AENEAS

from Rust Programs
to Pure Lambda Calculus

Son Ho (Inria),
Jonathan Protzenko (MSR)

Rust: low-level, memory safe language

In C:

```
uint32_t *choose(bool b, int32_t *x, int32_t *y)
{
    if b { return x; }
    else { return y; }
}
```

```
int32_t x = 0;
int32_t y = 1;
int32_t *z = choose(true, &x, &y);

*z = 2; // Updates x

// Observe the changes
assert(x == 2);
assert(y == 1);
```

Rust: low-level, memory safe language

Null? Dangling? Aliased?

In C:

```
uint32_t *choose(bool b, int32_t *x, int32_t *y)
{
    if b { return x; }
    else { return y; }
}
```

```
int32_t x = 0;
int32_t y = 1;
int32_t *z = choose(true, &x, &y);

*z = 2; // Updates x

// Observe the changes
assert(x == 2);
assert(y == 1);
```

Rust: low-level, memory safe language

Null? Dangling? Aliased?

In C:

```
uint32_t *choose(bool b, int32_t *x, int32_t *y)
{
    if b { return x; }
    else { return y; }
}
```

```
int32_t x = 0;
int32_t y = 1;
int32_t *z = choose(true, &x, &y);

*z = 2; // Updates x

// Observe the changes
assert(x == 2);
assert(y == 1);
```

Aliasing?

Rust: low-level, memory safe language

Null? Dangling? Aliased?

In C:

```c
uint32_t *choose(bool b, int32_t *x, int32_t *y)
{
    if b { return x; }
    else { return y; }
}
```

```c
int32_t x = 0;
int32_t y = 1;
int32_t *z = choose(true, &x, &y);

*z = 2; // Updates x

// Observe the changes
assert(x == 2);
assert(y == 1);
```

Aliasing?

In Rust:

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

```rust
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Updates x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
```

Rust: low-level, memory safe language

Null? Dangling? Aliased?

In C:

```c
uint32_t *choose(bool b, int32_t *x, int32_t *y)
{
    if b { return x; }
    else { return y; }
}
```

```c
int32_t x = 0;
int32_t y = 1;
int32_t *z = choose(true, &x, &y);

*z = 2; // Updates x

// Observe the changes
assert(x == 2);
assert(y == 1);
```

Aliasing?

Exclusive access

In Rust:

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

```rust
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Updates x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
```

Rust: low-level, memory safe language

Null? Dangling? Aliased?

In C:

```c
uint32_t *choose(bool b, int32_t *x, int32_t *y)
{
    if b { return x; }
    else { return y; }
}
```

```c
int32_t x = 0;
int32_t y = 1;
int32_t *z = choose(true, &x, &y);

*z = 2; // Updates x

// Observe the changes
assert(x == 2);
assert(y == 1);
```

Aliasing?

Lifetime ‘a

Exclusive access

In Rust:

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

```rust
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Updates x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
```

Rust: low-level, memory safe language

Null? Dangling? Aliased?

In C:

```
uint32_t *choose(bool b, int32_t *x, int32_t *y)
{
    if b { return x; }
    else { return y; }
}
```

```
int32_t x = 0;
int32_t y = 1;
int32_t *z = choose(true, &x, &y);

*z = 2; // Updates x

// Observe the changes
assert(x == 2);
assert(y == 1);
```

Aliasing?

Lifetime ‘a

Exclusive access

In Rust:

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

```
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Updates x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
```

‘a ends here (borrow checker)

Rust: low-level, memory safe language

In C:

Null? Dangling? Aliased?

```c
uint32_t *choose(bool b, int32_t *x, int32_t *y)
{
    if b { return x; }
    else { return y; }
}
```

```c
int32_t x = 0;
int32_t y = 1;
int32_t *z = choose(true, &x, &y);

*z = 2; // Updates x

// Observe the changes
assert(x == 2);
assert(y == 1);
```

Aliasing?

In Rust:

Lifetime ‘a

Exclusive access

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

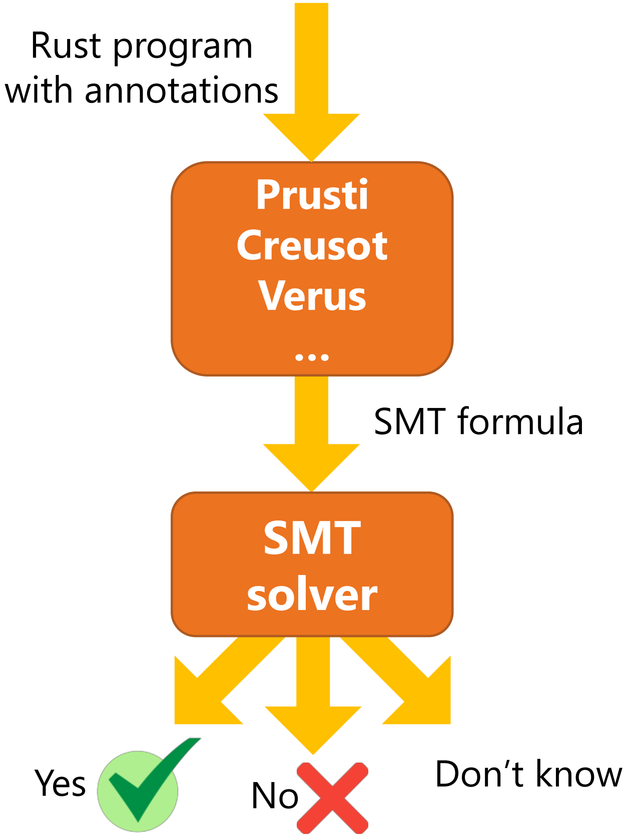
```rust
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Updates x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
```

‘a ends here (borrow checker)

Leverage Rust’s type system to ease verification (system programming, etc.)?

Leveraging Safe Rust in Verification

Explored design space:



