Towards a linear functional translation for borrowing

Sidney Congard

Towards a linear functional translation for borrowing

Sidney Congard

We present a functional translation of a subset of safe Rust programs, building upon the results of Aeneas [HP22]. It preserves linearity and captures a new feature, namely lifetime bounds. This is a work in progress: in particular, translation rules are not set yet. We discuss perspectives for this work at the end of the paper.

1 Introduction

Rust is, like C++, a systems programming language. It has references called mutable and immutable borrows, giving respectively a read & write and a read-only access to their underlying value. The Rust type system comes with a borrow checker ensuring that they're safely used, notably that values can be either mutated or aliased (i.e. shared) through borrows, but not both at the same time. It eliminates a large class of errors, such as data races or use-after-free.

Recent works exploit this guarantee to give high-level semantics to Rust programs, to facilitate their formal verification. Among those works, most give a logical encoding of Rust programs such as Creusot [DJM22] (using prophecy variables), Verus [LHC+23] (using ghost linear types) and Prusti [AMPS19] (using Rust types to apply some rules automatically). However, there is also Aeneas which instead translates Rust programs into pure functional programs, to embed them in proof assistants (initially F-Star, but now also Lean4, Coq and HOL4). Those works are complementary to RustBelt [JJKD18]: it also gives a semantic to *unsafe* code ignoring the borrow checker, but results in more difficult verification.

This paper builds upon the results of Aeneas, to propose a functional translation for a subset of safe Rust programs that handles lifetime bounds. Our version preserves the source program *linearity*, meaning that it doesn't duplicate or discard the borrowed values. For that, we present preliminary results centered around examples of translated programs:

- The second section "framework" presents the translation framework, starting with Rust borrow checker, Aeneas and then showing our translation.
- The third section "lifetime bounds" presents lifetime bounds and shows how the translation handles them. They aren't supported in Aeneas.
- The last section "work in progress and future perspectives" speaks about:
  - The soundness and expressivity of the translation with respect to source programs.
  - Effectful destructors and exceptions, leveraging the (for now unused) linearity from the translation.
  - Two features that aren't supported in Aeneas: nested borrows and polymorphism.

2 Framework

For simplicity, our source language is a stripped-down version of safe Rust: in particular, we're removing all features that we'll confidently be able to incorporate later in the translation. That includes several features supported by Aeneas such as loops, inductive types, immutable borrows, pair projections, arrays, the ambient state monad and some syntactic sugar. Also, assertions are only used to access values and to illustrate the current program state: their interpretation in the functional language doesn't matter. Finally, polymorphism is restricted to types without borrows: this limitation is discussed in the last section. In the rest of this paper, "borrow" will always mean "mutable borrow". Commented "drop" instructions are inserted by the compiler: they indicate that the dropped variable becomes inaccessible, either because it's going out of scope or because of borrow checker restrictions.

Polonius [Mat18], a new formulation for the borrow checker, ensures the borrow guarantees by binding each borrow to its origins with regions (also called lifetimes), i.e. sets of loans where a loan is a tag at a possible location of the borrowed value. Each region tells us that to access any loaned value in it, all borrows attached to this region must have been dropped: we can then "end" the region to retrieve the loaned values. That allows great flexibility compared to simple linear or affine types, as long as we respect this discipline of "stacked" borrows [JDKD19] where only the most recent borrows are available.

In the example below, after defining `a`, `b` and `c`, Polonius has two regions: `{a}`, attached to `b` and `{*b}`, attached to `c`. Only `c` does not contain any loan and is thus directly accessible. To access `b`, we must drop `c` first and similarly, to access `a` we must drop `b` and so drop also `c` (as we must access a value to drop it). After running the borrow checker, we obtain the inserted drop annotations for borrows. They do nothing at run-time as they only prevent subsequent borrow uses at compile-time.

```
let mut a = 1;
let b = &mut a; // borrows the value a
let c = &mut *b; // borrows the value previously borrowed by b
// drop c (ends the region {*b} to access b)
*b += 2;
// drop b (ends the region {a} to access a)
assert!(a == 3);
```

