Ownership Guided C to Rust Translation

Hanliang Zhang[1], Cristina David[1], Yijun Yu[2], and Meng Wang[1(✉)]

[1] University of Bristol, Bristol, UK
{pd21541,cristina.david,meng.wang}@bristol.ac.uk
[2] The Open University, Milton Keynes, UK
yijun.yu@open.ac.uk

Abstract. Dubbed a safer C, Rust is a modern programming language that combines memory safety and low-level control. This interesting combination has made Rust very popular among developers and there is a growing trend of migrating legacy codebases (very often in C) to Rust. In this paper, we present a C to Rust translation approach centred around static ownership analysis. We design a suite of analyses that infer ownership models of C pointers and automatically translate the pointers into safe Rust equivalents. The resulting tool, CROWN, scales to real-world codebases (half a million lines of code in less than 10 s) and achieves a high conversion rate.

1 Introduction

Rust [33] is a modern programming language which features an exciting combination of memory safety and low-level control. In particular, Rust takes inspiration from ownership types to restrict the mutation of shared state. The Rust compiler is able to statically verify the corresponding ownership constraints and consequently guarantee memory and thread safety. This distinctive advantage of provable safety makes Rust a very popular language, and the prospect of migrating legacy codebases in C to Rust is very appealing.

In response to this demand, automated tools translating C code to Rust emerge from both industry and academia [17,26,31]. Among them, the industrial strength translator C2Rust [26] rewrites C code into the Rust syntax while preserving the original semantics. The translation does not synthesise an ownership model and thus is not able to do more than replicating the unsafe use of pointers in C. Consequently, the Rust code must be labelled with the `unsafe` keyword which allows certain actions that are not checked by the compiler. More recent work focuses on reducing this unsafe labelling. In particular, the tool Laertes [17] aims to rewrite the (unsafe) code produced by C2Rust by searching the solution space guided by the type error messages from the Rust compiler. This is impressive, as for the first time proper Rust code beyond a line-by-line direct conversion from the original C source may be synthesised. On the other hand, the limit of the trial-and-error approach is also clear: the system does not support the reasoning of the generation process, nor create any new understanding of the target code (other than the fact that it compiles successfully).

In this paper, we take a more principled approach by developing a novel ownership analysis of pointers that is efficient (scaling to large programs (half a million LOC in less than 10 s)), sophisticated (handling nested pointers and inductively-defined data structures), and precise (being field and flow sensitive). Our ownership analysis makes a strengthening assumption about the Rust ownership model, which obviates the need for an aliasing analysis. While this assumption excludes a few safe Rust uses (see discussion in Sect. 5), it ensures that the ownership analysis is both scalable and precise, which is subsequently reflected in the overall scalability and precision of the C to Rust translation.

The primary goal of this analysis is of course to facilitate the C to Rust translation. Indeed, as we will see in the rest of the paper, an automated translation system is built to encode the ownership models in the generated Rust code which is then proven safe by the Rust compiler. However, in contrast to trying the Rust compiler as common in existing approaches [17,31], this analysis approach actually extracts new knowledge about ownership from code, which may lead to other future utilities including preventing memory leaks (currently allowed in safe Rust), identifying inherently unsafe code fragments, and so on. Our current contributions are:

– design a scalable and precise ownership analysis that is able to handle complex inductively-defined data structures and nested pointers. (Section 5)
– develop a refactoring technique for Rust leveraging ownership analyses to enhance code safety. While in this paper we focus on applying our technique to the translation from C to Rust, it can be used to improve the safety of any unsafe Rust code. (Section 6)
– implement a prototype tool (CROWN, standing for C to Rust OWNership guided translation) that translates C code into Rust with enhanced safety. (Section 7)
– evaluate CROWN with a benchmark suite including commonly used data structure libraries and real-world projects (ranging from 150 to half a million LOC) and compare the result with the state-of-the-art. (Section 8)

2 Background

We start by giving a brief introduction of Rust, in particular its ownership system and the use of pointers, as they are central to memory safety.

