choose<'a>(b : bool, x : &'a mut i32, y : &'a mut i32) -> &'a mut i32;
```

```
forall b x y x_v y_v.

  { (x ~~> x_v) * (y ~~> y_v) }

  choose(b, x, y)

  { fun z ->
    // Returned value
    (z ~~> choose_fwd b x_v y_v) *

    // Magic wand
    (forall z_v.
     (z ~~> z_v) --*
      (let (x_v', y_v') = choose_back b x_v y_v z_v in
       (x ~~> x_v') * (y ~~> y_v')
    )) }
```

Pure Translation – Lists & Recursion

```rust
pub enum List<T> {
  Cons(T, Box<List<T>>),
  Nil,
}

fn nth<'a, T>(l: &'a mut List<T>, i: u32) -> &'a mut T {
  match l {
    List::Nil => { panic!() }
    List::Cons(x, tl) => {
      if i == 0 { return x; }
      else { return nth(tl, i - 1); } } } }
```

```ocaml
let rec nth_fwd (l: list u32) (i: u32) : result u32 =
  match l with
  | Nil -> Fail
  | Cons x tl ->
    if i = 0 then return x
    else nth_fwd tl (i-1)

let rec nth_back (l: list u32) (i: u32) (n_x: u32) =
  match l with
  | Nil -> Fail
  | Cons x tl ->
    // If we reach the ith element:
    // replace "x" with its new value
    if i = 0 then Success (Cons n_x tl)
    // Otherwise continue diving into the list
    else
      match nth_back tl (i-1) n_x with
      | Fail -> Fail
      | Success n_tl ->
        // Replace "tl" with its new value
        Success (Cons x n_tl)
```

Ongoing work: Loops / Translating blocks

There is a general problem of translating blocks of code in isolation:

```
// Rust
let x = 0;
if b { x = 1; }
else { x = 2; }
let y = x + 1;
...
```

```
// Current translation
let x = 0;
if b then
    let x = 1 in
    let y = x + 1 in
    ...
else
    let x = 2 in
    let y = x + 1 in
    ...
```

```
// Target translation
let x = 0;
let x = if b then 1 else 2 in
let y = x + 1 in
...
```

Idea: define a “join” operation on environments

Discussion – Nested Borrows

- The real difficulty is dealing with **nested lifetimes**

```
fn id_mut_mut(ppx : &'a mut &'b mut u32)
  -> &'a mut &'b mut u32 {
    x
}

let mut x = 0;
let mut px = &mut x;
let ppx = &mut px;
let mppx = &mut (*ppx);
let ppy = id_mut_mut(move mppx);
**ppy = 1; // Updates x
let mut y = 2;
*ppy = &mut y; // ← Now, (**ppx) == y
assert!(**ppx == 2); // This ends 'a
**ppx = 3; // ← We update y!
assert!(x == 1); // This ends 'b
assert!(y == 3);
```

```
// If we allow arbitrary type instantiations, we
// can turn any innocent looking function into
// something very complex:
fn id<T>(x : T) -> T { x }

// This is similar to what is on the left:
...
let mppx = &mut (*ppx);
let ppy = id<&mut &mut u32>(move mppx);
...
```

Discussion – Nested Borrows – Use Cases

- What are the use cases for nested lifetimes (real question)?
- The only use case we are aware of is degenerate:

```
// This function is very idiomatic
fn f<'b, 'a>(ctx : &'b mut Ctx<'a>, ...) {
    // However the following “borrow overwrites” seem non-idiomatic:
    ctx.x = &mut ...;
}

fn g<'b, 'a>(...) -> &'b mut Ctx<'a>;
// The following “borrow overwrites” seem even less idiomatic:
ctx.x = &mut ...;
```

- Choose a subset which forbids the difficult operations?
- We are interested in use cases for “borrow overwrites”!

Pure Translation – Proof of Concept (loops) (i)

```
fn list_iterator_has_next<'a, T>(l: &'a List<T>) -> bool {
    match l {
        List::Nil => {
            return false;
        }
        List::Cons(_, _) => {
            return true;
        }
    }
}

fn list_iterator_get_next<'a, T>(l: &'a mut List<T>) -
> (&'a mut T, &'a mut List<T>) {
    match l {
        List::Nil => {
            panic!();
        }
        List::Cons(x, tl) => {
            return (x, tl);
        }
    }
}
```

```
let list_iterator_has_next (#a : Type0) (l : list a) : bool =
  match l with
  | [] -> false
  | x :: l -> true

let list_iterator_get_next_fwd (#a : Type0) (l : list a) : result (a & list a) =
  match l with
  | [] -> Fail
  | x :: tl -> Success (x, tl)

let list_iterator_get_next_back
  (#a : Type0) (l : list a) // Old value of the list
  (n_x : a) (n_tl : list a) :
  // return: the reconstructed list
  list a =
  n_x :: n_tl
```

Pure Translation – Proof of Concept (loops) (ii)

```
fn list_iterator_test1(mut l: &mut List<u32>) {
    while list_iterator_has_next(l) {
        let (x, tl) = list_iterator_get_next(l); // Reborrow (i)
        *x = *x + 1;
        l = tl; // Reborrow (ii)
    }
}
```

```
let rec list_iterator_test1 (l0 : list u32) : result (list u32) =
  // > while list_iterator_has_next(l) {
  if list_iterator_has_next l0 then
    begin
    // > let (x, tl) = list_iterator_get_next(l);
    // This introduces a borrow on l, that we designate
    // as the borrow (i)
    tmp <-- list_iterator_get_next_fwd l0;
    let (x0, tl0) = tmp in

    x1 <-- check_add x0 1; // > *x = *x + 1;
    let l1 = tl0 in // > l = tl; // Borrow (ii)

    // Recursive call: end of the loop
    // We get the new value of l
    l2 <-- list_iterator_test1 l1;

    // Reborrows expire: propagate the values back
    let tl1 = l2 in // End borrow (ii)
    let l3 = list_iterator_get_next_back l0 x1 tl1 in // End borrow (i)
    return l3
    end
  else
    return l0
```