In Rust function signatures, borrows must be attached to explicit regions parameters. That allows to describe the loans (i.e. possible origins) of the function output borrows in terms of the loans of the input borrows. All information about borrow dependencies is thus available in the signature, allowing the borrow checker to ignore the function implementation at its call sites. In the example below, after calling `pick_first` we have two regions: `{v}`, attached to `r` and `{u, *r}`, attached to `t`.

```
// The region 'a of the output groups together loans from x and y.
fn pick_first<'a, T>(x: &'a mut T, y: &'a mut T) -> &'a mut T {
    // drop y (goes out of scope)
    x
}
let (mut u, mut v) = (10, 20);
let r = &mut v;
let t = pick_first(&mut u, &mut *r);
*t += 1;
// drop t (ends the region {u, *r} to access r)
*r += 2;
// drop r (ends the region {v} to access v)
assert!(u == 11 && v == 22);
```

To translate such a program, dependencies between overlapping borrows are reified into functions that take the values under the recent borrows and give back the values under the older borrows. Thus, dropping a borrow results in explicitly returning its updated underlying value. But, to do that, contrarily to e.g. Creusot, Aeneas doesn't get enough information from the drops inserted by the borrow checker. So, Aeneas also plays the borrow checker role, carrying additional information to know how to return the dropped values to their origin.

It first translates the Rust source program into the Low-Level Borrow Calculus (abbreviated LLBC), a new formalism equipped with a functional operational semantics that explicits the loans for each borrow at run-time. Then, following its symbolic execution which approximates at compile-time the borrow corresponding loans, we can generate the functional translation of the source program in continuation-passing style. Aeneas translates each Rust function into:

- A forward function, which translates borrows by their underlying values.
- One backward function per region, which translates it as a function taking the same arguments as the forward function plus the values of the output borrows under the region, and returning the actualized values of the input borrows under the region.

The Aeneas translation of the previous program is shown below, as a reference before exposing our translation. A backward function is called when the corresponding region is ended (and so after the borrows attached to it are all dropped), to get the actualized values of the region loans before accessing them. Below, we see that because `pick_first_fwd` returns `x`, `pick_first_back` returns `ret` (the actualized value of `x`) and `y` (which didn't change since the call to `pick_first_fwd` as it's loaned).

```
let pick_first_fwd T (x y: T): T = x
let pick_first_back T (x y ret: T): T * T = (ret, y)

let (u, v) = (10, 20) in
let r = v in
let t = pick_first_fwd u r in
let t = t + 1 in
let (u, r) = pick_first_back u r t (* drop t *)
let r = r + 2 in in
let v = r in (* drop r *)
assert(u == 11 && v == 22)
```

Our translation comes closer to Polonius: we build linear functions $(A_i \times ...) \multimap (B_i \times ...)$ (i.e. functions which must be called exactly once and whose arguments cannot be duplicated or discarded) that play the role of regions enriched with information about how borrows are related to their loans, replacing backward functions. It consumes the tuple of values underlying the borrows attached to the region and returns the tuple of loaned values.

Given all the information about borrows and loans dependencies, we are now able to translate basic borrow operations:

- Creating a new borrow `&mut x` of type T is translated by the underlying value x and a new identity route $\lambda(x').(x') : (T) \multimap (T)$ which will transmit the updated value of x.
- Dropping the kth borrow `drop x` from the route $r : (A_0 \times ...A_n) \multimap B$ is translated by passing x in r with $r = \lambda(a_0...a_{k-1}, a_{k+1}...a_n).r(a_0...a_{k-1}, x, a_{k+1}...a_n)$.
- Ending the route $r : () \multimap (B_0 \times ...B_n)$ loaning the values of the variables $b_0, ...b_n$ is translated to `let` $(b_0, ...b_n) = r()$.

- Similar route manipulations are defined to permute their arguments or to compose them on a single input/output.