2.1 Rust Ownership Model

Ownership in Rust denotes a set of rules that govern how the Rust compiler manages memory [33]. The idea is to associate each value with a *unique* owner. This feature is useful for memory management. For example, when the owner goes out of scope, the memory allocated for the value can be automatically recycled.

```
1 let mut v = ...
2 let mut u = v;          // ownership is transferred to u
```

In the above snippet, the assignment of `v` to `u` also transfers ownership, after which it is illegal to access `v` until it is re-assigned a value again.

This permanent transfer of ownership gives strong guarantees but can be cumbersome to manage in programming. In order to allow sharing of values between different parts of the program, Rust uses the concept of *borrowing*, which refers to creating a *reference* (marked by an ampersand). A reference allows referring to some value without taking ownership of it. Borrowing gives the temporary right to read and, potentially, uniquely mutate the referenced value.

This concept of time creates another dimension of ownership management known as *lifetime*. For mutable references (as marked by `mut` in the above examples), only one reference is allowed at a time. But for immutable references (the ones without the `mut` marking), multiple of them can coexist as long as there isn't any mutable reference at the same time. As one can expect, this interaction of mutable and immutable references, and their lifetimes is highly non-trivial. In this paper, we focus on analysing mutable references.

2.2   Pointer Types in Rust

Rust has a richer pointer system than C. The primitive C-style pointers (written as `*const T` or `*mut T`) are known as *raw pointers*, which are ignored by the Rust compiler for ownership and lifetime checks. Raw pointers are a major source of unsafe Rust (more below). Idiomatic Rust instead advocates *box pointers* (written as `Box<T>`) as owning pointers that uniquely own heap allocations, as well as *references* (written as `&mut T` or `& T` as discussed in the previous subsection) as non-owning pointers that are used to access values owned by others. Rust also offers smart pointers for which the borrow rules are checked at runtime (e.g. `RefCell<T>`). We aim for our translation to maintain CPU time without additional runtime overhead, and therefore we do not refactor raw pointers into `RefCell<T>`s.

C-style array pointers are represented in Rust as references to arrays and slice references, with array bounds known at compile time and runtime, respectively. The creation of meta-data such as array bounds is beyond the scope of ownership analysis. In this work, we keep array pointers as raw pointers in the translated code.

2.3   Unsafe Rust

As a pragmatic design, Rust allows programs to contain features that cannot be verified by the compiler as memory safe. This includes dereferencing raw pointers, calling low level functions, and so on. Such uses must be marked with

the `unsafe` keyword and form fragments of *unsafe Rust*. It is worth noting that `unsafe` does not turn off all compiler checks; safe pointers are still checked.

Unsafe Rust is often used to implement data structures with complex sharing, overcome incompleteness issues of the Rust compiler, and support low-level systems programming [2,18]. But it can also be used for other reasons. For example, `c2rust` [26] directly translates C pointers into raw pointers. Without unsafe Rust, the generated code would not compile.

3 Overview

In this section, we present an overview of CROWN via two examples. The first example provides a detailed description of the `push` method for a singly-linked list, whereas the second shows a snippet from a real-world benchmark.

```c
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
```
(a) C code

```rust
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
```
(b) `c2rust` result

```rust
1  #[repr(C)]
2  pub struct Node {
3      pub data: i32,
4      pub next: Option<Box<Node>>,
5  }
6
7  #[repr(C)]
8  pub struct List {
9      pub head: Option<Box<Node>>,
10 }
11
12 pub unsafe extern "C" fn push(mut
          list: Option<&mut List>, mut
          new_data: i32) {
13     let mut new_node = Some(Box::new
          (<Node as Default>::default
          ()));
14     (*new_node.as_deref_mut().unwrap
          ()).data = new_data;
15     (*new_node.as_deref_mut().unwrap
          ()).next = (*list.
          as_deref_mut().unwrap()).
          head.take();
16     (*list.as_deref_mut().unwrap()).
          head = new_node;
17 }
18
```
(c) CROWN result

Fig. 1. Pushing into a singly-linked list

3.1 Pushing into a Singly-Linked List

The C code of function `push` in Fig. 1a allocates a new node where it stores the data received as argument. The new node subsequently becomes the head of `list`. This code is translated by `c2rust` to the Rust code in Fig. 1b. Notably, the `c2rust` translation is syntax-based and simply changes all the C pointers to `*mut` raw pointers. Given that dereferencing raw pointers is considered an unsafe operation in Rust (e.g. the dereferencing of `new_node` at line 16 in Fig. 1b), the `push` method must be annotated with the `unsafe` keyword (alternatively, it could be placed inside an `unsafe` block). Additionally, `c2rust` introduces two directives for the two struct definitions, `#[repr(C)]` and `#[derive(Copy, Clone)]`. The former keeps the data layout the same as in C for possible interoperation, and the latter instructs that the corresponding type can only be duplicated through copying.

