Rustbelt, a formalization of Rust type system

Ralf Jung et al. @ POPL'18

Presented by Yanning Chen @ ProSE Seminar

Features of Rust type system

- Ownership
- Mutable/Shared References
- Lifetimes
- Interior Mutability

Goal: Well-typed Rust programs should be *memory-safe*.

How? *Aliasing* and *mutation* cannot ocur at the same time on any given location.

Ownership

In Rust, a type represents:

- information on what values it can hold
- **ownership of the resources (e.g. memory)**

Fact: Rust uses an affine type system, i.e. a value can be used *at most* once.

Consequence: no two values can own the same resource, so *mutation* allowed but **no** *aliasing*.

```
let s1 = String::from("hello");
let s2 = s1; // s1 is moved to s2, i.e. s1 is used and no longer available
println!("{}", s1); // error: use of moved value: s1
```

Q: What happens when a value is weakened?

A: Underlying resources deallocated!

Mutable Reference

What we don't want to gain permanent access to a value?

```
fn Vec::push<T>(Vec<T>, T) -> Vec<T>

let v_ = Vec::push(v, 1);   // v is no longer available
```

Instead, Rust uses *mutable references*:

```
fn Vec::push<T>(&mut Vec<T>, T)

Vec::push(&mut v, 1);   // v is still available
```

A **mutable reference** grants *temporary exclusive access* (i.e. *borrowing*) to the value *for the duration of the function call*.

Result: still, *mutation* allowed but **no** *aliasing*.

Shared Reference

What if we want to access a value at multiple places?

Admit *aliasing*!

```rust
let v = vec![1];
// Two shared references to v are created
join(|| println!("{:?}", &v), || println!("{:?}", &v));
// Still have access to v at the main thread after references are dropped
Vec::push(&mut v, 2);
```

Result: for memory safety, allow *aliasing* but **no** *mutation*.

```rust
let v = vec![1];
let r = &v;             // temporary shared reference
Vec::push(&mut v, 2);   // error: active shared reference exists
println!("{:?}", r);    // shared reference ends here
```

Shared Reference - *Copy* types

What if we want to access a value at multiple places?
Admit *aliasing*!

Therefore, *shared references* can be freely duplicated, i.e. *unrestricted variables* in linear logic.

In Rust, *unrestricted types* are called *Copy* types.

Semantically, every type that can be duplicated via bit-wise copy is a *Copy* type.

- `&T` yes, because it's a shared pointer. `Int` yes, because it's a number.
- `&mut T` no, because it also holds exclusive access. `Vec<Int>` no, because it's a pointer to an heap array, and a bit-wise copy doesn't duplicate the underlying data.

Lifetimes

- **Ownership**: exclusive access, *mutation*
- **Mutable Reference**: *temporary* exclusive access, *mutation*
- **Shared Reference**: *temporary* shared access, *aliasing*

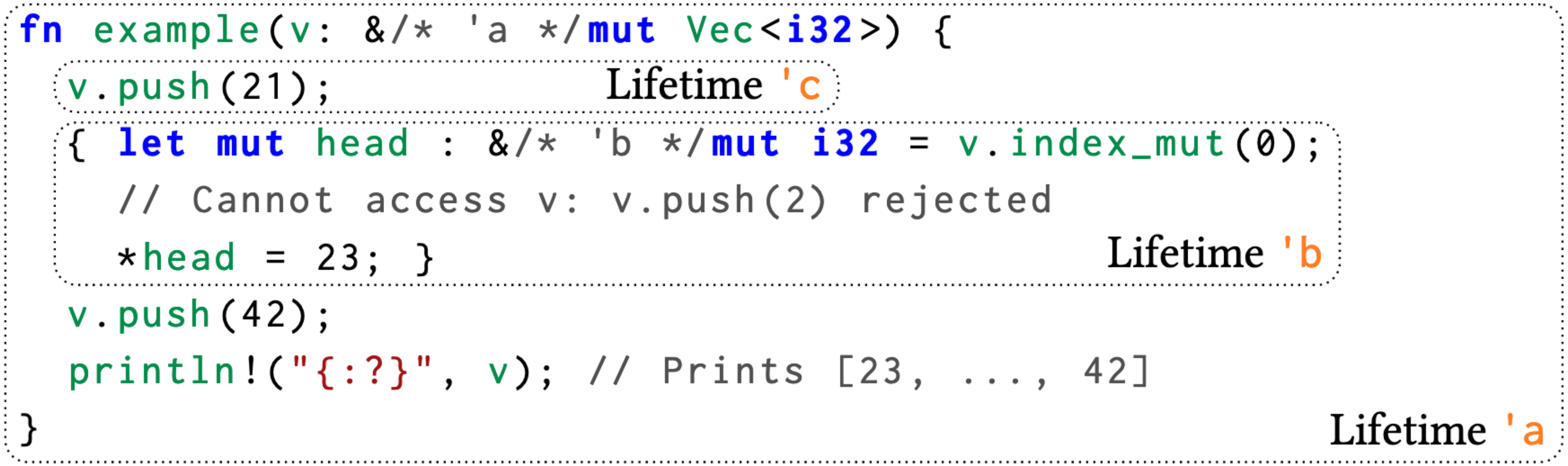
How to track if a reference is active? How long is *temporary*?