Each region parameter of a function is translated to a type parameter and in-&-out route arguments. Both route arguments output the type parameter, which is always instantiated with the tuple made from the types of the loaned values for the corresponding region. The input route arguments are the types of the function input borrows underlying values attached to the translated region, and similarly, the output route arguments are the types of the function output borrows underlying values attached to the region. In the translation below, the `pick_first` region `'a` is thus translated by

- The type parameter A.
- The input route `a` expecting the updated values under `x` and y.
- The output route built from `a` expecting the updated value under the returned borrow `x`.

To call a function, we must pass it new routes that group borrows under the same regions, such as λ`(u2, r2).(u2, r2)` in the translation below.

The last missing ingredient is to adapt a subset of LLBC annotating Rust function signatures, variables and regions to connect borrowed values to matching route arguments. While we don't expose formal translation rules in this paper as they are still being worked out, we give our minimal borrow calculus syntax in the table below.

`Sort`	`Variable`	`Grammar`
`VarId`	v	
`FunId`	f	
`RegionId`	α	
`BorrowId`	B_i	
`LoanId`	L_i	
`Value`	a	$\star \mid (a, a') \mid B_i\ a \mid L_i$
`Route`	r	$\overrightarrow{L_i} \multimap (\overrightarrow{B_i} \mid \alpha)$
`FunSignature`	s	$[\overrightarrow{\alpha}](\overrightarrow{a}, \overrightarrow{r}) \to (a, \overrightarrow{r})$
`Environment`	Ω	$\overrightarrow{f : s}, \overrightarrow{\alpha} \vdash \overrightarrow{v : (a \mid r)}$

Such environments annotate Rust variables and routes with a graph of borrow dependencies:

- Edges correspond to borrow and loan identifiers matching pairs B_i, L_i: each borrowed value is annotated with a B_i and must eventually fill the corresponding hole L_i. Note that this can lead to confusion as the route inputs are tagged with loans identifiers and their outputs with borrow identifiers: this is because the route acts like an indirection in which we give back borrowed values.
- Vertices correspond to either values or routes, which have more structure to retain information about compound values or regions. Star ($\star$) values are values without loans or borrows.

We can then define the dependencies of a given loan identifier $D(\Omega, L_i)$ as the set of variables reachable from it, by fetching the corresponding borrow identifier:

- $B_i \in a$ with $v : a \in \Omega$ a variable annotation, in which case $D(\Omega, L_i) := \{v\}$,
- $B_i \in \overrightarrow{B}$ with $r : \overrightarrow{L} \multimap \overrightarrow{B} \in \Omega$ a route annotation, in which case $D(\Omega, L_i) := \cup\{D(\Omega, L_j) \mid L_j \in \overrightarrow{L}\}$.

Except when dropping variables, those dependencies can only grow, but we must avoid making them grow more than necessary: that would prevent us from accessing some variables without a good reason, making the borrow checker too strict. This could happen by merging too many routes while joining different environments, an operation detailed in the next section.

Finally, we can define our main borrow calculus operations on the environment:

- When accessing a variable $v : L_i$, we fetch all the dependencies of loan identifiers in a and call their corresponding routes, removing them from the environment. The environment thus shrinks until there are no loan identifiers left in a.
- When borrowing a variable value $v : a$, we create a route $r : L_i \multimap B_j$ with fresh identifiers and a borrow $b : B_i\ a$. The variable value is replaced by the route output corresponding loan L_j.