While `c2rust` uses raw pointers in the translation, the ownership scheme in Fig. 1b obeys the Rust ownership model, meaning that the raw pointers could be translated to safe ones. A pointer to a newly allocated node is assigned to `new_node` at line 15. This allows us to infer that the ownership of the newly allocated node belongs to `new_node`. Then, at line 18, the ownership is transferred from `new_node` to `(*list).head`. Additionally, if `(*list).head` owns any memory object prior to line 17, then its ownership is transferred to `(*new_node).next` at line 17. This ownership scheme corresponds to safe pointer use: (i) each memory object is associated with a unique owner and (ii) it is dropped when its owner goes out of scope. As an illustration for (i), when the ownership of the newly allocated memory is transferred from `new_node` to `(*list).head` at line 18, `(*list).head` becomes the unique owner, whereas `new_node` is made invalid and it is no longer used. For (ii), given that argument `list` of `push` is an output parameter (i.e. a parameter that can be accessed from outside the function), we assume that it must be owning on exit from the method. Thus, no memory object is dropped in the `push` method, but rather returned to the caller.

CROWN infers the ownership information of the code translated by `c2rust`, and uses it to translate the code to safer Rust in Fig. 1c. As explained next, CROWN first retypes raw pointers into safe pointers based on the ownership information, and then rewrites their uses.

Retyping Pointers in Crown. If a pointer owns a memory object at *any point within its scope*, CROWN retypes it into a `Box` pointer. For instance, in Fig. 1c, local variable `new_node` is retyped to be `Option<Box<Node>>` (safe pointer types are wrapped into Option to account for null pointer values). Variable `new_node` is non-owning upon function entry, becomes owning at line 13 and ownership is transferred out again at line 16.

For struct fields, CROWN considers all the code in the scope of the struct declaration. If a struct field owns a memory object at *any point within the scope of its struct declaration*, then it is retyped to `Box`. In Fig. 1b, fields `next` and `head` are accessed via access paths `(*new_node).next` and `(*list).head`, and given ownership at lines 17 and 18, respectively. Consequently, they are retyped to `Box` at lines 4 and 9 in Fig. 1c, respectively.

A special case is that of output parameters, e.g. `list` in our example. For such parameters, although they may be owning, CROWN retypes them to `&mut` in order to enable borrowing. In `push`, the input argument `list` is retyped to `Option<&mut List>`.

Rewriting Pointer Uses in Crown. After retyping pointers, CROWN rewrites their uses. The rewrite process takes into consideration both their new type and the context in which they are being used. Due to the Rust semantics, the rewrite rules are slightly intricate (see Sect. 6). For instance, the dereference of `new_node` at line 14 is rewritten to `(*new_node).as_deref_mut().unwrap()` as it needs to be mutated and the optional part of the `Box` needs to be unwrapped. Similarly, at line 15, `(*list).head` is rewritten to be `((*list.as_deref_mut()).unwrap()).head.take()` as the LHS of the assignment expects a `Box` pointer.

After the rewrite performed by CROWN, the `unsafe` block annotation is not needed anymore. However, CROWN does not attempt to remove such annotations. Notably, safe pointers are always checked by the Rust compiler, even inside `unsafe` blocks.

