The original idea for this talk was "Local Reasoning in Any Language" to document how I think about code, regardless of the language I'm programming in. The talk gets mired in language details, so I'm doing multiple versions; the first was for C++, and this is the second for Rust. Except for the details, the rules in this talk apply to all languages. I present here how I map these ideas into Rust, and if you program in a different language, figure out a set of conventions to map the ideas into that language.

Some Links

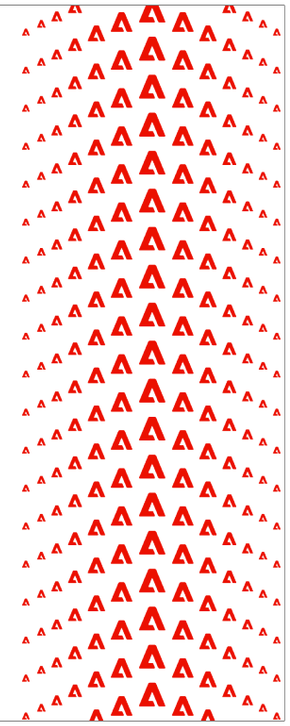
- https://sean-parent.stlab.cc/papers-and-presentations/
- https://www.hylo-lang.org/

- Local Reasoning Can Help Prove Correctness
- https://accu.org/journals/overload/33/188/teodorescu-parent/

The first link is to my vanity page - you'll find links to nearly all of my talks (at least all the ones I've been able to hunt down). This talk will be posted there soon.

Lucian Teh-oh-doh-RES-koo

Why Software Projects Fail



That is a big topic. I spent significant time looking for engineering analysis about why software projects fail. There are a _lot_ of management analyses, but I could not find a single paper with an engineering analysis. I will argue there is a specific point where engineering will, and often does, fail.

The failure occurs at the point where we lose the ability to reason _locally_ about code.

Why Software Projects Fail

"The greatest limitation in writing software is our ability to understand the systems we are creating."

– *A Philosophy of Software Design*, John Osterhaut

At some point, our ability to reason about the system breaks down. We can no longer understand what we are building or what effect a change will have, and we lose sight of the goal in the details of the code. Who here has worked on a software project that failed? Who has worked on a project where they felt lost? Uncertain of the effect any change in the code will have. Photoshop has over 30M lines of code... I can tell you that the engineering team feels uneasy every day.

Local Reasoning

Local reasoning *is the idea that the reader can make sense of the code directly in front of them, without going on a journey discovering how the code works.*

—Nathan Gitter

- The two units of code this talk is concerned with are:
  - Functions
  - Classes
- The API is the key to local reasoning

The solution is to construct systems that can be reasoned about locally. This is a talk about a practical approach to writing correct code. Formal methods mainly focus on how you prove the code you write is correct - here we focus on constructing correct code. STLab is looking to build a culture of correctness at Adobe. Local reasoning is a key part of that.

Functions

```
fn f() -> ()
```


Let's start with a simple function interface. <click>
Either this function does nothing, or whatever it does is entirely through side effects. Either way, we should document it. <click>
Now we can implement `f` <click>

Functions

```
/// Does nothing.
fn f() -> ()
```


Let's start with a simple function interface. <click>
Either this function does nothing, or whatever it does is entirely through side effects. Either way, we must document it. <click>
Now we can implement `f` <click>

Functions

```
/// Does nothing.
fn f() -> () {}
```


I hope everyone is convinced that `f()` is implemented correctly.

A specification, or contract, is a requirement for local reasoning. Without it, we cannot know what the operation is supposed to do or what a class represents.
Again: a contract is a requirement for local reasoning.

Suppose a piece of code has a contract, and everything it invokes also has one. In that case, we can read the implementation and verify that the function body is correct and fulfills the contract.

Let's make our function a little more complicated<click>

Contracts

- *Precondition*: A condition that must be true before an operation is called
- *Postcondition*: A condition that must be true after an operation, or the operation must result in an *error*
  - If the precondition is not satisfied, the postcondition is unspecified
- *Type Invariant*: A condition that must be true when an instance of a type is created, and is a postcondition of any mutating operation on the instance

The idea of contracts comes from Bertrand Meyer's Design by Contract. Contracts allow us to reason about code locally and are fundamental to local reasoning.

Functions

```
/// Returns the successor of `x`.
fn f(x: i32) -> i32 { x + 1 }
```


Still very simple, this code is easy to understand at a glance. It doesn't have a great name—but it does what the specification says.
But there is an issue. Do you see it? What happens if there is no successor of `x`?

The rule of contracts is that if preconditions are satisfied, and the function cannot satisfy the postconditions, the function must return an error. We can argue if the precondition here is implied by the specification, but let's make it explicit.

Functions

```
/// Returns the successor of `x`.
/// Precondition: `x < i32::MAX`.
fn f(x: i32) -> i32 { x + 1 }
```


The contract provides the scaffolding for us to reason about the function locally. The process is straightforward; for each operation, we need to ensure that the preconditions are satisfied by the arguments or the prior operation. The arguments play a key role in our ability to reason about an operation, so let's break down the different kinds of arguments.

Function Arguments

Function Arguments

```
/// Increments the value of `x` by 1.
/// Precondition: `x < i32::MAX`.
fn a(x: &mut i32) {
    *x += 1;
}
```


Here is the same operation but written as an _action_ that mutates the argument.
This function is still simple; is it correct?

In most other languages, this would add the precondition that no other thread is accessing `*x` when we update it. That would be a data race. The Rust borrow-checker verifies this implicit precondition so it doesn't need to be stated.