Rust program with annotations

Prusti
Creusot
Verus
…

SMT formula

SMT solver

Yes ✓

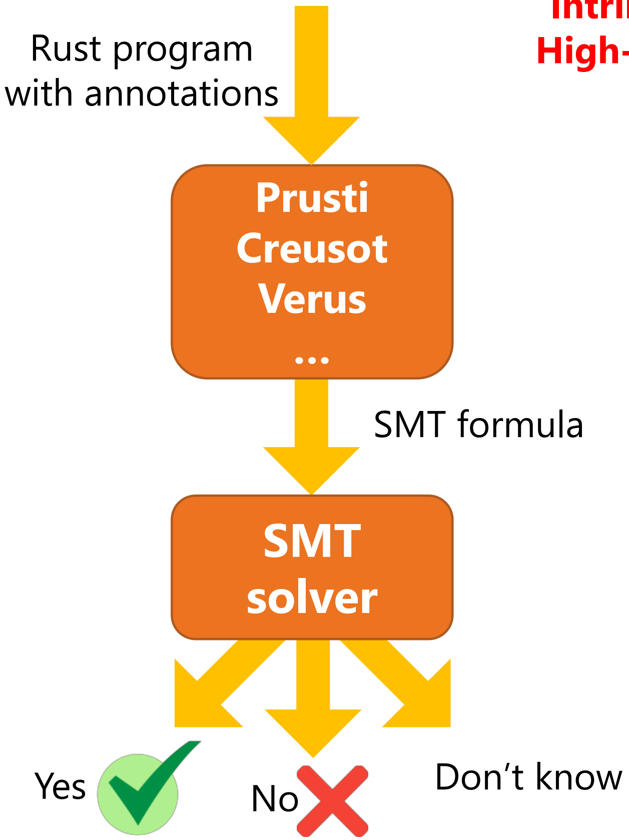
No ✗

Don’t know

Leveraging Safe Rust in Verification

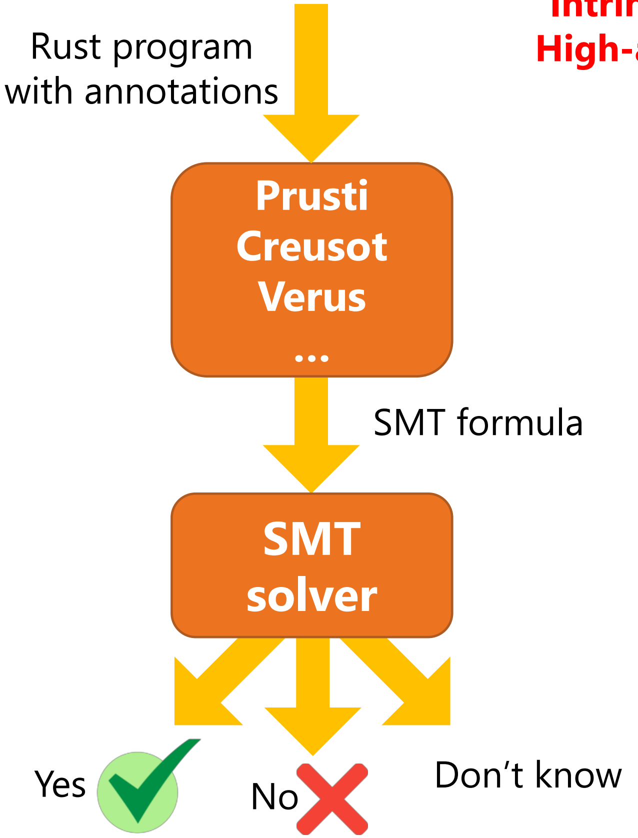
Explored design space:

Intrinsic proofs, High-automation



Leveraging Safe Rust in Verification

Explored design space:



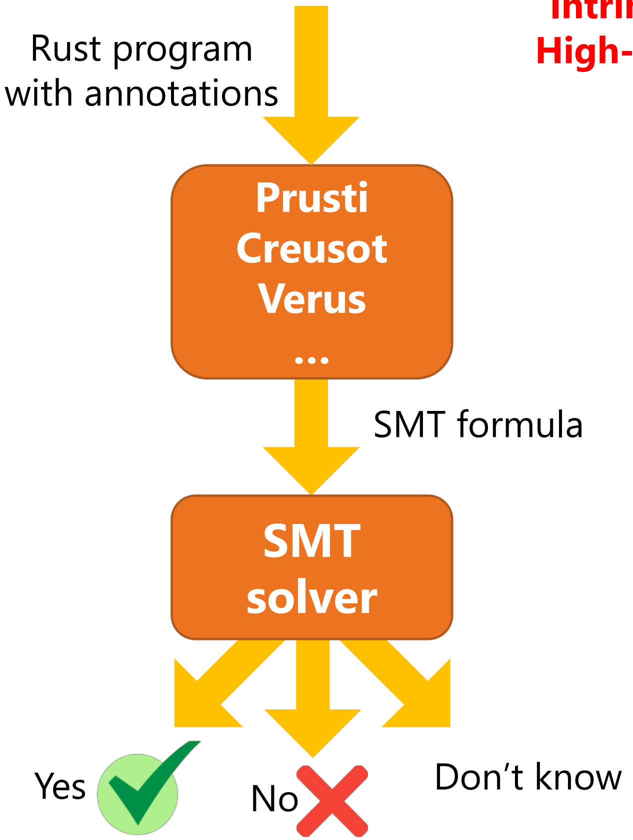
Intrinsic proofs,
High-automation

Extrinsic proofs,
Interaction with
tactics

Leveraging Safe Rust in Verification

Explored design space:

Intrinsic proofs, High-automation



Rust program with annotations

Prusti
Creusot
Verus
…

SMT formula

SMT solver

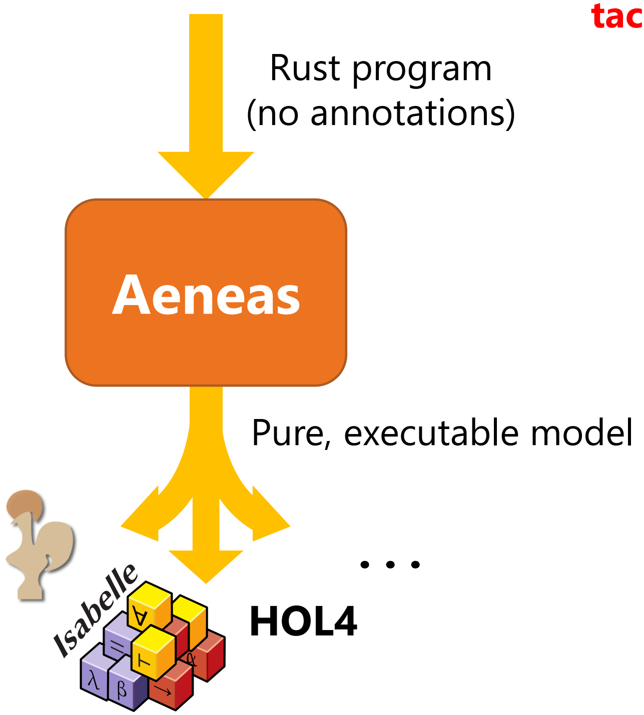
Yes ✓

No ✗

Don't know

Our work:

Extrinsic proofs, Interaction with tactics



Rust program (no annotations)

Aeneas

Pure, executable model

Isabelle

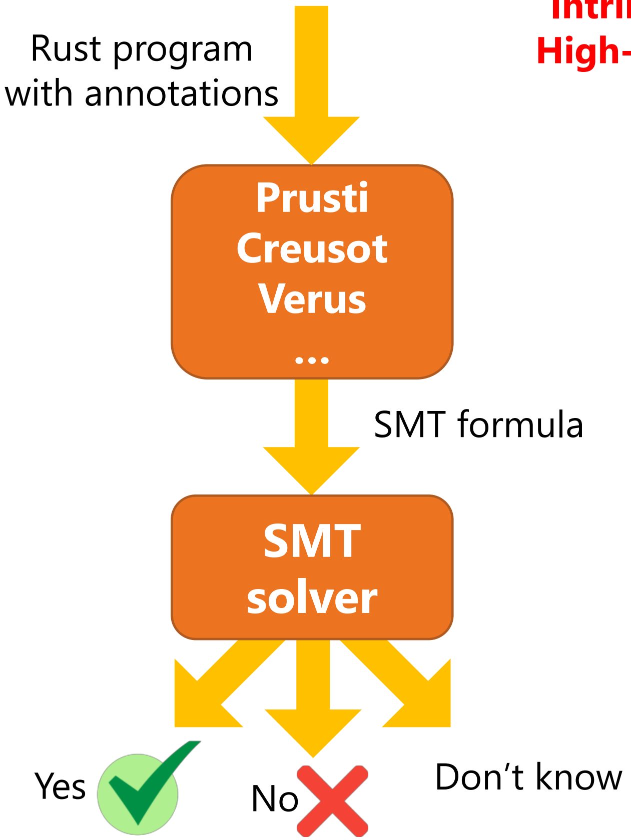
HOL4

…

Leveraging Safe Rust in Verification

Explored design space:

