Ownership Guided C to Rust Translation

Literature Review

Oct 03, 2025

Why is Rust Important

- Memory Safe
- Type Safe
- Comparable Performance to C

⇒ A lot of interest in converting C code to Rust code.

C ⇒ Rust Translation

C2Rust → Rewrites C code into Rust while preserving original semantics

Laertes → Rewrites unsafe code C2Rust into Safe Rust

- Uses type error messages from Rust compiler.
- Trial and Error based approach – Does not create any understanding of the code expect for the fact it compiles.

CROWN

- Builds an ownership model of pointers and rewrite C2Rust unsafe code into safe Rust.
- The analysis is:
  - Efficient (scales to half a million lines in 10 seconds)
  - Sophisticated (handles nested pointers)
  - Precise (field and flow sensitive)
    - Field Sensitive : Can distinguish b/w fields of struct
    - Flow Sensitive : Respects program control flow and updates accordingly

Background

Are you all familiar with all of these terms?

- Ownership
- Borrowing
- Lifetime
- Raw pointers
- Box Pointer

Background

Are you all familiar with all of these terms?

- Ownership
- Borrowing
- Lifetime
- Raw pointers
- Box Pointer
- Unsafe

Background

Are you all familiar with all of these terms?

- Ownership
- Borrowing
- Lifetime
- Raw pointers
- Box Pointer
- Unsafe

```
1  struct Node {
2      int data;
3      struct Node * next;
4  };
5
6  struct List {
7      Node * head;
8  };
9
10 void push(struct List* list, int
           new_data) {
11     struct Node* new_node = (struct
           Node*) malloc(sizeof(struct
           Node));
12     new_node->data  = new_data;
13     new_node->next = list->head;
14     list->head    = new_node;
15 }
16
```

Consider this C code for a LinkedList

```c
struct Node {
    int data;
    struct Node * next;
};

struct List {
    Node * head;
};

void push(struct List* list, int
        new_data) {
    struct Node* new_node = (struct
        Node*) malloc(sizeof(struct
        Node));
    new_node->data  = new_data;
    new_node->next = list->head;
    list->head    = new_node;
}
```

C code

```rust
#[repr(C)]
#[derive(Copy, Clone)]
pub struct Node {
    pub data: i32,
    pub next: *mut Node,
}

#[repr(C)]
#[derive(Copy, Clone)]
pub struct List {
    pub head: *mut Node,
}

pub unsafe extern "C" fn push(mut
        list: *mut List, mut
        new_data: i32) {
    let mut new_node = malloc(::std
        ::mem::size_of::<Node>() as
        libc::c_ulong) as *mut Node;
    (*new_node).data = new_data;
    (*new_node).next = (*list).head;
    (*list).head = new_node;
}
```

C2Rust Output Code

```c
struct Node {
    int data;
    struct Node * next;
};

struct List {
    Node * head;
};

void push(struct List * list, int
        new_data){
    struct Node* new_node = (struct
        Node*) malloc(sizeof(struct
        Node));
    new_node->data  = new_data;
    new_node->next = list->head;
    list->head = new_node;
}
```

C code

```rust
#[repr(C)]
#[derive(Copy, Clone)]
pub struct Node {
    pub data: i32,
    pub next: *mut Node,
}

#[repr(C)]
#[derive(Copy, Clone)]
pub struct List {
    pub head: *mut Node,
}

pub unsafe extern "C" fn push(mut
        list: *mut List, mut
        new_data: i32) {
    let mut new_node = malloc(::std
        ::mem::size_of::<Node>() as
        libc::c_ulong) as *mut Node;
    (*new_node).data = new_data;
    (*new_node).next = (*list).head;
    (*list).head = new_node;
}
```

C2Rust Output Code

Raw pointers being Dereferenced, Unsafe ….. Etc etc

```rust
#[repr(C)]
#[derive(Copy, Clone)]
pub struct Node {
    pub data: i32,
    pub next: *mut Node,
}

#[repr(C)]
#[derive(Copy, Clone)]
pub struct List {
    pub head: *mut Node,
}

pub unsafe extern "C" fn push(mut
        list: *mut List, mut
        new_data: i32) {
    let mut new_node = malloc(::std
        ::mem::size_of::<Node>() as
        libc::c_ulong) as *mut Node;
    (*new_node).data = new_data;
    (*new_node).next = (*list).head;
    (*list).head = new_node;
}
```