Rust disallows mutation under aliasing.
<click>

Rust Borrow Checker:

- Arguments passed to a function by mutable reference cannot be accessed by other threads or indirectly through another operation.
- Arguments passed to a function by reference cannot be written to, either directly or indirectly, for the duration of the call.

Why Mutation?

- Mutation is space efficient
- Mutation is *often*:
  - more performant.
  - simpler to reason about.

How would I replace the battery in my car, compared to rebuilding an equivalent vehicle with a new battery?

In situ operations are more space-efficient and may be more time-efficient.

Argument Qualifiers

- *let* arguments
  - `&T` (or `T` if it conforms to `Copy`)
- *inout* arguments
  - `&mut T`
- *sink* arguments
  - `T` (unless `T` conforms to `Copy`)

I'm borrowing the terminology for arguments from Hylo because Rust doesn't have a good nomenclature for them.

In Rust, if an object can be copied (as opposed to cloned), the copy operation is as inexpensive as consumption, and so the value is not consumed. This means that pass by value may mean "let" or "sink" depending on the type.

Argument Guarantees

- *let* arguments
  - Postcondition: The argument is not modified
- *inout* arguments
  - Postcondition: None (The argument may be modified)
- *sink* arguments
  - Postcondition: The argument is consumed

sink arguments are used when the argument is escaped - either stored or returned, possibly with modification.

A more complex action

```
/// Offsets the value of `x` by `n`.
/// Precondition: `(x + n) < INT_MAX`.
fn offset(x: &mut i32, n: &i32) {
  *x += n;
}
```

- What if this is called as:

```
let x = 2;
offset(&mut x, &x);

println!("{x}");
```


It "feels like" this should print 4.

A more complex action

```
error[E0502]: cannot borrow `x` as immutable because it is also
borrowed as mutable
   |
   |     offset(&mut x, &x);
   |     ------ ------  ^^ immutable borrow occurs here
   |     |      |
   |     |      mutable borrow occurs here
   |     mutable borrow later used by call
```


The postconditions for `x` and `n` conflict. x cannot both be unchanged and be the offset value.

A more complex action

```
let mut x = 2;
x += x;

println!("{x}");
```

4

This will print 4

A more complex action

```
let mut x = 2;
let x_ref = &mut x;
*x_ref += *x_ref;

println!("{x_ref}");
```

4

This will print 4

A more complex action

```
let mut x = 2;
let x_ref = &mut x;
*x_ref += *x_ref;

println!("{x_ref}");
```

4

The star is required because AddAssign is implemented for both reference and values on the right-hand side. The `*` forces the value case and does an implicit copy.

Adding the `*` in our offset function wouldn't help - because we break at the interface level.

A more complex action

```
// Offsets the value of `x` by `n`
// Precondition: `(x + n) < INT_MAX`
fn offset(x: &mut i32, n: &i32) {
    for _ in 0..*n {
        *x += 1;
    }
}
```

- What will this print?

```
let x = 2;
offset(&mut x, &x);

println!("{x}");
```


Consider this implementation. This is also a correct implementation.
If it weren't for the borrow checker, what would this print?
It would not print 4 - it would never terminate.

A more complex action

```
vector a{ 0, 1, 1, 0 };
erase(a, a[0]);
println("{}", a);
```

- What will this print?

[1, 1]
  or
[1, 0]

– *https://godbolt.org/z/z381aM1r7*

I'm showing the C++ code for this example to make a point, this is the only C++ code in this talk. The intention of this code is to erase all elements in a vector matching the first element. That is "erase all `0`s"
What will this print... <click>

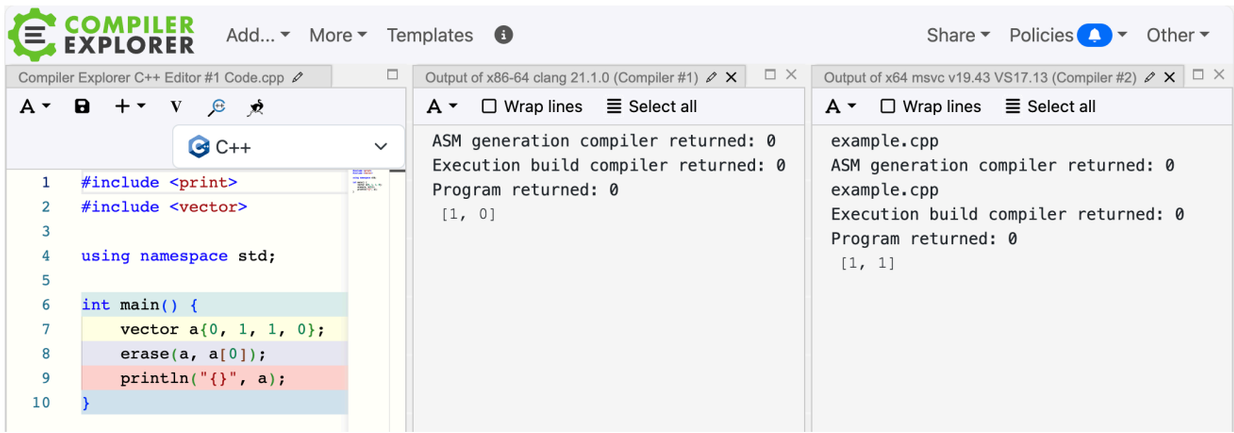
We would expect this, and maybe that is what it prints. But it might also print this <click>

Why? After the code removes the first element matching a[0] (0), a[0] holds a 1, so the remaining 1s are removed, leaving the trailing 0.

It is because of a clause in the standard that allows, but does not require, an operation to make a copy of its arguments, that the first answer is possible.
Both implementations conform to the standard.

If arguments are mutated under aliasing, local reasoning is broken for both the client and implementor.

