Local Reasoning in ~~Any Language~~ Rust

Sean Parent | Sr. Principal Scientist
Software Technology Lab

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/

Why Software Projects Fail

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

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

Functions

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

Functions

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

Functions

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

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

Functions

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

Functions

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

Function Arguments

Function Arguments

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

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.

Argument Qualifiers

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

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

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}");
```

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
```

A more complex action

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

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

```
4
```

A more complex action

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

println!("{x_ref}");

4
```

A more complex action

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

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

```
4
```

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}");
```

A more complex action

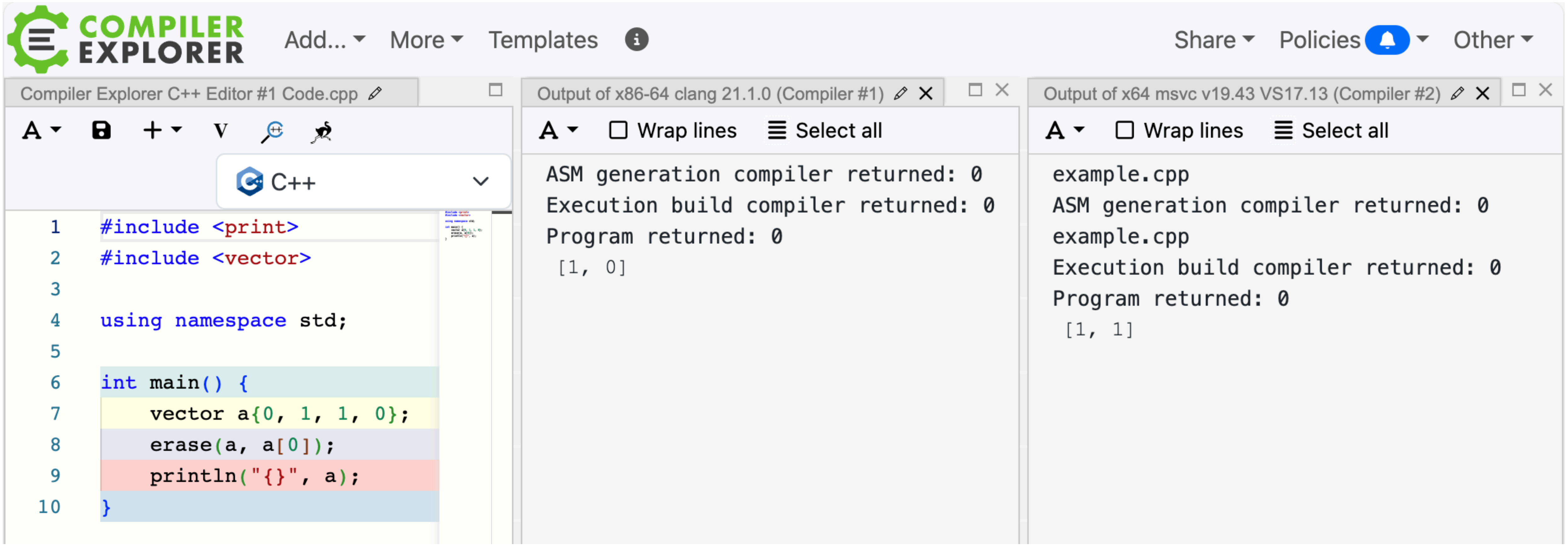
```
vector a{ 0, 1, 1, 0 };
erase(a, a[0]);
println("{}", a);
```

- What will this print?

```
[1, 1]
  or
[1, 0]
```

– *https://godbolt.org/z/z381aM1r7*

A more complex action


COMPILER EXPLORER
Add...
More
Templates
Share
Policies
Other
Compiler Explorer C++ Editor #1 Code.cpp
C++
Output of x86-64 clang 21.1.0 (Compiler #1)
Wrap lines
Select all
Output of x64 msvc v19.43 VS17.13 (Compiler #2)
Wrap lines
Select all

```cpp
#include <print>
#include <vector>

using namespace std;

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

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
 [1, 0]
```

```
example.cpp
ASM generation compiler returned: 0
example.cpp
Execution build compiler returned: 0
Program returned: 0
 [1, 1]
```

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*

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
```

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]
```

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

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*

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*

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

NOBODY EXPECTS
THE ALIASED PARAMETER
imgflip.com

Return Values

Return Values

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

Return-by-reference

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

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

```
[0, 1, 2, 42]
```

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

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
}
```

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);
```

