
# Functional translation of Rust programs

## Internship report

Sidney Congard

M2 - Logique Mathématique et Fondements de l'Informatique

Supervised by Bhargavan Khartikeyan at Prosecco

September 2022

# Contents



# 1 Context

## 1.1 Rust

Rust is a programming language started in 2006 by Graydon Hoare, exasperated by numerous crashes from his elevator firmware. Its first version has been published in 2014 and it has since become increasingly popular. Rust enables system programming by giving a great control over performances and memory management while having high-level features (ML-style type inference, typeclasses, ...) and a type system that rules out undefined behaviors, such a memory errors (use-after-free, access to initialized memory, ...) or data races.

Those errors are ruled out by the Rust borrow checker, a part of its type system which is based on the exclusion between aliasing and mutability: a value can be either borrowed by potentially multiple shared references (written `&T`) or by a single mutable reference (written `&mut T`), but not both at the same time.

```
let mut x = 2;
let p = &mut x;
// Modify x through p.
*p += 3;
// To access x, p is first dropped here:
// Two simultaneous mutable accesses to x is forbidden.
assert!(x == 5);
```

To track which values are borrowed by a given reference, each reference is associated to a lifetime which may become explicit in type definitions and function signatures. A lifetime is a name for a region (a set of data) of the program.

```
// Returns one of the given references.
fn choose<'a, T>(e: bool, x: &'a mut T, y: &'a mut T)
    -> &'a mut T
{
    if e { x } else { y }
}
```

For example, this function manipulate mutable references with an explicit lifetime `'a`. Because its output may alias either the value borrowed by `x` or the one borrowed by `y`, neither `x` nor `y` values must be accessible as long as the output is alive. Those dependencies are expressed by lifetimes: all outputs associated to a given lifetime must be dropped to be able to access the inputs underlying value.

```
let mut x = 2;
let mut y = 6;
let p = choose(true, &mut x, &mut y);
*p += 3;
// To access x, p is first dropped (and give also back y).
assert!(x == 5);
```

References can also be split by destructuring records or enumerations: we can go from `&mut (A, B)` to `(&mut A, &mut B)` but not the converse.

Moreover, a value under a reference can be re-borrowed: this is useful because a mutable reference cannot be copied, but the same value can be borrowed multiple times (if we get it back before each new borrow).

```
fn inc(x: &mut u32) {
    *x += 1;
}

let mut x = 2;
let mut y = 6;
let p = choose(true, &mut x, &mut y);
inc(p); // Compiled as "f(&mut *p)": we pass a new reference...
inc(p); // ...So p is again available after the call.
inc(p);
assert!(x == 5);
```

Then, unsafe code can be written to have more power, notably to allow us to manipulate raw pointers and union members. This is useful to locally bypass the borrow checker when it's too restrictive or to interface the program with foreign functions. The RustBelt [Col18] project proved the guarantees of the type system and formalized conditions under which unsafe code maintain the type system guarantees.

# 2 The Aeneas toolchain

The expressivity, control and guarantees offered by Rust are great, but we sometimes want to prove more properties about our programs. That's where verification toolchains come into play. However, proving properties about low-level programs (in, e.g., C) has historically been quite tedious, notably due to a lot of mundane, memory-related obligations popping up.

To avoid that, Aeneas [HP22] leverages the type system of Rust to express a program written in safe Rust into a pure, high-level representation where those obligations won't appear: this allows to bypass more heavyweight representations such as separation logic.

So, Aeneas is a toolchain published in June 2022 which translates a subset of safe Rust programs into equivalent functional programs in F$^\star$, a proof

assistant. It exploits Rust type system to produce a lightweight translation which is free of concerns about memory. For example, the code below is translated into the following functions:

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

fn call_choose() {
    let mut x = 2;
    let mut y = 6;
    let p = choose(true, &mut x, &mut y);
    *p += 3;
    assert!(x == 5);
}
```

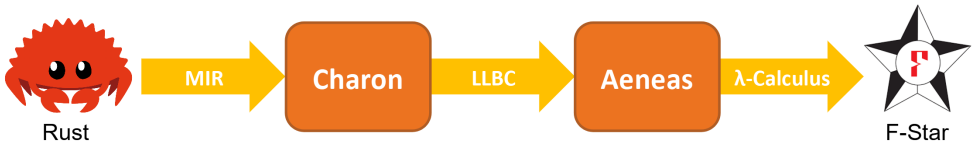
```
let choose_fwd (t : Type) (b : bool) (x : t) (y : t)
        : result t =
    if b then Return x else Return y

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

let call_choose_fwd : result unit =
    i <-- choose_fwd i32 true 2 6; (* monadic let *)
    z <-- i32_add i 3;
    (x0, y0) <-- choose_back i32 true 2 6 z;
    massert (x0 = 5); (* monadic assert *)
    Return ()
```

We see that the `choose` function is translated into two functions: the first one does the job of the choose function, while the second one retrieve the input updated values once its return value is dropped. So, for each lifetime

parameter, we have one corresponding backward function which tells us: "give me the outputs with this lifetime, I'll give you back the inputs with the same lifetime".

To do that, Rust programs are first translated into LLBC programs ("Low-Level Borrow Calculus", a new formalism) with Charon, a dedicated plugin for the Rust compiler of 10kLOC. Then, an OCaml program of 14kLOC which is also named Aeneas translates an LLBC program to lambda calculus then embeds it in a functional programming language.

Those steps allow for a modular approach that could be leveraged later by translating other low-level languages to LLBC, by performing other analysis on LLBC or targeting another language than F* for the functional translation.




## 2.1 LLBC

Syntactically, an LLBC program is a form of simplified Rust. Notably,

- It desugarizes the re-borrows: `inc(p)` becomes `inc(&mut *p)`.
- Expressions are limited: `g(f())` becomes `let x = f(); g()`.
- It does not support traits (Rust typeclasses).
- It explicits drops, copies and other implied operations.
- The set of variables stays the same in a function, but they can be inaccessible (written $\bot$).

The important part of LLBC lies in its operational semantics, which we see by examples below. The environment is written in comments, and omitted variables are set to $\bot$. First, here are mutable borrows:

```
let mut x = (1, 2);
// x ↦ (1, 2)

let mut y = &mut x;
// x ↦ loan^m l_0
// y ↦ borrow^m l_0 (1, 2)

let z = &mut (*y).0;
// x ↦ loan^m l_0
// y ↦ borrow^m l_0 (loan^m l_1, 2)
// z ↦ borrow^m l_1 1

// Read x ⇝ retrieve borrow^m l_0 ⇝ retrieve borrow^m l_1 :
assert!(x == (1, 2));
// x ↦ (1, 2)
```

So when a mutable reference is created, the borrowed value is moved in a new borrow, and a corresponding loan is made at its origin. When a loan is accessed, its corresponding borrow is dropped and its value is retrieved. As seen above, multiple borrows can be dropped at once.

In contrast, shared borrows don't take the value with them. Most of this report will deal with mutable references only.

```
let x = (1, 2);
// x ↦ (1, 2)

let y0 = &x;
// x ↦ loan^s {l_0} (1, 2)
// y0 ↦ borrow^s l_0

let y1 = copy y0;
// x ↦ loan^s {l_0, l_1} (1, 2)
// y0 ↦ borrow^s l_0
// y1 ↦ borrow^s l_1

let z = & (*y1).0;
// x ↦ loan^s {l_0, l_1} (loan^s {l_2} 1, 2)
```

```
// y0 ↦ borrow^s l_0
// y1 ↦ borrow^s l_1
// z ↦ borrow^s l_2

assert!(x.1 == 2);
// x ↦ (loan^s {l_2} 1, 2)
// z ↦ borrow^s l_2
```

This operational semantics is used in two ways: for a concrete execution and a symbolic execution. The concrete execution can be used to execute a given LLBC program, for example Aeneas use it to execute functions marked as unit tests. But here, we're mainly interested in the symbolic execution because it's used to translate the LBC program into a functional program.

The symbolic execution diverges from the concrete execution by executing all branches in pattern matching and approximating the environment at function calls. For example, this is the LLBC symbolic execution of the `call_choose` function from before:

```
// Signature of choose:
(bool, &'a mut u32, &'a mut u32) -> &'a mut u32

// Body of call_choose:

let mut x = 2;
let mut y = 6;
// The temporary arguments are in separate statements
// in LLBC, due to the limitation on expressions.
let px = &mut x;
let py = &mut y;
// x ↦ loan^m l_0, y ↦ loan^m l_1, px ↦ borrow^m l_0 2, yx ↦ borrow^m l_1 6

let p = choose(true, move px, move py);
// x ↦ loan^m l_0, y ↦ loan^m l_1, p ↦ borrow^m l_r (σ : u32)
// A{borrow^m l_0, borrow^m l_0, loan^m l_r}

*p += 3;
```

```
// Step 1:
// x ↦ loan^m l_0, y ↦ loan^m l_1, p ↦ borrow^m l_r (σ' : u32)
// A{borrow^m l_0, borrow^m l_1, loan^m l_r}
// Step 2:
// x ↦ loan^m l_0, y ↦ loan^m l_1, px' ↦ borrow^m l_0 σ_x, py' ↦ borrow^m l_1 σ_y
// Step 3:
// x ↦ σ_x, y ↦ loan^m l_1, py' ↦ borrow^m l_1 σ_y
assert!(x == 5);
```

We see that calling a function results in $\sigma$, a symbolic value output, and $A$, a region abstraction: this records the function borrows from its inputs and the function loans from its output. Then, when accessing $x$, we need to:

- Retrieve the value associated to the borrow $l_0$ (step 3).
- For that, we must first end the region abstraction holding $borrow^m\ l_0$ (step 2).
- For that, the region abstraction must get the value associated to its loans (step 1).

So, for each function call, we get one region abstraction by lifetime parameter which tracks the inputs and outputs of its corresponding backtrack function.

The translation of `call_choose` is done by following the symbolic execution. Notably, the line `(x0, y0) <-- choose_back i32 true 2 6 z` is obtained by translating the step 2: the outputs `x0` and `y0` correspond to the variables $px'$ and $py'$. The rest of this translation is quite straightforward.

The details on the translation (implemented by continuation passing style) can be found in [HP22]. One important point is that disjunctions in the control-flow are assumed to be in terminal position: that means that the continuations of `match`s will be duplicated in each branch, because environments are only duplicated and not joined (this is partly what I've worked on).

## 2.2 Charon

A Rust program is compiled through several representations from higher to lower level. Charon fetches a Rust program at the MIR step ("Mid-level

Intermediate Representation"), after that the type checker ran and translated high-level features (e.g. the lookup of typeclass functions) but before optimisations on the MIR.

The MIR code is a form of minimal Rust: operations and types are explicit, high-level features are desugared into lower-level operations, the control flow is done with `goto` instructions and the MIR encodes a CFG (Control-Flow Graph).

The first thing done is to register then reorder all types and functions declarations of the Rust program so that dependencies of a declaration occurs before it. In the case of (perhaps mutually) recursive declarations, they are grouped together. This is done because the target language generally don't support unordered declarations as Rust does. This reordering is made to be as stable as possible regarding the original declarations order.

Then, the MIR code is translated to a similar format (simply named "IR") but independent from the Rust compiler. This allows a complete control on this representation, for minor changes and custom identifiers.

After that, this code is translated to LLBC code by reconstructing the control-flow from the CFG to the AST control-flow (with pattern matching), without early returns. An option can be enabled to check that no code duplication occurs, to keep a good quality translation.

Finally, some micro-passes are applied to the LLBC code:

- Some operations are simplified.
- Assertions are reconstructed.
- Functions which return nothing return an unit type.
- Unused local variables are removed.

The resulting LLBC code is exported in JSON.

## 2.3 Aeneas

Aeneas parses the LLBC JSON and performs a symbolic execution on it, to transform it to a symbolic AST. Then, this symbolic AST is used to synthesize the (typed) lambda-calculus translation. Finally, this lambda-calculus is exported in F$^\star$. Functions without parameters marked as unit tests can also be executed with the concrete semantics to pass them.

Aeneas misses some features that can be provided by a purely safe Rust implementation, such as:

- Constants.
- Loops.
- Nested borrows (such as `&'a mut (T1, &'b mut T2)`).
- Borrows in enumerations, and so borrows in recursive types as well.
- Closures.
- Dynamic drops.
- Hierarchies between lifetimes.
- The high-level features coming with traits (Rust typeclasses).

# 3 Personal work

The four sections below goes from the most concrete to the most speculative work. The two first sections (constants and joins) took most of my time and are compatible with Aeneas formalism, whereas the two next present alternative formalisms to address the functional translation.

## 3.1 Constants

A limit of Aeneas is that declarations of global variables are not supported. Global mutable variables stay out of scope because they are only supported in unsafe code, so my first contribution was to add global constants to Aeneas. This conceptually small addition ended up in quite significant changes (the total changes being +3300 and -1800 LOC) which address some adjacent issues or limitations.

After those changes, Aeneas can translate the following file ...