A more complex action



Here is the code with two implementations of the standard library, each producing a different result.

A more complex action

```
vector a{ 0, 1, 1, 0 };
erase(a, int{a[0]});
println("{}", a);
```

- What will this print?

[1, 1]

– *https://godbolt.org/z/PYdj3W5Ma*

To get a defined result, we need to remove the aliasing by forcing a copy of `a[0]`.

C++ is littered with aliasing and lifetime-of-reference issues foot-guns.

A more complex action

```
let mut a = vec![0, 1, 1, 0];
a.retain(|x| *x != a[0]);
println!("{:?}", a);
```

- What will this print?

```
error: cannot borrow `a` as mutable because it is also borrowed as immutable
    |
    |     a.retain_mut(|x| *x != a[0]);
    |     ^^----------^---^^^^^^^-^^^^
    |     | |          |        |
    |     | |          |        first borrow occurs due to use of `a` in
closure
    |     | |          immutable borrow occurs here
    |     | immutable borrow later used by call
    |     mutable borrow occurs here
```


Here is the equivalent code in Rust - the Rust borrow checker prevents this code from compiling.

A more complex action

```
let mut a = vec![0, 1, 1, 0];
let first = a[0];
a.retain(|x| *x != first);
println!("{:?}", a);
```

- What will this print?

[1, 1]

We have to explicitly make a named copy in Rust.

Rust Borrow Checker:

- Referenced objects must be within the objects lifetime
- *inout* and *sink* arguments cannot be accessed except directly by the callee
- *let* arguments passed by reference cannot be mutated by either the caller or indirectly by the callee

The Rust borrow checker provides strong guarantees for references.

Law of Exclusivity

A variable cannot be accessed via a different name for the duration in which the same variable is being modified as an `inout` argument.

– *Swift 5 Exclusivity Enforcement*

In Swift, this is known as "The Law of Exclusivity", a term coined by John McCall.

Law of Exclusivity

If you have a mutable reference to a value, you can have no other references to that value.

– *The Rust Programming Language: References and Borrowing*

In Rust, the borrow checker enforces this restriction. C++ does not have such a restriction. We must rely on conventions and diligence. This phrasing is actually wrong - you can have all the references, you just can't use them.

The Law of Exclusivity is a General Law of Programming

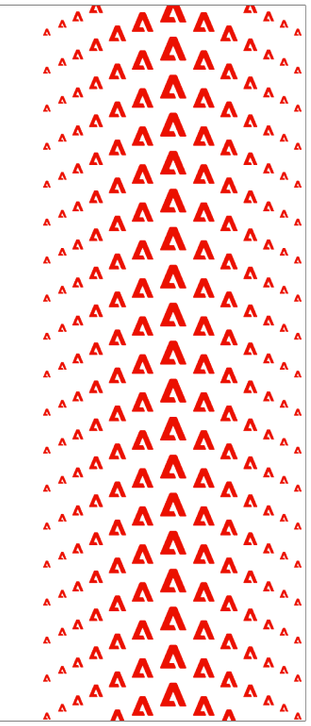
- In every mainstream language other than Rust and Swift, upholding the LoE is the programmer's responsibility

To reason about a function locally, you have to know that values don't change as a side effect of other operations.

What would you say in the contract? The documentation for C++ std::erase() is the implementation. This is not local reasoning.

To reason about a function locally, you have to know that things don't aliasing - what would you say in the contract?

Return Values



We haven't talked about function results yet.

Return Values

```
// Returns the successor of `x`.
// Precondition: `x < i32::MAX`
fn f(x: i32) -> i32 {
    x + 1
}
```


Let's go back to an early simple function. Here, we are returning a new value. Would it ever make sense to return a reference from a function?

Return-by-reference

```
let mut a = vec![0, 1, 2, 3];
*a.last_mut().unwrap() = 42;

println!("{:?}", a);
```

[0, 1, 2, 42]

Vec::last_mut() is an example of returning an optional reference. There are many examples of returning references in the standard library, including indexing, slices, and iterators.

When we return a reference, or anything containing a reference, we call that a _projection_.

Projection Qualifiers

- Projections qualifiers mirror argument qualifiers
  - *Constant* (&T) projections do not allow the projected object to be modified
  - *Mutable* (`&mut T`) projections allow the projected objects to be modified
- Returning by value is the mirror of *sink* arguments
  - *Value* (T) allows the object to be consumed

The fact that projection qualifiers mirror argument qualifiers is not a coincidence -
- Returning a constant projection pairs with a Let argument
- Returning a mutable projection pairs with an InOut argument
- Returning a value pairs with pairs with a sync argument

Projection Validity Enforced By Borrow Checker

- A projection is invalidated when:
  - The source of the projection is modified or destroyed

```
let mut a = vec![0];
let p = &a[0]; // p is a projection
a.push(1);     // ERROR
```

  - The lifetime of the object they are projected from ends

```
fn f() -> &String {
    let s = String::from("Hello");
    &s // ERROR
}
```


A mutation of the part is a mutation of the whole - unless you have special knowledge.

Projecting Multiple Values

- Iterators and slices may project a collection of values from an object
- They follow the same rules as reference projections

```
let mut a = vec![3, 2, 1, 0];
let src = &a[0..2];
let dst = &mut a[2..]; // ERROR
dst.copy_from_slice(src);

let mut a = vec![3, 2, 1, 0];
let (left, right) = a.split_at_mut(2); // Ensure non-overlapping
right.copy_from_slice(&left);
```


Let's say we wanted to copy the first half of a vector over the second. The obvious way runs into the borrow checker.

To mutate sections of a whole, we may need operations like `split` to convince the borrow-checker that the projections do not overlap. The borrow-checker knows that members do not overlap. This allows us to parallelize operations on non-overlapping components.