In our translation below, we comment most lines with the borrow calculus environment.

```
let pick_first T A (x y: T) (a: (T * T)→A): T * (T→A) =
    (* α ⊢ x : B_x⋆, y : B_y⋆, a : L_x × L_y ⊸ α *)
    let a = λret. a (ret, y) in (* drop y *)
    (* α ⊢ x : B_x⋆, a : L_x ⊸ α *)
    (x, a)

let (u, v) = (10, 20) in
(* ⊢ u : ⋆, v : ⋆ *)
let (r, dr) = (v, λv2. v2) in
(* ⊢ u : ⋆, v : L_1, r : B_0 ⋆, dr : L_0 ⊸ B_1 *)
let (t, dt) = pick_first u r (λ(u2, r2). (u2, r2)) in
(* ⊢ u : L_2, v : L_1, r : B_0 L_3, dr : L_0 ⊸ B_1, t : B_4 ⋆, dt : L_4 ⊸ B_2 × B_3 *)
let t = t + 1 in
let (u, r) = dt t (* drop t *)
(* ⊢ u : ⋆, v : L_1, r : B_0 ⋆, dr : L_0 ⊸ B_1 *)
let r = r + 2 in in
let v = dr r in (* drop r *)
(* ⊢ u : ⋆, v : ⋆ *)
assert(u == 11 && v == 22)
```

Identity routes (that we'll now create with "id") could be inlined to get closer to Aeneas forward and backward functions: above, `dr` could be left implicit and function routes such as `pick_first` third argument could be instantiated with `id`. This yields an equivalent, often more concise translation. We'll keep the style presented above to recognize more easily the translated region parameters in our translations, but we could use this inlined version for any example in this article.

```
(* obtained by inlining a := id in the previous "pick_first" definition *)
let pick_first_inl T (x y: T): T * (T→(T * T)) =
    let a = λret. (ret, y) in (* drop y *)
    (x, a)
```

3 Lifetime bounds

By default, regions mentioned in a function signature are disjoint. This is sometimes too restrictive, as illustrated by the second example below: there, we return either the first two or the last two borrows of a function. So, we want to explain to the borrow checker at the signature level that `ret.0` (the first output) may only overlap `x` and `y`, and similarly that

`ret.1` may only overlap `y` and `z`. This is used in the example to avoid dropping `ret.1` while accessing the value under `x`.

For that, we want to group `ret.0, x, y` under a region and `ret.1, y, z` under another one. However, borrows (in particular `y`) can only be placed in one region. So instead, we'll put it in a third region with *bounds* to the two other regions, meaning that both `ret.0` and `ret.1` may overlap `y`.

```
fn three_of_two<'a, 'c, 'b: 'a+'c, T>(i: bool, x: &'a mut T, y: &'b mut T,
    z: &'c mut T) -> (&'a mut T, &'c mut T) {
    if i { (x, y) }
    else { (y, z) }
}
let (mut u, mut v, mut w) = (10, 20, 30);
let (r, s) = three_of_two(true, &mut u, &mut v, &mut w);
*r += 1;
*s += 2;
// drop r
assert!(u == 11);
*s += 3;
// drop s
assert!(v == 25 && w == 30);
```

Each lifetime bound `'a: 'b` in a signature is translated as a dependency from the route `'b` to the route `'a`, i.e. as an argument between them. Its type is unknown from the caller's point of view and is therefore masked by an existential type.

Our translation handles the if/else construct by computing the join (a known problem in abstract interpretation) of the two branches, taking the union of their borrow dependencies: $\forall L_i.D(join(\Omega,\Omega'),L_i) = D(\Omega,L_i) \cup D(\Omega',L_i)$ (the environments must have the same loan & borrow identifiers). While the obtained routes will match the function return type in this example, we cannot guarantee this in general: see the discussion in future work.

```
let three_of_two T A C B (i: bool) (x y z: T) (a: T→A) (c: T→C) (b: T→B):
    ((T * T) * ∃R.∃S. (T→(A * R)) * (T→(C * S)) * ((R * S)→B)) =
    if i { ((x, y), λx2. (a x2, ()), λy2. (c z, y2), λ((), y2). b y2) }
    else { ((y, z), λy2. (a x, y2), λz2. (c z2, ()), λ(y2, ()). b y2) }

let (u, v, w) = (10, 20, 30) in
let ((r, s), dr, ds, dt) = three_of_two(true, u, v, w, id, id, id) in
let r = r + 1 in
let s = s + 2 in
let (u, t0) = dr r in (* drop r *)
assert(u == 11) in
let s = s + 3 in
let (w, t1) = ds s in (* drop s *)
let v = dt (t0, t1) in
assert(v == 25 && w == 30)
```

To implement this, we define new operations to manipulate routes:

- Adding a dependency between two routes $a \rightsquigarrow b$ is done by adding a unit output to a and a unit input to b, bound by a new pair of identifiers L_i/B_i.
- Masking a dependency between two routes is done by introducing a new existential type for their corresponding output and input.