```
const X: u32 = u32::MAX;

static Y: u32 = add(2, 3);

const fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

...To this F* file:

```
module Constants
// Contains common stuff such as the result monad or eval_global.
open Primitives

(** [core::num::u32::{8}::MAX] *)
let core_num_u32_max_body : result u32 = Return 4294967295
let core_num_u32_max_c : u32 = eval_global core_num_u32_max_body

(** [constants::X] *)
let x_body : result u32 = Return core_num_u32_max_c
let x_c : u32 = eval_global x_body

(** [constants::add] *)
let add_fwd (a : i32) (b : i32) : result i32 =
    i <-- i32_add a b;
    Return i

(** [constants::Y] *)
let y_body : result i32 =
    i <-- add_fwd 2 3;
    Return i
let y_c : i32 = eval_global y_body
```

More precisely, we want to translate global `const` and `static` variables from safe Rust programs and treat uniformly those two kinds of declaration. They are distinct in Rust because static variables are guaranteed to be unique

(with a single address): they are similar to inline variables in C++17 or static variables defined in a C function, while const values can be freely duplicated [1].

While safe Rust supports const variables parametrized by generic types, this feature is not added by this work. Now, we'll follow the translation of global variables from the Rust program to their representation in a functional programming language.

Most modifications happened in Charon. The first step is to be able to find the const and static global variables in the Rust MIR program. Statics were a bit tricky because they are treated by the compiler as a constant address to an allocation: the static is mentioned in a tag associated to this allocation.

Once they are found, global variables are registered in a new, third kind of declaration alongside types and functions. Here, I did some refactoring to reduce duplications between type and function declarations and to abstract common parts between function and global declarations. I also decoupled the mechanism to avoid infinite recursion from the depth-first visit of the Rust MIR.

Before this work on globals, external globals (such as `u32::max` from the standard library) were supported by being inlined there they are used. Now, external globals are exported separately, as shown in the example above.

Then, we need to add globals in the LLBC language. For that, I updated it with global declarations and an instruction to assign a global. The instruction is separated from a regular assignment as the result from a tradeoff between a higher-level langage and more atomic operations.

I also removed constant ADTs ("algebraic data types", types from enumerations and records or tuples) from LLBC: this was inherited from Rust MIR but not really wanted in LLBC where no difference is made between run-time and compile-time values. To do that, I implemented a micro-pass to change them to regular ADTs.

Then, in Aeneas, I adapted LLBC to the removal of constant ADTs, the global declarations and the global assignments. I first proposed to treat glob-

[1] See `https://rust-lang.github.io/rfcs/0246-const-vs-static.html` for more details.

als as functions without arguments, but in the end I treated them separately to exploit the fact that they cannot fail.

Indeed, by default every function is wrapped in a result monad which accounts for possible panics of the Rust program (e.g. when an addition overflows). However, const or static globals are initialized at compile-time, so they do not panic. To exploit this, their declaration is separated in two:

- The global body, which is treated like a regular function without arguments.
- The global "real" declaration (the one used to refer to the global), which normalizes the global body and unwraps its value.

This also normalizes the constant bodies. With this last step, the translation is finally complete.

## 3.2 Joins

Another limit of Aeneas concerns the way borrows are approximated: every execution path of a function goes under symbolic execution because the approximation only takes place at function call sites. In presence of multiple pattern matching, that leads to a combinatorial explosion of branches, which harms both the translation performances and, more importantly, the quality of the translation.