Lifetime Annotations

- Mostly inferred
  - Use `'a` for projection objects

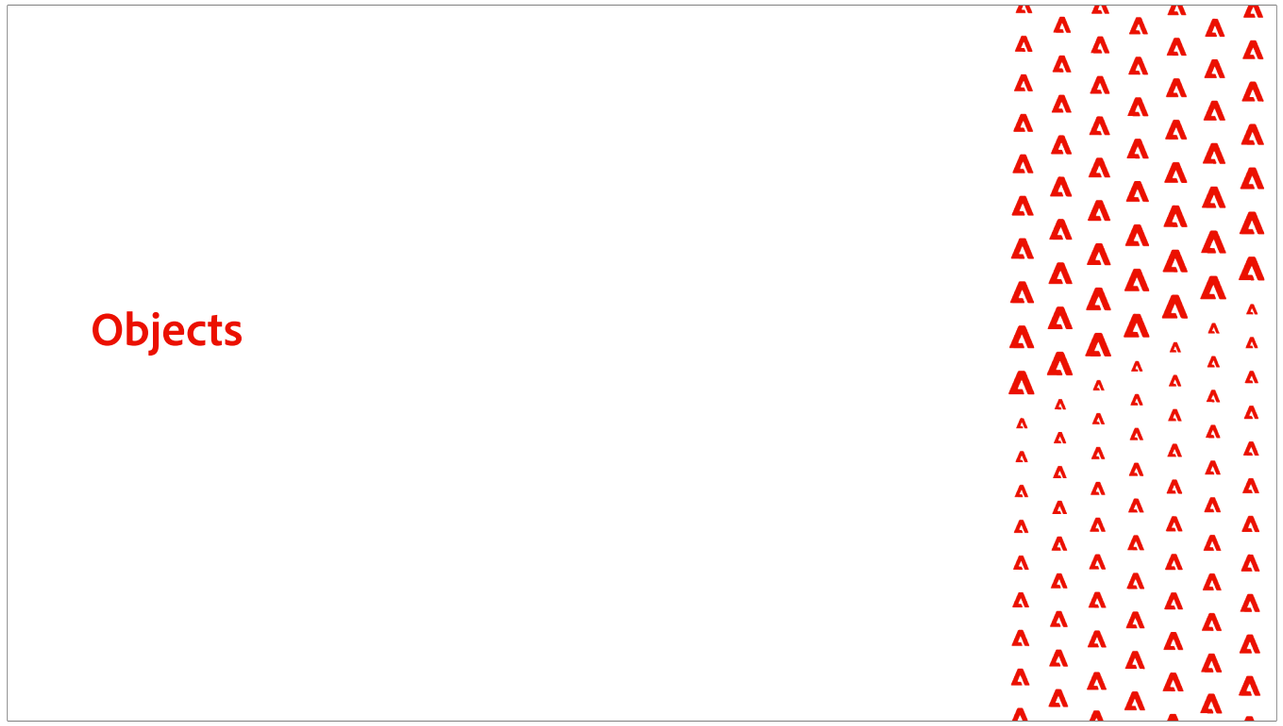
```
struct EveryNth<'a, T> { slice: &'a [T], n: usize, i: usize }

impl<'a, T> Iterator for EveryNth<'a, T> {
    type Item = &'a T;
    fn next(&mut self) -> Option<Self::Item> {
        (self.i < self.slice.len()).then(|| {
            let item = &self.slice[self.i];
            self.i += self.n;
            item
        })
    }
}
```


Lifetime annotations are mostly inferred; you may need them if you create an object as a projection. <click>

Here is a very terse example of an iterator that returns every Nth element from a slice: the lifetime annotations or the bold `'a`s.

Objects



Argument independence allows us to reason about a function in isolation, but for that to work, our objects must be independent.

Objects

```
fn f(a: &mut Box<Widget>) -> ();
```

- What is the *type* of the argument for `f()`?

- To understand `f()` we need to understand the intended *extent* of `a`

This seems like a ridiculous question - of course, the type is a mutable reference to a Box<Widget>!
But is the thing being modified the Box or the Widget?

The rules in Rust around references mostly prevent us from creating objects with unknown extent - but will still need to be clear in our interfaces. Pass the narrowest extent possible for the operation - avoid Optional, Box, Arc, Result, and even Vec in interfaces. If "Box<Widget>" _is_ the whole that is being operated on, then it should be a separate type, encapsulating the Box.

Composite Objects and Whole-Part Relationships

- A *composite object* is made up of other objects, called its *parts*.
- The whole–part relationship satisfies the four properties of *connectedness*, *noncircularity*, *disjointness*, and *ownership*

```
let a = vec![0, 1, 2, 3];

struct Person {
    name: String,
    id: i32,
}
let b = Person {
    name: "John".to_string(),
    id: 0,
};
```


a is a composite object with 4 integer part
b is a composite object with two named parts

Disjointness - logically disjoint under mutation from other objects - not necessarily other parts. Immutable and copy-on-write objects may share storage.

Pointers, shared, unique or otherwise, witness a relationship. Which may, or may not, be a whole-part relationship. In an interface, their meaning is ambiguous, and they are best avoided. Alone, they are disconnected from any whole.

Objects

```
fn a(p: &mut Widget) { ... }
```

- This should only modify an instance of `Widget`
- And could be expressed as:

```
fn f(p: Widget) -> Widget { ... }
```


In general, I prefer the sink/return-by-value form over mutation for composition and readability. But Rust lacks an in-place transform for members or parts.
But we need to talk about non-whole part relationships.

