Translating C to Safe Rust

Mayank Keoliya

Theodore Leebrant

NUS National University of Singapore

School of Computing

Translating ~~C~~ `RefCell` to Safe ~~Rust~~ `GhostCell`

Mayank Keoliya

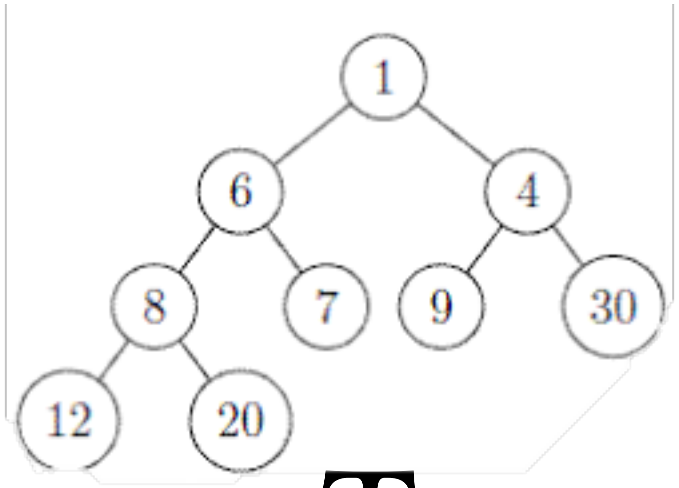
Theodore Leebrant

Translating ~~C~~ *`RefCell`* to Safe ~~Rust~~ *`GhostCell`*

Improving Rust Performance by Type-Directed Refactoring

Mayank Keoliya

Theodore Leebrant



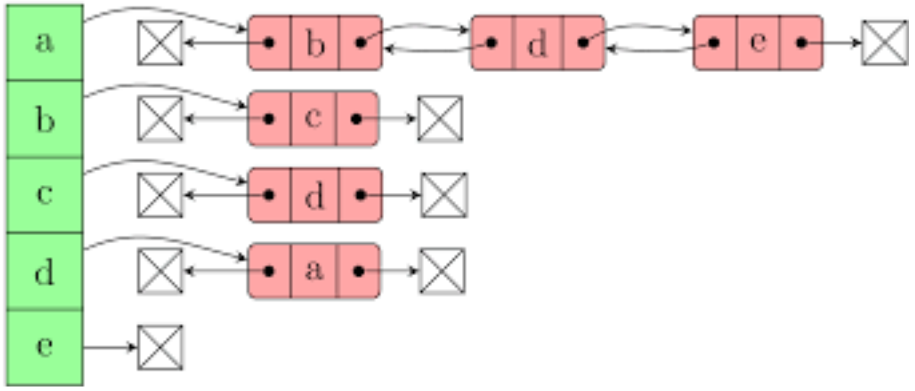
RefCell *GhostCell*

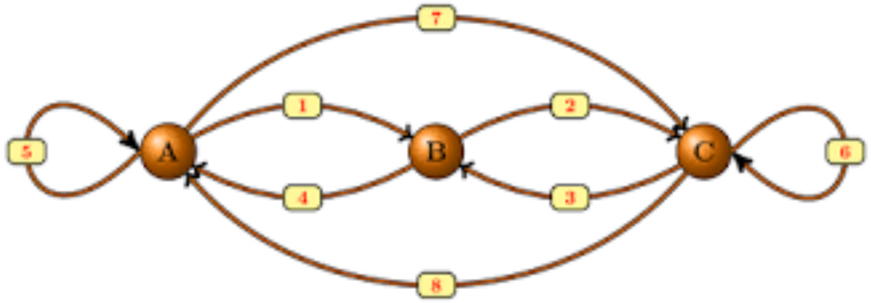
Translating ~~C~~ to Safe R~~u~~st

Improving Rust Performance by Type-Directed Refactoring

Mayank Keoliya

Theodore Leebrant





Contents

Motivation

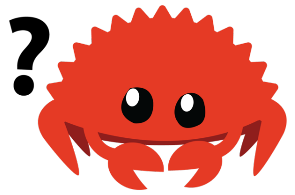
Translating C to Rust

```c
typedef struct Node {
    void* data;
    struct Node* next;
    struct Node* prev;
} Node;
```

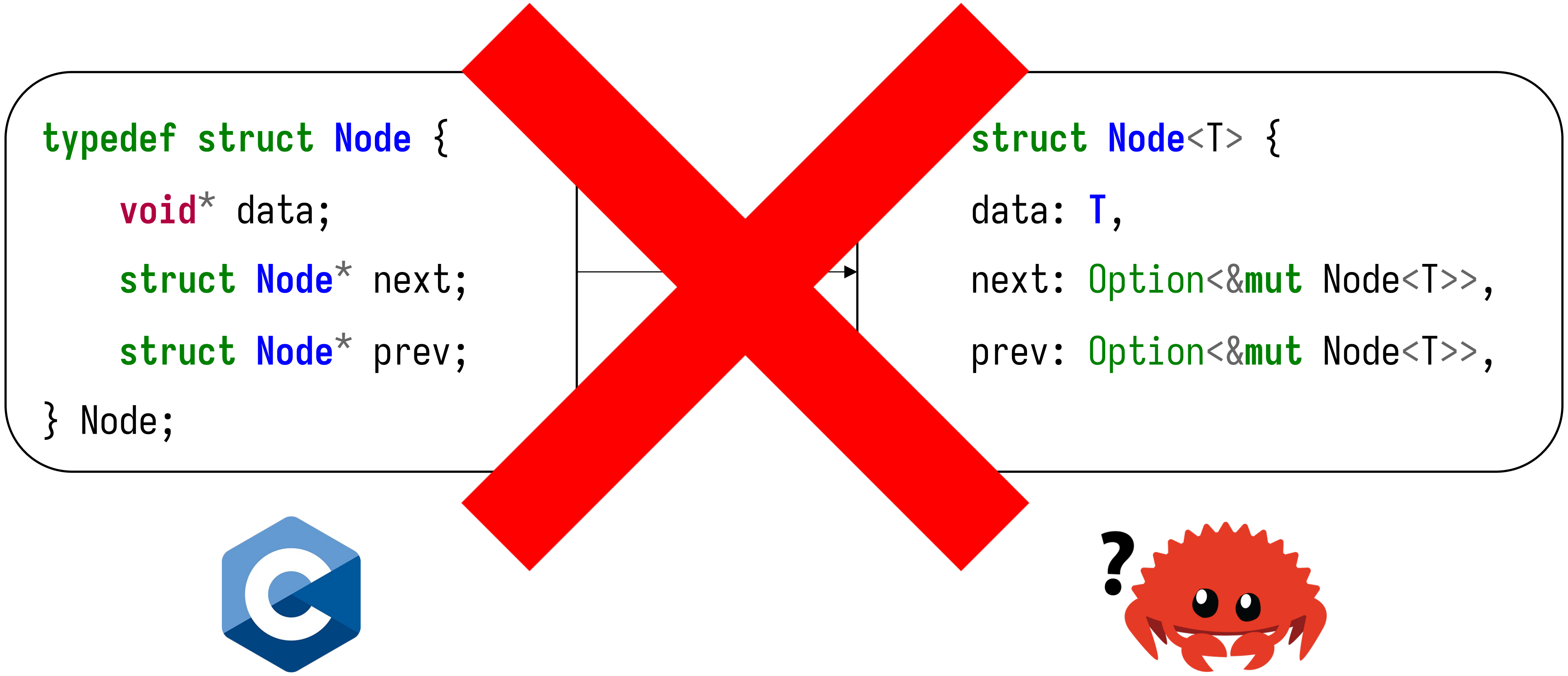
Translating C to Rust: A Naïve Attempt

```c
typedef struct Node {
    void* data;
    struct Node* next;
    struct Node* prev;
} Node;
```

```rust
pub struct Node<T> {
    data: T,
    next: Option<&mut Node<T>>,
    prev: Option<&mut Node<T>>,
}
```



Translating C to Rust: A Naïve Attempt



```c
typedef struct Node {
    void* data;
    struct Node* next;
    struct Node* prev;
} Node;
```

```rust
struct Node<T> {
data: T,
next: Option<&mut Node<T>>,
prev: Option<&mut Node<T>>,
```

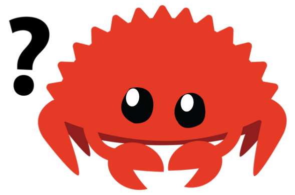
Translating C to Rust: A Naïve Attempt

```
typedef struct Node {
    void* data;
    struct Node
    struct Node   prev;
} Node;
```

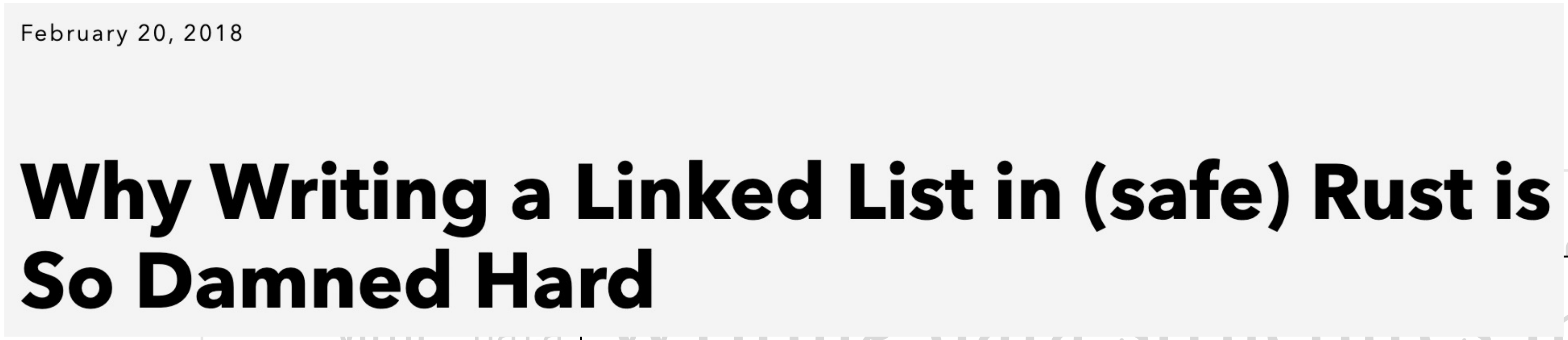
```
struct Node<T> {
                   t Node<T>>,
    prev: Option<&mut Node<T>>,
```

Writing data structures* in Rust is incredibly hard!

*with internal pointers

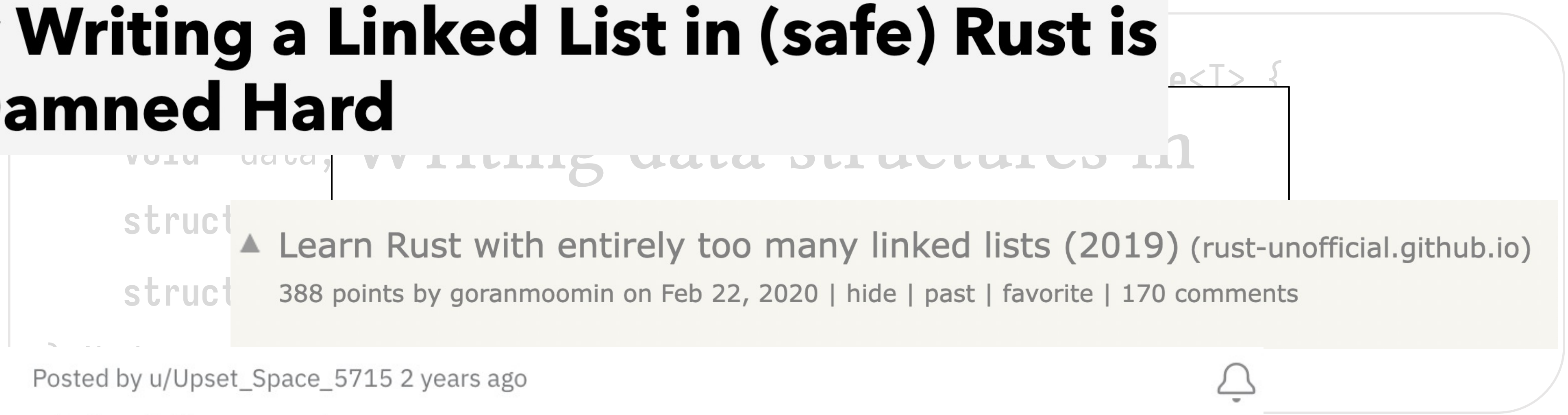


Translating C to Rust: A Naïve Attempt



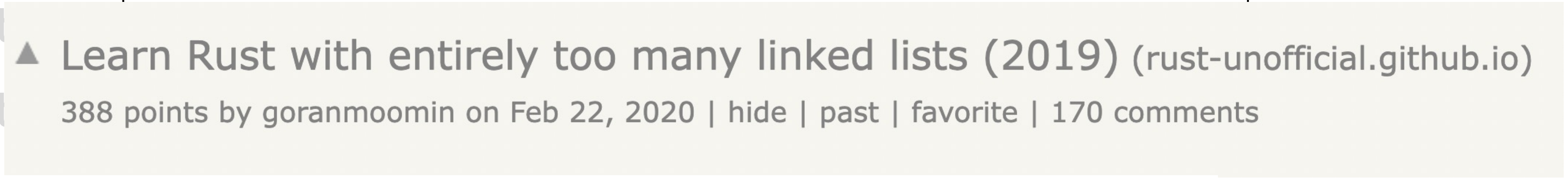
February 20, 2018

Why Writing a Linked List in (safe) Rust is So Damned Hard



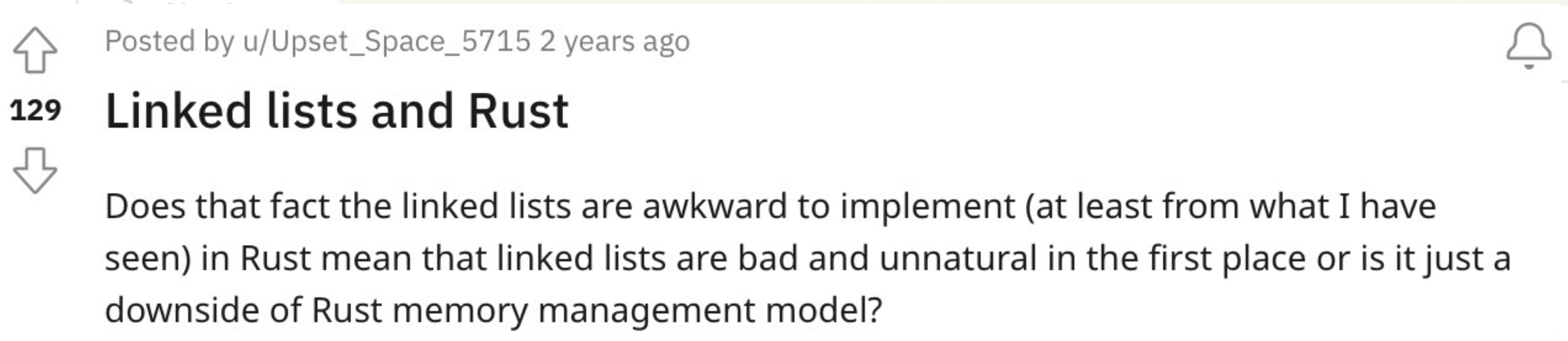
```
void *data;
struct
struct
```

Writing data structures in



▲ Learn Rust with entirely too many linked lists (2019) (rust-unofficial.github.io)

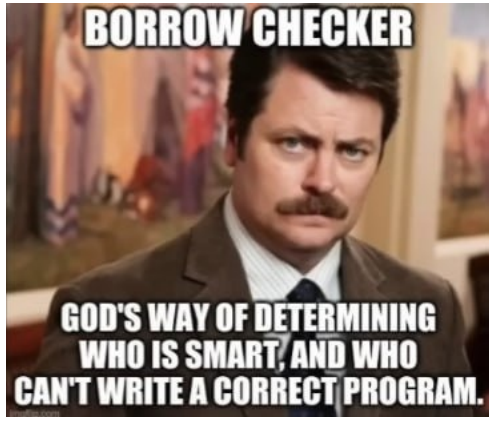
388 points by goranmoomin on Feb 22, 2020 | hide | past | favorite | 170 comments



Posted by u/Upset_Space_5715 2 years ago

129 **Linked lists and Rust**

Does that fact the linked lists are awkward to implement (at least from what I have seen) in Rust mean that linked lists are bad and unnatural in the first place or is it just a downside of Rust memory management model?