For example, here is a Rust program followed by some pseudo-code for its translation:

```
if a { f() };
if b { g() };
h();

if a
then f ; if b
  then g ; h
  else h
else if b
  then g ; h
  else h
```

We see that the rest of the function (here named `h`) is already duplicated four times. On top of that, loops are not supported because the current method would create an infinity of execution paths:

```
loop {
  if a { break };
  f();
}
g();
```

```
if a then g
else f ; if a
  then g
  else f ; if a
    then g
    else f ; ...
```

My second contribution is a join operation to merge environments. This is also a first step towards loops: we would then need to compute a fixpoint on the environment joins to then translate loops into recursive functions whose input is the program environment.

Then, joins mark a shift in the way borrows are treated because they will be approximated at the level of control-flow branches (in loops and branches) on top of function calls. A join is defined as an n-ary commutative operation on environments which yields a new environment. The 0-ary case is not defined.

I'll present first the binary version for a lighter formalism: the n-ary version is then straightforward but nonetheless required because the join operation is not associative. The joined environments are from the same function, so they must have the same variables with the same types. However, they may have different regions. We'll see how to join the environments from the following code:

```
let ab1 = (a1, b1);
let ab2 = (a2, b2);
let p: &mut (A, B);
let q: &mut A;
```

```
// ab1 ↦ (a1, b1)
// ab2 ↦ (a2, b2)
if e {
    p = &mut ab1;
    q = &mut ab2.0;
    // ab1 ↦ loan^m l0
    // ab2 ↦ (loan^m l1, b2)
    // p ↦ borrow^m l0 (a1, b1)
    // q ↦ borrow^m l1 a2
}
else {
    p = &mut ab2;
    q = &mut ab1.0;
    // ab1 ↦ (loan^m l2, b1)
    // ab2 ↦ loan^m l3
    // p ↦ borrow^m l3 (a2, b2)
    // q ↦ borrow^m l2 a1
}
// Which environment ?
```

Intuitively, we can see an environment as an hypergraph whose vertices are values and edges are region abstractions borrowing and lending values (or simple loan identifiers for 1-to-1 cases). Then, the join operation takes hypergraphs with the same vertices and do the union of their abstractions. That may create region abstractions when 1-to-1 cases become more complex.

More precisely, the join of two environments $E_0$ and $E_1$ is the environment where

- Each variable $v_i$ has the value given by the join of the values from variables with the same identifier in $E_0$ and $E_1$: in pseudo-code, $v_i \mapsto E_0.v_i \cup E_1.v_i$.
- Region abstractions are those from $E_0$ or $E_1$. In case the abstraction comes from both $E_0$ and $E_1$, they are joined.

So, it remains to define joins on values of the same type and on abstractions. For that, I gave rewrite rules which aims to ultimately eliminate all occurrences of "$\cup$". If no rule can be applied but some "$\cup$" remains, it's treated as a type error. This happens for example if exactly one of the two

values is $\bot$: the same variables must be alive at the join. So, this cannot supports dynamic joins [2] without further additions.

```
// The joined environment before rewriting "∪"s.
```
$ab1 \mapsto (loan^m\ l_0) \cup ((loan^m\ l_2, b_1))$
$ab2 \mapsto ((loan^m\ l_1, b_2)) \cup (loan^m\ l_3)$
$p \mapsto (borrow^m\ l_0\ (a_1, b_1)) \cup (borrow^m\ l_3\ (a_2, b_2))$
$q \mapsto (borrow^m\ l_1\ a_2) \cup (borrow^m\ l_2\ a_1)$

The joined environment from the example above will be expanded after explaining the rules. Some conventions are used for the rewriting rules below:

- $s, s_0, s_1, \ldots$ are symbols.
- $l, l_0, l_1, \ldots$ are loan identifiers.
- $r, r_0, r_1, \ldots$ are region abstraction identifiers.
- $x, x_0, x_1, \ldots$ are values from the left environment (i.e. the join first input).
- $y, y_0, y_1, \ldots$ are values from the right environment.
- Identifiers appearing in the right part of reduction rules are fresh identifiers.

Asymmetric rules are implicitly duplicated with swapped arguments.

Bottom: $\bot \cup \bot \rightsquigarrow \bot$

Symbols:

- $s_0 \cup s_1 \rightsquigarrow s_2$
- $s_0 \cup y, \bot \notin y \wedge borrow \notin y \wedge loan \notin y \rightsquigarrow s_1$
- $s \cup (y_0, \ldots y_n)$, `let` $y = (y_0, \ldots y_n)$ `in` $\bot \notin y \wedge (borrow \in y \vee loan \in y) \rightsquigarrow (s_0, \ldots s_n) \cup (y_0, \ldots y_n)$

[2] See `https://rust-lang.github.io/rfcs/0320-nonzeroing-dynamic-drop.html#how-dynamic-drop-semantics-works` for a description of dynamic drops.

The conditions to apply the second rule holds because it is not sound to simplify loans or borrows. The third rule is simply the symbolic expansion of $s$ when it matches a tuple containing some loans or borrows.

Tuples: $(x_0, \ldots x_n) \cup (y_0, \ldots y_n) \rightsquigarrow (x_0 \cup y_0, \ldots x_n \cup y_n)$

Region abstractions: Unless they have the same definition, the join fails. It may be possible to lighten this condition by doing the union of their values and assigning a backward function that calls the backward function of the region corresponding to the taken branch, but that need more testing. That allows us to curry some values given back in some (but not all) joined branches.

Mutable loans:

- $(mut\_loan\ l) \cup (mut\_loan\ l) \rightsquigarrow mut\_loan\ l$
- $(mut\_loan\ l) \cup x, \bot \notin x \wedge loan \notin x \rightsquigarrow mut\_loan\ l$
- $(mut\_loan\ l) \cup x, \bot \notin x \wedge loan \in x \rightsquigarrow mut\_loan\ l_0$ and a new region $r\{mut\_borrow\ l_0\ s_0, mut\_loan\ l, loans(x)...\}$

In the second and third rules, we can assume that $borrow \notin x$ because nested borrows are not supported. In the n-ary version, the third case is picked when there are different loans among the joined values, and the new region abstraction retrieves loans from all joined values.