C2Rust Output Code

```rust
#[repr(C)]
pub struct Node {
    pub data: i32,
    pub next: Option<Box<Node>>,
}

#[repr(C)]
pub struct List {
    pub head: Option<Box<Node>>,
}

pub unsafe extern "C" fn push(mut
        list: Option<&mut List>, mut
         new_data: i32) {
    let mut new_node = Some(Box::new
        (<Node as Default>::default
        ()));
    (*new_node.as_deref_mut().unwrap
        ()).data = new_data;
    (*new_node.as_deref_mut().unwrap
        ()).next = (*list.
        as_deref_mut().unwrap()).
        head.take();
    (*list.as_deref_mut().unwrap()).
        head = new_node;
}
```

CROWN output Code

```
1  typedef
2  struct zzzz {
3      Char         *name;
4      struct zzzz *link;
5  }
6  Cell;
7  [...]
8  Cell* aa = argList;
9  while (aa != NULL) {
10     Cell* aa2 = aa->link;
11     if (aa->name)
12         free(aa->name);
13     free(aa);
14     aa = aa2;
15 }
16 [...]
17
```

What if there is a free in a loop?

Cell is a Linkedlist like data structure that holds a list of argument names. This snippet frees that argument list.

```c
typedef
struct zzzz {
    Char        *name;
    struct zzzz *link;
}
Cell;
[...]
Cell* aa = argList;
while (aa != NULL) {
    Cell* aa2 = aa->link;
    if (aa->name)
        free(aa->name);
    free(aa);
    aa = aa2;
}
[...]
```

C code

```rust
#[derive(Copy, Clone)]
#[repr(C)]
pub struct zzzz {
    pub name: *mut Char,
    pub link: *mut zzzz,
}
pub type Cell = zzzz;
[...]
let mut aa: *mut Cell = argList;
while !aa.is_null() {
    let mut aa2 = (*aa).link;
    if !(*aa).name.is_null() {
        free((*aa).name as *mut libc
        ::c_void);
    }
    free(aa as *mut libc::c_void);
    aa = aa2;
}
[...]
```

C2Rust code

```rust
#[derive(Copy, Clone)]
#[repr(C)]
pub struct zzzz {
    pub name: *mut Char,
    pub link: *mut zzzz,
}
pub type Cell = zzzz;
[...]
let mut aa: *mut Cell = argList;
while !aa.is_null() {
    let mut aa2 = (*aa).link;
    if !(*aa).name.is_null() {
        free((*aa).name as *mut libc
        ::c_void);
    }
    free(aa as *mut libc::c_void);
    aa = aa2;
}
[...]
```

C2Rust code

```rust
#[repr(C)]
pub struct zzzz {
    pub name: *mut /* owning */ Char
        ,
    pub link: Option<Box<zzzz>>,
}
pub type Cell = zzzz;
[...]
let mut aa: Option<Box<Cell>> =
        argList;
while !aa.as_deref().is_none() {
    let mut aa2 = (*aa.as_deref_mut
        ().unwrap()).link.take();
    if !(*aa.as_deref().unwrap()).
        name.is_null {
        free((*aa.as_deref().unwrap
        ()).name as *mut libc::
        c_void);
    }
    aa = aa2;
}
[...]
```

CROWN code

Cool, but how does CROWN actually work?

- Applies several static analysis on MIR to infer properties of pointers:

Cool, but how does CROWN actually work?

- Applies several static analysis on MIR to infer properties of pointers:
  1. Ownership Analysis : Infer's about each pointer whether it is owning/non-owning at each location.

Cool, but how does CROWN actually work?

- Applies several static analysis on MIR to infer properties of pointers:
  1. Ownership Analysis: Infer's about each pointer whether it is owning/non-owning at each location.
  2. Mutability Analysis: Infer's which pointers are used as mutable.