3.2 Freeing an Argument List in bzip2

We next show the transformation of a real-world code snippet with a loop structure: a piece of code in `bzip2` that frees argument lists. `bzip2` defines a singly-linked list like structure, `Cell`, that holds a list of argument names. In Fig. 2, we extract from the source code a snippet that frees the argument lists. Here, the local variable `argList` is an already constructed argument list, and `Char` is a type alias to C-style characters. As a note, `Cell` in Figs. 2b and 2c does not refer to Rust's `std::cell::Cell`.

```
 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
```

(a) C definition

```
 1  #[derive(Copy, Clone)]
 2  #[repr(C)]
 3  pub struct zzzz {
 4      pub name: *mut Char,
 5      pub link: *mut zzzz,
 6  }
 7  pub type Cell = zzzz;
 8  [...]
 9  let mut aa: *mut Cell = argList;
10  while !aa.is_null() {
11      let mut aa2 = (*aa).link;
12      if !(*aa).name.is_null() {
13          free((*aa).name as *mut libc
             ::c_void);
14      }
15      free(aa as *mut libc::c_void);
16      aa = aa2;
17  }
18  [...]
19
```

(b) `c2rust` result

```
 1  #[repr(C)]
 2  pub struct zzzz {
 3      pub name: *mut /* owning */ Char
         ,
 4      pub link: Option<Box<zzzz>>,
 5  }
 6  pub type Cell = zzzz;
 7  [...]
 8  let mut aa: Option<Box<Cell>> =
         argList;
 9  while !aa.as_deref().is_none() {
10      let mut aa2 = (*aa.as_deref_mut
         ().unwrap()).link.take();
11      if !(*aa.as_deref().unwrap()).
         name.is_null {
12          free((*aa.as_deref().unwrap
             ()).name as *mut libc::
             c_void);
13      }
14      aa = aa2;
15  }
16  [...]
17
```

(c) CROWN result

Fig. 2. Freeing an argument list

CROWN accurately infers an ownership scheme for this snippet. Firstly, ownership of `argList` is transferred to `aa`, which is to be freed in the subsequent loop. Inside the loop, ownership of `link` accessed from `aa` is firstly transferred to `aa2`, then ownership of `name` accessed from `aa` is released in a call to `free`. After the conditional, ownership of `aa` is also released. Last of all, `aa` regains ownership from `aa2`.

Handling of Loops. For loops, CROWN only analyses their body once as that will already expose all the ownership information. For inductively defined data structures such as `Cell`, while further unrolling of loop bodies explores the data structures deeper, it does not expose any new struct fields: pointer variables and pointer struct fields do not change ownership between loop iterations. Additionally, CROWN emits constraints that equate the ownership of all local pointers at the loop entry and exit. For example, the ownership statuses of `aa` and `aa2` at loop entry are made equal with those at loop exit, and inferred to be owning and non-owning, respectively.

Handling of Null Pointers. It is a common C idiom for pointers to be checked against null after malloc or before free: `if !p.is_null()  free(p)` `;`. This could be problematic since the then-branch and the else-branch would have conflicting ownership statuses for `p`. We adopt a similar solution as [24]: we insert an explicit null assignment in the null branch `if !p.is_null()  free` `(p); else  p = ptr::null_mut();`. As we treat null pointers as both owning and non-owning, the ownership of `p` will be dictated by the non-null branch, enabling CROWN to infer the correct ownership scheme.

Translation. With the above ownership scheme, CROWN performs the rewrites as in Fig. 2c. Note that we do not attempt to rewrite `name` since it is an array pointer (see Sect. 7 for limitations).

4   Architecture

In this section, we give a brief overview of CROWN's architecture. CROWN takes as input a Rust program with unsafe blocks, and outputs a safer Rust program, where a portion of the raw pointers have been retyped as safe ones (in accordance to the Rust ownership model), and their uses modified accordingly. In this paper we focus on applying our technique to programs automatically translated by `c2rust`, which maintain a high degree of similarity to the original C ones, where the C syntax is replaced by Rust syntax.

CROWN applies several static analyses on the MIR of Rust to infer properties of pointers:

– **Ownership analysis:** computes ownership information about the pointers in the code, i.e. for each pointer it infers whether it is owning/non-owning at particular program locations.
– **Mutability analysis:** infers which pointers are used to modify the object they point to (inspired by [22,25]).
– **Fatness analysis:** distinguishes array pointers from non-array pointers (inspired by [32]).