Mutable borrows:

- $(mut\_borrow\ l\ x) \cup (mut\_borrow\ l\ y) \rightsquigarrow mut\_borrow\ l(x \cup y)$
- $(mut\_borrow\ l_0\ x_0) \cup (mut\_borrow\ l_1\ y_0) \rightsquigarrow mut\_borrow\ l_2\ s_0$ and a new region $r\{mut\_borrow\ l_0\ s_1, mut\_borrow\ l_1\ s_2, mut\_loan\ l_2\}$

The join main rules are the one creating new region abstractions (in loans and borrows): it takes the graph of borrows from both environments and add regions for loans going to different places and borrows from different places.

Shared borrows are treated similarly to mutable borrows. There are some remaining limitations when a safe Rust program weakens a mutable borrow into a shared borrow in one of the joined branches, because it will prevent to

eliminate the join of shared and mutable borrows and be treated as a type error.

Now, backwards functions should be associated to region abstractions to retrieve the values. I will present those for the n-ary version, in the case of `match`. A region abstraction can be seen as a signature for a backward function whose arguments are the region loans and the outputs are the region borrows.

Then, a region $r\{mut\_borrow\ l, mut\_loan\ l_0, \ldots mut\_loan\ l_n\}$ from the third rules of loans applied to $x_0 \cup \ldots \cup x_k$ receives the backward function $\lambda(l_0, \ldots l_n).\texttt{match e } \{\texttt{case } 0 \Rightarrow x_0[l_i/loan\ l_i], \ldots \texttt{case } k \Rightarrow x_k[l_i/loan\ l_i]\}$ where "e" is the discriminant of the matched enumeration. "$[l_i/loan\ l_i]$" means that every loan in a joined value is replaced by the corresponding backward function argument. The same function is created for shared loans.

Finally, a region $r\{mut\_loan\ l, mut\_borrow\ l_0, \ldots mut\_borrow\ l_n\}$ from the third rules of borrows applied to $mut\_borrow\ x_0 \cup \ldots \cup mut\_borrow\ x_n$ receives the backward function $\lambda(l).\texttt{match e } \{\texttt{case } 0 \Rightarrow (l, x_1, \ldots x_n), \texttt{case } 1 \Rightarrow (x_0, l, \ldots x_n), \ldots \texttt{case } n \Rightarrow (x_0, x_1, \ldots l)\}$. The same function is created for shared borrows.

The backward function for loans fills holes in the expression from the executed branch, while the backward function for borrows gives back the borrowed values except the one from the executed branch, to give the updated value instead. We can now complete the join:

$ab1 \mapsto (loan^m\ l_0) \cup ((loan^m\ l_2, b_1))$
$ab2 \mapsto ((loan^m\ l_1, b_2)) \cup (loan^m\ l_3)$
$p \mapsto (borrow^m\ l_0\ (a_1, b_1)) \cup (borrow^m\ l_3\ (a_2, b_2))$
$q \mapsto (borrow^m\ l_1\ a_2) \cup (borrow^m\ l_2\ a_1)$

```
// Join the ab1 values:
// Yields a new loan and its new region.
```
$ab1 = loan^m\ l_4$
$ab2 = ((loan^m\ l_1, b_2)) \cup (loan^m\ l_3)$
$p = (borrow^m\ l_0\ (a_1, b_1)) \cup (borrow^m\ l_3\ (a_2, b_2))$
$q = (borrow^m\ l_1\ a_2) \cup (borrow^m\ l_2\ a_1)$
$R0$ {
  $borrow^m\ l_4$,
  $loan^m\ l_0$,

```
  loan^m l_2,
  // "l_0" and "(l_2, b_1)" are the joined values of ab1,
  // after the substitution for their loans.
  λ(l_0, l_2). match e with
    | true  → l_0
    | false → (l_2, b_1)
}

// Join the remaining values:
ab1 = loan^m l_4
ab2 = loan^m l_5
p = borrow^m l_6σ_0
q = borrow^m l_7σ_1
R0 {
  borrow^m l_4,
  loan^m l_0,
  loan^m l_2,
  λ(l_0, l_2). match e with
    | true  → l_0
    | false → (l_2, b_1)
}
R1 {
  borrow^m l_5,
  loan^m l_1,
  loan^m l_3,
  λ(l_1, l_3). match e with
    | true  → (l_1, b_2)
    | false → l_3
}
R2 {
  borrow^m l_0, // Borrowed (a_1, b_1)
  borrow^m l_3, // Borrowed (a_2, b_2)
  loan^m l_6,
  // Recall the resulting square, constant
  // everywhere but on the diagonal.
  λ(l_6). match e with
    | true  → (l_6, (a_2, b_2))
    | false → ((a_1, b_1), l_6)
```

```
}
R3 {
  borrow^m l_1, // Borrowed a_2
  borrow^m l_2, // Borrowed a_1
  loan^m l_7,
  λ(l_7). match e with
    | true  → (l_7, a_1)
    | false → (a_2, l_7)
}
```