BORROW CHECKER
GOD'S WAY OF DETERMINING WHO IS SMART, AND WHO CAN'T WRITE A CORRECT PROGRAM.

Translating C to Rust: Mechanized?

```c
typedef struct Node {
    void* data;
    struct Node* next;
    struct Node* prev;
} Node;
```

Immunant's
c2rust

```rust
pub struct Node {
    data: *mut libc::c_void,
    next: *mut Node,
    prev: *mut Node,
}
```

Translating C to Rust: Mechanized?

```c
struct Node *createNode(int *data)
{
    struct Node *newNode =
      (struct Node*) malloc(
          sizeof(struct Node));
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}
```

Immunant's
c2rust

```rust
pub unsafe fn createNode(
     mut data: *mut libc::c_void
) -> *mut Node {
    let mut newNode: *mut Node =
        malloc(::std::mem::size_of::<Node>()
            as libc::c_ulong) as *mut Node;
    let ref mut fresh0 = (*newNode).data;
    *fresh0 = data;
    return newNode;
}
```

Translating C to Rust: Mechanized?

```
struc
{
    s



    r
    r
    r
    r
}
```

Translating C to Rust: Safety with `Rc+RefCell`

```c
typedef struct Node {
    void* data;
    struct Node* next;
    struct Node* prev;
} Node;
```

```rust
pub type NodePtr<T> =
    Arc<RefCell<Node<T>>>;

pub struct Node<T> {
    data: T,
    prev: Option<NodePtr<T>>,
    next: Option<NodePtr<T>>,
}
```

Translating C to Rust: Safety with Rc+RefCell

```c
struct Node *createNode(int *data)
{
    struct Node *newNode =
      (struct Node*) malloc(
          sizeof(struct Node));
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}
```

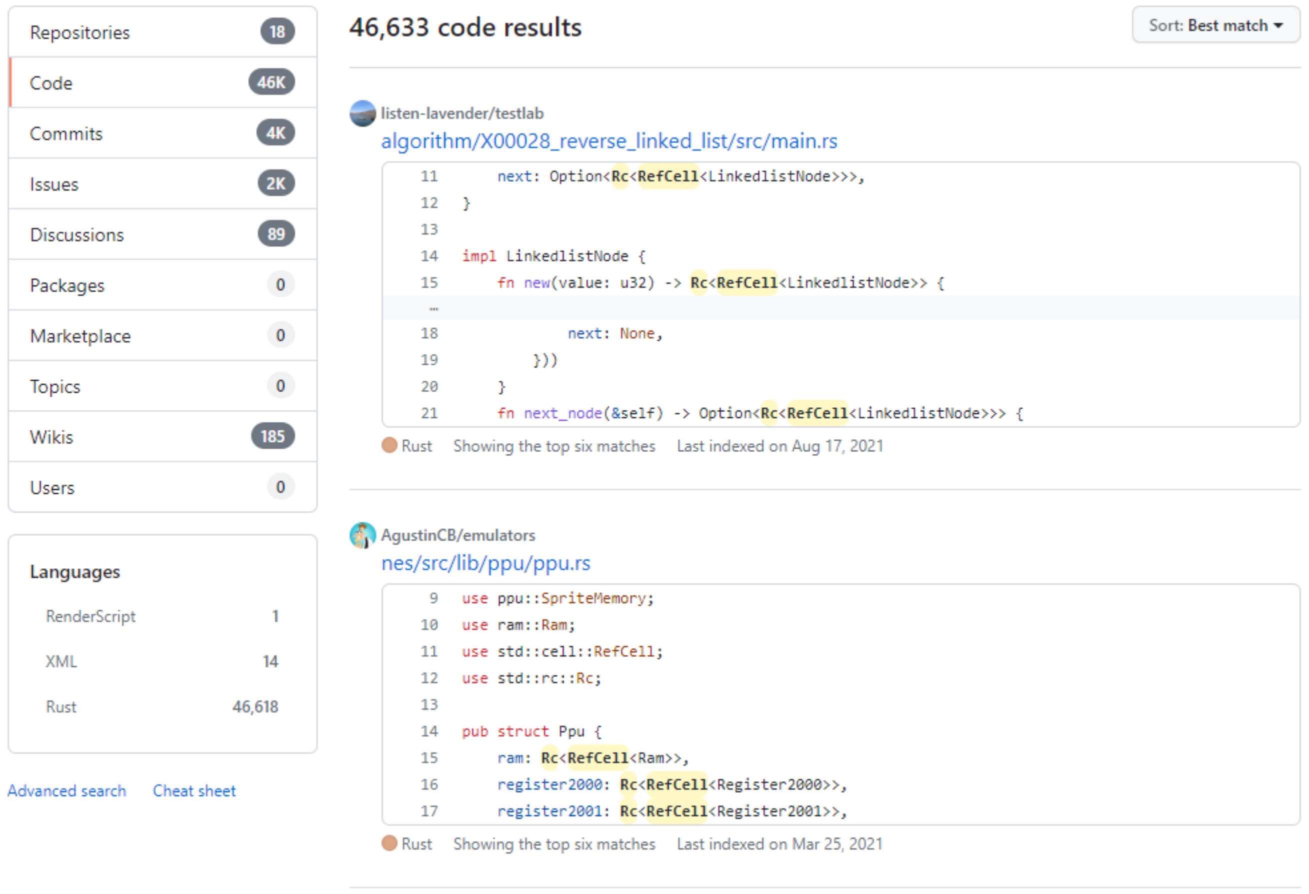
```rust
pub fn createNode<T>(value: T) -> NodePtr<T> {
    Rc::new(RefCell::new(
        Self {
            data: value,
            prev: None,
            next: None,
        }
    ))
}
```

Translating C to Rust: Tradeoffs of Safety

	C	Unsafe Rust	Safe Rust (`Rc+RefCell`)
Time	1.57 ms	1.51 ms	2.48 ms

Median time to insert 100,000 nodes into a doubly-linked list

Prevalence of the `Rc<RefCell>` Pattern



Repositories	18
Code	46K
Commits	4K
Issues	2K
Discussions	89
Packages	0
Marketplace	0
Topics	0
Wikis	185
Users	0

Languages

RenderScript	1
XML	14
Rust	46,618

Advanced search Cheat sheet

46,633 code results

Sort: Best match

listen-lavender/testlab
algorithm/X00028_reverse_linked_list/src/main.rs

```
    next: Option<Rc<RefCell<LinkedlistNode>>>,
}

impl LinkedlistNode {
    fn new(value: u32) -> Rc<RefCell<LinkedlistNode>> {
…
            next: None,
        }))
    }
    fn next_node(&self) -> Option<Rc<RefCell<LinkedlistNode>>> {
```

● Rust Showing the top six matches Last indexed on Aug 17, 2021

AgustinCB/emulators
nes/src/lib/ppu/ppu.rs

```
use ppu::SpriteMemory;
use ram::Ram;
use std::cell::RefCell;
use std::rc::Rc;

pub struct Ppu {
    ram: Rc<RefCell<Ram>>,
    register2000: Rc<RefCell<Register2000>>,
    register2001: Rc<RefCell<Register2001>>,
```

● Rust Showing the top six matches Last indexed on Mar 25, 2021

GitHub search result for “`Rc<RefCell`” having .rs file extension

So how can we improve the `Rc<RefCell>` pattern?

Essence of Rust: Ownership and Borrowing

- **Ownership**: who owns a value?
    - Every value has an owner
    - There can only be one owner at a time
    - When the owner goes out of scope, value is dropped

Essence of Rust: Ownership and Borrowing

- **Ownership**: who owns a value?
  - Every value has an owner
  - There can only be one owner at a time
  - When the owner goes out of scope, value is dropped

- **Borrowing**: using a value without owning it
  - Immutable borrows using `&`, mutable borrows using `&mut`
  - Always need to satisfy Aliasing XOR Mutability (AXM) principle

Ownership of Heap-allocated Values

Box

- Single ownership
- Little overhead
- Deallocation when out of scope

Rc / Arc

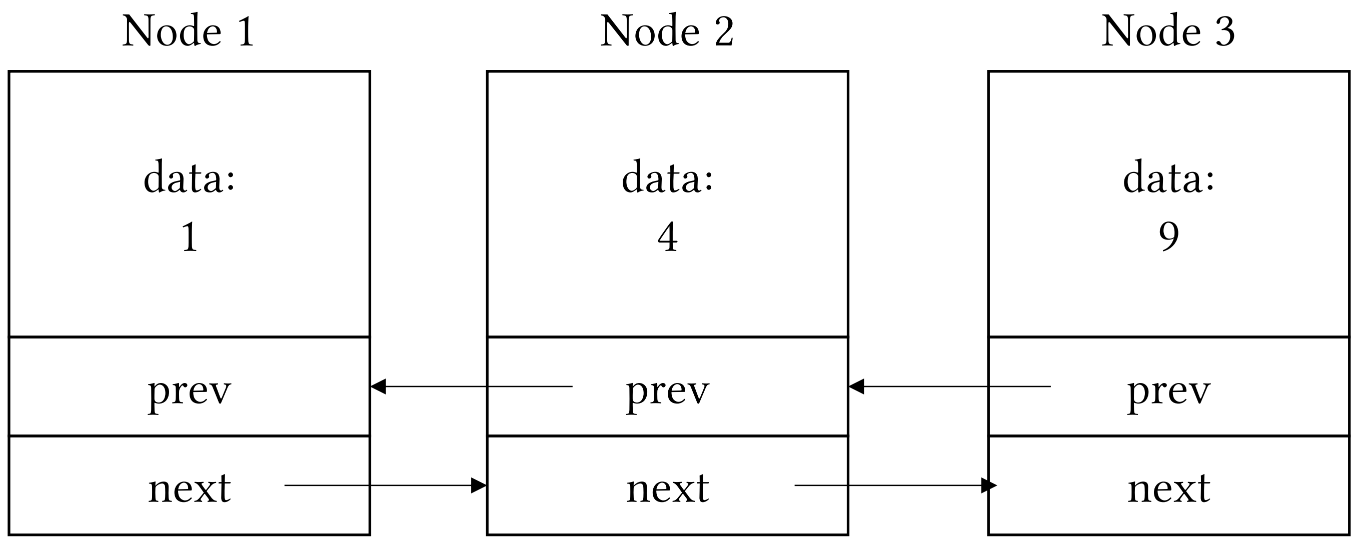
- Reference counted: multiple ownership
- Runtime overhead on counting
- Deallocation when count is zero

Arena

- Region-based memory allocation
- Ownership by the arena itself
- All deallocation happens when arena goes out of scope

Borrowing a Heap-allocated Value

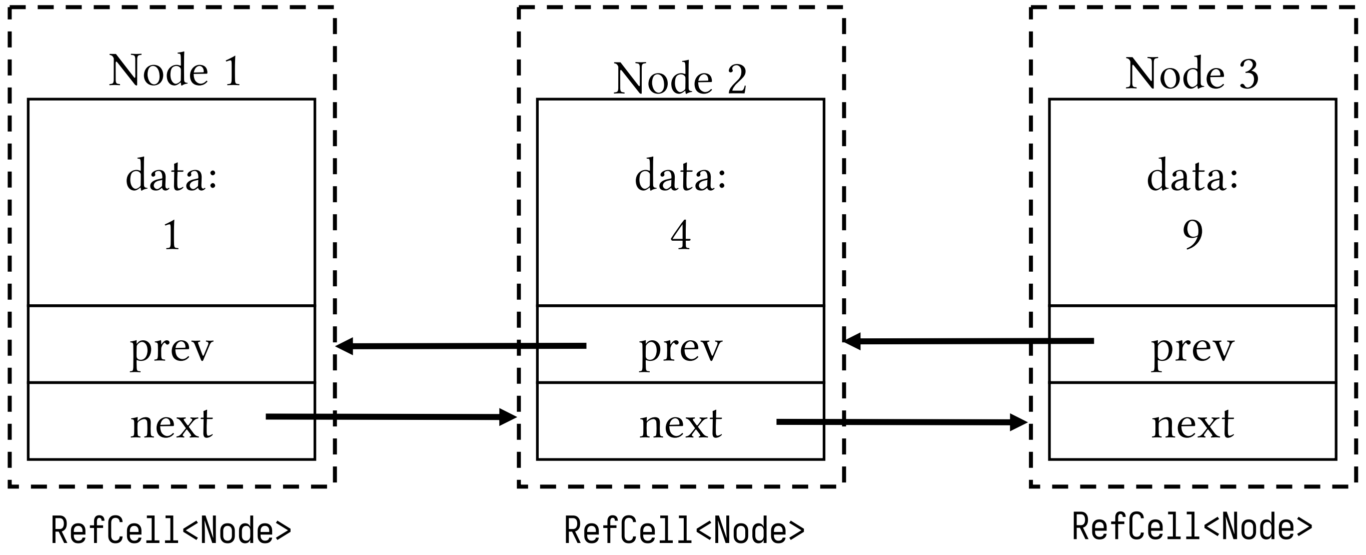
- Usual borrowing fails on reference-based data structures with aliasing:



- We can move the responsibility from compile-time to runtime by using Interior Mutability

Interior Mutability

- Allows getting a mutable reference to data from an immutable reference of the wrapper



- Now `Node1.next` and `Node3.prev` holds an immutable reference of `Node2`'s wrapper!