The results of these analyses are summarised as type qualifiers [21]. A type qualifier is an atomic property (i.e., ownership, mutability, and fatness) that 'qualifies' the standard pointer type. These qualifiers are then utilised for pointer retyping. For example, an owning, non-array pointer is retyped to `Box`. After pointers have been retyped, CROWN rewrites their usages accordingly.

5   Ownership Analysis

The goal of our ownership analysis is to compute an ownership scheme for a given program that obeys the Rust ownership model, if such a scheme exists. The ownership scheme contains information about whether pointers in the program are owning or non-owning at particular program locations. At a high-level, our analysis works by generating a set of ownership constraints (Sect. 5.2), which are

then solved by a SAT solver (Sect. 5.3). A satisfying assignment for the ownership constraints is an ownership scheme that obeys the Rust semantics.

Our ownership analysis is flow and field sensitive, where the latter enables inferring ownership information for pointer struct fields. To satisfy field sensitivity, we track ownership information for *access paths* [10,14,29]. An access path represents a memory location by the way it is accessed from an initial, base variable, and comprises of the base variable and a sequence of field selection operators. For the program Fig. 1b, some example access paths are `new_node` (consists only of the base variable), `(*new_node).next`, and `(*list).head`. Our analysis associates an ownership variable with each access path, e.g. `p` has associated ownership variable $\mathbb{O}_p$, and `(*p).next` has associated ownership variable $\mathbb{O}_{(*p).next}$. Each ownership variable can take the value 1 if the corresponding access path is owning, or 0 if it is non-owning. By ownership of an access path we mean the ownership of the field (or, more generally, pointer) accessed last through the access path, e.g. the ownership of `(*new_node).next` refers to the ownership of field `next`.

5.1 Ownership and Aliasing

One of the main challenges of designing an ownership analysis is the interaction between ownership and aliasing. To understand the problem, let us consider the pointer assignment at line 3 in the code listing below. We assume that the lines before the assignment allow inferring that `q`, `(*q).next` and `r` are owning, whereas `p` and `(*p).next` are non-owning. Additionally, we assume that the lines after the assignment require `(*p).next` to be owning (e.g. `(*p).next` is being explicitly freed). From this, an ownership analysis could reasonably conclude that ownership transfer happens at line 3 (such that `(*p).next` becomes owning), and the inferred ownership scheme obeys the Rust semantics.

```
1    let p, r, q : *mut Node;
2    // p and (*p).next non-owning; q, (*q).next and r owning
3    (*p).next = r;
4    // (*p).next must have ownership
```

Let's now also consider aliasing. A possible assumption is that, just before line 3, `p` and `q` alias, meaning that `(*p).next` and `(*q).next` also alias. Then, after line 3, `(*p).next` and `(*q).next` will still alias (pointing to the same memory object). However, according to the ownership scheme above, both `(*p).next` and `(*q).next` are owning, which is not allowed in Rust, where a memory object must have a unique owner. This discrepancy was not detected by the ownership analysis mimicked above. The issue is that the ownership analysis ignored aliasing. Indeed, ownership should not be transferred to `(*p).next` if there exists an owning alias that, after the ownership transfer, continues to point to the same memory object as `(*p).next`.

Precise aliasing information is very difficult to compute, especially in the presence of inductively defined data structures. In the current paper, we alleviate the need to check aliasing by making a strengthening assumption about

the Rust ownership model: we restrict the way in which pointers can acquire ownership along an access path, thus limiting the interaction between ownership and aliasing. In particular, we introduce a novel concept of *ownership monotonicity*. This property states that, along an access path, the ownership values of pointers can only decrease (see Definition 1, where *is_prefix*(a, b) returns true if access path a is a prefix of b, and false otherwise – e.g. *is_prefix*(p, (*p).next) $= true$). Going back to the previous code listing, the ownership monotonicity implies that, for access path (*p).next we have $\mathbb{O}_{\texttt{p}} \geq \mathbb{O}_{\texttt{(*p).next}}$, and for access path (*q).next we have $\mathbb{O}_{\texttt{q}} \geq \mathbb{O}_{\texttt{(*q).next}}$. 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.