Those rules can yield regions whose loans/arguments come from other regions. In those cases, in can be useful to inline those arguments to "flatten" the graph of borrows, so that all regions take their arguments from values. This can be done by "inlining" in a region the arguments from the regions it depends on, as shown below:

```
// Starting from the previous environment.
// We can inline dependencies in R0 from other regions:
R0 {
  borrow^m l_4,
  loan^m l_6,
  loan^m l_7,
  λ(l_6, l_7).
    let l_0 = R2(l_6).0 in
    let l_2 = R3(l_7).1 in
    match e with
    | true  → l_0
    | false → (l_2, b1)
}

// We unfold R2 and R3 backward functions ...
R0 {
  borrow^m l_4,
  loan^m l_0,
  loan^m l_2,
  λ(l_6, l_7).
    let l_0 = match e with
    | true  → l_6
```

```
    | false → (a1, b1)
    in
    let l_2 = match e with
    | true  → a1
    | false → l_7
    in
    match e with
    | true  → l_0
    | false → (l_2, b1)
}

// ... And simplify R0 backward function.
R0 {
  borrow^m l_4,
  loan^m l_6,
  loan^m l_7,
  λ(l_6, l_7).
    match e with
    | true  → l_6
    | false → (l_7, b1)
}

// Let's retrieve ab1 value to check the join:
// We feed the value of borrows l_6 and l_7 to R0 ...
ab1 = R0(match e with
    | true  → (a1, b1)
    | false → (a2, b2)
        , match e with
    | true  → a2
    | false → a1
)

// ... Unfold R0 backward function ...
ab1 = match e with
    | true  → (a1, b1)
    | false → (a1, b1)

// ... And simplify the resulting expression.
```

```
// That's indeed ab1 value!
ab1 = (a1, b1)
```

I have not yet defined how those backward functions precisely fit in the current backward function translation.

Subsequent work on loops is not done. The next step would be to compute a fixpoint on the environment with successive joins to determine the environment at the start of the loop. The environment at the end of the loop is simply a join of paths ending on a break statement. Then, the loop body can be translated to a recursive function. More work would be needed to allow inner return statements and nested loops with labelled break or continue statements.

## 3.3 Typed loan identifiers

To implement joins, my first approach as to type the loan and borrow identifiers with the set of possible identifiers. That would allow to

- Simplify at any time the environment values.
- Split the computational content of an environment from its type.
- Aim to capture additional features such as borrows in enumerations or nested borrows.
- Control the loss of precision during joins.

However, it makes too much changes to be incorporated to Aeneas.

Loan identifiers are renamed holes. The key idea is to introduce a subtyping relation on environments, so we can weaken them to loose in precision: given two environments of type $E_0, E_1$ such that $E_0 \leq: E_1$, we have a coercion operation written $(e : E_0)$ `as` $E_1$ which transport the values of $e$ in $E_1$ with the identity function.

Then, $E_0 \vee E_1$ gives the join of two compatible environments: the most specific supertype $E_{0 \vee 1}$, meaning that $E_0, E_1 \leq: E_{0 \vee 1} \wedge \forall E. E_0, E_1 \leq: E \Rightarrow E_{0 \vee 1} \leq: E$. This operation is associative and commutative, so we can ignore in which order we merge the environments (at the end of a `match` or at the beginning of a `loop`).

An environment is a set of variables, a variable being a pair of an identifier and a typed value: $\{v \leftarrow (x : T), \ldots\} : env$. Two environments are compatible if their variables have the same identifiers and compatible types. Then, we have $E_0 \leq: E_1$ iff environments are compatible and we have $T_0 \leq: T_1$ for each (common) variable $v \leftarrow (x_0 : T_0), v \leftarrow (v_1 : T_1)$.

So, it remains to define the union, the compatibility relation and the subtyping relation on types. Compared to Aeneas, loan and borrow types are now typed with the set of possible holes for the loan or borrow. So first, we introduce new rules to manipulate holes:

H-NEW $\dfrac{}{\Gamma \vdash h : hole, h \notin \Gamma}$

H-EMPTY $\dfrac{}{\{\} : holes}$

H-POINT $\dfrac{\Gamma \vdash h : hole}{\Gamma \vdash \{h\} : holes}$

H-UNION $\dfrac{\Gamma \vdash A : holes \qquad \Delta \vdash B : holes}{\Gamma, \Delta \vdash A \cup B : holes}$

When calling functions, lifetime parameters are interpreted as sets of holes in the context. Those are the only possible open terms for hole sets. The holes of references associated to those lifetimes are then the only possible open terms for holes. Hole sets are given a normal form by separating concrete holes, open terms for holes and open terms for hole sets in three different sets. This form is preserved by unions is facilitates the evaluation of the $\subseteq$ predicate.

We can now define types including mutable loans and borrows. Their shared flavor has not been investigated.

MLOAN-TYPE $\dfrac{\Gamma \vdash A : type, H : holes}{\Gamma \vdash A < H : type}$

A loan of type $A < H$ is a value of type $A$ accessible once the holes in $H$ are filled. For this reason, we can always weaken a type by adding holes to it. Moreover, types can also be weakened by propagating holes from one of its inner type to an outer type. For example, $(A < \{h\}, B) \leq: (A, B) < \{h\}$.

Then, we can determine the union of two loans: $(A_0 < H_0) \vee (A_1 < H_1) \rightsquigarrow (A_0 \vee A_1) < (H_0 \cup H_1)$. If only one value is a loan, we first change

the type of the other value $A$ to $A < \{\}$ : those types are equivalent as they have the same values.

$$\textsc{Mborrow-type}\ \frac{\Gamma \vdash A : \mathit{type}, H : \mathit{holes}}{\Gamma \vdash A > H : \mathit{type}}$$

A borrow of type $A > H$ is a value of type $A$ that will fill one of the holes in $H$ once the borrow is dropped. So, we can weaken a borrow by adding holes which are not filled in $H$, in the same way as for loans. The union rule follows the one for loans: $(A_0 > H_0) \vee (A_1 > H_1) \rightsquigarrow (A_0 \vee A_1) > (H_0 \cup H_1)$. Note that when a borrow is created, its hole set is a singleton: borrows with an empty set occur when constructing an enumeration which has another constructor parametrized by some region. For example, the value `Option<&'a mut T>::None` has the sum type $() \mid (A > \{\})$.