Answer: equip each reference with a *lifetime*.

```
&'a mut T   // mutable reference with lifetime 'a
&'b T       // shared reference with lifetime 'b
```

Lifetimes

`index_mut: for<'b> fn(&'b mut Vec<i32>, usize) -> &'b mut i32.`



```
fn example(v: &/* 'a */mut Vec<i32>) {
  v.push(21);
  { let mut head : &/* 'b */mut i32 = v.index_mut(0);
    // Cannot access v: v.push(2) rejected
    *head = 23; }
  v.push(42);
  println!("{:?}", v); // Prints [23, ..., 42]
}
```

- the output of `index_mut` has the same lifetime as the input.
- passing `v` to `index_mut`, we create a lifetime `'b` for `v` and `head`.
- to call `push` we need to create a mutable reference, whose lifetime overlaps with `'b`.

Interior Mutability

Q: What if we need shared mutable state? i.e. multi-thread queue?

A: Add primitives that allow *mutation* through *shared references*, i.e. *interior mutability*.

```
Cell::set(&Cell<T>, T)
Cell::get(&Cell<T>) -> T
```

```
let c1 : &Cell<i32> = &Cell::new(1);
let c2 : &Cell<i32> = &c1;
c1.set(2);
println!("{}", c2.get()); // 2
```

Oops! *Aliasing* and *mutation* at the same time!

`Cell` is implemented using **unsafe** code, i.e. opting *out* of the type system.

Interior Mutability

If you think about it, `Cell` is still safe to use.

```
Cell::set(&Cell<T>, T)
Cell::get(&Cell<T>) -> T
```

`Cell` can only hold *Copy* types, and returns a copy of the value when `get` is called.

No way to alias the inner data semantically!

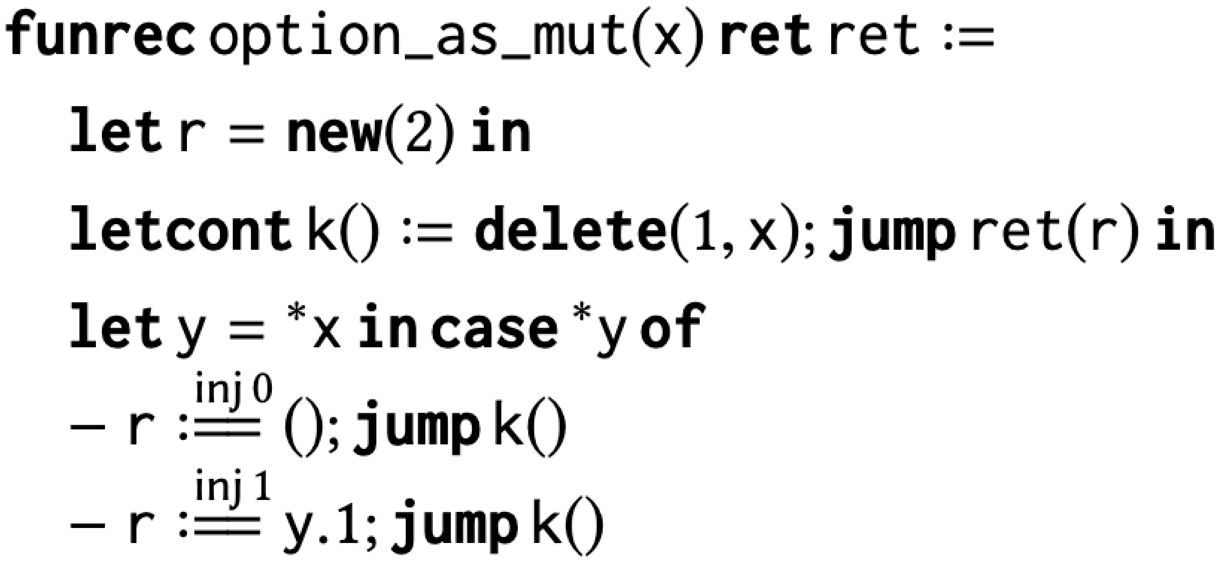
Formalization of Rust: Challenges

- Complex language: imperative, traits, …
- *Unsafe* types: opting out of syntactic typing rules