Intrinsic proofs, High-automation



Rust program with annotations

Prusti
Creusot
Verus
…

SMT formula

SMT solver

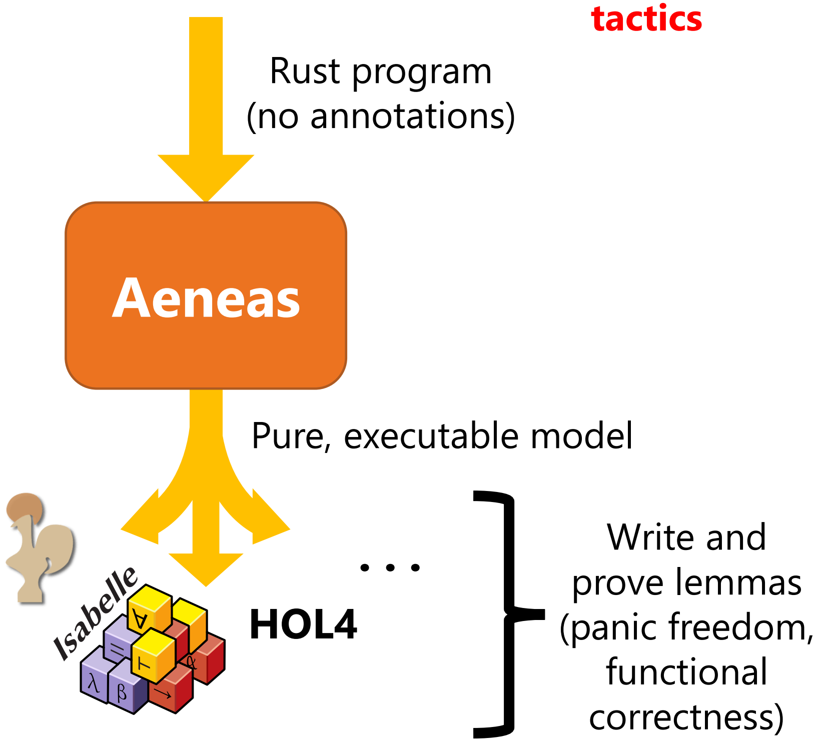
Yes ✓

No ✗

Don't know

Our work:

Extrinsic proofs, Interaction with tactics



Rust program (no annotations)

Aeneas

Pure, executable model

HOL4

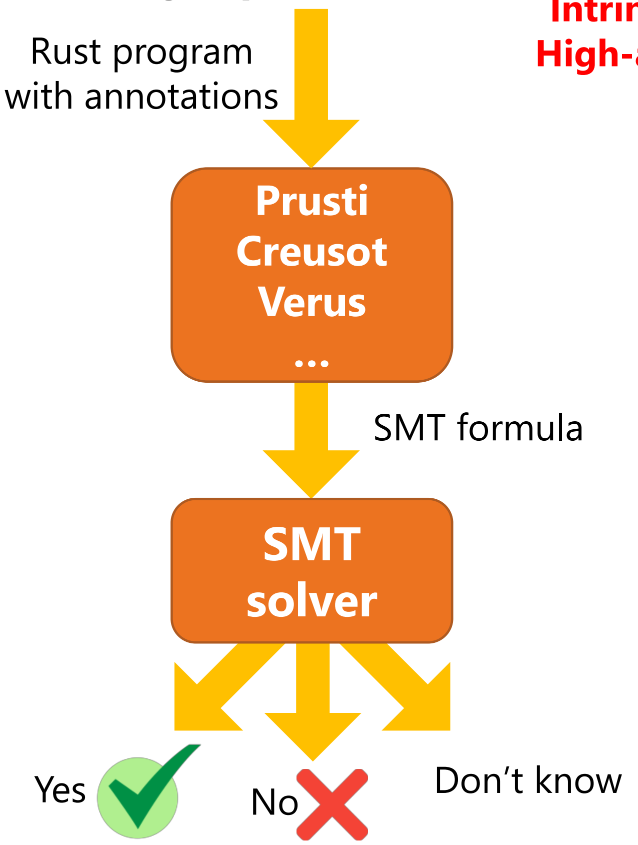
…

Write and prove lemmas (panic freedom, functional correctness)

Leveraging Safe Rust in Verification

Explored design space:

Intrinsic proofs, High-automation



Rust program with annotations

Prusti Creusot Verus …

SMT formula

SMT solver

Yes ✓

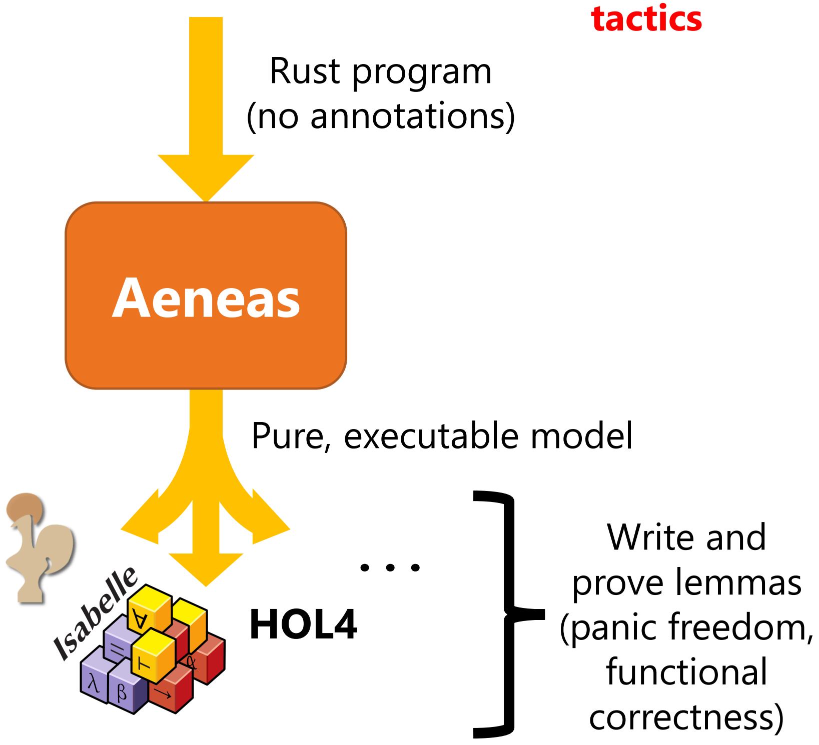
No ✗

Don't know

Our work:

(similar to **Electrolysis**)

Extrinsic proofs, Interaction with tactics



Rust program (no annotations)

Aeneas

Pure, executable model

HOL4

…

Write and prove lemmas (panic freedom, functional correctness)

Translating safe Rust to pure

Rust:

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

```
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Update x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
...
```

Translating safe Rust to pure

Rust:

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

```
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Update x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
...
```

Translation:

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

```
let x = 0 in
let y = 1 in
let z = choose true x y in
...
```

Translating safe Rust to pure

Rust:

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

```
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Update x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
...
```

Translation:

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

```
let x = 0 in
let y = 1 in
let z = choose true x y in

let z = 2 in

...
```

Translating safe Rust to pure

Rust:

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

```
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Update x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
...
```

‘a ends here

Translation:

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

```
let x = 0 in
let y = 1 in
let z = choose true x y in

let z = 2 in

...
```

?