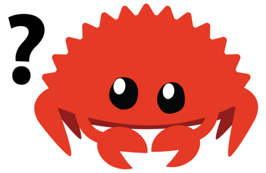
How RefCell Maintains Safety

- Usual borrow checker behaviour:

```rust
let mut s = String::from("hi");
let mut y = &mut s;
let mut z = &s;
println!("{y}");
println!("{z}");
```

```
error[E0502]: cannot borrow `s` as immutable because it is
also borrowed as mutable

error: could not compile `main` due to previous error
```



- RefCell behaviour:

```rust
let c = RefCell::new(5);
let m = c.borrow_mut();
let b = c.borrow();
```

panic!!

Wrappers with Interior Mutability Patterns

`RefCell`

- Allows getting interior references by `borrow()` or `borrow_mut()` calls
- Checks AXM at runtime
- Not thread-safe

`Mutex / RwLock`

- Thread-safe version of `RefCell`
- Locks interior nodes for every access (`Mutex`) or depending on the access type (`RwLock`)

GhostCell: A Better Wrapper

- Moves AXM checking back from runtime to compile-time
- GhostCell provides interior mutability while ensuring AXM ***on the whole data-structure*** during compile-time
    - Result: fast code that is still safe! (c.f. the *RustHornBelt* project)

	Mutex	RefCell	GhostCell
Rc	30.78 ms	9.18 ms	**3.68 ms**
Arena	7.95 ms	4.5 ms	**0.47 ms**

Median time of inserting 100 000 nodes on different doubly-linked list implementations

GhostCell: Separating Permissions from Data

- **Permission** refers to mutability / accessibility of references
- `GhostCell` enforces permissions at the data-structure level, instead of individual cells / nodes.
  - i.e. provides **coarse-grained** checks, but more restricted semantics
  - Permissions are tied to a special `GhostToken`
  - Contrast to `RefCell`, which is **fine-grained** and applies **dynamic checks** to individual node accesses

GhostCell: Separating Permission from Data

- GhostCell labels all cells in a data structure with a unique *brand*
  - Brands are implemented using lifetime parameters

- Each instance of a data structure carries a brand
  - `GhostCell<‘id, T>` is a cell type for wrapping data of type `T` that belongs to a data structure with brand `‘id`

- Accesses to a branded struct requires a token with the same brand
  - `GhostToken<‘id>` is the token signifying permission to access data of type `GhostCell<‘id, T>`

GhostCell: Separating Permission from Data

- GhostCell enables compile-time AXM checks by using *lifetime brands*

The *brand* used

```
pub struct Node<'id, T> {
    data: T,
    prev: Option<NodePtr<'id, T>>,
    next: Option<NodePtr<'id, T>>,
}
type NodePtr<'id, T> = Arc<GhostCell<'id, Node<'id, T>>>;
```

GhostCell: Separating Permission from Data

- Accesses to these structures require a token with the **same brand**

```
type NodePtr<'id, T> = Arc<GhostCell<'id, Node<'id, T>>>;

impl<'id, T> Node<'id, T> {
pub fn print_content(node: &NodePtr<'id, T>, token: &GhostToken<'id>) {
    let node_inner = node.borrow(token);
    println!("{node_inner}");
}}
```

GhostCell: Separating Permission from Data

- Accesses to these structures require a token with **correct permission**

```
fn main() {
  GhostToken::new(|t| {
    let mut xs = List::new(..);
    let x : &GhostCell<Node> = xs.head;
    Node::print_content(x, &t);
    let x_inner = x.borrow_mut(&mut t);
    x_inner.data = 5;
  });
}
```

Immutable access requires immutable token reference

Mutable access requires mutable token reference

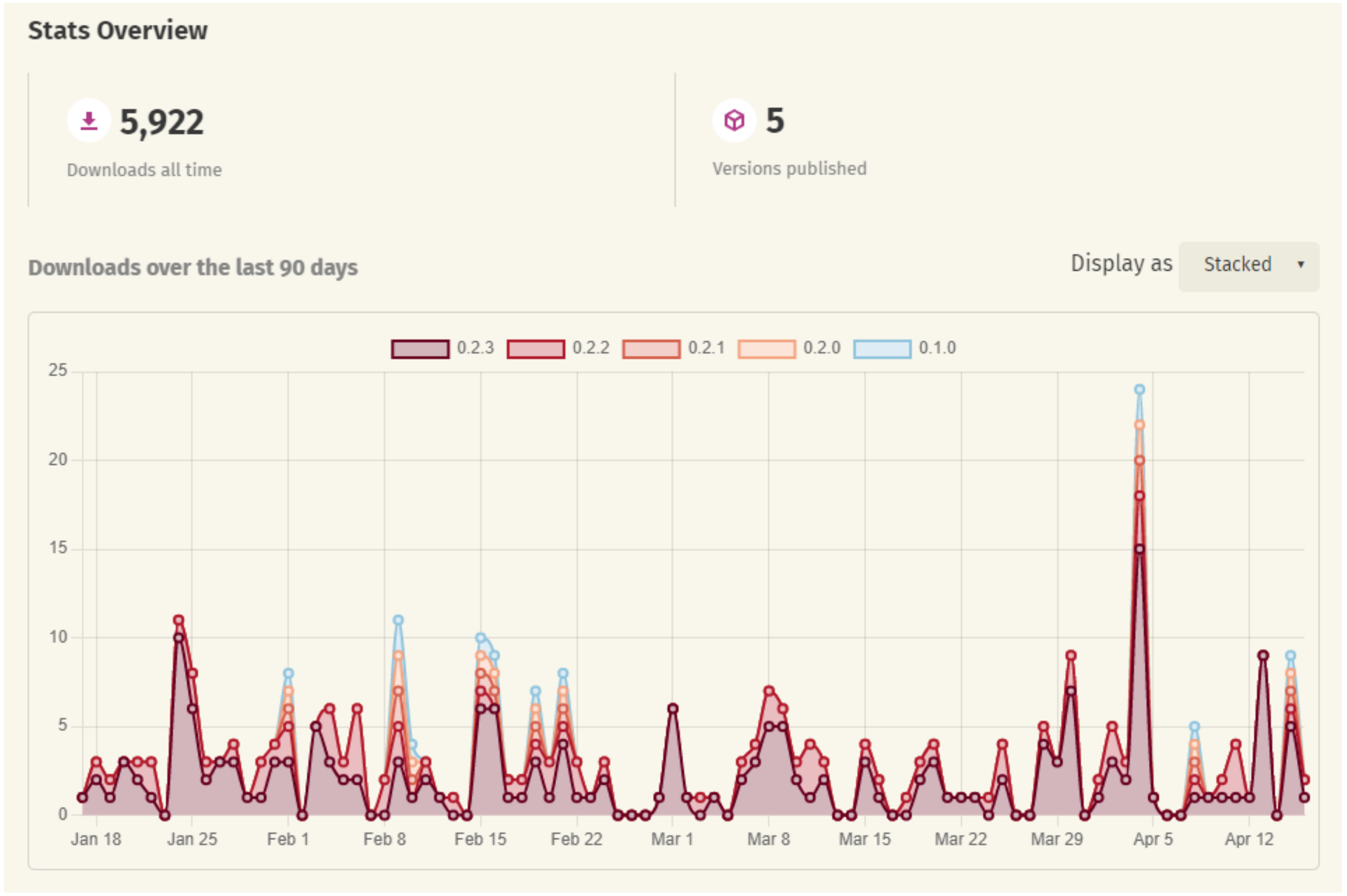
GhostCell: Separating Permission from Data

- Coarse-grained AXM enforcement, on the level of data structure

```
fn main() {
  GhostToken::new(|t| {
    let mut xs = List::new(..);
    let x : &GhostCell<Node> = xs.head;
    let y : &GhostCell<Node> = xs.tail;
    let x_inner = x.borrow_mut(&mut t);
    let y_inner = y.borrow_mut(&mut t);
    x_inner.data = 5;
  });
}
```

Two concurrent mutable borrows to the same data structure: AXM violation!

GhostCell: Usage


Stats Overview

5,922
Downloads all time

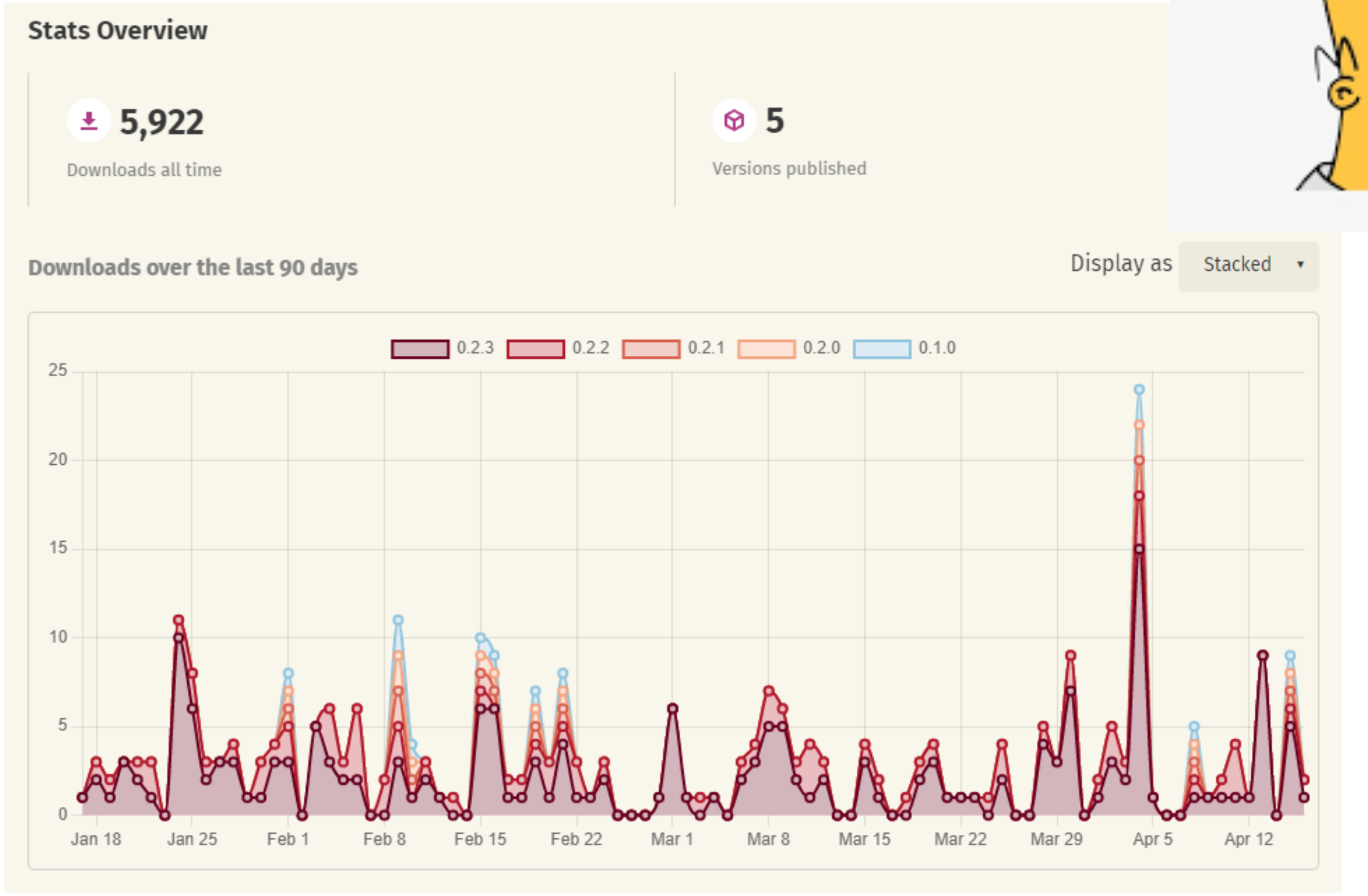
5
Versions published

Downloads over the last 90 days

Display as Stacked

Picture credits: https://crates.io/crates/ghost-cell

GhostCell: Usage



Stats Overview

5,922
Downloads all time

5
Versions published

Downloads over the last 90 days

Display as Stacked

Picture credits: https://crates.io/crates/ghost-cell

Problem Statement

We want to transform
a `RefCell`-based data structure

to

its `GhostCell` counterpart.

Why? Performance & safety!

Literature Review and Related Works

The Landscape of Data Structure Refinement

1. Laertes (OOPSLA 2021)
   - Minimizing raw pointer usage in automatically generated Rust code
2. Primrose (The Art, Science, and Engineering of Programming 2023)
   - Choosing container types based on semantic properties
3. Artemis (ESEC/FSE 2018)
   - Choosing the best container type based on performance
4. Various works on type systems

Laertes (Emre et al, 2021; 2023)

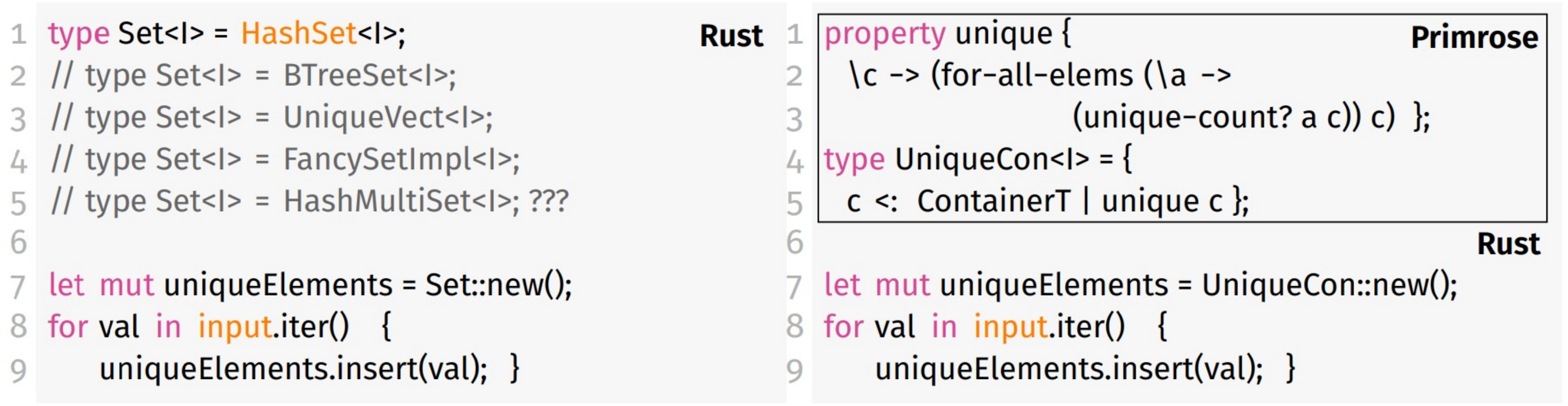
```
pub struct node_t {
  pub left: *mut node_t,
  pub right: *mut node_t,
  pub value: c_int,
}
```

```
pub struct node_t<’a1, ‘a2> {
  pub left: Option<&’a1 mut node_t<’a1, ‘a2>> ,
  pub right: Option<&’a1 mut node_t<’a1, ‘a2>>,
  pub value: Option<&’a2 mut c_int>,
}
```