Challenge: complex language

Solution: work on a subset of Rust intermediate representation called λ_{Rust}.

```
fn option_as_mut<'a>
    (x: &'a mut Option<i32>) ->
    Option<&'a mut i32> {
  match *x {
    None => None,
    Some(ref mut t) => Some(t)
  }
}
```



funrec option_as_mut(x) **ret** ret :=

let r = **new**(2) **in**

letcont k() := **delete**(1, x); **jump** ret(r) **in**

let y = *x **in case** *y **of**

– r $\overset{\text{inj } 0}{:==}$ (); **jump** k()

– r $\overset{\text{inj } 1}{:==}$ y.1; **jump** k()

Type system of λ_{Rust}

Observation: local variables of Rust are also addressable.

Simplification: treat local variables as heap-allocated, i.e. *pointer* types.

- Primitives: `bool`, `int`
- Pointers:
  1. $\mathbf{own}\ \tau$: pointer with full ownership of an allocation containing a value of type τ
  2. $\&^{\kappa}_{\mathbf{mut}/\mathbf{shr}}\tau$: mutable/shared reference with lifetime κ to a value of type τ
- Other types: Π, Σ, $\rightarrow$, μ, ...

Note: Types of local variables of Rust programs are all *pointer* types.

Not describing in detail due to time limit.

Challenge: *unsafe* types

Unsafe types opts out of typing rules, so no way to prove safety from the rules!

Solution: take the *semantic* approach.

- **syntactic typing**: terms the typing rules allow to be of type τ
- **semantic typing**: terms that are safe to be treated as type τ

Semantic typing

What is a type? a certain set of values, or, a predicate on values.

Example: in lambda calculus with booleans,

- $[|\mathrm{Bool}|](v) := v = \mathrm{true} \lor v = \mathrm{false}.$
- $[|A * B|](v) := \exists v_1, v_2 .\, v = (v_1, v_2) \land [|A|](v_1) \land [|B|](v_2).$

Challenges to model Rust type system

- How to describe *ownership*?
- How to describe *temporary* access?
- How to deal with *interior mutability*?

Challenge: How to describe *ownership*?

What is a type? a certain set of values, or, a predicate on values.

What predicate? using which logic?

Separation Logic!

Separation Logic 101

A logic that describes a *heap*.

- $\mathtt{emp}$: empty heap
- $x \mapsto v$: heap with a single cell at address x containing value v
- $P * Q$: heap that can be *split* into two parts, one satisfying P and the other satisfying Q (like *conjunction*, but disjoint)
- $P \mathrel{-\!\!*} Q$: heap that, *disjointly* combined with another heap satisfying P, satisfies Q (like *implication*, but disjoint)

Separation Logic 101

Separation logic is a substructural logic.

Example: Consider the following heap: $x = 1$.
$x \mapsto 1$ holds, but $x \mapsto 1 * x \mapsto 1$ does not. Thus, no *contraction*.

Also, after an implication is applied to a value, the value is *consumed*.

Separation Logic 101

A logic that describes a *heap*.

Separation logic is a substructural logic.

- Rust types: a type represents ownership of a resource, and the type system is affine.
- Separation logic: a predicate represents a resource, and the logic is *affine*.

Perfect logic to describe Rust types!

P.S. 🤓👆 *Not every separation logic is affine, but the one used in Rustbelt, i.e. Iris, is.*

Interpreting Rust types: primitives

Associate every type τ to an *Iris* (separation logic) predicate on values.

$$[|\tau|].\mathrm{own} : \mathit{list}\ \mathit{Val} \to \mathit{Prop}$$

(Ignore why we name it "own" for now, will be explained later.)