Translating safe Rust to pure

Rust:

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

```
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Update x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
...
```

Translation:

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

```
let x = 0 in
let y = 1 in
let z = choose true x y in

let z = 2 in

let (x, y) = ?? in
...
```

Translating safe Rust to pure

Rust:

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

```
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Update x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
...
```

Translation:

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

```
let x = 0 in
let y = 1 in
let z = choose true x y in

let z = 2 in

let (x, y) = (z, y) in
...
```

Translating safe Rust to pure

Rust:

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

```
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Update x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
...
```

Translation:

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

```
let x = 0 in
let y = 1 in
let z = choose true x y in

let z = 2 in

let (x, y) = if true then (z, y) else (x, z) in
...
```

Translating safe Rust to pure

Rust:

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

```
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Update x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
...
```

Translation:

```
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)
```

```
let x = 0 in
let y = 1 in
let z = choose_fwd true x y in

let z = 2 in

let (x, y) = choose_back true x y z in
...
```

Translating safe Rust to pure

Rust:

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

```
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Update x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
...
```

Translation:

```
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)
```

```
let x = 0 in
let y = 1 in
let z = choose_fwd true x y in

let z = 2 in

let (x, y) = choose_back true x y z in
...
```

Modular translation with *forward* and *backward* functions

Translating safe Rust to pure

Rust:

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

Types derived from Rust signature *only*

```
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Update x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
...
```

Translation:

```
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)
```

```
let x = 0 in
let y = 1 in
let z = choose_fwd true x y in

let z = 2 in

let (x, y) = choose_back true x y z in
...
```

Modular translation with *forward* and *backward* functions

Recursive Functions

Rust:

```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::Cons(x, tl) => {
      if i == 0 {
        return x;
      }
      else {
        return nth(tl, i - 1);
      }
    }
    List::Nil => { panic!() }
  }
}
```

Recursive Functions

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::Cons(x, tl) => {
      if i == 0 {
        return x;
      }
      else {
        return nth(tl, i - 1);
      }
    }
    List::Nil => { panic!() }
  }
}
```

Translation:

```
let rec nth_fwd (t : Type) (l : list_t t) (i : u32) : result t =
  begin match l with
  | ListCons x tl ->
    if i = 0
    then Return x
    else begin i0 <-- u32_sub i 1; nth_fwd t tl i0 end
  | ListNil -> Fail Failure
  end
```

Recursive Functions

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::Cons(x, tl) => {
      if i == 0 {
        return x;
      }
      else {
        return nth(tl, i - 1);
      }
    }
    List::Nil => { panic!() }
  }
}
```

Translation:

```
let rec nth_fwd (t : Type) (l : list_t t) (i : u32) : result t =
  begin match l with
  | ListCons x tl ->
    if i = 0
    then Return x
    else begin i0 <-- u32_sub i 1; nth_fwd t tl i0 end
  | ListNil -> Fail Failure
  end
```

```
let rec nth_back (t : Type) (l : list_t t) (i : u32) (ret : t) :
  result (list_t t) =
  begin match l with
  | ListCons x tl ->
    if i = 0
    then Return (ListCons ret tl)
    else begin
      i0 <-- u32_sub i 1;
      tl0 <-- nth_back t tl i0 ret;
      Return (ListCons x tl0) end
  | ListNil -> Fail Failure
  end
```

Recursive Functions

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::Cons(x, tl) => {
      if i == 0 {
        return x;
      }
      else {
        return nth(tl, i - 1);
      }
    }
    List::Nil => { panic!() }
  }
}
```

Translation:

```
let rec nth_fwd (t : Type) (l : list_t t) (i : u32) : result t =
  begin match l with
  | ListCons x tl ->
    if i = 0
    then Return x
    else begin i0 <-- u32_sub i 1; nth_fwd t tl i0 end
  | ListNil -> Fail Failure
  end
```

```
let rec nth_back (t : Type) (l : list_t t) (i : u32) (ret : t) :
  result (list_t t) =
  begin match l with
  | ListCons x tl ->
    if i = 0
    then Return (ListCons ret tl)
    else begin
      i0 <-- u32_sub i 1;
      tl0 <-- nth_back t tl i0 ret;
      Return (ListCons x tl0) end
  | ListNil -> Fail Failure
  end
```

Forward and backward functions behave like **lenses**

Opaque (External) Functions

Rust (external dependency):

```
struct S { x: i32 /* private field */ }
fn create(x: i32) -> S;
fn get_field<'a>(s : &'a mut S) -> &'a mut i32;
```

Rust (local crate):

```
fn f() {
  let mut s = create(0);
  let x = get_field(&mut s);
  *x += 1;
}
```

Opaque (External) Functions

Rust (external dependency):

```
struct S { x: i32 /* private field */ }
fn create(x: i32) -> S;
fn get_field<'a>(s : &'a mut S) -> &'a mut i32;
```

Rust (local crate):

```
fn f() {
  let mut s = create(0);
  let x = get_field(&mut s);
  *x += 1;
}
```

Translation (in an interface file):

```
type S
val create_fwd : i32 -> result S
val get_field_fwd : S -> result i32
val get_field_back : S -> i32 -> result S
```

Translation (in an implementation file):

```
let f_fwd =
  s <-- create_fwd 0;
  x <-- get_field_fwd s;
  x0 <-- i32_add x 1;
  s <-- get_field_back s x0;
  Return ()
```

Opaque (External) Functions

Rust (external dependency):

```
struct S { x: i32 /* private field */ }
fn create(x: i32) -> S;
fn get_field<'a>(s : &'a mut S) -> &'a mut i32;
```

Rust (local crate):

```
fn f() {
  let mut s = create(0);
  let x = get_field(&mut s);
  *x += 1;
}
```

Translation (in an interface file):

```
type S
val create_fwd : i32 -> result S
val get_field_fwd : S -> result i32
val get_field_back : S -> i32 -> result S
```

Translation (in an implementation file):

```
let f_fwd =
  s <-- create_fwd 0;
  x <-- get_field_fwd s;
  x0 <-- i32_add x 1;
  s <-- get_field_back s x0;
  Return ()
```

Rust signatures efficiently capture the **effectful behavior**

Loops

Rust:

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

pub fn nth<T>(mut ls: &mut List<T>, mut i: u32)
  -> &mut T {
  loop {
    match ls {
      List::Cons(x, tl) => {
        if i == 0 { return x; }
        else {
          ls = tl;
          i -= 1;
          continue;
        }
      }
      List::Nil => { panic!() }
    }
  }
}
```

Loops

Rust:

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

pub fn nth<T>(mut ls: &mut List<T>, mut i: u32)
  -> &mut T {
  loop {
    match ls {
      List::Cons(x, tl) => {
        if i == 0 { return x; }
        else {
          ls = tl;
          i -= 1;
          continue;
        }
      }
      List::Nil => { panic!() }
    }
  }
}
```

Translation:

```
let rec nth_loop_fwd
  (t : Type) (ls : list_t t) (i : u32) : result t =
  begin match ls with
  | ListCons x tl ->
    if i = 0 then Return x
    else begin i0 <-- u32_sub i 1; nth_loop_fwd t tl i0 end
  | ListNil -> Fail Failure
  end

let nth_fwd t ls i = nth_loop_fwd t ls i
```

Loops

Rust:

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

pub fn nth<T>(mut ls: &mut List<T>, mut i: u32)
  -> &mut T {
  loop {
    match ls {
      List::Cons(x, tl) => {
        if i == 0 { return x; }
        else {
          ls = tl;
          i -= 1;
          continue;
        }
      }
      List::Nil => { panic!() }
    }
  }
}
```