Objects, Copies, and Argument Independence

- To reason locally:
  - Objects used as arguments must be independent under mutation to uphold the Law of Exclusivity
  - Copies are equal and logically disjoint

[Say more here... To reason locally...]

Achieving Independence

- No mutation
- No sharing
- Copy-on-write (mutation un-shares)
- Borrowing (no mutation if shared)

No mutation is the functional programming approach.

Static single assignment (SSA) is the no-sharing approach

Swift relies heavily on copy-on-write. It uses function bundles to associate a transform and action. When modifying an object, if the object is uniquely owned, the action is used. Otherwise, the transform is used, and the "copy" is free. For example, if we have a copy-on-write dynamic array that is shared, and we insert an element, instead of copying, then inserting, we copy up to the insertion point, then move in the elements to be inserted, the copy the remaining elements.

Rust relies on borrow to (mostly) statically ensure. Rust also has the Cow type for copy-on-write.

Extending Independence with Mutation

- A mutable object may extend permission for mutation to its parts through projections
  - So long as those projections do not overlap

In Rust this is done with ownership and borrowing, in Swift (and Hylo), this is done through whole/part relationships and projections. The advantage of Hylo is fewer annotations.
In C++, the developer must manage projections. In reference-semantic languages (Jave, JavaScript, Python...), immutability is the only scalable option.

whole/part examples

```
#[derive(Debug, Clone, PartialEq)]
pub struct Whole {
    part: Part,
}

impl Whole {
    pub fn new(state: State) -> Self {
        Self {
            part: Part::from(state),
        }
    }
}
```


This is the canonical template for a class. Well-behaved parts compose into well-behaved wholes. Copy and equality are part-wise. We don't need a default constructor unless there is a meaningful default value.

whole/part examples

```
#[derive(Debug, Clone, PartialEq)]
pub struct Whole {
    shared_part: Arc<Part>,
}

impl Whole {
    pub fn new(state: State) -> Self {
        Self {
            shared_part: Arc::new(Part::from(state)),
        }
    }
}
```


The parts may be shared (immutable in Rust)

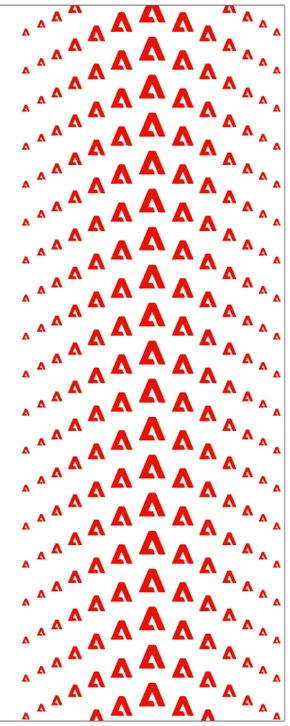
whole/part examples

```
#[derive(Debug, Clone, PartialEq)]
pub struct Whole {
    remote_part: Box<Part>,
}

impl Whole {
    pub fn new(state: State) -> Self {
        Self {
            remote_part: Box::new(Part::from(state)),
        }
    }
}
```


The parts may be heap allocated...

Errors & Safety



But there is more to a system than just a bunch of objects - the objects are often somehow *related*.

Errors

- *Postcondition*: A condition that must be true after an operation, **or the operation must result in an *error***
  - If the precondition is not satisfied, the postcondition is unspecified
- *Type Invariant*: A condition that must be true when an instance of a type is created, and **is a postcondition of any mutating operation** on the instance

These are general rules - any particular operation may provide stronger guarantees

Errors

- An object under mutation when an error occurs is left in an unspecified state
  - The class invariants may not hold
- Any object under mutation when an error occurs must be discarded

These are general rules - any particular operation may provide stronger guarantees
By dropping or replacing, as if by assignment, with a known value

Error Handling Composes

- Local objects are discarded during propagation
- Discarding *inout* arguments is the responsibility of the caller
- We only need to be concerned with objects passed as inout arguments to a failed operation when error propagation stops

```
fn a(&mut self) -> Result<()> {
    let mut x = f()?.g()?;
    h(&mut x)?;
    h(self)?;
    Ok(())
}
```


These are general rules - any particular operation may provide stronger guarantees
By dropping or replacing, as if by assignment, with a known value

Error Handling Composes

```
fn transaction(&mut self) -> () {
    let tmp = self.clone();
    if let Err(e) = self.op() {
        notify(e);
        *self = tmp;
    }
}
```


This is how we can stop error propagation and leave the system in a known state. Cow is a tool that can be used to make the clone efficient. See the history feature in Photoshop.

Safety

- For an unsafe operation the default contract is:
  - If the precondition is not satisfied, the postcondition is **undefined**
- Avoid using unsafe, but if you do
  - Keep the scope of the unsafe block as small as possible
  - Include a comment that states the preconditions for the unsafe code and why, or under what circumstances, they are satisfied
  - If there are *circumstances*, the calling operation also must be marked as unsafe with documented preconditions

The circumstances would then be a precondition of the caller.

Safety

```
/// Pushes a value into storage and registers its dropper.
fn push_storage<T: 'static>(&mut self, value: T) {
    self.storage.push(value);

    self.dropper.push(|storage, p| unsafe {
        // `drop_in_place` is only invoked with the corresponding
        // value in `storage` - see invariants.
        storage.drop_in_place::<T>(p)
    });
}
```


The circumstances would then be a precondition of the caller.