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
        })
    }
}
```

Objects

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`

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,
};
```

Objects

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

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

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

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

Achieving Independence

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

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

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),
        }
    }
}
```

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)),
        }
    }
}
```

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)),
        }
    }
}
```

Errors & Safety

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

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

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(())
}
```

Error Handling Composes

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

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

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)
    });
}
```

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

Ps Adobe Photoshop

Artwork by **Leandro Alzate**

Extrinsic Relationships

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

- 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*

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.

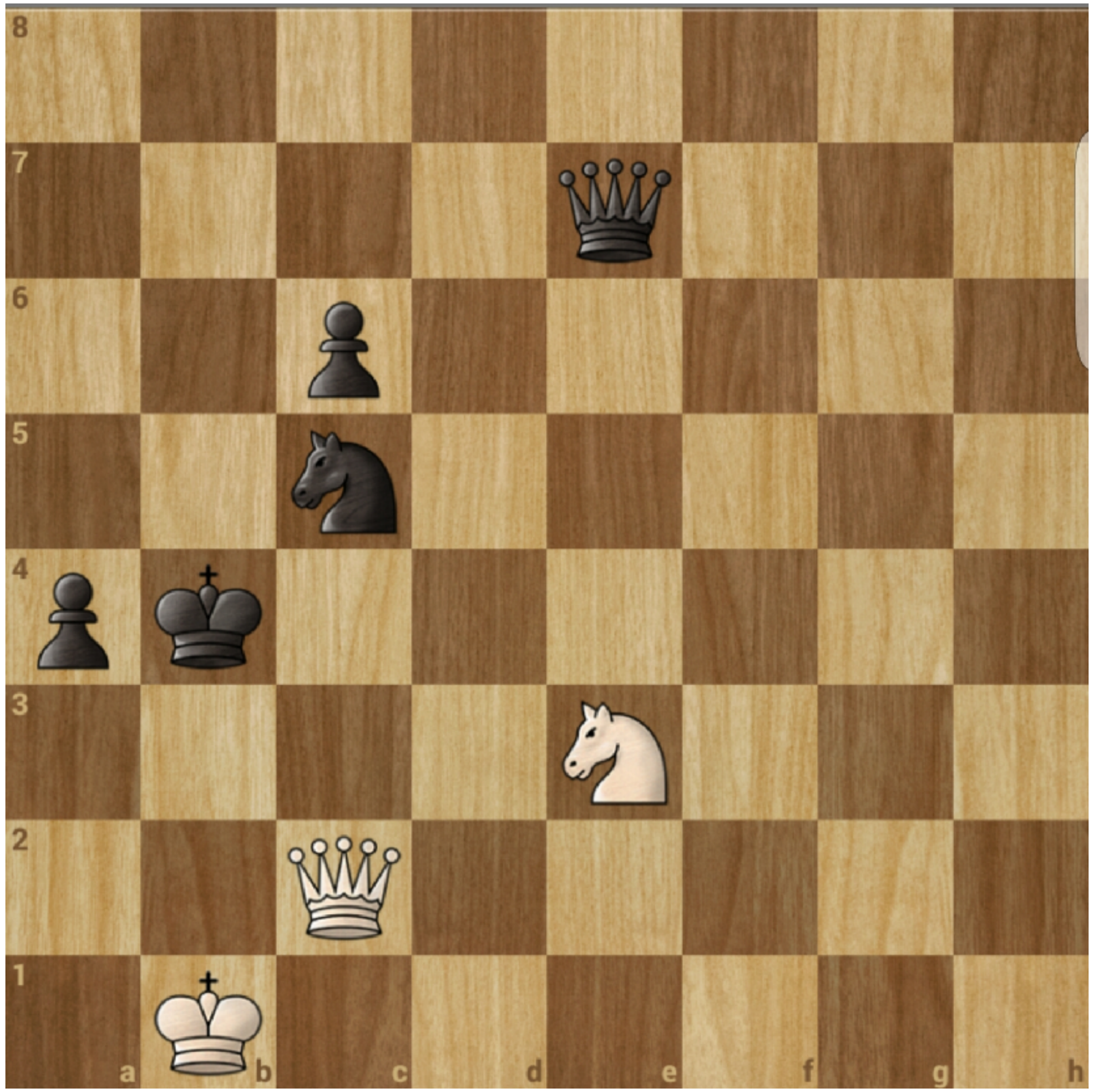
Local Reasoning and Extrinsic Relationship

- 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

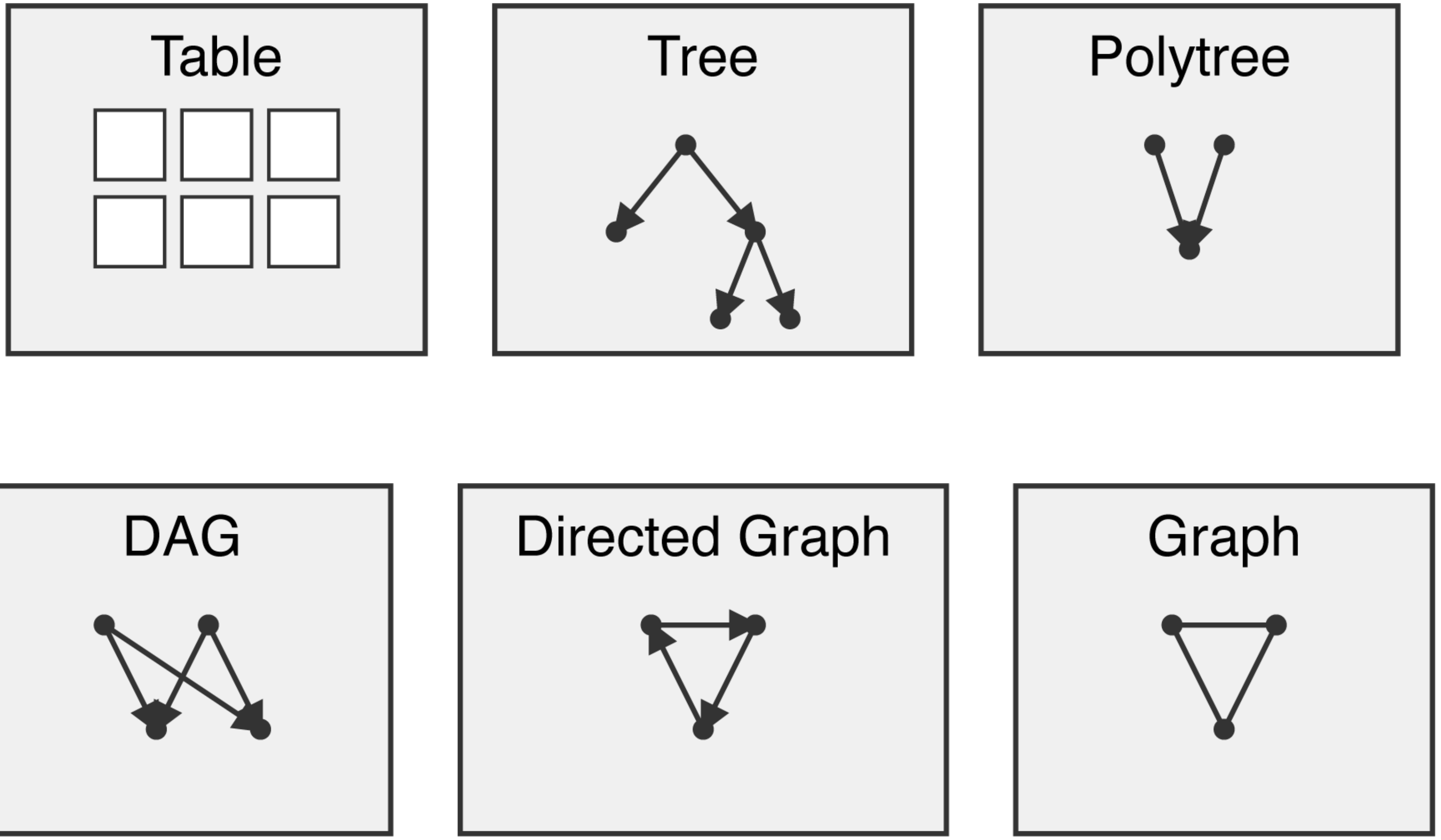
Class Invariants

- 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

An Analogy



Structural Complexity ≈ Reasoning Complexity



Free Relationships

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)

Free relationships

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

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*

CALM

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

– *Keeping CALM: WhenDistributed Consistency is Easy*

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

Russian Coat Check Algorithm

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

Russian Coat Check Algorithm

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

Russian Coat Check Algorithm

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

Russian Coat Check Algorithm

0	3	4	8	9	
a	d	e	i	j	

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

Ps Adobe Photoshop

Adobe
Bē
Artwork by Leandro Alzate