Translation:

```
let rec nth_loop_fwd
  (t : Type) (ls : list_t t) (i : u32) : result t =
  begin match ls with
  | ListCons x tl ->
    if i = 0 then Return x
    else begin i0 <-- u32_sub i 1; nth_loop_fwd t tl i0 end
  | ListNil -> Fail Failure
  end

let nth_fwd t ls i = nth_loop_fwd t ls i
```

```
let rec nth_loop_back
  (t : Type) (ls : list_t t) (i : u32) (ret : t) :
  result (list_t t) =
  begin match ls with
  | ListCons x tl ->
    if i = 0 then Return (ListCons ret tl)
    else begin
      i0 <-- u32_sub i 1;
      tl0 <-- nth_loop_back t tl i0 ret;
      Return (ListCons x tl0) end
  | ListNil -> Fail Failure
  end

let nth_back t ls i ret = nth_loop_back t ls i ret
```

Loops

Rust:

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

pub fn nth<T>(mut ls: &mut List<T>, mut i: u32)
  -> &mut T {
  loop {
    match ls {
      List::Cons(x, tl) => {
        if i == 0 { return x; }
        else {
          ls = tl;
          i -= 1;
          continue;
        }
      }
      List::Nil => { panic!() }
    }
  }
}
```

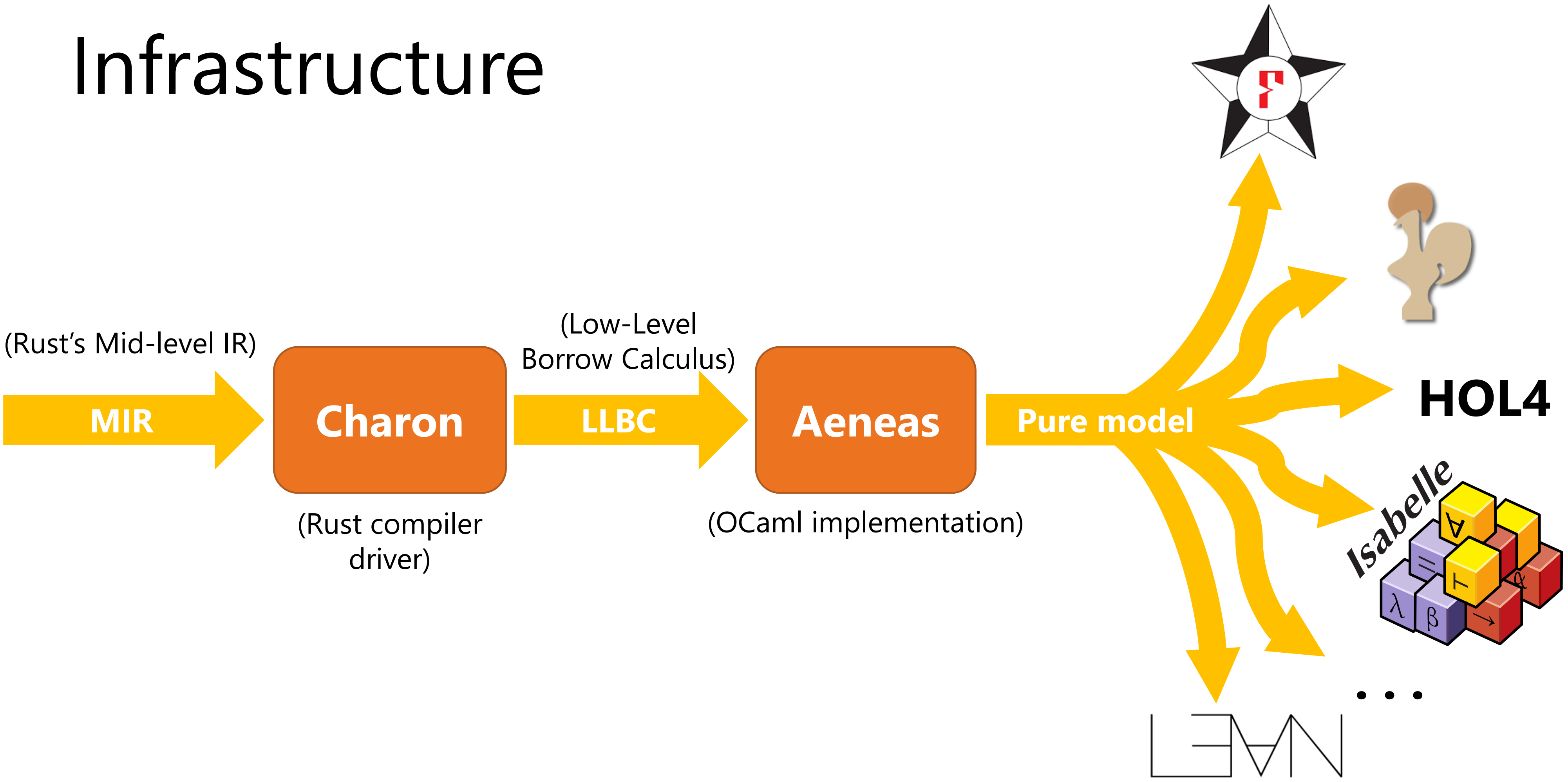
Translated functions are
similar to the recursive case

Translation:

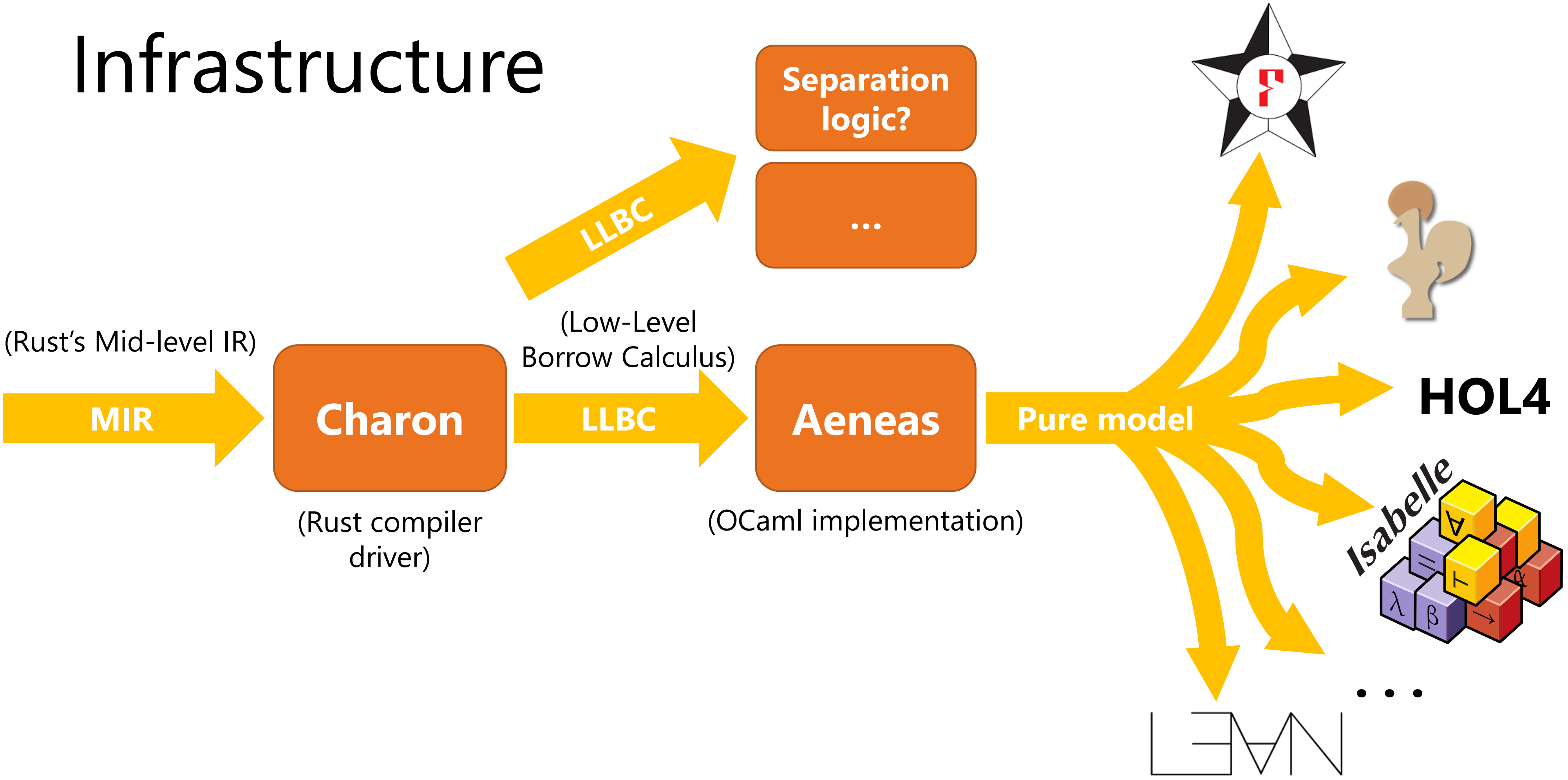
```
let rec nth_loop_fwd
  (t : Type) (ls : list_t t) (i : u32) : result t =
  begin match ls with
  | ListCons x tl ->
    if i = 0 then Return x
    else begin i0 <-- u32_sub i 1; nth_loop_fwd t tl i0 end
  | ListNil -> Fail Failure
  end