- Optimistically rewrite pointers into `Option<&T>`
- Iteratively passes these rewrites to the compiler until it compiles
- Targeted for automated code generation from the `c2rust` tool

https://doi.org/10.1145/3485498
https://doi.org/10.1145/3586046

Primrose (Qin, O'Connor, and Steuwer 2023)


Rust

```
type Set<I> = HashSet<I>;
// type Set<I> = BTreeSet<I>;
// type Set<I> = UniqueVect<I>;
// type Set<I> = FancySetImpl<I>;
// type Set<I> = HashMultiSet<I>; ???

let mut uniqueElements = Set::new();
for val in input.iter() {
    uniqueElements.insert(val); }
```

Primrose

```
property unique {
  \c -> (for-all-elems (\a ->
                (unique-count? a c)) c) };
type UniqueCon<I> = {
  c <: ContainerT | unique c };
```

Rust

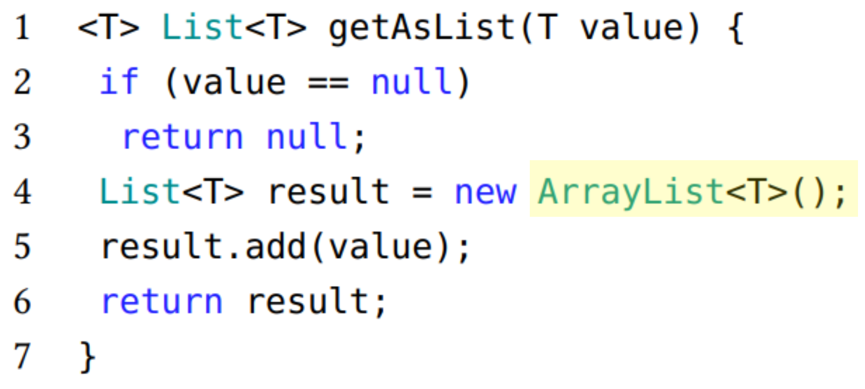
```
let mut uniqueElements = UniqueCon::new();
for val in input.iter() {
    uniqueElements.insert(val); }
```

- Choosing the optimal container type for a Data Structure, based on
  - Syntactic Properties: API exposed by the container type
  - Semantic Properties specified by user
  - Runtime performance

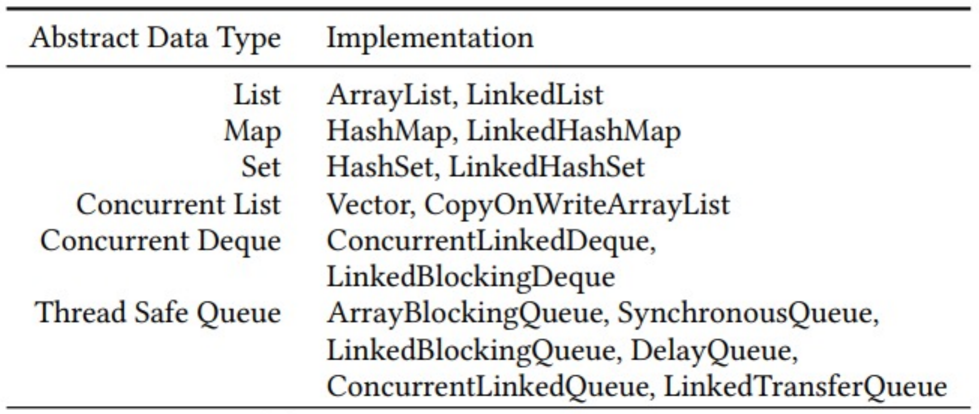
- Language-agnostic, prototyped for Rust

https://doi.org/10.22152/programming-journal.org/2023/7/11

Artemis (Basios et al, 2018)



```
<T> List<T> getAsList(T value) {
 if (value == null)
  return null;
 List<T> result = new ArrayList<T>();
 result.add(value);
 return result;
}
```



Abstract Data Type	Implementation
List	ArrayList, LinkedList
Map	HashMap, LinkedHashMap
Set	HashSet, LinkedHashSet
Concurrent List	Vector, CopyOnWriteArrayList
Concurrent Deque	ConcurrentLinkedDeque, LinkedBlockingDeque
Thread Safe Queue	ArrayBlockingQueue, SynchronousQueue, LinkedBlockingQueue, DelayQueue, ConcurrentLinkedQueue, LinkedTransferQueue

- Choosing container types which share a common interface
- Finds the optimal data structure based on performance in test suites
- Implemented for Java

https://doi.org/10.1145/3236024.3236043

Breaking News: the 100,000th New Type System

- **A Flexible Type System for Fearless Concurrency** (Milano 2022)
  - New type system to mark local heaplets, provides language primitives too (e.g. "**if disconnected")**
    - *Recent work into using "ghost cells" to achieve cyclic data structure patterns is encouraging, but remains above the annotation budget that we believe is desirable for such common data structures*

- **Rusty Links in Local Chains** (Noble 2023)
  - *Distinguishes local intra-thread ownership from global inter-thread ownership*
    - *Many programmers find Rust hard to learn and to use correctly. Rust's version of ownership types bans common idioms such as circular or doubly-linked lists, to the point where the **difficulty of implementing a data structure often taught at first year has now become an Internet trope**. A number of solutions have been proposed for these problems, including incorporating a garbage collector [15], careful library design, **phantom types [33],** or proving unsafe Rust code correct.*

Our Contribution (I)

- First to explore automated refinement of containers to improve performance in Rust

- Similar to Primrose and Artemis: refining structures based on exposed API
  - Without the need to describe semantic properties
  - Applied to Rust

Our Contribution (II)

Identified novel problem area

Implemented GHOSTCELLIFY

Future extension for other `Cells`

Overview

Running Example

- Live Demo!

GHOSTCELLIFY: Rewriting `RefCell` to `GhostCell`

- 4-step process:
  - Sanitizer
  - Brand introduction
  - Rewriting implementation methods and traits
  - Rewriting client code

Step 1: Sanitizer

- Reject `RefCell` code which could not be rewritten as `GhostCell`

Step 1: Sanitizer

- Reject `RefCell` code which could not be rewritten as `GhostCell`

```
fn set_both(node1_ptr: &NodePtr,
            node2_ptr: &NodePtr, v: T) {
    let node1 = node1_ptr.borrow_mut();
    let node2 = node2_ptr.borrow_mut();
    node1.data = v.clone();
    node2.data = v.clone();
    node1.next = node2_ptr.clone();
}
```

Case 1: two pointers from the same instance of a data structure

Step 1: Sanitizer

- Reject `RefCell` code which could not be rewritten as `GhostCell`

```
fn set_both(node1_ptr: &NodePtr,
            node2_ptr: &NodePtr, v: T) {
    let node1 = node1_ptr.borrow_mut();
    let node2 = node2_ptr.borrow_mut();
    node1.data = v.clone();
    node2.data = v.clone();
    node1.next = node2_ptr.clone();
}
```

Case 1: two pointers from the same instance of a data structure

These calls would require the same token: two concurrent mutable borrows!

Step 1: Sanitizer

- Reject `RefCell` code which could not be rewritten as `GhostCell`

```
fn set_both(node1_ptr: &NodePtr,
            node2_ptr: &NodePtr, v: T) {
    let node1 = node1_ptr.borrow_mut();
    let node2 = node2_ptr.borrow_mut();
    node1.data = v.clone();
    node2.data = v.clone();
    node1.next = node2_ptr.clone();
}
```

Case 2: two pointers from the two distinct instances of a data structure

Step 1: Sanitizer

- Reject `RefCell` code which cannot fit `GhostCell` semantics

```
fn set_both(node1_ptr: &NodePtr,
            node2_ptr: &NodePtr, v: T) {
    let node1 = node1_ptr.borrow_mut();
    let node2 = node2_ptr.borrow_mut();
    node1.data = v.clone();
    node2.data = v.clone();
    node1.next = node2_ptr.clone();
}
```

Case 2: two pointers from the two distinct instances of a data structure

`node1` and `node2` have different brands (lifetimes), so this is a type mismatch!

Step 2: Introducing Brands

- We need to introduce brands to the data structure definition.

Step 2: Introducing Brands

- We need to introduce brands to the data structure definition

```
pub type NodePtr<T> =
    Rc<RefCell<Node<T>>>;

pub struct Node<T> {
    data: T,
    prev: Option<NodePtr<T>>,
    next: Option<NodePtr<T>>,
}

pub struct List<T> {
    head: Option<NodePtr<T>>,
    last: Option<NodePtr<T>>
}
```

Step 2: Introducing Brands

- We need to introduce brands to the data structure definition

```rust
pub type NodePtr<T> =
    Rc<RefCell<Node<T>>>;

pub struct Node<T> {
    data: T,
    prev: Option<NodePtr<T>>,
    next: Option<NodePtr<T>>,
}

pub struct List<T> {
    head: Option<NodePtr<T>>,
    last: Option<NodePtr<T>>
}
```

```rust
pub type NodePtr<‘id, T> =
    Rc<RefCell<Node<‘id, T>>>;

pub struct Node<‘id, T> {
    data: T,
    prev: Option<NodePtr<‘id, T>>,
    next: Option<NodePtr<‘id, T>>,
}

pub struct List<‘id, T> {
    head: Option<NodePtr<‘id, T>>,
    last: Option<NodePtr<‘id, T>>
}
```

Step 3: Rewriting Impl. Methods & Traits

- Operations involving the data structure needs to have token reference

`RefCell` API	`GhostCell` API
`pub fn new(value: T) -> RefCell<T>`	`pub fn new(value: T) -> GhostCell<‘id, T>`
`pub fn borrow(&self) -> Ref<‘_, T>`	`pub fn borrow(&self, &GhostToken<‘id>) -> &T`
`pub fn borrow_mut(&self) -> RefMut<‘_, T>`	`pub fn borrow_mut(&self, &mut GhostToken<‘id>) -> &mut T`

Step 3: Rewriting Impl. Methods & Traits

- Operations involving the data structure needs to have token reference

`RefCell` API	`GhostCell` API
`pub fn new(value: T) -> RefCell<T>`	`pub fn new(value: T) -> GhostCell<‘id, T>`
`pub fn borrow(&self) -> Ref<‘_, T>`	`pub fn borrow(&self, &GhostToken<‘id>) -> &T`
`pub fn borrow_mut(&self) -> RefMut<‘_, T>`	`pub fn borrow_mut(&self, &mut GhostToken<‘id>) -> &mut T`

Step 3: Rewriting Impl. Methods & Traits

- Operations involving the data structure needs to have token reference

```
impl<'id, T> Node<'id, T> {
pub fn print_content(node: &NodePtr<'id, T>) {
    let node_inner = node.borrow();
    println!("{node_inner}");
}}
```

Branded `RefCell` implementation

Step 3: Rewriting Impl. Methods & Traits

- Operations involving the data structure needs to have token reference

```
impl<'id, T> Node<'id, T> {
pub fn print_content(node: &NodePtr<'id, T>) {
    let node_inner = node.borrow();
    println!("{node_inner}");
}}
```

Branded `RefCell` implementation

```
impl<'id, T> Node<'id, T> {
pub fn print_content(node: &NodePtr<'id, T>, token: & GhostToken<'id>) {
    let node_inner = node.borrow(token);
    println!("{node_inner}");
}}
```

`GhostCell` implementation

Step 3: Rewriting Impl. Methods & Traits

- Operations involving the data structure needs to have token reference

```
impl<'id, T> List<'id, T> {
pub fn insert(&mut self, val: T) {
    let new_node = Node::new(val);
    if self.last.is_none() {
        ... ()
    }
    let last_node = self.last.unwrap();
    Node::insert_next(
        &last_node, new_node.clone());
    self.last = Some(new_node);
}}
```

Step 3: Rewriting Impl. Methods & Traits

- Operations involving the data structure needs to have token reference

```
impl<'id, T> List<'id, T> {
pub fn insert(&mut self, val: T) {
    let new_node = Node::new(val);
    if self.last.is_none() {
        ... ()
    }
    let last_node = self.last.unwrap();
    Node::insert_next(
        &last_node, new_node.clone());
    self.last = Some(new_node);
}}
```

Step 3: Rewriting Impl. Methods & Traits

- Operations involving the data structure needs to have token reference

```
impl<'id, T> List<'id, T> {
pub fn insert(&mut self, val: T) {
    let new_node = Node::new(val);
    if self.last.is_none() {
        ... ()
    }
    let last_node = self.last.unwrap();
    Node::insert_next(
        &last_node, new_node.clone());
    self.last = Some(new_node);
}}
```

Step 3: Rewriting Impl. Methods & Traits

- Operations involving the data structure needs to have token reference

```
impl<'id, T> List<'id, T> {
pub fn insert(&mut self, val: T) {
    let new_node = Node::new(val);
    if self.last.is_none() {
        ... ()
    }
    let last_node = self.last.unwrap();
    Node::insert_next(
        &last_node, new_node.clone());
    self.last = Some(new_node);
}}
```

```
impl<'id, T> Node<'id, T> {
pub fn insert_next(
    node1: &NodePtr<'id, T>,
    node2: NodePtr<'id, T>
) {
    ...
    let mut node2_inner = node2.borrow_mut();
    node2_inner.prev = Some(node1.clone());
    node2_inner.next = node1_old_next;
}}
```

Step 3: Rewriting Impl. Methods & Traits

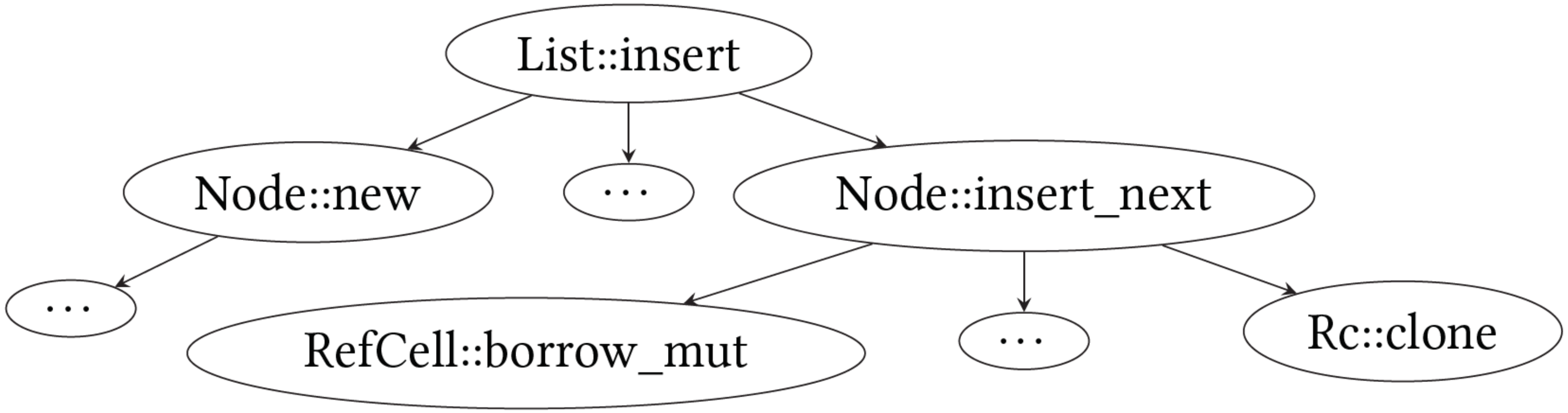
- Operations involving the data structure needs to have token reference

```
impl<'id, T> List<'id, T> {
pub fn insert(&mut self, val: T) {
    let new_node = Node::new(val);
    if self.last.is_none() {
        ... ()
    }
    let last_node = self.last.unwrap();
    Node::insert_next(
        &last_node, new_node.clone());
    self.last = Some(new_node);
}}
```

```
impl<'id, T> Node<'id, T> {
pub fn insert_next(
    node1: &NodePtr<'id, T>,
    node2: NodePtr<'id, T>
) {
    ...
    let mut node2_inner = node2.borrow_mut();
    node2_inner.prev = Some(node1.clone());
    node2_inner.next = node1_old_next;
}}
```