- Composing routes along a dependency $a \rightsquigarrow b$, if a has a single output or b has a single input, is done by composing them as expected. The condition ensures that all loan dependencies $D(L)$ are preserved so that this rule can be applied whenever wanted, as a normalization rule. This is not the case in general, if the single input or output condition is not verified.

Then, we can translate the if/else clauses with the following steps. They are interleaved with the environment annotations and the generated code for our example in if/else branches, where new expressions fill the previous hole $\Diamond$ to detail the expression translating `three_of_two` before inlining sub-expressions.

`if:` `let ret = (x, y) in` $\Diamond$
$ret : (B_x\star, B_y\star), z : B_z\star, a : L_x \multimap \alpha, b : L_y \multimap \beta, c : L_z \multimap \gamma$
`else:` `let ret = (y, z) in` $\Diamond$
$ret : (B_{y'}\star, B_{z'}\star), x : B_x\star, a : L_x \multimap \alpha, b : L_{y'} \multimap \beta, c : L_{z'} \multimap \gamma$

1. Before joining the two environments, we drop values which are not shared: in our example, `z` in the first branch and `x` in the second branch.

   `if:` `let c = c z in` $\Diamond$
   $ret : (B_x\star, B_y\star), a : L_x \multimap \alpha, b : L_y \multimap \beta, c : \gamma$
   `else:` `let a = a x in` $\Diamond$
   $ret : (B_{y'}\star, B_{z'}\star), a : \alpha, b : L_{y'} \multimap \beta, c : L_{z'} \multimap \gamma$

2. We insert identity routes serving as indirections for the new loans and borrows coming from each branch, to allow many-to-many dependencies between loans and borrows.

   `if:` `let (r0, r1) = (id, id) in` $\Diamond$
   $ret : (B_0\star, B_1\star), a : L_x \multimap \alpha, b : L_y \multimap \beta, c : \gamma, r0 : L_0 \multimap B_x, r1 : L_1 \multimap B_y$
   `else:` `let (r0, r1) = (id, id) in` $\Diamond$
   $ret : (B_0\star, B_1\star), a : \alpha, b : L_{y'} \multimap \beta, c : L_{z'} \multimap \gamma, r0 : L_0 \multimap B_{y'}, r1 : L_1 \multimap B_{z'}$

3. We add on those identity routes the dependencies coming from both branches. At this point, the annotations have the same shape in both branches, so we can rename matching pairs of loan/borrow identifiers to equate them (the renaming is not needed for our example).

   `if:` `let r0 = λx. (r0 x, ()) in let b = λ((), y). b y in`
   `let r1 = λy. (r1 y, ()) in let c = λ(). c in` $\Diamond$
   `else:` `let r0 = λy. ((), r0 y) in let a = λ(). a in`
   `let r1 = λz. ((), r1 z) in let b = λ(y, ()). b y in` $\Diamond$
   $ret : (B_0\star, B_1\star), a : L_x \multimap \alpha, b : L_y \times L_{y'} \multimap \beta, c : L_{z'} \multimap \gamma,$
   $r0 : L_0 \multimap B_x \times B_{y'}, r1 : L_1 \multimap B_y \times B_{z'}$

4. We normalize all routes with the composition rule, which generates the same code for both branches as their environments are the same.