let nth_fwd t ls i = nth_loop_fwd t ls i
```

```
let rec nth_loop_back
  (t : Type) (ls : list_t t) (i : u32) (ret : t) :
  result (list_t t) =
  begin match ls with
  | ListCons x tl ->
    if i = 0 then Return (ListCons ret tl)
    else begin
      i0 <-- u32_sub i 1;
      tl0 <-- nth_loop_back t tl i0 ret;
      Return (ListCons x tl0) end
  | ListNil -> Fail Failure
  end

let nth_back t ls i ret = nth_loop_back t ls i ret
```

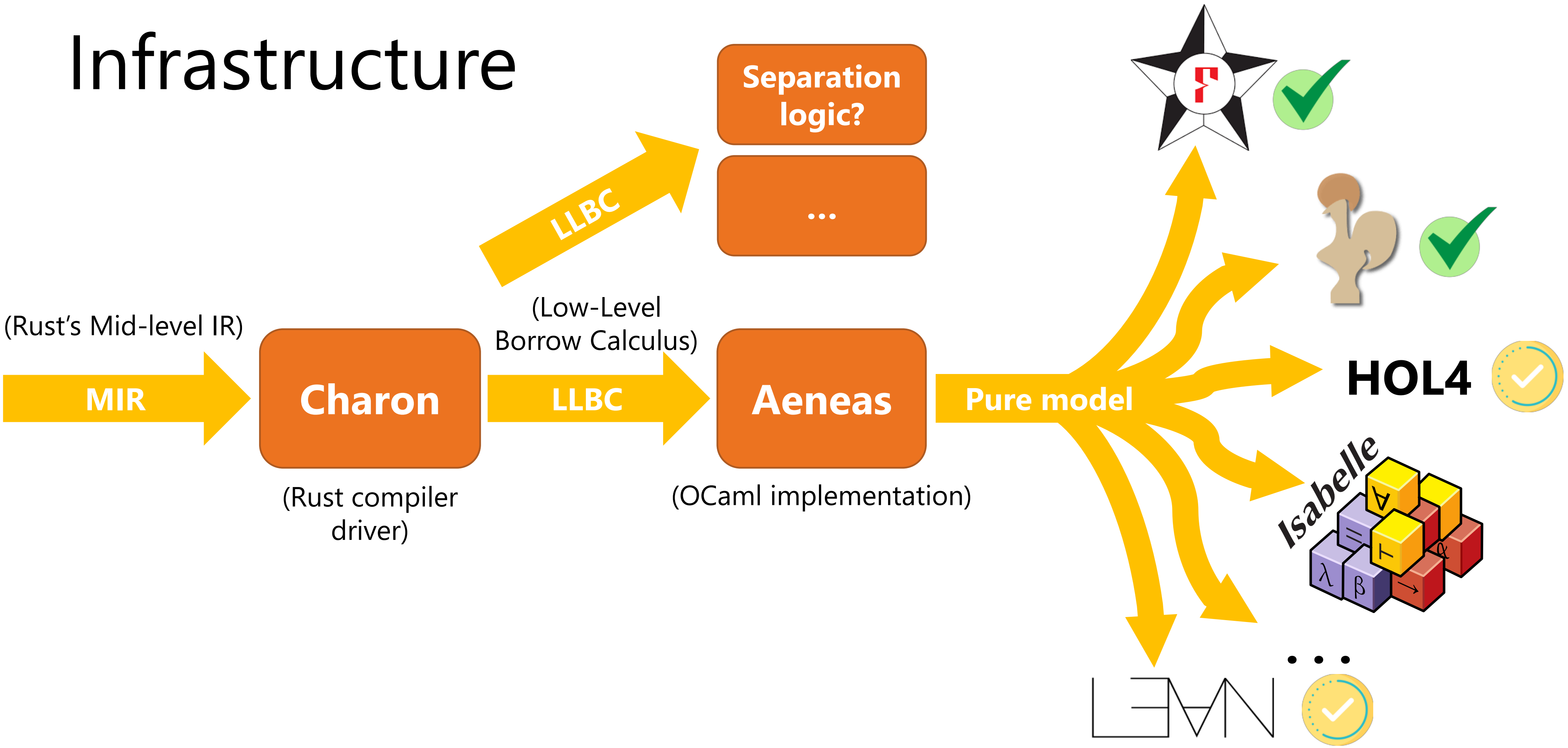
Infrastructure



Infrastructure



Infrastructure



(Rust's Mid-level IR)

MIR

Charon

(Rust compiler driver)

(Low-Level Borrow Calculus)

LLBC

LLBC

Separation logic?

...

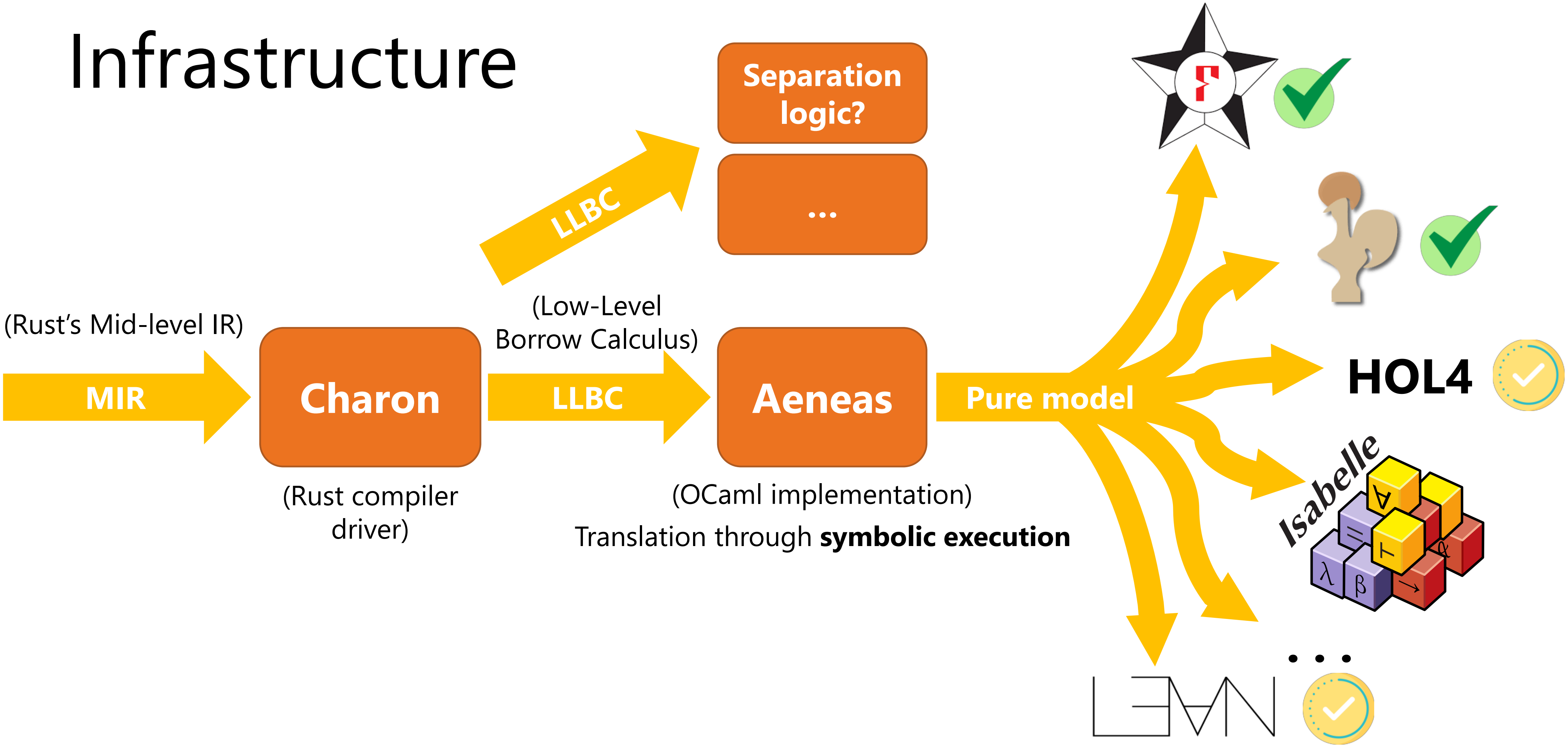
Aeneas

(OCaml implementation)

Pure model

HOL4

Infrastructure