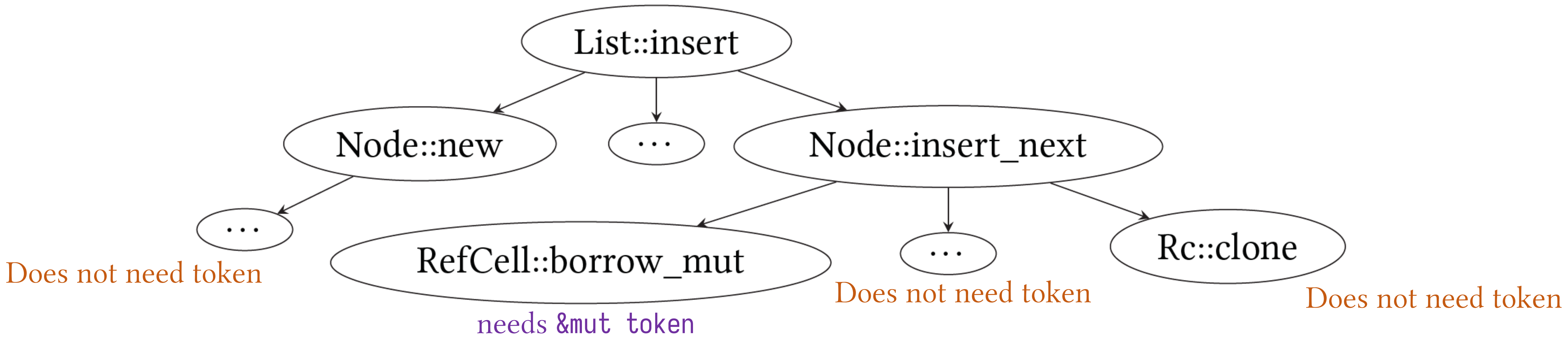
Step 3: Rewriting Impl. Methods & Traits

- Build a call graph



Step 3: Rewriting Impl. Methods & Traits

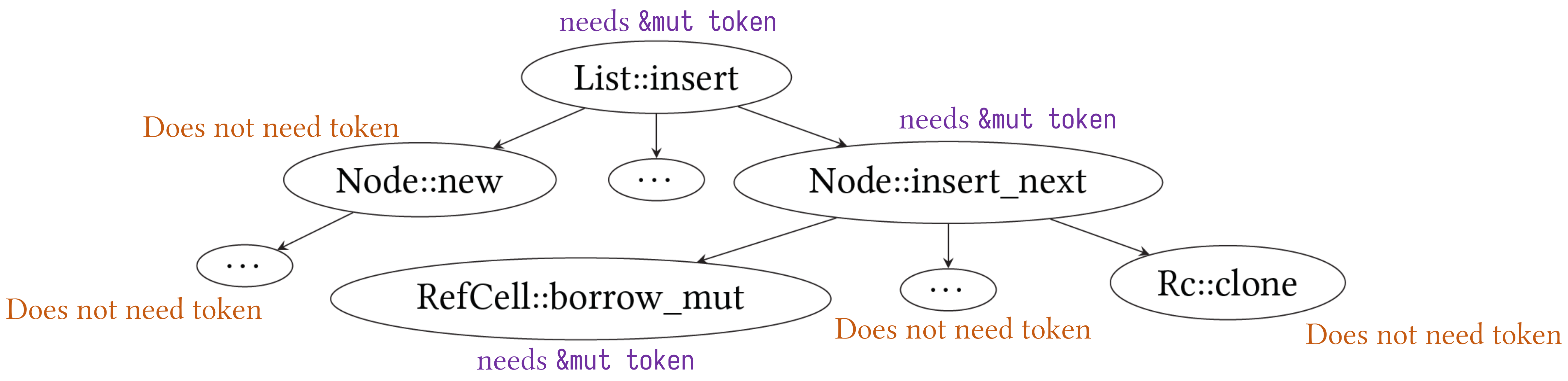
- Annotate the calls with the corresponding token permission needed



Step 3: Rewriting Impl. Methods & Traits

- Propagate up:

$$\text{parent's permission} = \max_{c \in \text{children}} \text{permission}(c)$$



Step 3: Rewriting Impl. Methods & Traits

- Rewrite based on the call graph

```
impl<'id, T> List<'id, T> {
pub fn insert(
    &mut self, val: T,
    tok: &mut GhostToken<‘id>
) {
    let new_node = Node::new(val);
    …
    Node::insert_next(
        &last_node, new_node.clone(), tok);
    …
}}
```

```
impl<'id, T> Node<'id, T> {
pub fn insert_next(
    node1: &NodePtr<'id, T>,
    node2: NodePtr<'id, T>,
    tok: &mut GhostToken<‘id>
) {
    ...
    let mut node2_inner = node2.borrow_mut(tok);
    node2_inner.prev = Some(node1.clone());
    node2_inner.next = node1_old_next;
}}
```

Step 4: Rewriting Client Code

- Introduce token by means of closure

Step 4: Rewriting Client Code

- Introduce token by means of closure

```rust
fn main() {


  let mut a = List::new(..);
  a.insert(5);
  let a_head = a.head.borrow_mut();
  Node::map(&a_head, |x| x + 1);


}
```

Step 4: Rewriting Client Code

- Create a fresh token scoped to a closure

```rust
fn main() {

  let mut a = List::new(..);
  a.insert(5);
  let a_head = a.head.borrow_mut();
  Node::map(&a_head, |x| x + 1);

}
```

```rust
fn main() {
  GhostToken::new(|t| {
    let mut a = List::new(..);
    a.insert(5);
    let a_head = a.head.borrow_mut();
    Node::map(&a_head, |x| x + 1);
  });
}
```

Step 4: Rewriting Client Code

- Update function calls based on function definition and callgraph

```
fn main() {
  GhostToken::new(|t| {
    let mut a = List::new(..);
    a.insert(5);
    let a_head = a.head.borrow_mut();
    Node::map(&a_head, |x| x + 1);
  });
}
```

Step 4: Rewriting Client Code

- Introduce token references accordingly

```
fn main() {
  GhostToken::new(|t| {
    let mut a = List::new(..);
    a.insert(5);
    let a_head = a.head.borrow_mut();
    Node::map(&a_head, |x| x + 1);
  });
}
```

```
fn main() {
  GhostToken::new(|t| {
    let mut a = List::new(..);
    a.insert(5);
    let a_head = a.head.borrow_mut(&mut t);
    Node::map(&a_head, |x| x + 1);
  });
}
```

Ghostcellify

High-Level Algorithm

```
Procedure Ghostcellify($P$)
    Input: Program $P$
    Output: New program $\mathcal{P}'$
    $\mathcal{S} \leftarrow$ StructDefs($P$)
    $\mathcal{I} \leftarrow$ Impl($P$)
    $\mathcal{C} \leftarrow$ ClientCode($P$)
    if not Sanitize($\mathcal{P}$) then          } §4.1
        Abort()
    GenerateBrands($\mathcal{S} \Downarrow_r \mathcal{S}'$)   } §4.2
    TransformImpl($\mathcal{I} \rightsquigarrow \mathcal{I}'$)   } §4.3
    TransformClient($\mathcal{C} \Rightarrow_d \mathcal{C}'$)   } §4.4
    $\mathcal{P}' \leftarrow \mathcal{S}' \cup \mathcal{I}' \cup \mathcal{C}'$
```

Symbol	Meaning
$\mathcal{P}$	Program AST (RefCell-based) §4.0
$e : \mathrm{T}$	Sanitize §4.1
$\Downarrow_r$	Brand Inference §4.2
$\rightsquigarrow$	Rewrite Impl. Methods §4.3
$\Rightarrow_d$	Rewrite client code §4.4

§4.0. Rust syntax

```
Procedure Ghostcellify(P)
    Input: Program $P$
    Output: New program $\mathcal{P}'$
    $\mathcal{S} \leftarrow$ StructDefs($P$)
    $\mathcal{I} \leftarrow$ Impl($P$)
    $\mathcal{C} \leftarrow$ ClientCode($P$)
    if not Sanitize($\mathcal{P}$) then
        Abort()
    GenerateBrands($\mathcal{S} \Downarrow_r \mathcal{S}'$)
    TransformImpl($\mathcal{I} \rightsquigarrow \mathcal{I}'$)
    TransformClient($\mathcal{C} \Rightarrow_d \mathcal{C}'$)
    $\mathcal{P}' \leftarrow \mathcal{S}' \cup \mathcal{I}' \cup \mathcal{C}'$
```

Symbol	Meaning
$\mathcal{P}$	Program AST (RefCell-based) §4.0
$e : \mathrm{T}$	Sanitize §4.1
$\Downarrow_r$	Brand Inference §4.2
$\rightsquigarrow$	Rewrite Impl. Methods §4.3
$\Rightarrow_d$	Rewrite client code §4.4

Simplified Rust Syntax

v		variable names
E	$::=$	$v \mid (\overline{v}) \mid [\overline{v}] \mid v \oplus v$
		$\mid \texttt{fn}\ (\overline{v})\{E\} \mid \texttt{drop}(v)$
		$\mid \texttt{let}\ [\texttt{mut}]\ v = E$
		$\mid \texttt{\&}[\texttt{mut}]\ v$
		$\mid E;E$

t		user-defined types
$'id$		lifetimes
T	$::=$	$() \mid \text{primitive} \mid t$
		$\mid (\overline{T}) \mid [\overline{T}]$
		$\mid (\overline{T}) \rightarrow T \mid T\texttt{<}\overline{id}, \overline{T}\texttt{>}$
		$\mid \texttt{ref}\ T \mid \texttt{ref mut}\ T$

§4.1. Sanitizer

```
Procedure Ghostcellify(P)
    Input: Program $P$
    Output: New program $\mathcal{P}'$
    $\mathcal{S} \leftarrow$ StructDefs($P$)
    $\mathcal{I} \leftarrow$ Impl($P$)
    $\mathcal{C} \leftarrow$ ClientCode($P$)
    if not Sanitize($\mathcal{P}$) then
        Abort()
    GenerateBrands($\mathcal{S} \Downarrow_r \mathcal{S}'$)
    TransformImpl($\mathcal{I} \rightsquigarrow \mathcal{I}'$)
    TransformClient($\mathcal{C} \Rightarrow_d \mathcal{C}'$)
    $\mathcal{P}' \leftarrow \mathcal{S}' \cup \mathcal{I}' \cup \mathcal{C}'$
```

Symbol	Meaning
$\mathcal{P}$	Program AST (RefCell-based)
$e : \mathrm{T}$	Sanitize §4.1
$\Downarrow_r$	Brand Inference §4.2
$\rightsquigarrow$	Rewrite Impl. Methods §4.3
$\Rightarrow_d$	Rewrite client code §4.4

Primer: Simplified Borrowing Rules for Rust

- Γ: typing environment, mapping expression to types
- Ω: constraints on references to values
- Δ: store environment mapping places to values

S-REF
$$\frac{}{\texttt{ref mut}\ T <: \texttt{ref}\ T}$$

T-IMMUT-BORROW
$$\frac{\Gamma \vdash v : T' \qquad \Delta(p) = v \qquad \texttt{mut ref}\ p \notin \Omega \qquad \Omega \cup \{\texttt{ref}\ p\}; \Gamma[x : \texttt{ref}\ T'] \vdash E : T}{\Omega, \Gamma \vdash \texttt{let}\ x = \ \&v; E \ : \ T}$$

T-MUT-BORROW
$$\frac{\Gamma \vdash v : T' \qquad \Delta(p) = v \qquad \texttt{ref}\ p \notin \Omega \qquad \Omega \cup \{\texttt{ref}\ p\}; \Gamma[x : \texttt{mut ref}\ T'] \vdash E : T}{\Omega, \Gamma \vdash \texttt{let}\ x = \ \&\texttt{mut}\ v; E \ : \ T}$$

T-SCOPE
$$\frac{\Omega \vdash E_1 : T_1; \Omega_1 \qquad \Omega_1 \vdash E_2 : T_2; \Omega_2 \qquad \ldots \qquad \Omega_{n-1} \vdash E_n : T_n; \Omega_n \qquad \Omega' = \Omega \cup \Omega_n \setminus \Omega_{n-1}}{\Omega, \Gamma \vdash \{E_1; E_2; \ldots E_n\}E \ : \ T; \Omega'}$$

Primer: Simplified Borrowing Rules for Rust

- Γ: typing environment, mapping expression to types
- Ω: constraints on references to values
- Δ: store environment, mapping places to values

S-REF

$$\frac{}{\texttt{ref mut}\ T <: \texttt{ref}\ T}$$

T-IMMUT-BORROW

$$\frac{\Gamma \vdash v : T' \qquad \Delta(p) = v \qquad \texttt{mut ref}\ p \notin \Omega \qquad \Omega \cup \{\texttt{ref}\ p\}; \Gamma[x : \texttt{ref}\ T'] \vdash E : T}{\Omega, \Gamma \ \vdash\ \texttt{let}\ x = \ \texttt{\&}v; E \ :\ T}$$

T-MUT-BORROW

$$\frac{\Gamma \vdash v : T' \qquad \Delta(p) = v \qquad \texttt{ref}\ p \notin \Omega \qquad \Omega \cup \{\texttt{ref}\ p\}; \Gamma[x : \texttt{mut ref}\ T'] \vdash E : T}{\Omega, \Gamma \ \vdash\ \texttt{let}\ x = \ \texttt{\&mut}\ v; E \ :\ T}$$

T-SCOPE

$$\frac{\Omega \vdash E_1 : T_1; \Omega_1 \qquad \Omega_1 \vdash E_2 : T_2; \Omega_2 \qquad \ldots \qquad \Omega_{n-1} \vdash E_n : T_n; \Omega_n \qquad \Omega' = \Omega \cup \Omega_n \setminus \Omega_{n-1}}{\Omega, \Gamma \ \vdash\ \{E_1; E_2; \ldots E_n\} E \ :\ T; \Omega'}$$

Primer: Simplified Borrowing Rules for Rust

- Γ: typing environment, mapping expression to types
- Ω: constraints on references to values
- Δ: store environment, mapping places to values