Definition 1 (Ownership monotonicity). *Given two access paths a and b, if is_prefix(a, b), then $\mathbb{O}_a \geq \mathbb{O}_b$.*

Ownership monotonicity is stricter than the Rust semantics, causing our analysis to reject two scenarios that would otherwise be accepted by the Rust compiler (see discussion in Sect. 5.4). In this work, we made the design decision to use ownership monotonicity over aliasing analysis as it allows us to retain more control over the accuracy of the translation. Conversely, using an aliasing analysis would mean that the accuracy of the translation is directly dictated by the accuracy of the aliasing analysis (i.e. false alarms from the aliasing analysis [23, 40] would result in CROWN not translating pointers that are actually safe). With ownership monotonicity, we know exactly what the rejected valid ownership schemes are, and we can explicitly enable them (again, see discussion in Sect. 5.4).

5.2   Generation of Ownership Constraints

During constraint generation, we assume a given k denoting the length of the longest access path used in the code. This enables us to capture the ownership of all the access paths exposed in the code. Later in this section, we will discuss the handling of loops, which may expose longer access paths.

Next, we denote by $\mathcal{P}$ the set of all access paths in a program, *base_var*(a) returns the base variable of access path a, and $|a|$ computes the length of the access path a in terms of applied field selection operators from the base variable. In the context of the previous code listing, *base_var*((*p).next) = p, *base_var*(p) = p, $|\texttt{p}| = 1$ and $|\texttt{(*p).next}| = 2$. Then, we define $ap(v, lb, ub)$ to return the set of access paths with base variable v and length in between lower bound lb and upper bound ub: $ap(v, lb, ub) = \{a \in \mathcal{P} | base\_var(a) = v \wedge lb \leq |a| \leq ub\}$. For illustration, we have $ap(\texttt{p}, 1, 2) = \{\texttt{p}, \texttt{(*p).next}\}$.

Ownership Transfer. The program instructions where ownership transfer can happen are (pointer) assignment and function call. Here we discuss assignment and, due to space constraints, we leave the rules for interprocedural ownership analysis in the extended version [41]. Our rule for ownership transfer at assignment site follows Rust's `Box` semantics: when a `Box` pointer is moved, the

ASSIGN
$$v = base\_var(\mathtt{p}), \quad w = base\_var(\mathtt{q}),$$
$$a \in ap(v, |\mathtt{p}|, k), \quad b \in ap(w, |\mathtt{q}|, k), \quad c \in ap(v, 1, |\mathtt{p}|-1), \quad d \in ap(w, 1, |\mathtt{q}|-1)$$
$$|a| - |\mathtt{p}| = |b| - |\mathtt{q}|, |c| = |d|$$
$$is\_prefix(p, a), \; is\_prefix(q, b), \; is\_prefix(c, p), \; is\_prefix(d, q)$$
$$C' = C \cup \{\mathbb{O}_a = 0 \wedge \mathbb{O}_{a'} + \mathbb{O}_{b'} = \mathbb{O}_b \; \wedge \mathbb{O}_{c'} = \mathbb{O}_c \wedge \mathbb{O}_{d'} = \mathbb{O}_d\}$$
$$\overline{C \vdash \mathtt{p = q;} \Rightarrow C'}$$

Fig. 3. Ownership constraint generation for assignment

object it points to is moved as well. For instance, in the following Rust pseudocode snippet:

```
1 let p,q: Box<Box<i32>>;
2 p = q; // ownership transfer occurs
3 // the use of q and *q is disallowed
```

when ownership is transferred from `q` to `p`, `*q` also loses ownership. Except for reassignment, the use of a `Box` pointer after it lost its ownership is disallowed, hence the use of `q` or `*q` is forbidden at line 3.

Consequently, we enforce the following *ownership transfer rule*: if ownership transfer happens for a pointer variable (e.g. `p` and `q` in the example), then it must happen for all pointers reachable from that pointer (e.g. `*p` and `*q`). The ownership of pointer variables from which the pointer under discussion is reachable remains the same (e.g. if ownership transfer happens for some assignment `*p = *q` in the code, then `q` and `p` retain their respective previous ownership values).