Cool, but how does CROWN actually work?

- Applies several static analysis on MIR to infer properties of pointers:
  1. Ownership Analysis: Infer's about each pointer whether it is owning/non-owning at each location.
  2. Mutability Analysis: Infer's which pointers are used as mutable.
  3. Fatness Analysis: Distinguishes array pointers to non-array pointers.

Ownership Analysis

- Contains information whether points are owning or non-owning at particular locations.
- Basically creates a set of ownership constraints which are than solved by a SAT solver.
- A satisfying assignment for these ownership constraints in an ownership scheme that obeys Rust semantics.

Ownership Analysis

- This is field and flow sensitive.
  - For field sensitivity, they track ownership for `access paths`.
  -

Ownership Analysis

- This is field and flow sensitive.
  - For field sensitivity, they track ownership for `access paths`.
  - Examples access path are new_node, (*new_node).next and (*list.head).
  - Associate an ownership variable with each access path.
  - Ownership of access path = Ownership of the field accessed last through the path (i.e, in *(n).next, it will refer to ownership of next.)

```
1  #[repr(C)]
2  #[derive(Copy, Clone)]
3  pub struct Node {
4      pub data: i32,
5      pub next: *mut Node,
6  }
7
8  #[repr(C)]
9  #[derive(Copy, Clone)]
10 pub struct List {
11     pub head: *mut Node,
12 }
13
14 pub unsafe extern "C" fn push(mut
       list: *mut List, mut
       new_data: i32) {
15     let mut new_node = malloc(::std
       ::mem::size_of::<Node>() as
       libc::c_ulong) as *mut Node;
16     (*new_node).data = new_data;
17     (*new_node).next = (*list).head;
18     (*list).head = new_node;
19 }
20
```

Ownership and Aliasing

- Let's assume before assignment we can infer q (*q).nest and r are owning, while p and (*p).next are non-owning.

```
…
let p, r, q: *mut Node;
(*p).next = r
// *(p).next must have ownership
…
```

Ownership and Aliasing

- Let's assume before assignment we can infer q (*q).nest and r are owning, while p and (*p).next are non-owning.
- What if *p and *q alias?

```
…
let p, r, q: *mut Node;
(*p).next = r
// *(p).next must have
ownership
…
```

Ownership and Aliasing

- Let's say before line 2, p and q are aliasing. {(*p).next and (*q).next also alias}
- At line 3, both (*p).next and (*q).next both will be owning. Which should not happen and the above model cannot capture this.

```
…
1.  let p, r, q: *mut Node;
2.  (*p).next = r
3.  // *(p).next must have ownership
…
```

Ownership Monotonicity

- Restrict the way in each pointers can acquire ownership, thus limiting the interaction b/w ownership and aliasing.
- Stricter than Rust semantics

Given two access paths a and b, if is prefix(a, b), then O(a) ≥ O(b).

“The ownership values of a pointer can only decrease. “

What does this mean 🥲

The ownership monotonicity implies that, for access path (*p).next we have O(p) ≥ O((*p).next), and for access path (*q).next we have O(q) ≥ O((*q).next). This means that, if (*p).next is allowed to take ownership, then p must already be owning. Consequently, all aliases of p must be non-owning, which means that all aliases of (*p).next, including (*q).next, are non-owning.

Generation of Ownership Constraints

1. Ownership Transfer : The program instruction where transfer can happen are (pointer) assignment and function call.
2. Rule for ownership transfer at assignment time follow Box semantics:
    a. When a box pointer is moved, the object it points to it moved as well.
    b. Ownership transfer is for all layers.