Summary

Summary

- Local Reasoning is built on:
  - Whole-part relationships
  - Projections for local operations

- Use these and embrace the Borrow Checker as a tool to achieve correctness

About the artist

Leandro Alzate

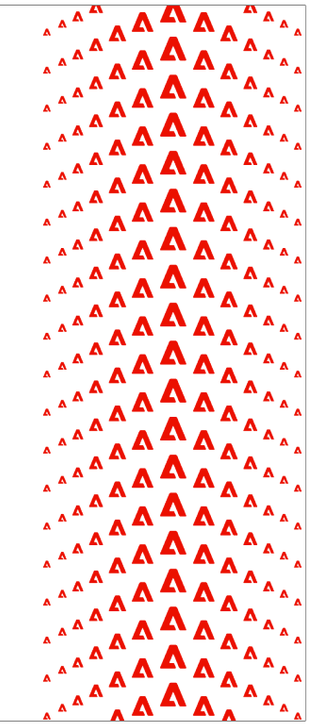
Berlin-based illustrator Leandro Alzate mixes bright color palettes and stylized characters in his fanciful work for editorial and advertising clients. He draws inspiration from observing the ways people interact, and combines that with his passion for architectural shapes and spaces. He created this piece for the German Ministry of Economy to encourage people to explore work-from-home career opportunities. Working with brushes and vector shapes, Alzate created this piece entirely in Adobe Photoshop.

Made with

Adobe Photoshop


Adobe
Artwork by Leandro Alzate

Extrinsic Relationships



But there is more to a system than just a bunch of objects - the objects are often somehow *related*.

Extrinsic Relationships

- An *extrinsic relationship* is a relationship that is not a whole-part relationship

```
let a = vec![0, 1, 2, 3];
```

  - *`a[0]` is before `a[1]`* is an extrinsic relationship

Relationships exist all over in the code - the main challenge in programming isn't in functions or classes, but in finding and managing the _essential relationships_.

Relationships

- A relationship is a connection between elements of two sets
  - For every relationship, there is a corresponding binary predicate. i.e., `is_married(a, b)`
- A relationship between objects may be severed by modifying or destroying either object
- A relationship may be *witnessed* by an object such as a pointer or **index**
  - An object that is a witness to a severed relationship may be *invalid*

I'm emphasizing index - sometimes memory-safe or functional languages are described as solving the problems with pointers. They solve the memory-safety problems and assist with local reasoning by forcing resolution to go through an owning object. But they do not ensure correctness (because correctness doesn't compose), and can be used to break local reasoning.

In any Turing complete language, you can build a C machine and write buggy C code. [Move this sentence]

If I have an index to the largest element of an array, and I change the element such that it is no longer the largest, my index, as a witness to the relationship, is invalid.

The severing of relationships and invalidation of witnesses we describe as "spooky action at a distance."

An invalid witness is one for which there is no (direct) way to determine whether the corresponding relationship is severed. Any result depending on an invalid witness is unspecified.

You Have an Extrinsic Relationship If...

- Your class stores a reference.
- Your class stores a key, index, or ID that refers to another object.
- You reference a (mutable) global variable.
- You use any primitive facility to share mutable data (Cell, RefCell, Atomic, Mutex, RwLock)

- Shared mutable access breaks local reasoning even if the exclusivity is checked at runtime.

Local Reasoning and Extrinsic Relationship

- To reason *locally* about extrinsic relationships, they should be encapsulated in a class
- The relationships are maintained between the parts by the class
- The class ensures the validity and correctness of the relationships by controlling access to the related objects
  - The validity and correctness of the relationships are the class invariants
- An intrusive witness in a part should only be manipulated by the owning class, and explicitly severed if the object is moved or copied outside the whole
- Containers are examples of classes that manage extrinsic relationships between their parts

Class invariants are extrinsic relationships on or between parts that always hold (for some definition of always).

Private access is used to protect the class invariants.

By encapsulating, we mean managing all the elements involved in a relationship. Those elements are _parts_.

"explicitly severed," such as by nulling a pointer or optional, or assigning a sentinel value such as a negative index to represent severed.

Linked list example. Splicing doesn't entangle lists—a container view of the world.

Class Invariants

- A class invariant is a general postcondition on any mutating operation
  - If members involved in the invariant are private, this invariant must only be shown to hold on mutating operations that have access to the members
- Care must be taken not to invoke an external operation with a reference to the object while the object invariants are violated
  - The borrow checker prevents most accidental cases

Class invariants are extrinsic relationships on or between parts that always hold (for some definition of always).

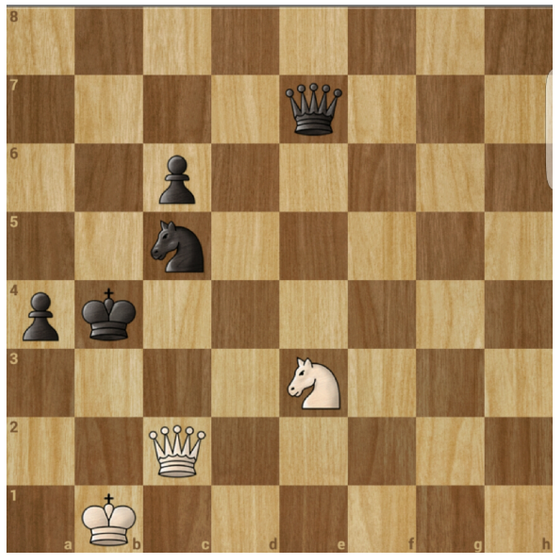
Private access is used to protect the class invariants.

By encapsulating, we mean managing all the elements involved in a relationship. Those elements are _parts_.

"explicitly severed," such as by nulling a pointer or optional, or assigning a sentinel value such as a negative index to represent severed.

Linked list example. Splicing doesn't entangle lists—a container view of the world.

An Analogy



I started this talk by claiming that software projects fail because local reasoning breaks down. Failing to manage extrinsic relationships is where that happens.