Finally, two types are compatible if they are the same modulo their holes in borrows and loans ($A$ being compatible with $A < H$). This corresponds to the existence of their join.

Now, to handle the functional translation, we need to recover or replace backward functions. They are two candidates for that:

A first way would be to keep all holes local, so that when ending a borrow, we can immediately fill its corresponding hole. This "eager" approach eliminates backward functions but pass and return additional data from functions. For that, we define a *mold* function which determines the additional data from a function argument type (we consider here that a function takes exactly one argument to simplify):

- $\mathit{mold}((A, B)) \rightsquigarrow (\mathit{mold}(A), \mathit{mold}(B))$
- $\mathit{mold}(A \mid B) \rightsquigarrow \mathit{mold}(A) \mid \mathit{mold}(B)$
- $\mathit{mold}(A < H)$ is undefined (function arguments cannot have loans).
- For an atomic or generic type, $\mathit{mold}(A) \rightsquigarrow ()$
- $\mathit{mold}(A > H) \rightsquigarrow ((A < H), \mathit{mold}(A))$

The last rule is the more important, it says that a borrow comes with its hole and the mold of its underlying value. For example, given a nested borrow `&'a mut (A, &'b mut B)`, we get:

$mold((A, B > H_1) > H_0)$
$= ((A, B > H_1) < H_0, ((), (B < H_1, ())))$
$\simeq ((A, B > H_1) < H_0, B < H_1)$.

This is because we pass both the hole corresponding to a given borrow and the holes which may be filled by re-borrowing some fields under the given borrow:

```
fn relocate(x: &'a mut (A, &'b mut B), y: &'b mut B) {
    // Here holes, l_0, l_1 and l_2 are symbolic values.
    // l_0 ∈ H_0 and l_1, l_2 ∈ H_1.
    // x ↦ borrow^m l_0 (σ_0, borrow^m l_1 σ_1) : (A, B > H_1) > H_0
    // y ↦ borrow^m l_2 σ_2 : B > H_1
    // mold(x) ↦ (loan^m l_0, loan^m l_1) : ((A, B > H_1) < H_0, B < H_1)
    // mold(y) ↦ loan^m l_2 : B < H_1
    *x.1 = move y;
    // x ↦ borrow^m l_0 (σ_0, borrow^m l_2 σ_2) : (A, B > H_1) > H_0
    // mold(x) ↦ (loan^m l_0, σ_1) : ((A, B > H_1) < H_0, B)
    // mold(y) ↦ loan^m l_2 : B < H_1

    drop x;
    // mold(x) ↦ ((σ_0, borrow^m l_2 σ_2), σ_1) : ((A, B > H_1), B)
    // mold(y) ↦ loan^m l_2 : B < H_1
    // The mold values are returned from the function.
    // In the concrete evaluation, they replace the mold input values.
    // In the symbolic evaluation, they may just remove some H_0 loans.
}
```

This approach needs some more work, but seems to work well even when adding features such as nested borrows or borrows in recursive types. However, its calculus is not fit for lambda calculus and the translation is quite heavyweight, due to the additional data which come around at function calls.

So while this may be a good approach to specify a borrow checker for Rust, a lazier approach would be preferable for the functional translation.

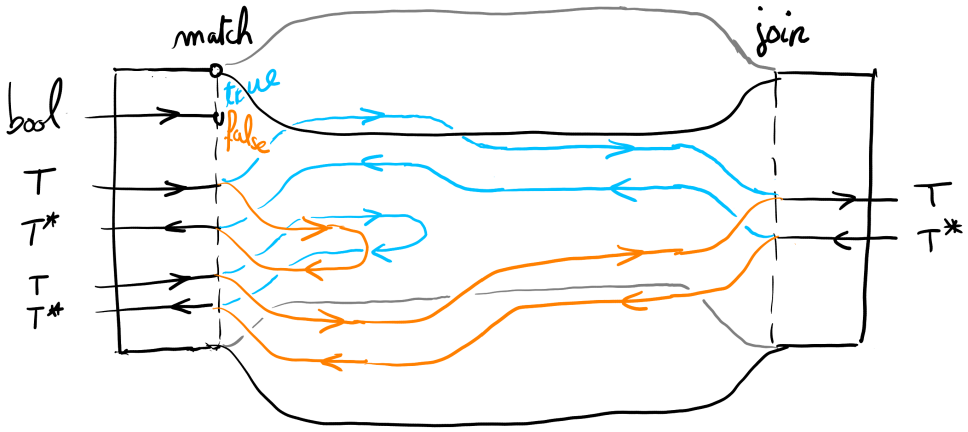
The second approach is to consider mold values as backward functions, which are built from the operations such as drops inside the function. More work is also required to explore this case.

## 3.4 Denotational semantics

To interpret a LLBC program, I also began a graphical approach to capture its denotation while minimizing sequentiality: values are interpreted by oriented strings (from left to right) and functions by boxes, taking arguments on their left and outputting results on their right. The model is not yet fixed but the needed operations suggest some monoidal category.

References are pairs of values, one of them going backwards, which is its dual object. They are created by a demi-circle to the left (an unit operation) and dropped with a demi-circle to the right (an eval operation). Product types can be eliminated by splitting its values while sum types are eliminated by splitting the environnement (their merge correspond to the join operation).