   `let r0 = (λu. let (v, w) = r0 u in (a v, w)) in`
   `let r1 = (λu. let (v, w) = r1 u in (v, c w)) in` $\Diamond$
   $ret : (B_0\star, B_1\star), b : L_y \times L_{y'} \multimap \beta, r0 : L_0 \multimap \alpha \times B_{y'}, r1 : L_1 \multimap B_y \times \gamma$

5. We mask dependencies between routes with new existential types. At this point, the routes have the same type in both branches and can be returned along each if/else result. In the example, some re-ordering of routes and their outputs is needed before returning the if/else result from the function, which is guided by its signature (and inlined in the translation for simplicity).

   `(ret, b, r0, r1)`

4 Work in progress and future perspectives

4.1 Translation soundness and expressivity

To ensure that the translation is correct, we should aim for a theorem ensuring that the source program behaves like its translated program. For that, the source program should be interpreted with an imperative operational semantics (like the one given by RustBelt or tree borrows [Vil23]), and the translated program with a standard functional semantics. If the theorem is too complicated, we can aim for an operational semantics closer to a dynamic borrow checker instead (like LLBC or stacked borrows). To see that the translation captures closely Rust programs, we may also investigate denotational completeness for some Rust function types T, i.e. that every element of type $[\![T]\!]$ is the denotation of a translated source program of type T.

We also want to characterize the translation expressivity: as it is currently, it is determined empirically by looking at which programs the translation accepts. This is very similar to what happens with Rust borrow checker. A first step would be to study the differences between Polonius and our translation. The main difficulty seems to accommodate the lost equivalences when enriching regions/relations to routes/functions.

With regions, any two different ways to present the same dependencies between some borrows and loans are equivalent: we obtain different function signatures that can accept the same programs. In particular, we can call the first function inside the second and vice-versa. With routes, two different presentations of the same dependencies may not be equivalent, preventing the translation of the second function call in the first one or vice-versa.

So either we settle on a translation more restrictive than Polonius, which can be acceptable given that the counter-examples require complex signatures that we rarely find in Rust programs, or we try to rewrite the signatures before translating them. To make further progress, it will be necessary to understand better the structure underlying routes.

4.2 Destructors and effects

Rust sees its values as resources: they are linear values equipped with a destructor, i.e. a function that can consume them and may have an effect to clean up the program state (e.g. a dropped file handle closes its underlying file). The call to the destructor is inserted by the compiler for variables going out of scope, making the type system affine. They are also called when an exception (named "panic" in Rust) is raised.

Like Aeneas, this translation was focused on the notion of borrows. However, Rust also has effectful destructors and exceptions (named "panics") which play a major role in Rust programs:

- Some use emergent patterns, such as typestates [SY86], which exploit Rust linearity and destructors to maintain guarantees about the program state. This allows to implement state machines at the type level.
- Some must be fault-tolerant, by being in a suitable state after encountering an exception. This can happen at FFI boundaries or for programs with multiple threads. This is especially important for the concurrency model of Rust which allows to share mutable data with a mutex: when an exception happens in a subroutine, it poisons the mutex to emit an error at subsequent accesses to the mutex. We thus prevent deadlocks or accesses to an invalid state. This error handling is similar to the mechanism from this paper on affine sessions [MV14], where errors do not prevent progress properties of the whole program.

Those kinds of guarantees aren't preserved in the previously mentioned works, notably Aeneas and RustBelt (where it is an explicit limitation).

Some recent works ([CFMM16], [CMM18]) study the interactions between destructors, effects, and exceptions with denotational semantics for linear logic where a suitable notion of resources can be modeled. Our long-term objective is to leverage our translation linearity to reconcile those works with our handling of borrows, to have a renewed understanding of Rust programming paradigm, and to be able to improve the verification of Rust programs.

4.3 Additional features

Other features from Rust are not present in Aeneas paper. Among those, we investigate nested borrows, i.e. borrows of another borrow, and polymorphism (which was restricted to values without borrows).