This is why it is essential to avoid creating _potential_ extrinsic relationships where a whole/part relationship would suffice. This is why we don't want incidental data structures, but we want data structures encapsulated in a class, so we can reason about the relationships locally.

Even if we are well principled in doing this. If every component we write has a well-defined contract. Extrinsic relationships are _hard_. Computer scientists are bad at relationships.

To illustrate - consider this chess board. Not any chess board or chess game, just _this_ board. There are four distinct pieces. King, queen, knight, pawn. Seven classes if we encode color in the class. We can represent these as very simple types.

Chess pieces do not have state. You might say "position is their state" - but that is false. The relationship is extrinsic to the part, even if I use an intrusive witness to represent the relationship.

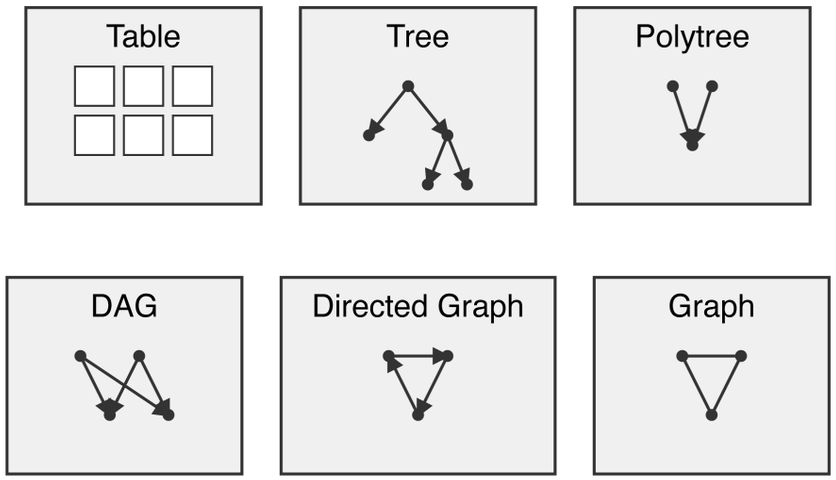
This is an end-game. We are not concerned with pawns moving 1 or 2 spaces, en passant, or castling. The board is only 8x8 - just 64 cells containing one of eight pieces or nothing.

The relationship between the piece and the board is explicit. But that creates relationships and potential relationships between the pieces. Is another piece along a line of immediate attack? How do those relationships change if I move a piece? Is a king in check? In mate?

Even with this simple example, I cannot reason through the relationship considering every possible move independently. There are just 30 allowed moves for white to take _next_, ruling out invalid moves, and about 30 responses to consider for each...

Now consider a system with shared mutable references, state-owned by more than one class, and multiple instances... If you think, "it isn't that hard" add concurrency and non-determinism.

Structural Complexity ≈ Reasoning Complexity



Tables and Trees are both expressible directly as whole-part relationships.

With poly-trees, joins must be managed - this is the structure of race conditions and LoE violations. Joins are potential contradictions. Joins should be explicit and managed (last-one-in wins is almost always a bad join).

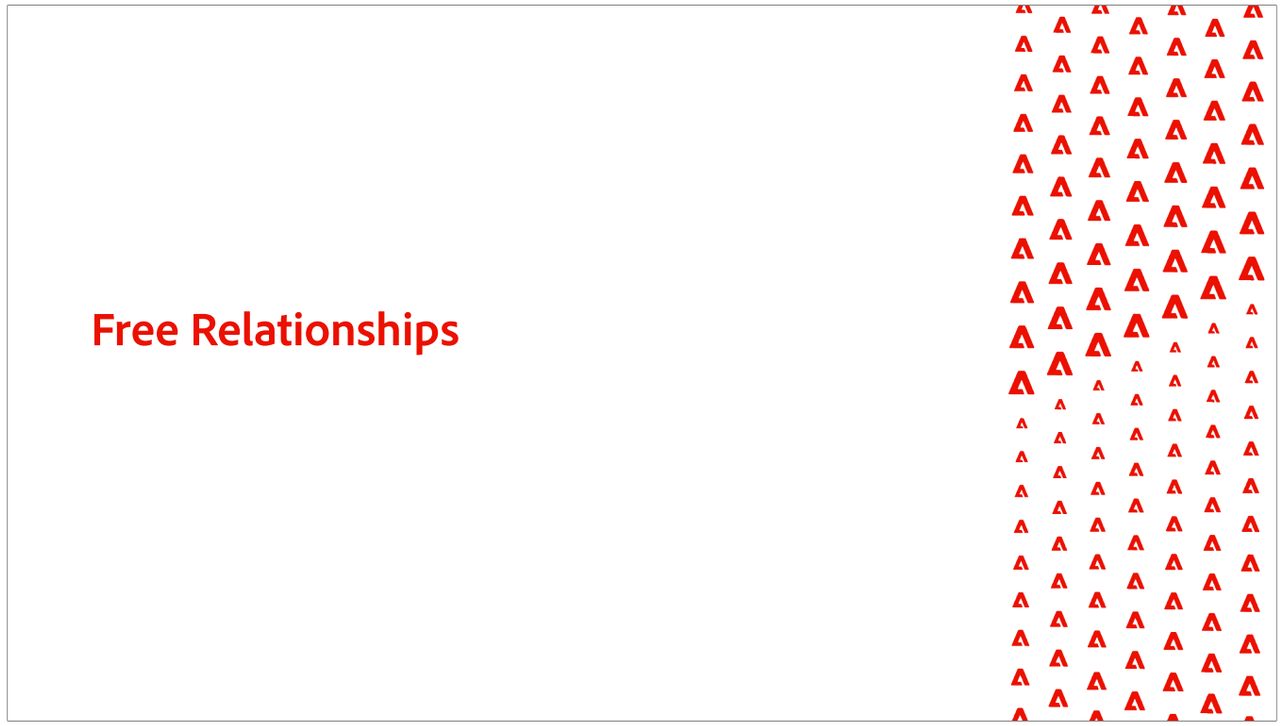
DAGs can rejoin - if not directed, they would contain cycles. They introduce consistency concerns (is the data calculated on this path consistent with this other path). The s-combinator is a diamond-shaped DAG relationship. S-combinators allow us to build a system *effectively* Turing-complete without iteration or recursion.