- $[|\mathbf{bool}|].\mathrm{own}(\bar{v}) := \bar{v} = [\mathbf{true}] \vee \bar{v} = [\mathbf{false}]$
- $[|\tau_1 \times \tau_2|].\mathrm{own}(\bar{v}) := \exists \bar{v_1}, \bar{v_2}.\, \bar{v} = \bar{v_1} \mathbin{+\!\!+} \bar{v_2} * [|\tau_1|].\mathrm{own}(\bar{v_1}) * [|\tau_2|].\mathrm{own}(\bar{v_2})$

Notice: $*$ is *separating conjunction*, meaning its two oprands are disjoint in memory.

Interpreting Rust types: *Copy* types

Recall: types that can be freely duplicated via bit-wise copy are *Copy* types.

Consequence: given $[|\tau|].\mathrm{own}(\bar{v})$, we can freely duplicate the proposition, recovering contraction rule on the type.

Proposition that can be freely copied (i.e. $P \vdash P * P$*) is called a persistent proposition.*

Therefore, the interpretation of *Copy* types can always be written as:

$[|\tau|].\mathrm{own}(\bar{v}) := \exists v.\, \bar{v} = [v].\, *\Phi_\tau(v)$, where Φ_τ is a persistent proposition.

E.g. for $\tau = \mathbf{bool}$, $\Phi_{\mathbf{bool}}(v) := v = [\mathbf{true}] \vee v = [\mathbf{false}]$, which is trivially persistent because it's not describing any resource.

Interpreting Rust types: owned pointers

Associate every type τ to an *Iris* (separation logic) predicate on values.

$$[\![\mathbf{own}\ \tau]\!].\mathrm{own}(\bar{v}) := \exists \ell\,.\,\bar{v} = [\ell\,] * \exists \bar{w}.\,\ell \mapsto \bar{w} * [\![\tau]\!].\mathrm{own}(\bar{w})$$

- $\exists \ell\,.\,\bar{v} = [\ell\,]$: $\bar{v}$ contains a single address ℓ .
- $\ell \mapsto \bar{w}$: heap at address ℓ contains value $\bar{w}$.
- $[\![\tau]\!].\mathrm{own}(\bar{w})$: $\bar{w}$ can be seen as a value of type τ.

Notice: $*$ is *separating conjunction*, meaning location ℓ and memory region representing $\bar{w}$ are disjoint.

† Interpreting Rust types: owned pointers (for *Copy* types)

$$[|\mathbf{own}\ \tau|].\mathrm{own}(\bar{v}) := \exists \ell\,.\,\bar{v} = [\ell\,] * \exists \bar{w}.\,\ell \mapsto \bar{w} * [|\tau|].\mathrm{own}(\bar{w})$$

Recall: types that can be duplicated via bit-wise copy are *Copy* types.

Try to duplicate $[|\mathbf{own}\ \tau|].\mathrm{own}(\bar{v})$:

- $\exists \ell\,'.\,\bar{v} = [\ell\,']$: can always find another address $\ell\,'$. (assume no allocation failure)
- $\exists \bar{w}'.\,\ell\,' \mapsto \bar{w}'$: let $\bar{w}' = \bar{w}$ up to bit-wise copy.
- $[|\tau|].\mathrm{own}(\bar{w}')$: holds because τ can be duplicated by bit-wise copy.

Property: for any *Copy* type τ, predicate $[|\mathbf{own}\ \tau|](\bar{v})$ can be freely duplicated.

Interpreting Rust types: mutable references

What's the difference between *mutable references* and *owned pointers*?

- *owned pointers*: ownership for an unlimited time
- *mutable references*: ownership for *a limited period of time*

Challenge: how to describe *temporary* ownership?

Recall how we tracked references in Rust type system: *lifetimes*.

Solution: lifetime logic.

Full borrow predicate

P: separation assertion representing ownership of some resource
$\&_{\mathbf{full}}^{\kappa} P$: assertion representing ownership of P *during lifetime* κ

Intuition: P holds only when κ is active.

We'll head back to the precise definition of lifetime logic later.

Interpreting Rust types: mutable references

$\&^{\kappa}_{\mathbf{mut}}\tau$: mutable reference with lifetime κ to a value of type τ

Meaning: ownership of a value of type τ for the duration of lifetime κ.

- $[\![\mathbf{own}\ \tau]\!].\mathrm{own}(\bar{v}) := \exists \ell\,.\,\bar{v} = [\ell\,] * \exists \bar{w}.\,\ell \mapsto \bar{w} * [\![\tau]\!].\mathrm{own}(\bar{w})$
- $[\![\&^{\kappa}_{\mathbf{mut}}\tau]\!].\mathrm{own}(\bar{v}) := \exists \ell\,.\,\bar{v} = [\ell\,] * \&^{\kappa}_{\mathbf{full}}(\exists \bar{w}.\,\ell \mapsto \bar{w} * [\![\tau]\!].\mathrm{own}(\bar{w}))$

Interpreting Rust types: shared references

$$[\![\&^{\kappa}_{\mathbf{shr}}\tau]\!].\mathrm{own}(\bar{v}) :=?$$

Question: what can we say about shared references *universally*?