(Rust's Mid-level IR)

MIR

Charon

(Rust compiler driver)

(Low-Level Borrow Calculus)

LLBC

LLBC

Separation logic?

…

Aeneas

(OCaml implementation)

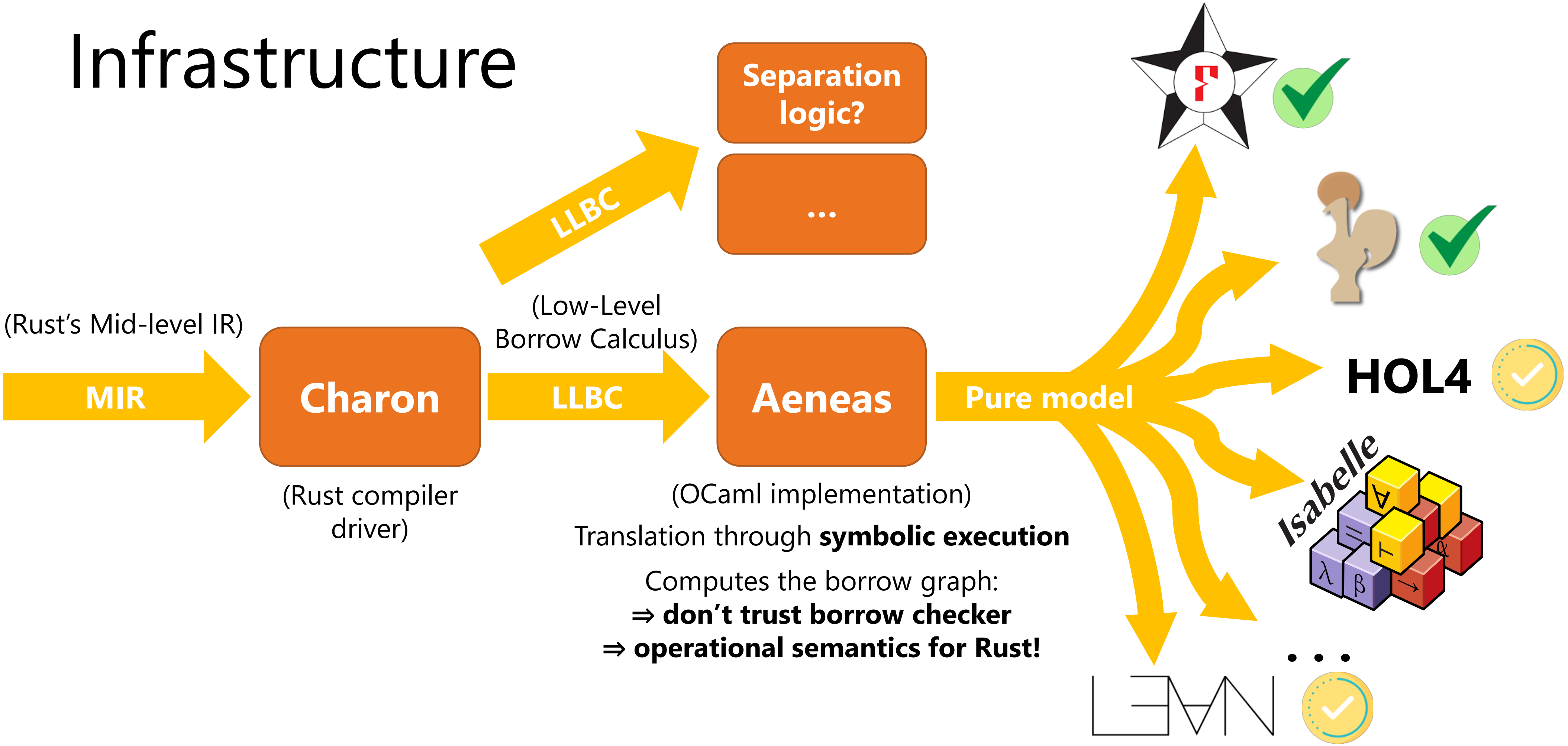
Translation through **symbolic execution**

Pure model

HOL4

…

Infrastructure



(Rust's Mid-level IR)

MIR

Charon

(Rust compiler driver)

(Low-Level Borrow Calculus)

LLBC

LLBC

Separation logic?

...

Aeneas

(OCaml implementation)

Translation through **symbolic execution**

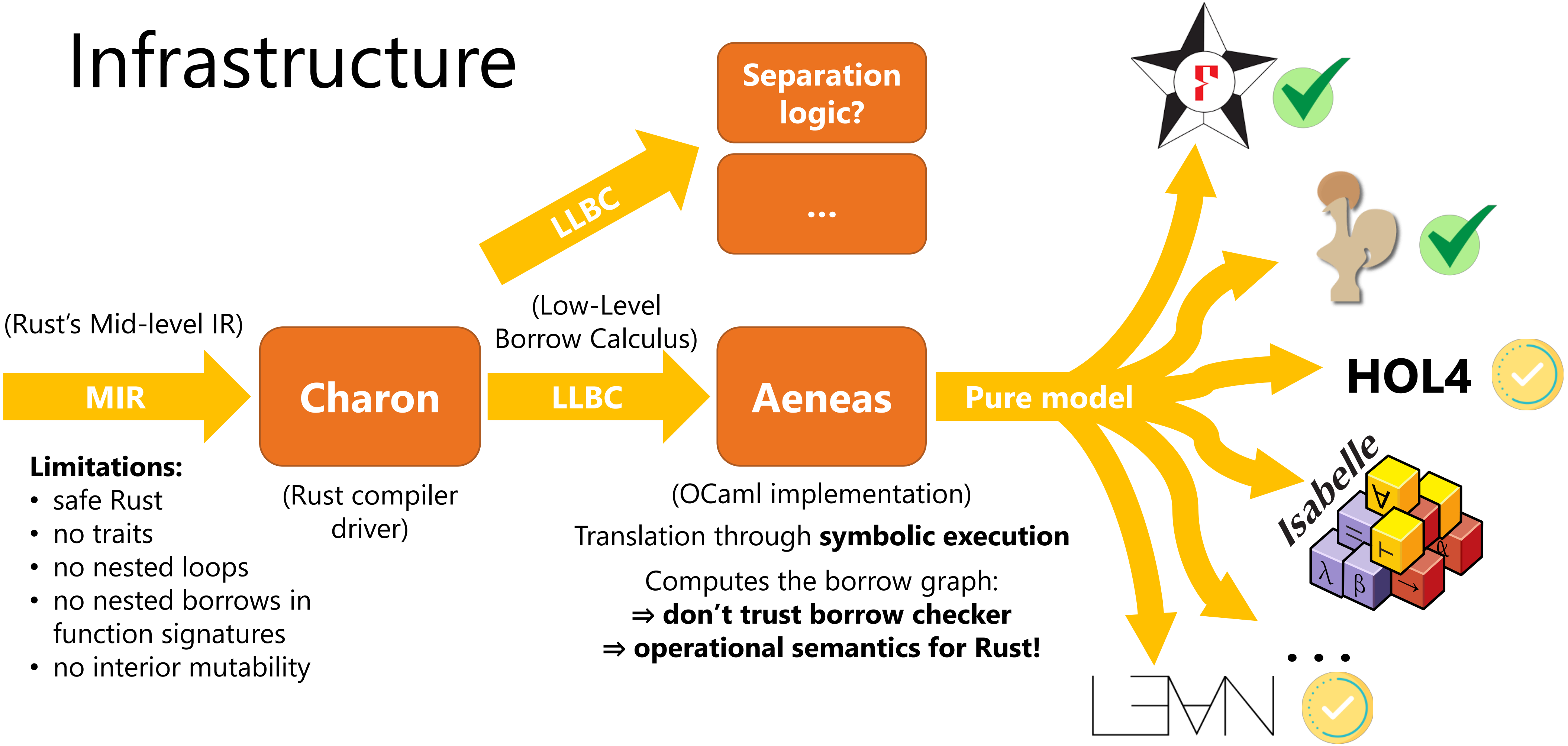
Computes the borrow graph:

⇒ **don't trust borrow checker**

⇒ **operational semantics for Rust!**

Pure model

Infrastructure



(Rust's Mid-level IR)

MIR

Charon

(Rust compiler driver)

Limitations:

- safe Rust
- no traits
- no nested loops