Directed graphs: Cycles should be factored out and replaced with a single node. "No raw loops" includes structural loops. Reasoning about traversing cycles requires proving termination and loop invariants.

Chessboards are filled with essential cyclic relationships - the queen is related to the king and the king to the queen. Chaotic systems are cyclic relationships. Bugs that don't reproduce are chaotic.

Finally, we have undirected graphs - these can be viewed as all possible directed graphs.

Free Relationships



In the 80s and into the 90s, there was a view that you could build systems at scale consisting of networks of objects. The entire OOP ethos was built around this idea. The view was always flawed but persists in reference-semantic languages.

The 80s programmers were the hippies from the 60s and 70s. Free the objects!

Free relationships

- A *free relationship* is an extrinsic relationship that is not managed between parts of an object
- Rust does not require extrinsic relationships to be encapsulated; these may be witnessed by:
  - References, iterators, slices, and indices
  - Mutex, Atomic, and RefMut
- Safety properties are a tool to assist with local reasoning
  - Within any safe Turing-complete system, you can implement a C machine and write unsafe code (see asm.js)

All too often, we circumvent safety and local reasoning for efficiency or expressibility - but doing so is fraught with issues.

Free relationships

- Are there structures we can reason about locally composed of free relationships?

We only have local knowledge of each object, which follows a set of rules.

CALM

"Question: What is the family of problems that can be consistently computed in a distributed fashion without coordination, and what problems lie outside that family?"

– *Keeping CALM: WhenDistributed Consistency is Easy*

This paper is from 2019. Relatively recent.

CALM

"A program has a consistent, coordination-free distributed implementation if and only if it is monotonic."

– *Keeping CALM: WhenDistributed Consistency is Easy*

Consistency As Logical Monotonicity (CALM).

CALM

- Conflict-free replicated data types(CRDTs) provide a framework for monotonic programming patterns
- An immutable variable is a monotonic pattern that transitions from undefined to its final value and never returns. Immutable variables generalize to immutable data structures

Immutable globals are okay. They don't require any additional coordination. A monotonic system can never repeat its state. This is related to the class ABA problem in concurrent programming.

In 2008, I gave a Google tech talk on a possible future of software development. I conjectured that _at_ some scale, we require coordination-free computation. That scale is determined by the latency required for coordination. Significant progress has been made in recent years in this space (see CRDT and operational transforms), but many open issues remain. But we now know the bounds within which we are working.

Russian Coat Check Algorithm

0	1	2	3	4	5	6	7
a	b	c	d	e	f	g	h

In my relationships talk I presented this structure. The Russian Coat Check Algorithm (so named because I thought of it while watching how a woman managed coats at a coat check in Russia and as a nod to the Russian Peasant Algorithm - aka Egyptian Multiplication). Probably every coat check algorithm, but we don’t have coats in CA

ordered is a relationship we can exploit

[every add is a wide gap]

Russian Coat Check Algorithm

0	1	2	3	4	5	6	7
a	x	x	d	e	x	x	x

In my relationships talk I presented this structure. The Russian Coat Check Algorithm (so named because I thought of it while watching how a woman managed coats at a coat check in Russia and as a nod to the Russian Peasant Algorithm - aka Egyptian Multiplication).

ordered is a relationship we can exploit
Probably every coat check algorithm but we don’t have coats in CA

[every add is a wide gap]

Russian Coat Check Algorithm

0	1	2	3	4	5	6	7
a	x	x	d	e	x	x	x

ordered is a relationship we can exploit
Probably every coat check algorithm but we don’t have coats in CA

[every add is a wide gap]

Russian Coat Check Algorithm

0	3	4	8	9	
a	d	e	i	j	

This structure is monotonic. It will never repeat the same state. A given element has 3 states and never cycles.

- Has not been there
- Is there
- Was there

A Russian Coat Check can be safely shared without coordination. Generation counts in heap allocation is a related mechanism that avoids the need for explicit coordination.

CALM is a tool to help you reason about what can be meaningfully shared and provides a framework for how to reason about objects required to be shared.

Summary

Existing Code

- Be conservative
- Avoid modifying shared data (the borrow checker makes this the default)
- Avoid creating new sharing, including by holding indices
- Remember the power of preconditions and push responsibility to the caller

Summary

- Interfaces should make the scope of the operation clear
- Projections provide an efficient way to achieve value semantics and manipulate parts
- Let Rust uphold the Law of Exclusivity
- Implementors provide types with value semantics
- Confine extrinsic relationships between parts within a class
  - As the relationships between parts scale, seek a general solution

About the artist

Leandro Alzate

Berlin-based illustrator Leandro Alzate mixes bright color palettes and stylized characters in his fanciful work for editorial and advertising clients. He draws inspiration from observing the ways people interact, and combines that with his passion for architectural shapes and spaces. He created this piece for the German Ministry of Economy to encourage people to explore work-from-home career opportunities. Working with brushes and vector shapes, Alzate created this piece entirely in Adobe Photoshop.

Made with

Adobe Photoshop


Adobe
Bē
Artwork by Leandro Alzate