S-REF

$$\frac{}{\texttt{ref mut}\ T <: \texttt{ref}\ T}$$

T-IMMUT-BORROW

$$\frac{\Gamma \vdash v : T' \qquad \Delta(p) = v \qquad \texttt{mut ref}\ p \notin \Omega \qquad \Omega \cup \{\texttt{ref}\ p\}; \Gamma[x : \texttt{ref}\ T'] \vdash E : T}{\Omega, \Gamma \ \vdash\ \texttt{let}\ x = \ \&v; E \ :\ T}$$

T-MUT-BORROW

$$\frac{\Gamma \vdash v : T' \qquad \Delta(p) = v \qquad \texttt{ref}\ p \notin \Omega \qquad \Omega \cup \{\texttt{ref}\ p\}; \Gamma[x : \texttt{mut ref}\ T'] \vdash E : T}{\Omega, \Gamma \ \vdash\ \texttt{let}\ x = \ \&\texttt{mut}\ v; E \ :\ T}$$

T-SCOPE

$$\frac{\Omega \vdash E_1 : T_1; \Omega_1 \qquad \Omega_1 \vdash E_2 : T_2; \Omega_2 \qquad \ldots \qquad \Omega_{n-1} \vdash E_n : T_n; \Omega_n \qquad \Omega' = \Omega \cup \Omega_n \setminus \Omega_{n-1}}{\Omega, \Gamma \ \vdash\ \{E_1; E_2; \ldots E_n\} E \ :\ T; \Omega'}$$

Primer: GhostCell Borrow Semantics

- Γ: typing environment, mapping expression to types
- Ω: constraints on references to values
- Δ: store environment, mapping places to values

GC-IMMUT-BORROW

$$\frac{\Gamma \vdash v : \texttt{GhostCell<}'id, T\texttt{>} \qquad \Gamma \vdash token : \texttt{\&GhostToken<}'id\texttt{>} \\ \Delta(p) = token \qquad \texttt{mut ref}\ p \notin \Omega \qquad \Omega' = \Omega \cup \{\texttt{ref}\ p\}}{\Omega; \Gamma \vdash v.\texttt{borrow}(token) : \texttt{\&}T; \Omega'}$$

GC-MUT-BORROW

$$\frac{\Gamma \vdash v : \texttt{GhostCell<}'id, T\texttt{>} \qquad \Gamma \vdash token : \texttt{\&mut GhostToken<}'id\texttt{>} \\ \Delta(p) = token \qquad \texttt{ref}\ p \notin \Omega \qquad \Omega' = \Omega \cup \{\texttt{mut ref}\ p\}}{\Omega; \Gamma \vdash v.\texttt{borrow\_mut}(token) : \texttt{\&mut}T; \Omega'}$$

Primer: GhostCell Borrow Semantics

- Γ: typing environment, mapping expression to types
- Ω: constraints on references to values
- Δ: store environment, mapping places to values

GC-IMMUT-BORROW

$$\frac{\begin{array}{c}\Gamma \vdash v : \texttt{GhostCell<}'id, T\texttt{>} \qquad \Gamma \vdash token : \texttt{\&GhostToken<}'id\texttt{>} \\ \Delta(p) = token \qquad \texttt{mut ref}\ p \notin \Omega \qquad \Omega' = \Omega \cup \{\texttt{ref}\ p\}\end{array}}{\Omega; \Gamma \vdash v.\texttt{borrow}(token) : \texttt{\&}T; \Omega'}$$

GC-MUT-BORROW

$$\frac{\begin{array}{c}\Gamma \vdash v : \texttt{GhostCell<}'id, T\texttt{>} \qquad \Gamma \vdash token : \texttt{\&mut GhostToken<}'id\texttt{>} \\ \Delta(p) = token \qquad \texttt{ref}\ p \notin \Omega \qquad \Omega' = \Omega \cup \{\texttt{mut ref}\ p\}\end{array}}{\Omega; \Gamma \vdash v.\texttt{borrow\_mut}(token) : \texttt{\&mut}T; \Omega'}$$

New Borrow Rules for `RefCell`

- Γ: typing environment, mapping expression to types
- Ω: constraints on references to values
- S: set of data structure instances
- B: mapping from data structure instance to a brand
- $\mathfrak{F}$: the fields of a data structure instance

RC-IMMUT-BORROW

$$\frac{\Gamma \vdash v : \texttt{RefCell<}T\texttt{>} \qquad v \in \mathfrak{F}[t] \qquad t \in S \qquad B[t] = p \qquad \texttt{mut ref } p \notin \Omega \qquad \Omega' = \Omega \cup \{\texttt{ref } p\}}{\mathfrak{F}; \Omega; \Gamma \ \vdash\ v.\texttt{borrow()} :\ \& T;\ \Omega'}$$

RC-MUT-BORROW

$$\frac{\Gamma \vdash v : \texttt{RefCell<}T\texttt{>} \qquad v \in \mathfrak{F}[t] \qquad t \in S \qquad B[t] = p \qquad \texttt{ref } p \notin \Omega \qquad \Omega' = \Omega \cup \{\texttt{mut ref } p\}}{\mathfrak{F}; \Omega; \Gamma \ \vdash\ v.\texttt{borrow\_mut()} :\ \&\texttt{mut } T;\ \Omega'}$$

New Borrow Rules for `RefCell`

- Γ: typing environment, mapping expression to types
- Ω: constraints on references to values
- S: set of data structure instances
- B: mapping from data structure instance to a brand
- 𝔉: the fields of a data structure instance

RC-IMMUT-BORROW

$$\frac{\Gamma \vdash v : \texttt{RefCell<}T\texttt{>} \qquad v \in \mathfrak{F}[t] \qquad t \in S \qquad B[t] = p \qquad \texttt{mut ref } p \notin \Omega \qquad \Omega' = \Omega \cup \{\texttt{ref } p\}}{\mathfrak{F}; \Omega; \Gamma \ \vdash \ v\texttt{.borrow()} : \ \&\, T;\ \Omega'}$$

RC-MUT-BORROW

$$\frac{\Gamma \vdash v : \texttt{RefCell<}T\texttt{>} \qquad v \in \mathfrak{F}[t] \qquad t \in S \qquad B[t] = p \qquad \texttt{ref } p \notin \Omega \qquad \Omega' = \Omega \cup \{\texttt{mut ref } p\}}{\mathfrak{F}; \Omega; \Gamma \ \vdash \ v\texttt{.borrow\_mut()} : \ \&\texttt{mut } T;\ \Omega'}$$

New Borrow Rules for `RefCell`

- Γ: typing environment, mapping expression to types
- Ω: constraints on references to values
- S: set of data structure instances
- B: mapping from data structure instance to a brand
- $\mathfrak{F}$: the fields of a data structure instance

RC-IMMUT-BORROW

$$\frac{\Gamma \vdash v : \texttt{RefCell<}T\texttt{>} \qquad v \in \mathfrak{F}[t] \qquad t \in S \qquad B[t] = p \qquad \texttt{mut ref}\ p \notin \Omega \qquad \Omega' = \Omega \cup \{\texttt{ref}\ p\}}{\mathfrak{F};\Omega;\Gamma \ \vdash\ v\texttt{.borrow()} :\ \texttt{\&}\,T;\ \Omega'}$$

RC-MUT-BORROW

$$\frac{\Gamma \vdash v : \texttt{RefCell<}T\texttt{>} \qquad v \in \mathfrak{F}[t] \qquad t \in S \qquad B[t] = p \qquad \texttt{ref}\ p \notin \Omega \qquad \Omega' = \Omega \cup \{\texttt{mut ref}\ p\}}{\mathfrak{F};\Omega;\Gamma \ \vdash\ v\texttt{.borrow\_mut()} :\ \texttt{\&mut}\ T;\ \Omega'}$$

New Borrow Rules for `RefCell`

- Bridges the semantics gap between `RefCell` and `GhostCell`
  - **Fine-grained** permissions for each node (`RefCell`) vs. **coarse-grained** permissions for whole DS (`GhostCell`)

- Only applies to `RefCells` used in data structures
  - Allows for a **constrained static analysis**

- If code passes sanitizer → it can be `GhostCell`-ified!

§4.2. Brand Inference

```
Procedure Ghostcellify($P$)
    Input: Program $P$
    Output: New program $\mathcal{P}'$
    $\mathcal{S} \leftarrow$ StructDefs($P$)
    $\mathcal{I} \leftarrow$ Impl($P$)
    $C \leftarrow$ ClientCode($P$)
    if not Sanitize($\mathcal{P}$) then
        Abort()
    GenerateBrands($\mathcal{S} \Downarrow_r \mathcal{S}'$)
    TransformImpl($\mathcal{I} \rightsquigarrow \mathcal{I}'$)
    TransformClient($C \Rightarrow_d C'$)
    $\mathcal{P}' \leftarrow \mathcal{S}' \cup \mathcal{I}' \cup C'$
```

Symbol	Meaning
$\mathcal{P}$	Program AST (RefCell-based)
$e : \mathrm{T}$	Sanitize §4.1
$\Downarrow_r$	Brand Inference §4.2
$\rightsquigarrow$	Rewrite Impl. Methods §4.3
$\Rightarrow_d$	Rewrite client code §4.4

§4.2. Brand Inference (I)

- We need to add brands to the struct definitions.
    - **But how many?**
        - **Won’t one brand suffice?** *(c.f. GhostCell paper, some use-cases from GitHub)*

§4.2. Brand Inference (II)

- We need to add brands to the struct definitions.
  - **But how many?**
    - **Won’t one brand suffice?** *(c.f. GhostCell paper, some use-cases from GitHub)*
    - **In simple cases, yes.**

§4.2. Brand Inference (III)

- We need to add brands to the struct definitions.
  - **But how many?**
    - **Won't one brand suffice?** *(c.f. GhostCell paper, common use-cases from GitHub)*
    - **In simple cases, yes.**

```
pub type NodePtr<'id, T> =
    Rc<GhostCell<?, Node<?, T>>>;


pub struct Node<'id, T> {
    data: T,
    prev: Option<NodePtr<?, T>>,
    next: Option<NodePtr<?, T>>,
}
```

```
pub struct List<'id, T> {
    head: Option<NodePtr<?, T>>,
    last: Option<NodePtr<?, T>>
}
```

§4.2. Brand Inference (IV)

- We need to add brands to the struct definitions.
  - **But how many?**
    - **Won't one brand suffice?** *(c.f. GhostCell paper, common use-cases from GitHub)*
    - **In simple cases, yes.**

```
pub type NodePtr<‘id, T> =
    Rc<GhostCell<‘id, Node<‘id, T>>>;


pub struct Node<‘id, T> {
    data: T,
    prev: Option<NodePtr<‘id, T>>,
    next: Option<NodePtr<‘id, T>>,
}
```

```
pub struct List<‘id, T> {
    head: Option<NodePtr<‘id, T>>,
    last: Option<NodePtr<‘id, T>>
}


let n1 = Node::new(1);  # brand ‘id
let n2 = Node::new(2);  # brand ’id
List::init(n1, n2);
```

§4.2. Brand Inference (V)

- We need to add brands to the struct definitions.
  - **But how many?**
    - **Won’t one brand suffice?** *(c.f. GhostCell paper, common use-cases from GitHub)*
    - **In simple cases, yes.**

```
pub type NodePtr<‘id, T> =
    Rc<GhostCell<‘id, Node<‘id, T>>>;


pub struct Node<‘id, T> {
    data: T,
    prev: Option<NodePtr<‘id, T>>,
    next: Option<NodePtr<‘id, T>>,
}
```

```
pub struct List<‘id, T> {
    head: Option<NodePtr<‘id, T>>,
    last: Option<NodePtr<‘id, T>>
}

let n1 = Node::new(1);   # brand ‘id
let n2 = Node::new(2);   # brand ’id
List::init(n1, n2);
```

✓

§4.2. Brand Inference (VI)

- We need to add brands to the struct definitions.
  - **But how many?**
    - **Won’t one brand suffice?** *(c.f. GhostCell paper, common use-cases from GitHub)*
    - **In simple cases, yes.**

```
pub type NodePtr<‘id, T> =
    Rc<GhostCell<‘id, Node<‘id, T>>>;


pub struct Node<‘id, T> {
    data: T,
    prev: Option<NodePtr<‘id, T>>,
    next: Option<NodePtr<‘id, T>>,
}
```

```
pub struct List<‘id, T> {
    head: Option<NodePtr<‘id, T>>,
    last: Option<NodePtr<‘id, T>>
}


let n1 = Node::new(1);  # brand ‘id
let n2 = Node::new(2);  # brand ‘id2
List::init(n1, n2);
```

§4.2. Brand Inference (VII)

- We need to add brands to the struct definitions.
  - **But how many?**
    - **Won't one brand suffice?** *(c.f. GhostCell paper, common use-cases from GitHub)*
    - **In simple cases, yes.**

```
pub type NodePtr<'id, T> =
    Rc<GhostCell<'id, Node<'id, T>>>;


pub struct Node<'id, T> {
    data: T,
    prev: Option<NodePtr<'id, T>>,
    next: Option<NodePtr<'id, T>>,
}
```

```
pub struct List<'id, T> {
    head: Option<NodePtr<'id, T>>,
    last: Option<NodePtr<'id, T>>
}


let n1 = Node::new(1);  # brand 'id
let n2 = Node::new(2);  # brand 'id2
List::init(n1, n2);
```

§4.2. Brand Inference (VIII)

- We need to add brands to the struct definitions.
  - **But how many?**
    - **Won’t one brand suffice?** *(c.f. GhostCell paper, common use-cases from GitHub)*
    - **Not in more complex settings** where we store ***auxiliary data*****!** (can’t modify both)

```
pub type NodePtr<‘id, T> = ...
pub type StatsPtr<‘id> = ...
pub struct Node<‘id, T> {
    data: T,
    stats: StatsPtr<‘id >
    prev: Option<NodePtr<‘id, T>>,
    next: Option<NodePtr<‘id, T>>,
}
```

§4.2. Brand Inference (IX)

- We need to add brands to the struct definitions.
  - **But how many?**
    - **Won't one brand suffice?** *(c.f. GhostCell paper, common use-cases from GitHub)*
    - **Not in more complex settings** where we store ***auxiliary data*****!** (can't modify both)

```
pub type NodePtr<'id, T> = ...          fn update(&self, token) {
pub type StatsPtr<'id> = ...                let prev = node.prev.borrow_mut(token);
pub struct Node<'id, T> {                   let stats = node.stats.borrow_mut(token);
    data: T,                                modify_prev(prev);
    stats: StatsPtr<'id >                   modify_stats(stats);
    prev: Option<NodePtr<'id, T>>,      }
    next: Option<NodePtr<'id, T>>,
}
```

§4.2. Brand Inference (X)