- no nested borrows in function signatures
- no interior mutability

LLBC

(Low-Level Borrow Calculus)

LLBC

Separation logic?

...

Aeneas

(OCaml implementation)

Translation through **symbolic execution**

Computes the borrow graph:
⇒ **don't trust borrow checker**
⇒ **operational semantics for Rust!**

Pure model

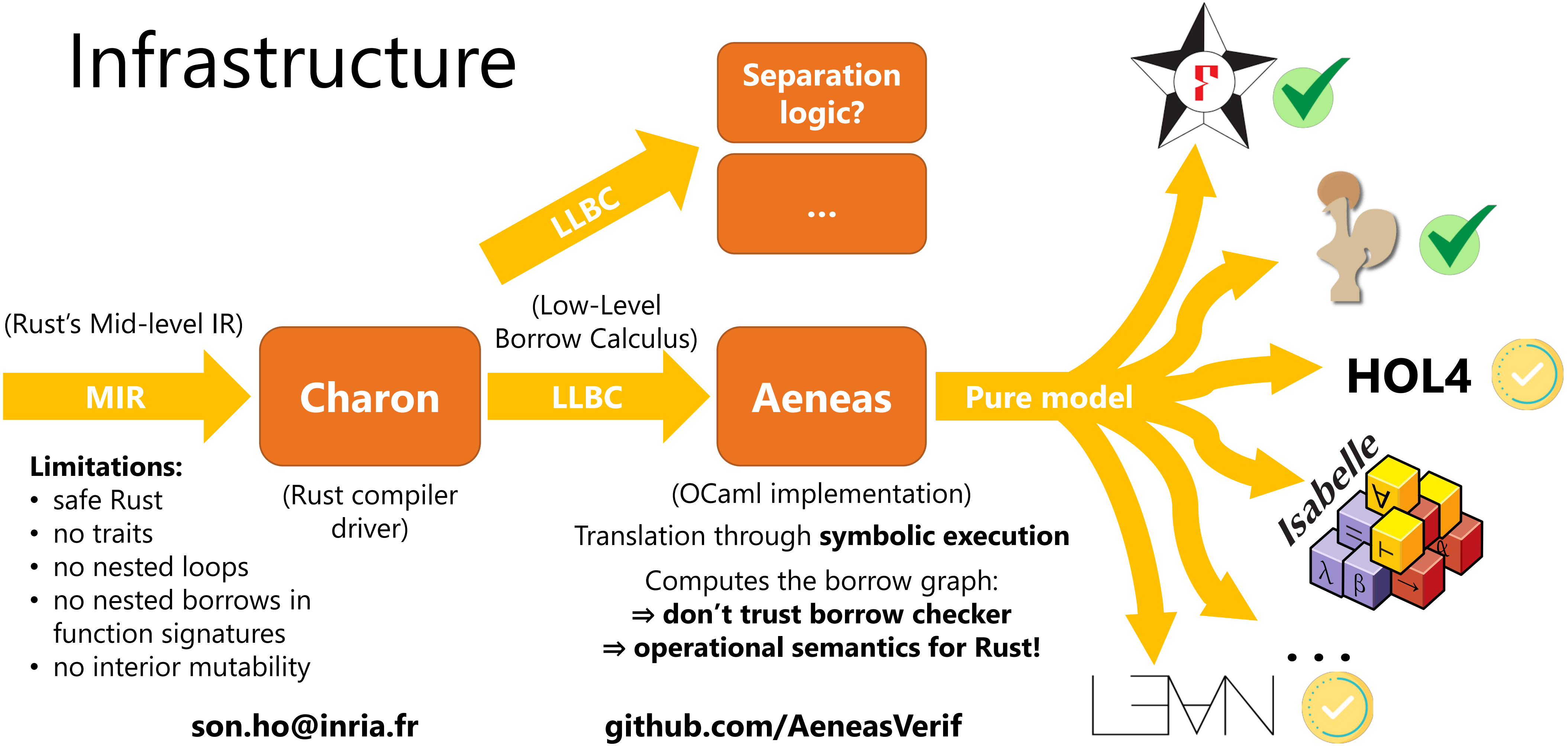
HOL4

Isabelle

LEAN

...

Infrastructure



Limitations:

- safe Rust
- no traits
- no nested loops
- no nested borrows in function signatures
- no interior mutability

Translation through **symbolic execution**

Computes the borrow graph:
⇒ **don't trust borrow checker**
⇒ **operational semantics for Rust!**

son.ho@inria.fr

github.com/AeneasVerif

Aeneas: Rust Verification by Functional Translation, ICFP 2022