1. they are pointers
2. they are *Copy* types, i.e. can be freely duplicated
3. they can be created by downgrading a *mutable reference*
4. for *Copy* τ, we can bit-wise copy the value it points to and get a new τ

Not so interesting! Is that true?

Interior mutability!

How to deal with *interior mutability*?

Many types have their own sharing reference behavior deviating from the universal rules!

Solution: let every type define their own sharing reference behavior, i.e. *sharing predicate*.

- **owned predicate** $[|\tau|].\mathrm{own}(\bar{v})$: describe values $\bar{v}$ that can be considered as type τ
- **sharing predicate** $[|\tau|].\mathrm{shr}(\kappa, \ell\ )$: describe a location ℓ and lifetime κ to be considered as type $\&^{\kappa}_{\mathbf{shr}}\tau$

Leveraging the sharing predicate to describe the behavior of shared references.

$$[|\&^{\kappa}_{\mathbf{shr}}\tau|].\mathrm{own}(\bar{v}) := \exists \ell\ .\ \bar{v} = [\ell\ ] * [|\tau|].\mathrm{shr}(\kappa, \ell\ )$$

Interpreting Rust types: shared references

Leveraging the sharing predicate to describe the behavior of shared references.

$$[\![\&^{\kappa}_{\mathbf{shr}}\tau]\!].\mathrm{own}(\bar{v}) := \exists \ell\,.\,\bar{v} = [\ell\,] * [\![\tau]\!].\mathrm{shr}(\kappa, \ell\,)$$

Laws for sharing predicates:

1. ~~they are pointers~~: already satisfied by the definition of sharing predicate
2. they ~~are *Copy* types~~ can be freely duplicated: $[\![\tau]\!].\mathrm{shr}(\kappa, \ell\,)$ must be persistent.
3. they can be created by downgrading a *mutable reference*:
   $$[\![\&^{\kappa}_{\mathbf{mut}}\tau]\!].\mathrm{own}([\ell\,]) * [\kappa]_q \ \twoheadrightarrow\!\!*\ \ [\![\tau]\!].\mathrm{shr}(\kappa, \ell\,) * [\kappa]_q$$

$[\kappa]_q$ *is a token that asserts the lifetime* κ *is active, and we'll talk about it later.*

Interpreting Rust types: shared references

4. for *Copy* τ, we can bit-wise copy the value it points to and get a new τ.

Recall: for *Copy* types τ,

$[|\tau|].\,\mathrm{own}(\bar{v}) := \exists v.\, \bar{v} = [v].\, *\Phi_\tau(v)$, where Φ_τ is a persistent proposition.

Define:

$$[|\tau|].\,\mathrm{shr}(\kappa, \ell\ ) := \exists v.\, \&^{\kappa}_{frac}(\ell\ \mapsto^{q}\ v) * \Phi_\tau(v)$$

Interpreting Rust types: shared references

Define: for *Copy* types τ,

$$[\![\tau]\!].\mathrm{shr}(\kappa, \ell) := \exists v.\, \&^{\kappa}_{frac}(\ell \mapsto^{q} v) * \Phi_{\tau}(v)$$

Recall: for mutable references,

$$[\![\&^{\kappa}_{\mathbf{mut}} \tau]\!].\mathrm{own}(\bar{v}) := \exists \ell\,.\, \bar{v} = [\ell\,] * \&^{\kappa}_{\mathbf{full}}(\exists \bar{w}.\, \ell \mapsto \bar{w} * [\![\tau]\!].\mathrm{own}(\bar{w}))$$

Intuition: † *fractured borrow* $\&^{\kappa}_{frac} P$ also represents ownership P during lifetime κ, but:

- is *persistent*, because it represents a shared borrow, while full borrow is not
- only grants a fraction of its content ($\mapsto^{q}$)

†: *no need to understand the details. Just treat them as **full borrows**.*

Lifetime logic

Things we used but not defined yet:

- **Full borrow** $\&^{\kappa}_{\mathbf{full}}P$: assertion representing ownership of P *during lifetime* κ
- † **Fractured borrow** $\&^{\kappa}_{frac}P$: assertion representing ownership of P *during lifetime* κ, but only grants a fraction of its content
- **Lifetime token** $[\kappa]_q$: token that asserts the lifetime κ is active

Lifetime logic

```rust
let mut v = Vec::new();
v.push(0);
{ // <- Vec<i32>
  let mut head = v.index_mut(0);
  *head = 23;
}
println!("{:?}", v);
```

given that

```rust
index_mut: for<'a> fn(&'a mut Vec<i32>, usize) -> &'a mut i32
```

Lifetime logic

```
index_mut: for<'a> fn(&'a mut Vec<i32>, usize) -> &'a mut i32
```

```
{
  let mut head = v.index_mut(0); // <- Vec<i32>
```

- need to provide `'a`
- need to pass a mutable reference of lifetime `'a`

Lifetime logic

```
index_mut: for<'a> fn(&'a mut Vec<i32>, usize) -> &'a mut i32
```

```
{
  let mut head = v.index_mut(0); // <- Vec<i32> ∗ [κ] ∗ ([κ] −∗ [†κ])
```

- need to provide `'a`

  $\textsc{LftL-begin}$: $\texttt{True} \twoheadrightarrow \exists \kappa.\ [\kappa]_1 * ([\kappa]_1 \twoheadrightarrow [\dagger\kappa])$

  Can always

  - create a lifetime token $[\kappa]_1$, accompanied by
  - a way to end it $[\kappa]_1 \twoheadrightarrow [\dagger\kappa]$. ($[\dagger\kappa]$ is a token that asserts the lifetime κ has ended.)

Lifetime logic

```
index_mut: for<'a> fn(&'a mut Vec<i32>, usize) -> &'a mut i32
```

```
{
  let mut head = v.index_mut(0); // <- &κ mut Vec<i32> ∗ [κ] ∗ ([†κ] −∗ Vec<i32>) ∗ ([κ] −∗ [†κ])
```

- need to provide `'a` (done)
- need to pass a mutable reference of lifetime `'a`

$$\textsc{LftL-borrow}: P \twoheadrightarrow\!\!\ast\; \&^{\kappa}_{\mathbf{full}} P \ast ([\dagger\kappa] \twoheadrightarrow\!\!\ast\; P)$$

Given an owned resource P, can split it into

  - a full borrow $\&^{\kappa}_{\mathbf{full}} P$, and
  - an *inheritance* $[\dagger\kappa] \twoheadrightarrow\!\!\ast\; P$ that can retrieve P back after κ dies.

Lifetime logic: separating conjunction

$\textsc{LftL-begin}$: $\texttt{True} \;\twoheadrightarrow\!\!*\; \exists\kappa.\, [\kappa]_1 * ([\kappa]_1 \twoheadrightarrow\!\!* [\dagger\kappa])$

$\textsc{LftL-borrow}$: $P \;\twoheadrightarrow\!\!*\; \&^{\kappa}_{\mathbf{full}} P * ([\dagger\kappa] \twoheadrightarrow\!\!* P)$

- Sep logic $P * Q$: *heap* that can be *split* into two *disjoint* parts, one satisfying P and the other satisfying Q
- Lifetime logic $P * Q$: *time* that can be *split* into two *disjoint* parts, one satisfying P (when κ is alive) and the other satisfying Q (when κ is dead)

Lifetime logic: frame rule

It's important for P and Q to be *disjoint*.

Consider $P \land Q$ and $P * Q$.

$$\frac{P \vdash P' \quad Q \vdash Q'}{P * Q \vdash P' * Q'} \quad \text{(can be seen as)} \quad \frac{\forall x, \{P(x)\}\, c\, \{P'(x)\} \quad \forall x, \{Q(x)\}\, c\, \{Q'(x)\}}{\forall x, \{(P * Q)(x)\}\, c\, \{(P' * Q')(x)\}}$$

But for $P \land Q$,

$$\frac{\forall x, \{P(x)\}\, c\, \{P'(x)\} \quad \forall x, \{Q(x)\}\, c\, \{Q'(x)\}}{\forall x, \{P(x) \land Q(x)\}\, c\, \{?\}}$$

What if P and Q describes some shared resource, and while $P \vdash P'$, c modifies something that invalidates Q?