- We need to add brands to the struct definitions.
  - **But how many?**
    - **Won't one brand suffice?** *(c.f. GhostCell paper, common use-cases from GitHub)*
    - **Not in more complex settings** where we store ***auxiliary data*****!** (can't modify both)

```
pub type NodePtr<'id, T> = ...
pub type StatsPtr<'id> = ...
pub struct Node<'id, T> {
    data: T,
    stats: StatsPtr<'id >
    prev: Option<NodePtr<'id, T>>,
    next: Option<NodePtr<'id, T>>,
}
```

```
fn update(&self, token) {
    let prev = node.prev.borrow_mut(token);
    let stats = node.stats.borrow_mut(token);
    modify_prev(prev);
    modify_stats(stats);
}
```

§4.2. Brand Inference (XI)

- We need to add brands to the struct definitions.
  - **But how many?**
    - **Won't one brand suffice?** *(c.f. GhostCell paper, common use-cases from GitHub)*
    - **Not in more complex settings** where we store ***auxiliary data*****!** (can't modify both)

```
pub type NodePtr<‘id, ‘id2, T> = ...
pub type StatsPtr<‘id> = ...

pub struct Node<‘id, ‘id2, T> {
    data: T,
    stats: StatsPtr<‘id>
    prev: Option<NodePtr<‘id, ‘id2, T>>,
    next: Option<NodePtr<‘id, ’id2, T>>,
}
```

```
fn update(&self, token, token2) {
    let prev = node.prev.borrow_mut(token);
    let stats = node.stats.borrow_mut(token);
    modify_prev(prev);
    modify_stats(stats);
}
```

§4.2. Brand Inference (XI)

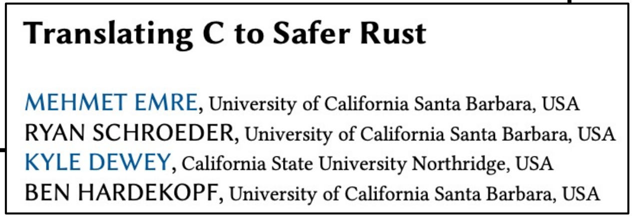
- We need to add brands to the struct definitions.
  - **But how many?**
    - **Won’t one brand suffice?** *(c.f. GhostCell paper, common use-cases from GitHub)*
    - **Not in more complex settings** where we store ***auxiliary data*!** (can’t modify both)

```
pub type NodePtr<‘id, ‘id2, T> = ...
pub type StatsPtr<‘id> = ...

pub struct Node<‘id, ‘id2, T> {
    data: T,
    stats: StatsPtr<‘id>
    prev: Option<NodePtr<‘id, ‘id2, T>>,
    next: Option<NodePtr<‘id, ’id2, T>>,
}
```

```
fn update(&self, token, token2) {
    let prev = node.prev.borrow_mut(token);
    let stats = node.stats.borrow_mut(token);
    modify_prev(prev);
    modify_stats(stats);
}
```

✓



Translating C to Safer Rust

MEHMET EMRE, University of California Santa Barbara, USA
RYAN SCHROEDER, University of California Santa Barbara, USA
KYLE DEWEY, California State University Northridge, USA
BEN HARDEKOPF, University of California Santa Barbara, USA

§4.2. Brand Inference (XIII)

UniqueFields(Node)
= Node, Stats

B-PRIM-TY

$$\frac{T \in Prim}{T \rightharpoonup_{\mathrm{b}} T;}$$

$\mathrm{B_i}$ = 'id, 'id2

T-REF-CELL

$$\frac{T \rightharpoonup_{\mathrm{b}} T'; B_T \qquad \overline{f_i} = \mathtt{UniqueFields(T)} \qquad \overline{b_i}\, fresh}{\mathtt{RefCell} < \mathtt{T} > \rightharpoonup_{\mathrm{b}} \mathtt{GhostCell} < \overline{\mathtt{b_i}}, \mathtt{T'} >; \mathtt{B_T} \cup \overline{\mathtt{b_i}}}$$

T-STRUCT-DEF

$$\frac{T_i \rightharpoonup_{\mathrm{b}} T_i'; B_{T_i} \qquad B = \bigcup_i B_{T_i}}{\mathtt{struct\ S} < \overline{\mathtt{A_i}} > \{\overline{\mathtt{f_i : T_i}}\} \rightharpoonup_{\mathrm{b}} \mathtt{struct\ S} < \overline{\mathtt{A_i}}, \mathtt{B} > \{\overline{\mathtt{f_i : T_i'}}\}; \mathtt{B}}$$

```
pub struct Node<‘id, ‘id2, T> {
    data: T,
    stats: StatsPtr<‘id>
    prev: Option<NodePtr<‘id, T>>,
    next: Option<NodePtr<‘id, T>>,
}
```

§4.3. Rewrite Impl. Methods

```
Procedure Ghostcellify($P$)
    Input: Program $P$
    Output: New program $\mathcal{P}'$
    $\mathcal{S} \leftarrow$ StructDefs($P$)
    $\mathcal{I} \leftarrow$ Impl($P$)
    $\mathcal{C} \leftarrow$ ClientCode($P$)
    if not Sanitize($\mathcal{P}$) then
        Abort()
    GenerateBrands($\mathcal{S} \Downarrow_r \mathcal{S}'$)
    TransformImpl($\mathcal{I} \rightsquigarrow \mathcal{I}'$)
    TransformClient($\mathcal{C} \Rightarrow_d \mathcal{C}'$)
    $\mathcal{P}' \leftarrow \mathcal{S}' \cup \mathcal{I}' \cup \mathcal{C}'$
```

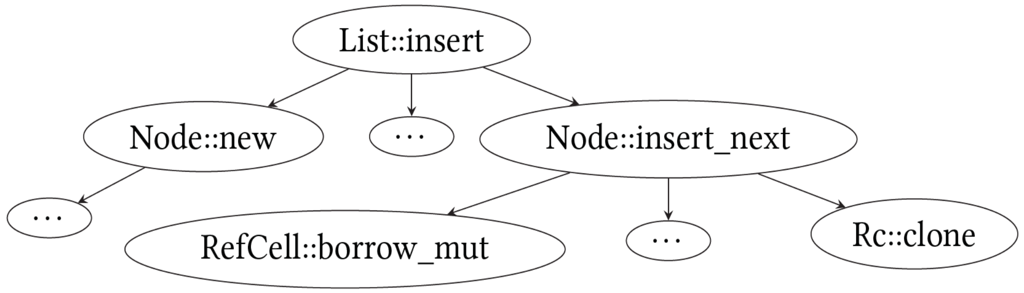
Symbol	Meaning
$\mathcal{P}$	Program AST (RefCell-based)
$e : \mathrm{T}$	Sanitize §4.1
$\Downarrow_r$	Brand Inference §4.2
$\rightsquigarrow$	Rewrite Impl. Methods §4.3
$\Rightarrow_d$	Rewrite client code §4.4

§4.3. Rewrite Impl. Methods (I)

- We perform the following steps:

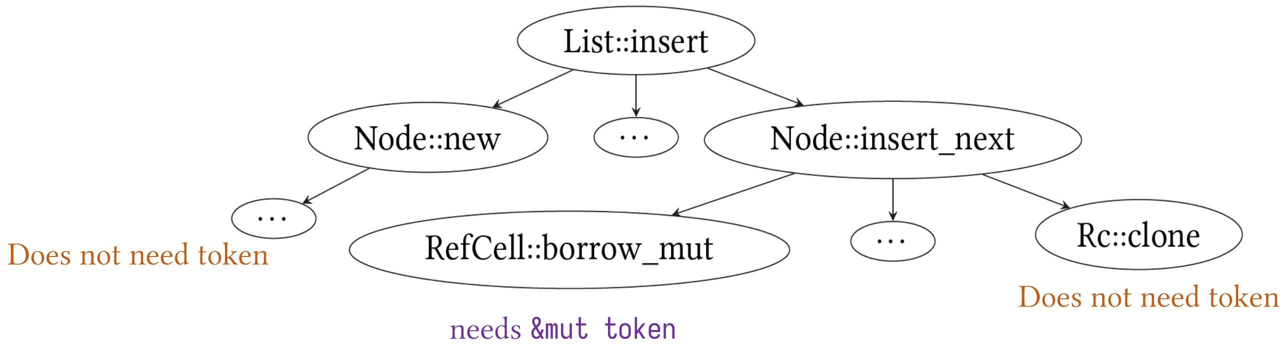
§4.3. Rewrite Impl. Methods (I)

- We perform the following steps:
    1. Construct a call-graph



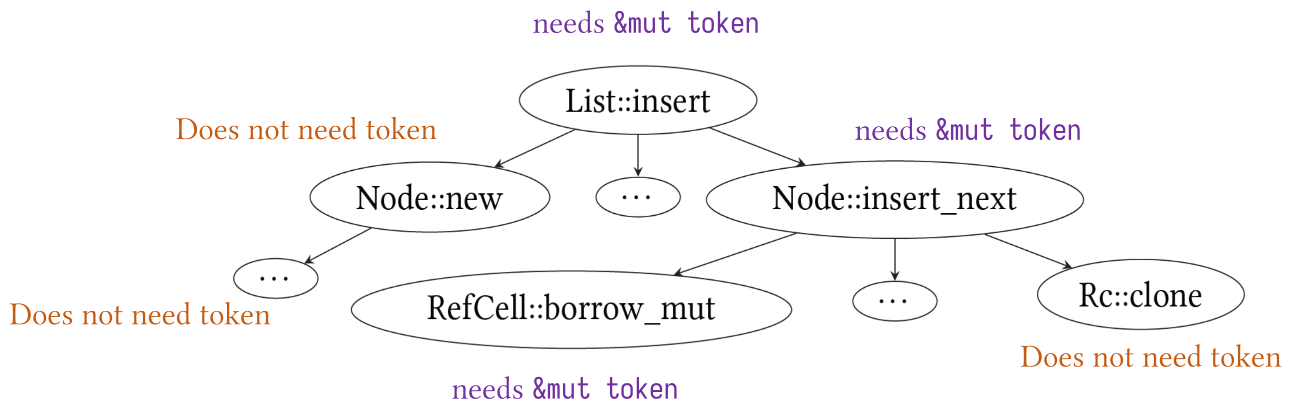
§4.3. Rewrite Impl. Methods (I)

- We perform the following steps:
  1. Construct a call-graph
  2. Annotating the leaves with the correct token permission



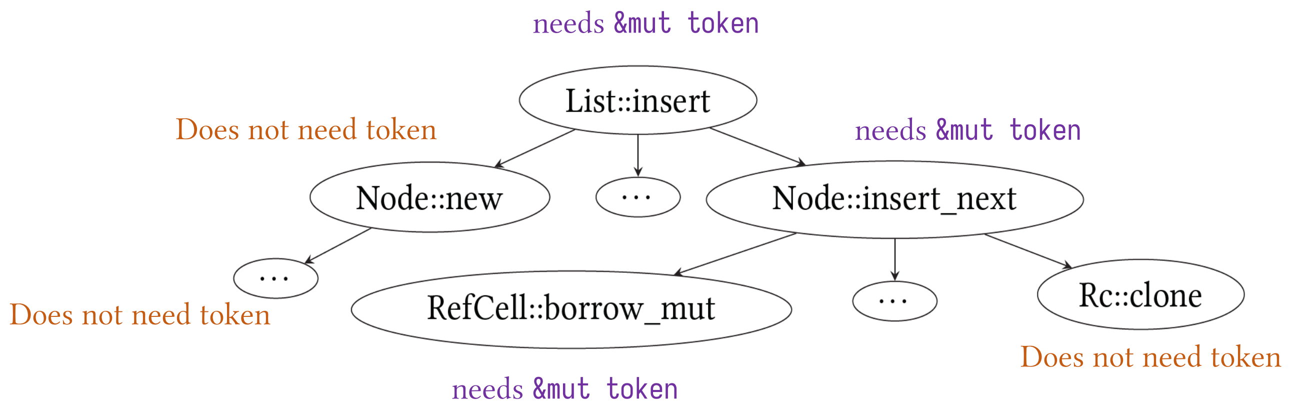
§4.3. Rewrite Impl. Methods (I)

- We perform the following steps:
  1. Construct a call-graph
  2. Annotating the leaves with the correct token permission
  3. Propagating the token permission up the graph



§4.3. Rewrite Impl. Methods (I)

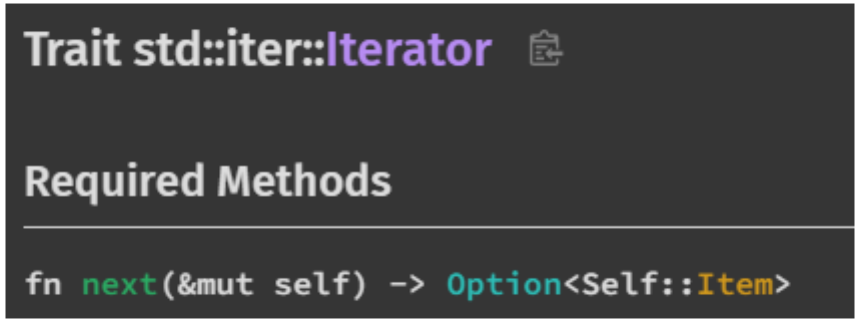
- We perform the following steps:
  1. Construct a call-graph
  2. Annotating the leaves with the correct token permission
  3. Propagating the token permission up the graph
  4. Rewriting the code using information from call graph



§4.3. Rewrite Impl. Traits (II)

- Traits require specific method signatures
  - We cannot supply tokens by adding them to the signature

```rust
impl<T> Iterator for List<T> {
    fn next(&mut self) -> Option<&T> {
        if let Some(x) = self.head {
            let res = x.borrow();
            self.head = res.next;
            return Some(res);
        }
        return None;
}}
```



Trait std::iter::Iterator

Required Methods

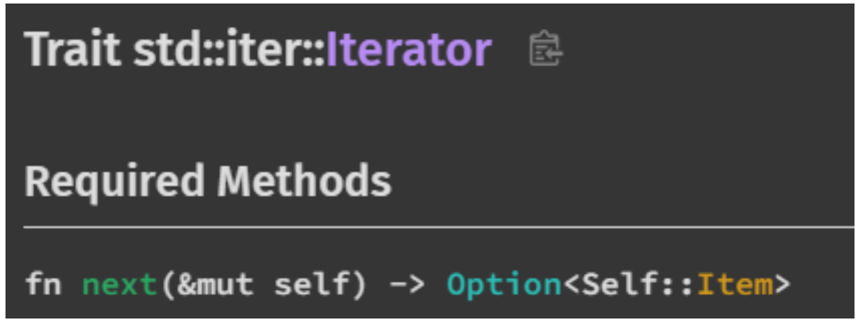
```rust
fn next(&mut self) -> Option<Self::Item>
```

Image credit: https://doc.rust-lang.org/std/iter/trait.Iterator.html

§4.3. Rewrite Impl. Traits (III)

- Traits require specific method signatures
  - We cannot supply tokens by adding them to the signature

```
impl<T> Iterator for List<T> {
    fn next(&mut self) -> Option<&T> {
        if let Some(x) = self.head {
            let res = x.borrow();
            self.head = res.next;
            return Some(res);
        }
        return None;
}}
```