Nested borrows are treated similarly to simple borrows, except that their value is consumed by the outer borrow route, and then by the inner borrow route. They are thus carrying those two destinations around, like in the example below. With this representation of nested borrows, `&'a mut T` and `&'a mut (&'a mut T)` become equivalent: the inner borrow is dropped at the same time as the outer borrow. The example below shows how a function with nested borrows can be translated and used.

```
fn nested_swaps<'a, 'b, 'c, T>(x: &'a mut (&'c mut T), y: &'b mut (&'c mut T))
{
    swap(x, y);   // Exchanges the value *x with *y.
    swap(*x, *y); // Exchanges the value **x with **y.
}
let (mut u, mut v) = (10, 20);
let mut r = &mut u;
let mut s = &mut v;
nested_swap(&mut r, &mut s);
assert!(*r == 10 && *s == 20);
// drop r, s
assert!(u == 20 && v == 10);

let nested_swaps T A B C (x y: T) (a: T→A) (b: T→B) (c: (T*T)→C):
        A * B * ((T*T)→C) =
    let (x, y, c) = (y, x, λ(x2, y2). c (y2, x2)) in (* swap (x, y) *)
    let (x, y) = (y, x) in (* swap ( *x, *y ) *)
    let a = a x in (* drop x *)
    let b = b y in (* drop y *)
    (a, b, c)

let (u, v) = (10, 20) in
let (r, dr) = (u, id) in
let (s, ds) = (v, id) in
let (r, s, drs) = nested_swaps r s id id (λ(r3, s3). (dr r3, ds s3)) in
assert(r == 10 && s == 20) in
let (u, v) = drs (r, s) in (* drop r, s *)
assert(u == 20 && v == 10)
```

The issue with polymorphism is that, when a type parameter is instantiated with a type T containing borrows, we must track the commutations and drops of values of type T. Our solution is thus to add a route consuming values of type T as if it was a borrow. That way, the caller gets back the route having consumed the dropped values, and expecting the returned values of type T.

It now remains to establish proper rules and exploit the translation linearity.

References

[AMPS19] Vytautas Astrauskas, Peter Müller, Federico Poli et Alexander J. Summers : Leveraging rust types for modular specification and verification. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[CFMM16] Pierre-Louis Curien, Marcelo Fiore et Guillaume Munch-Maccagnoni : A Theory of Effects and Resources: Adjunction Models and Polarised Calculi. *In Proc. POPL*, 2016.

[CMM18] Guillaume Combette et Guillaume Munch-Maccagnoni : A resource modality for raii (abstract). Rapport technique, avril 2018.

[DJM22] Xavier Denis, Jacques-Henri Jourdan et Claude Marché : Creusot: a Foundry for the Deductive Verification of Rust Programs. *In ICFEM 2022 - 23th International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science, Madrid, Spain, octobre 2022. Springer Verlag.

[HP22] Son Ho et Jonathan Protzenko : Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022.

[JDKD19] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang et Derek Dreyer : Stacked borrows: An aliasing model for rust. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[JJKD18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers et Derek Dreyer : Rustbelt: securing the foundations of the rust programming language. *PACMPL*, 2(POPL):66:1–66:34, 2018.

[LHC+23] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno et Chris Hawblitzel : Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1), apr 2023.

[Mat18] Nicholas Matsakis : An alias-based formulation of the borrow checker, 2018.

[MV14] Dimitris Mostrous et Vasco Thudichum Vasconcelos : Affine Sessions. *In* David Hutchiso, Takeo Kanade, ernhard Steffe, Demetri Terzopoulos, Doug Tyga, Gerhard Weikum, Eva Küh, Rosario Pugliese, Josef Kittle, Jon M. Kleinberg, Alfred Kobsa, Friedemann Matte, John C. Mitchell, Moni Nao, Oscar Nierstrasz et C. Pandu Ranga, éditeurs : *16th International Conference on Coordination Models and Languages (COORDINATION)*, volume LNCS-8459 de *Coordination Models and Languages*, pages 115–130, Berlin, Germany, juin 2014. Springer.

[SY86] Robert E. Strom et Shaula Yemini : Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[Vil23] Neven Villani : Tree borrows, a new aliasing model for rust, 2023.