Lifetime logic: separating conjunction

$\textsc{LftL-begin}$: $\texttt{True} \twoheadrightarrow\!\!\ast\ \exists\kappa.\ [\kappa]_1 \ast ([\kappa]_1 \twoheadrightarrow\!\!\ast [\dagger\kappa])$

$\textsc{LftL-borrow}$: $P \twoheadrightarrow\!\!\ast\ \&^{\kappa}_{\mathbf{full}} P \ast ([\dagger\kappa] \twoheadrightarrow\!\!\ast P)$

Whatever we do about $\&^{\kappa}_{\mathbf{full}} P$, we can always get back the *inheritance*.

Lifetime logic

```
index_mut: for<'a> fn(&'a mut Vec<i32>, usize) -> &'a mut i32
```

```
{
  let mut head = v.index_mut(0); // <- inside `index_mut`
```

1. split input `&κ Vec<i32>` into the accessed `&κ i32` and the rest `&κ Vec<i32>`
2. return `&k i32` to the caller, and drop the rest

Lifetime logic

```
index_mut: for<'a> fn(&'a mut Vec<i32>, usize) -> &'a mut i32
```

```
{
  let mut head = v.index_mut(0); // <- inside `index_mut`
```

1. split input `&κ Vec<i32>` into the accessed `&κ i32` and the rest `&κ Vec<i32>`
   $\textsc{LftL-bor-split}$: $\&^{\kappa}_{\mathbf{full}}(P * Q) \vdash \&^{\kappa}_{\mathbf{full}} P * \&^{\kappa}_{\mathbf{full}} Q$
2. return `&κ i32` to the caller, and drop the rest
   $P * Q \vdash P$, because *Iris* is an affine logic

```
{
  let mut head = v.index_mut(0); // <- &κ mut i32 ∗ [κ] ∗ ([†κ] −∗ Vec<i32>) ∗ ([κ] −∗ [†κ])
```

Lifetime logic

```
let mut head = v.index_mut(0);
*head = 23;   // <- i32 ∗ (i32 −∗ &κ mut i32 ∗ [κ]) ∗ ([†κ] −∗ Vec<i32>) ∗ ([κ] −∗ [†κ])
```

Need to access the resource of mutable reference `head`.

$\textsc{LftL-bor-acc}$: $\&^{\kappa}_{\mathbf{full}} P * [\kappa]_q \;\; -\!\!* \;\; P * (P -\!\!* \&^{\kappa}_{\mathbf{full}} P * [\kappa]_q)$

Given a full borrow $\&^{\kappa}_{\mathbf{full}} P$ and a witness $[\kappa]_q$ that shows κ is active,

- can access the resource P, accompanied by
- an *inheritance* $P -\!\!* \&^{\kappa}_{\mathbf{full}} P * [\kappa]_q$ that can retrieve mutable reference and *lifetime token back* after the access

It's important to return things you borrowed!: *lifetime token* is such a certificate.

Lifetime logic

```
  *head = 23;   // <- &κ mut i32 * [κ] * ([†κ] -* Vec<i32>) * ([κ] -* [†κ])
}
```

```
  *head = 23;
}               // <- [κ] * ([†κ] -* Vec<i32>) * ([κ] -* [†κ])
```

```
  *head = 23;
}               // <- ([†κ] -* Vec<i32>) * [†κ]
```

```
  *head = 23;
}               // <- Vec<i32>
```

† Lifetime logic

Fractured borrow $\&^{\kappa}_{frac}$ vs **Full borrow** $\&^{\kappa}_{\mathbf{full}}$

- **Fractured borrow**s are persistent: can be accessed simultaneously by multiple parties (freely duplicatable), but do not have full access, i.e. only a fraction of the resource.
- It's always possible to take a little bit of a resource from a **Fractured borrow**, no matter how many times it's been borrowed.

Intuition:

- from a full borrow with full lifetime $[\kappa]_1$, by downgrading it to a fractured borrow, we can get a fraction of it, thus getting fractional lifetime $[\kappa]_q$, e.g. $[\kappa]_{0.1}$, which is shorter than $[\kappa]_1$.
- The semantics guarantees that we can always get a tiny bit of resource of lifetime $[\kappa]_\epsilon$ from a fractured borrow.

Proof of soundness

Typing judgments are defined as

$$\mathbf{L}|\mathbf{T} \vdash \mathrm{I} \dashv x.\,\mathbf{T}'$$

- $\mathbf{L}$ lifetime context
- $\mathbf{T}$ type context
- I instruction

After the instruction, the type context is updated to $\mathbf{T}'$ with new variable x added.