Trait std::iter::Iterator

Required Methods

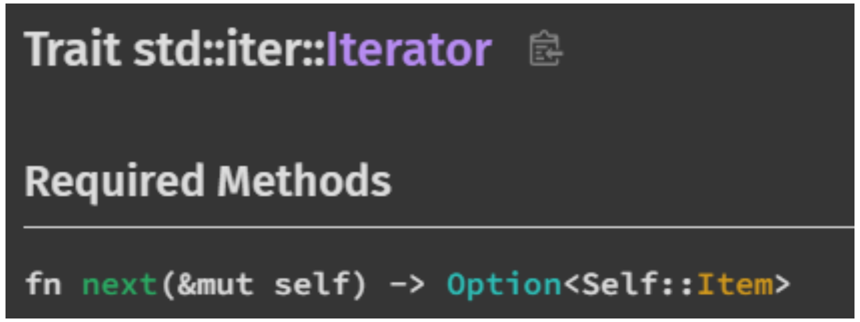
```
fn next(&mut self) -> Option<Self::Item>
```

Image credit: https://doc.rust-lang.org/std/iter/trait.Iterator.html

§4.3. Rewrite Impl. Traits (IV)

- Traits require specific method signatures
  - We cannot supply tokens by adding them to the signature

```
impl<'id, T> Iterator for List<'id, T> {
    fn next(&mut self, tok: &mut GhostToken<'id>) -> Option<&T> {
        if let Some(x) = self.head {
            let res = x.borrow(tok);
            self.head = res.next;
            return Some(res);
        }
        return None;
}}
```


Trait std::iter::Iterator

Required Methods

```
fn next(&mut self) -> Option<Self::Item>
```

Image credit: https://doc.rust-lang.org/std/iter/trait.Iterator.html

§4.3. Rewrite Impl. Traits (V)

- Traits require specific method signatures
  - We cannot supply tokens by adding them to the signature
  - **Solution: wrap the data structure with the token**

```rust
pub struct IteratorWrapper<'id, T> {
    inner: Rc<GhostCell<'id, T>>,
    token: &mut GhostToken<'id>
}
```

```rust
impl<'id, T> Iterator for IteratorWrapper<'id, T> {
    fn next(&mut self) -> Option<&T> {
        if let Some(x) = self.head {
            let res = x.borrow();
            self.head = res.next;
            return Some(res);
        }
        return None;
}}
```

§4.3. Rewrite Impl. Traits (VI)

- Traits require specific method signatures
  - We cannot supply tokens by adding them to the signature
  - **Solution: wrap the data structure with the token**

```rust
pub struct IteratorWrapper<'id, T> {
    inner: Rc<GhostCell<'id, T>>,
    token: &mut GhostToken<'id>
}
```

```rust
impl<'id, T> Iterator for IteratorWrapper<'id, T> {
    fn next(&mut self) -> Option<&T> {
        if let Some(x) = self.head {
            let res = x.borrow();
            self.head = res.next;
            return Some(res);
        }
        return None;
}}
```

§4.3. Rewrite Impl. Traits (VII)

- Traits require specific method signatures
    - We cannot supply tokens by adding them to the signature
    - **Solution: wrap the data structure with the token**

```
pub struct IteratorWrapper<'id, T> {
    inner: Rc<GhostCell<'id, T>>,
    token: &mut GhostToken<'id>
}
```

```
impl<'id, T> Iterator for IteratorWrapper<'id, T> {
    fn next(&mut self) -> Option<&T> {
        if let Some(x) = self.head {
            let res = x.borrow(self.token);
            self.head = res.next;
            return Some(res);
        }
        return None;
}}
```

§4.4. Rewrite Client Code

```
Procedure Ghostcellify(P)
    Input: Program P
    Output: New program $\mathcal{P}'$
    $\mathcal{S} \leftarrow$ StructDefs(P)
    $\mathcal{I} \leftarrow$ Impl(P)
    $C \leftarrow$ ClientCode(P)
    if not Sanitize($\mathcal{P}$) then
        Abort()
    GenerateBrands($\mathcal{S} \Downarrow_r \mathcal{S}'$)
    TransformImpl($\mathcal{I} \rightsquigarrow \mathcal{I}'$)
    TransformClient($C \Rightarrow_d C'$)   ⟵
    $\mathcal{P}' \leftarrow \mathcal{S}' \cup \mathcal{I}' \cup C'$
```

Symbol	Meaning
$\mathcal{P}$	Program AST (RefCell-based)
$e : \mathrm{T}$	Sanitize §4.1
$\Downarrow_r$	Brand Inference §4.2
$\rightsquigarrow$	Rewrite Impl. Methods §4.3
$\Rightarrow_d$	Rewrite client code §4.4

§4.4. Rewrite Client Code (I)

- We need to create tokens to brand the data-structures.

```
fn main() {
    let mut a, b, c = List::new(..),
                      List::new(..)
                      List::new(..);
    let c_head = c.head.borrow_mut();
    a.merge(b_head);
    Node::map(&c_head, |x| x + 1);
}
```

§4.4. Rewrite Client Code (II)

- We need to create tokens to use the data-structures.
  - Can we just create one token?
    - **Not if we want multiple mutable references to *different* instances of a struct.**

```
fn main() {
    let mut a, b, c = List::new(..),
                      List::new(..)
                      List::new(..);
    let c_head = c.head.borrow_mut();
    a.merge(b_head);
    Node::map(&c_head, |x| x + 1);
}
```

§4.4. Rewrite Client Code (III)

- We need to create tokens to use the data-structures.
  - Can we just create one token?
    - **Not if we want multiple mutable references to *different instances of a struct.***

```
fn main() {
    let mut a, b, c = List::new(..),
                      List::new(..)
                      List::new(..);
    let c_head = c.head.borrow_mut();
    a.merge(b_head);
    Node::map(&c_head, |x| x + 1);
}
```

```
fn main() {
GhostToken::new(|t1| {
    let mut a, b, c = List::new(..),
                      List::new(..)
                      List::new(..);
    let c_head = c.head.borrow_mut(t1);
    a.merge(b_head, t1);
    Node::map(&c_head, |x| x + 1, t1);
});
```

§4.4. Rewrite Client Code (IV)

- We need to create tokens to use the data-structures.
  - Can we just create one token?
    - **Not if we want multiple mutable references to *different instances of a struct.***

```rust
fn main() {
    let mut a, b, c = List::new(..),
                      List::new(..)
                      List::new(..);
    let c_head = c.head.borrow_mut();
    a.merge(b_head);
    Node::map(&c_head, |x| x + 1);
}
```

```rust
fn main() {
GhostToken::new(|t1| {
    let mut a, b, c = List::new(..),
                      List::new(..)
                      List::new(..);
    let c_head = c.head.borrow_mut(t1);
    a.merge(b_head, t1);
    Node::map(&c_head, |x| x + 1, t1);
});
```

§4.4. Rewrite Client Code (V)

- We need to create tokens to use the data-structures.
  - Can we just create one token?
    - **Not if we want multiple mutable references to *different instances of a struct.***

```
fn main() {
    let mut a, b, c = List::new(..),
                      List::new(..)
                      List::new(..);
    let c_head = c.head.borrow_mut();
    a.merge(b_head);
    Node::map(&c_head, |x| x + 1);
}
```

```
GhostToken::new(|t1| {
    GhostToken::new(|t2| {
    let mut a, b, c = List::new(..),
                      List::new(..)
                      List::new(..);
    let c_head = c.head.borrow_mut(t2);
    a.merge(b_head, t1);
    Node::map(&c_head, |x| x + 1, t2);
})});
```

§4.4. Rewrite Client Code (VI)

- We need to create tokens to use the data-structures.
  - Can we just create one token?
    - **Not if we want multiple mutable references to *different instances of a struct*.**

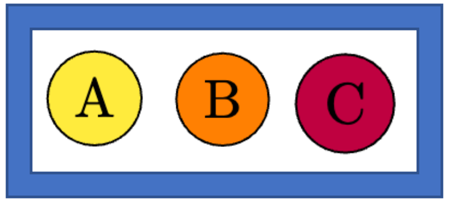
```rust
fn main() {
    let mut a, b, c = List::new(..),
                      List::new(..)
                      List::new(..);
    let c_head = c.head.borrow_mut();
    a.merge(b_head);
    Node::map(&c_head, |x| x + 1);
}
```

```rust
GhostToken::new(|t1| {
    GhostToken::new(|t2| {
     let mut a, b, c = List::new(..),
                       List::new(..)
                       List::new(..);
     let c_head = c.head.borrow_mut(t2);
     a.merge(b_head, t1);
     Node::map(&c_head, |x| x + 1, t2);
})});
```

Partition the heap + Assign brands

§4.4. Rewrite Client Code (VII)

- ~~Static~~ Dynamic Analysis over Rust's MIR (LLVM-like IR)



D-ALLOC

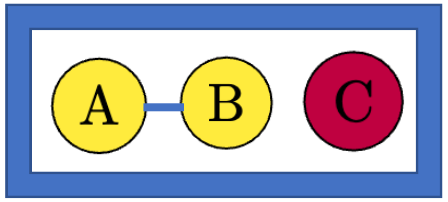
$$\frac{\mathcal{M}' = \mathcal{M}[x \rightarrow m] \qquad T \in \mathcal{T}}{V, E, \mathcal{M}; \texttt{let } x = \texttt{ALLOC(T)} \Rightarrow_{\mathrm{d}} \mathrm{V} \cup \{\texttt{m}\}, \mathrm{E}, \mathcal{M}'}$$

D-WRITE

$$\frac{\mathcal{M}(v) = m_v \qquad \mathcal{M}(x) = m_x}{V, E, \mathcal{M}; *x \ = \ v \Rightarrow_{\mathrm{d}} V, E \cup (m_v, m_x), \mathcal{M}}$$

D-READ

$$\frac{\mathcal{M}(x) = m_x \qquad \mathcal{M}' = \mathcal{M}[y \rightarrow m_x]}{V, E, \mathcal{M}; \texttt{let } y = *\mathrm{x} \Rightarrow_{\mathrm{d}} \mathrm{V}, \mathrm{E}, \mathcal{M}'}$$



Built-in (Milano 2022)'s type system

Evaluation

Benchmarks

- Currently, GHOSTCELLIFY translates 4 data-structures written using RefCell
    - Doubly-linked list
    - Graph (adjacency list)
    - Binary tree (with parent pointers)
    - Skiplist

- We also exercised GHOSTCELLIFY on multiple versions of each data-structure, by adding:
    - Adding auxiliary fields (e.g. `stats`)
    - Trait implementations
    - Generics

Performance Improvement on Benchmarks

	Rc-RefCell	Rc-GhostCell
DList	9.18	**3.68** ✓
DList-Aux	15.42	**6.43** ✓
Graph	15.18	**11.72** ✓
Bree	20.76	**9.19** ✓

Median time for inserting 100,000 nodes (ms)

Overview of Benches (Snippets)

```
pub struct BTree<T> {
    root: Option<Rc<RefCell<Node>>>,
}
pub struct Node<T> {
    children: [Option<Rc<RefCell<Node>>; 2],
    parent: Option<Rc<RefCell<Node>>,
    key: i32
}
…
7 methods ✓, 4 traits ✓, 2 traits ✗
```

Binary Tree

Traits such as **Display** and **Debug** could not be implemented for **BTree.**

Similar case for **Skiplist** and **Graph**.

Bench: Failure mode

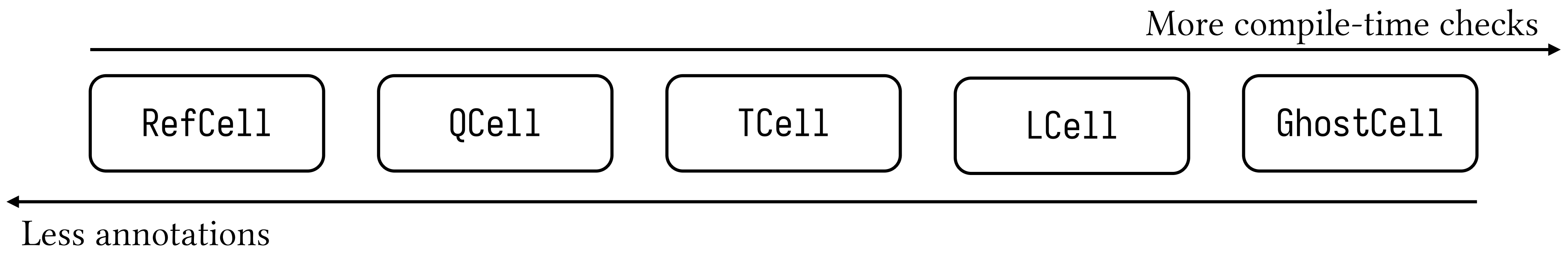
- Circle-DLL
  - Why? Does not pass sanitizer check, since it holds multiple mutable references, and **violates the lifted RefCell borrowing rules**

```
impl CircleDLL {
  pub fn remove(&mut self)  {
    ....
    self.last.borrow_mut().next = Some(next);
    Self.next.borrow_mut().last = Some(last);
}
```

Future Work

Other Cell-types

- Various tradeoffs for performance and safety guarantees
- Token-based Cells akin to `GhostCell`:
  - `QCell`: token based on integer ID
  - `TCell`: token based on marker type
  - `LCell`: token based on lifetime
- Different APIs & semantics compared to `RefCell` / `GhostCell`



Other Memory Management Schemes

- Extension to other memory management schemes, particularly **Arena**
- In particular the various Arena schemes:
  - `typed-arena`
  - `bumpalo`
  - `id_arena`

	Mutex	**RefCell**	**GhostCell**
Rc	30.78 ms	9.18 ms	3.68 ms
Arena	**7.95 ms**	**4.5 ms**	**0.47 ms**

Median time of inserting 100 000 nodes on different doubly-linked list implementations

Extensions

- Expanding to other Cell-types and memory-management schemes

	RefCell	GhostCell	QCell	TCell	LCell	Mutex
Rc	🔵	✅	?	?	?	?
Arena	?	?	?	?	?	?

Limitations & Conclusion

Limitations of Our Work

- Conservative static analysis for sanitizer
    - No support for mutually-recursive structs due to ambiguous brand assignment

- Only works on structures defined within the same module
    - Limitation due to the usage of `rustc_lint`

- Supports only a subset of Rust syntax, excludes:
    - Multiple generic parameters
    - Traits auto-derived from the `#derive` attribute
    - Client code not encapsulated in tests or `main()`

Conclusion

- In this work, we presented **GHOSTCELLIFY**, a tool to make Rust code more performant and safer by rewriting `RefCell` → `GhostCell`
  - To our knowledge, our tool is the first such work on type-driven transformation of Rust containers
- We formalized the semantics for a subset of Rust, `GhostCell` and devised borrow rules for `RefCell`
- We believe our framework can extend / generalize to various Cell-types and Memory Management schemes across the Rust landscape

Thank you!

Performance Comparisons

	Rc-RefCell	Rc-GhostCell
DList	9.18	**3.68**
DList-Aux	15.42	**6.43**
Graph	15.18	**11.72**
Bree	20.76	**9.19**

Median time for inserting 100,000 nodes (ms)