For example, this is how the `choose` function is represented:




A sheet or plan corresponds to an environment and a vertical slice of it to the environment at a certain state: in particular, boxes (left and right) boundaries fix the passing environment type. Backward functions corresponds to the part of this function which manipulates backward values. We can remark that a loop would be represented by a cylinder, as the environment sheet rolls back to the loop beginning.

Then, a program is correct when it is sequentializable, i.e. when it can be continuously deformed to eliminate all backward values: this amounts to performing the functional translation. That means that no function input depends on any of its output: this is a global acyclicity condition. Another condition concerns the fact that values must outlive references on it: that

means that a value dropped in a box cannot depend on a backward value exiting the box.

The borrow checker rules and regions are understood as a mean to check those conditions in a compositional way by approximating the possible entanglements inside boxes in a "predicative" way (values live in a greater scope than their references). It remains to understand how those contraints fit in this setting ...

An interesting point with this model is that it seems to be able to manipulate loans (i.e. backward values) explicitly: this would allow to use some safe patterns that I was using in C++ and that I miss in Rust. It contrasts with the predicative way of handling references described above: for example, we would be able to allocate a collection in a function, borrow one of its elements then return the collection along the reference. Several Rust libraries address this limitation for very specific cases.

```
// Pseudo-Rust code for the pattern above:
// We can return a reference and its borrowed value.
fn make_heap_ref<T>(x: T)
    -> exists 'a.(Vec<loan 'a mut T>, &'a mut T)
{
    // Move x on the heap (with an owned allocation).
    let mut b = Box::new(x);
    // Take a reference on the moved value.
    let r = b.as_mut();
    (b, r)
}
```

This also illustrates some important parts to interpret: moving values in Rust is not an innocuous operation because references on the value are invalidated. However, that doesn't necessarily apply to moved value members which are not themselves moved, such as the inner `Box` value above.

# 4 Related works

The idea to exploit Rust type system to ease the verification of Rust programs is not new:

Electrolysis [Seb16] is a framework which translates some safe Rust programs in Lean by treating references as lenses. It supports fewer programs than Aeneas, for example the `choose` function is not translated.

Otherwise, there are two ongoing projects to verify annotated Rust programs which cover (among other things) constants and loops:

Prusti [Ast+22] encodes Rust program in Viper, a toolchain that allows to reason on program states with notions of permissions and ownership. Them it uses Rust type system to automatically apply some rules to ease the verification.

There is also Creusot [DJM21], an encoding of Rust programs in Why3 based on prophecy variables, an higher-level logical representation of Rust borrows. Its interface is similar to Prusti: it takes annotated programs and produce intrinsic/automated proofs, as opposed to Aeneas or Electrolysis (untouched programs, external proofs).

Finally, the RustBelt [Col18] project allows to verify Rust programs, including their unsafe parts. For this reason, this framework uses a more heavyweight encoding of the program and is more seen as a complementary work (to address the small, tedious unsafe parts of the code) than an alternative.

# 5 Future works

Most future works concerns Aeneas, which is built in an incremental way:

- The most direct continuation would be to precise and implement the join operation in Aeneas then work on loops.
- There are a lot of features to add, listed in the Aeneas subsection: typeclasses, closures, nested borrows, lifetimes subtyping, ...
- Soundness proofs are always welcomed.
- Aeneas aims to ramp up the verification with bigger projects: a verified implementation of b$\epsilon$ tree is ongoing.

Then, I'm also interested in alternative formalism to address the functional translation, as well as the questions about typechecking and denotational semantics starting from a similar language to LLBC:

Typed joins suggest that a borrow checker is feasible by filling holes eagerly (to avoid backward functions). I began writing an implementation of a type checker on a small LLBC variant to explore this case.

Also, did not have the time to pass a lot of time on the denotational semantics but I believe that searching safe patterns in this kind of model can be fruitful to interpret more low-level programs, and this mirrors Rust capacities to offer novel safe interfaces with unsafe implementations.

A possible plan would be to design a category $\mathcal{C}$ with local rules to approximate the global invariants described on diagrams so that any morphism in it is a valid program, then translate those morphisms with a faithful functor in a category without duals and close to actual proof assistants. That would allow to extend the set of verifiable imperative programs by extending $\mathcal{C}$.

# References


[Ast+22] Vytautas Astrauskas et al. "The Prusti Project: Formal Verification for Rust". English. In: *NASA Formal Methods*. Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer, 2022, pp. 88–108. ISBN: 9783031067723. DOI: 10.1007/978-3-031-06773-0_5.

[Col18] Adrain Colyer. "RustBelt: Securing the Foundations of the Rust Programming Language". In: *the morning paper* (Jan. 2018). URL: https://blog.acolyer.org/2018/01/18/rustbelt-securing-the-foundations-of-the-rust-programming-language/.

[DJM21] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. *The CREUSOT Environment for the Deductive Verification of Rust Programs*. Research Report RR-9448. Inria Saclay - Île de France, Dec. 2021. URL: https://hal.inria.fr/hal-03526634.

[HP22] Son Ho and Jonathan Protzenko. "Aeneas: Rust Verification by Functional Translation". In: *ICFP 2022*. Aug. 2022. URL: https://www.microsoft.com/en-us/research/publication/aeneas-rust-verification-by-functional-translation/.

[Seb16] Ulrich Sebastian. “Electrolysis: Simple Verification of Rust Programs via Functional Purification”. MA thesis. Karlsruhe Institute of Technology, 2016. URL: https://github.com/Kha/electrolysis.