Proof of soundness

Interpretation of typing judgments:

$$\mathbf{L}|\mathbf{T} \vDash \mathrm{I} \dashv\ x.\, \mathbf{T}' := \{[|\mathbf{L}|]_\gamma * [|\mathbf{T}|]_\gamma\}\ \mathrm{I}\ \{\exists v.\ [|\mathbf{L}|]_\gamma * [|\mathbf{T}'|]_{\gamma[x \leftarrow v]}\}$$

- Interpreted as a separation logic triple
- $[|\mathbf{T}|]$ uses interpretation of types described earlier

Proof of soundness

1. **FTLR** (Foundamental Theorem of Logical Relations):
   $\forall \mathbf{L}, \mathbf{T}, \mathrm{I}. \quad \mathbf{T}'. \mathbf{L} | \mathbf{T} \vdash \mathrm{I} \dashv x. \mathbf{T}' \Rightarrow \mathbf{L} | \mathbf{T} \vDash \mathrm{I} \Dashv x. \mathbf{T}'$
   Syntactic typing rules are sound w.r.t. semantic typing rules.
2. **Adequacy**: a semantically well-typed program never gets stuck (no invalid memory access or data race).

Collary: every rust program that consists of *syntactically* well-typed *safe* code and *semantically* well-typed *unsafe* code, is safe to execute.

Conclusion

- Rust type system: ownership, mutable/shared references, lifetime, interior mutability
- Formalization: λ_{Rust}, $\mathbf{own}\ \tau$, $\&^{\kappa}_{\mathbf{mut}/\mathbf{shr}}$. Unsafe types? *Semantic typing*!
- Semantic typing:
  - Separation logic
  - $[|\tau|].\mathrm{own}(\bar{v})$, $[|\tau|].\mathrm{shr}(\kappa, \ell\ )$ (for interior mutability)
  - $\&^{\kappa}_{\mathbf{full}}P$, $\&^{\kappa}_{frac}P$, $[\kappa]_q$? *Lifetime logic*!
- Lifetime logic by example
  - Fractured borrow: persistent + fractional (inclusion) lifetime
- Soundness proof:
  - Judgment interpreted as separation logic triple
  - FTLR (syntactic -> semantic) + Adequacy (semantic -> runtime)

Appendix: Lifetime logic meets Interior Mutability

Example: Mutex is a product of flag (true: locked, false: unlocked) and the resource.

$$[\![\mathbf{mutex}(\tau)]\!].\mathrm{own}(\bar{v}) := [\![\mathbf{bool} \times \tau]\!].\mathrm{own}(\bar{v})$$

$$\begin{aligned}
&[\![\mathbf{mutex}(\tau)]\!].\mathrm{shr}(\kappa, \ell) := \&^{\kappa}_{\mathbf{atom}}(\\
&\quad \ell \mapsto \mathbf{true} \vee \\
&\quad \ell \mapsto \mathbf{false} * \&^{\kappa}_{\mathbf{full}}(\exists \bar{v}.\, (\ell + 1) \mapsto \bar{v} * [\![\tau]\!].\mathrm{own}(\bar{v})) \\
&)
\end{aligned}$$

Atomic persistent borrow $\&^{\kappa}_{\mathbf{atom}} P$: assertion representing ownership of P that *cannot be accessed for longer than one single instruction cycle*. Can be freely duplicated.

Appendix: Lifetime logic meets Interior Mutability

Example: Mutex is a product of flag (true: locked, false: unlocked) and the resource.

$$
\begin{aligned}
&[\![\mathbf{mutex}(\tau)]\!].\mathrm{shr}(\kappa, \ell\,) := \&^{\kappa}_{\mathbf{atom}}(\\
&\quad \ell\, \mapsto \mathbf{true} \vee \\
&\quad \ell\, \mapsto \mathbf{false} * \&^{\kappa}_{\mathbf{full}}(\exists \bar{v}.\, (\ell\, + 1) \mapsto \bar{v} * [\![\tau]\!].\mathrm{own}(\bar{v})) \\
&)
\end{aligned}
$$

Atomic persistent borrow $\&^{\kappa}_{\mathbf{atom}} P$: assertion representing ownership of P that *cannot be accessed for longer than one single instruction cycle*. Can be freely duplicated.

- When unlocked, one thread borrows it, takes its inner full borrow away, and set lock flag. Other threads can't observe an intermediate state due to atomicity.
- Later, another thread tries to borrow it, but the lock flag is set.
- When the first thread releases the lock, it put back the full borrow so another thread can use it.