Fancy Math

ASSIGN

$$v = base_var(\mathtt{p}), \quad w = base_var(\mathtt{q}),$$
$$a \in ap(v, |\mathtt{p}|, k), \quad b \in ap(w, |\mathtt{q}|, k), \quad c \in ap(v, 1, |\mathtt{p}|-1), \quad d \in ap(w, 1, |\mathtt{q}|-1)$$
$$|a| - |\mathtt{p}| = |b| - |\mathtt{q}|, |c| = |d|$$
$$is_prefix(p, a), \ is_prefix(q, b), \ is_prefix(c, p), \ is_prefix(d, q)$$
$$C' = C \cup \{\mathbb{O}_a = 0 \wedge \mathbb{O}_{a'} + \mathbb{O}_{b'} = \mathbb{O}_b \ \wedge \mathbb{O}_{c'} = \mathbb{O}_c \wedge \mathbb{O}_{d'} = \mathbb{O}_d\}$$

$$C \vdash \mathtt{p = q;} \Rightarrow C'$$

Page 10, last para

Theorm's

- If ownership is transferred from p to q, then, by the ASSIGN rule and ownership monotonicity, ownership also transfers between corresponding pointers on all access paths a and b: O(a)' = O(b) and O(b)' = 0.
- Under ownership monotonicity, if all allocated memory objects have a unique owner before a pointer assignment, then they will also have a unique owner after the assignment.

Maybe an example will help?

- Consider we have *p and *q of type *mut Node.
- Path, we are concerned with are : p, q, (*p).next and (*q).next

```
1 p = q;          // $\mathbb{O}_{p_1} = 0 \wedge \mathbb{O}_{p_2} + \mathbb{O}_{q_2} = \mathbb{O}_{q_1} \wedge$
2                 // $\mathbb{O}_{(*p_1).next} = 0 \wedge \mathbb{O}_{(*p_2).next} + \mathbb{O}_{(*q_2).next} = \mathbb{O}_{(*q_1).next}$
3 (*p).next = (*q).next;
4                 // $\mathbb{O}_{p_3} = \mathbb{O}_{p_2} \wedge \mathbb{O}_{q_3} = \mathbb{O}_{q_2} \wedge$
5                 // $\mathbb{O}_{(*p_2).next} = 0 \wedge \mathbb{O}_{(*p_3).next} + \mathbb{O}_{(*q_3).next} = \mathbb{O}_{(*q_2).next}$
```

Limitations of Ownership Monotonicity

- Mut Reference as as argument

  → List in our linkedlist example

  → base variable is mutable and non-owning but pointer reaching from it maybe be owning.

  → This can not be modeled by this stricter rule, this is done explicitly at translation stage instead.

Limitations of Ownership Monotonicity

- Local Borrows
  - local_borrow is non-owning.
  - *local_borrow is owning.

They ignore such cases.

```
let local_borrow = &mut n;

*local_borrow = Box::new(1);
```

C → Rust Translation

- Retyping Pointers
- *mut T corresponds to local_borrow case.

	Non-array pointers
Owning	`Option<Box<T>>`
Non-owning	`*mut T` or `Option<&mut T>`

C → Rust Translation

- Re-writing Pointer Uses
- BoxCtxt which requires Box pointers, MutCtxt which requires &mut references, ConstCtxt which requires & references, and RawCtxt which requires raw pointers.

	`Option<Box<T>>`	`Option<&mut T>`	`*mut T`
BoxCtxt	`p.take()`	⊥	`Some(Box::from_raw(p))`
MutCtxt	`p.as_deref_mut()`	`p.as_deref_mut()`	`p.as_mut()`
ConstCtxt	`p.as_deref()`	`p.as_deref()`	`p.as_ref()`
RawCtxt	`to_raw(&mut p)`	`to_raw(&mut p)`	`p`

- The columns correspond to the new type of the pointer to be rewritten, and rows represent possible contexts

C → Rust Translation

- Re-writing Pointer Uses
- BoxCtxt which requires Box pointers, MutCtxt which requires &mut references, ConstCtxt which requires & references, and RawCtxt which requires raw pointers.

	`Option<Box<T>>`	`Option<&mut T>`	`*mut T`
BoxCtxt	`p.take()`	⊥	`Some(Box::from_raw(p))`
MutCtxt	`p.as_deref_mut()`	`p.as_deref_mut()`	`p.as_mut()`
ConstCtxt	`p.as_deref()`	`p.as_deref()`	`p.as_ref()`
RawCtxt	`to_raw(&mut p)`	`to_raw(&mut p)`	`p`

- The columns correspond to the new type of the pointer to be rewritten, and rows represent possible contexts