Possible Ownership Transfer at Pointer Assignment: The ownership transfer rule at pointer assignment site is captured by rule ASSIGN in Fig. 3. The judgement $C \vdash \mathtt{p = q;} \Rightarrow C'$ denotes the fact that the assignment is analysed under the set of constraints C, and generates C'. We use prime notation to denote variables after the assignment. Given pointer assignment `p = q`, a and b represent all the access paths respectively starting from `p` and `q`, whereas c and d denote the access paths from the base variables of `p` and `q` that reach `p` and `q`, respectively. Then, equality $\mathbb{O}_{a'} + \mathbb{O}_{b'} = \mathbb{O}_b$ captures the possibility of ownership transfer for all access paths originating at `p` and `q`: (i) If transfer happens then the ownership of b transfers to a' ($\mathbb{O}_{a'} = \mathbb{O}_b$ and $\mathbb{O}_{b'} = 0$). (ii) Otherwise, the ownership values are left unchanged ($\mathbb{O}_{a'} = \mathbb{O}_a$ and $\mathbb{O}_{b'} = \mathbb{O}_b$). The last two equalities, $\mathbb{O}_{c'} = \mathbb{O}_c \wedge \mathbb{O}_{d'} = \mathbb{O}_d$, denote the fact that, for both (i) and (ii), pointers on access paths c and d retain their previous ownership. Note that "+" is interpreted as the usual arithmetic operation over $\mathbb{N}$, where we impose an implicit constraint $0 \leq \mathbb{O} \leq 1$ for every ownership variable $\mathbb{O}$.

C Memory Leaks: In the ASSIGN rule, we add constraint $\mathbb{O}_a = 0$ to C' in order to force a to be non-owning before the assignment. Conversely, having a owning before being reassigned via the assignment under analysis signals a memory

leak in the original C program. Given that in Rust memory is automatically returned, allowing the translation to happen would change the semantics of the original program by fixing the memory leak. Instead, our design choice is to disallow the ownership analysis from generating such a solution. As we will explain in Sect. 8, we intend for our translation to preserve memory usage (including possible memory leaks).

Simultaneous Ownership Transfer Along an Access Path: One may observe that the constraints generated by ASSIGN do not fully capture the stated ownership transfer rule. In particular, they do not ensure that, whenever ownership transfer occurs from p to q, it also transfers for all pointers on all access paths a and b. Instead, this is implicitly guaranteed by the ownership monotonicity rule, as stated in Theorem 1.

Theorem 1 (Ownership transfer). *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: $\mathbb{O}_{a'} = \mathbb{O}_b$ and $\mathbb{O}_{b'} = 0$. (proof in the extended version [41])*

Ownership and Aliasing: We saw in Sect. 5.1 that aliasing may cause situations in which, after ownership transfer, the same memory object has more than one owner. Theorem 2 states that this is not possible under ownership monotonicity.

Theorem 2 (Soundness of pointer assignment under ownership monotonicity). *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. (proof in the extended version [41])*

Intuitively, Theorem 2 enables a pointer to acquire ownership without having to consider aliases: after ownership transfer, this pointer will be the unique owner. The idea resembles that of strong updates [30].

Additional Access Paths: As a remark, it is possible for p and q to be accessible from other base variables in the program. In such cases, given that those access paths are not explicitly mentioned at the location of the ownership transfer, we do not generate new ownership variables for them. Consequently, their current ownership variables are left unchanged by default.

Ownership Transfer Example. To illustrate the ASSIGN rule, we use the singly-linked list example below, where we assume that p, q are both of type `*mut Node`. Therefore, we will have to consider the following four access path `p, q, (*p).next, (*q).next`. In SSA-style, at each line in the example, we generate new ownership variables (by incrementing their subscript) for the access paths mentioned at that line. For the first assignment, ownership transfer can happen between p and q, and `(*p).next` and `(*q).next`, respectively. For the second assignment, ownership can be transferred between `(*p).next` and `(*q).next`, while p and q must retain their previous ownership.

```
1  p = q;      //  𝕆_{p_1} = 0 ∧ 𝕆_{p_2} + 𝕆_{q_2} = 𝕆_{q_1}  ∧
2              //  𝕆_{(*p_1).next} = 0 ∧ 𝕆_{(*p_2).next} + 𝕆_{(*q_2).next} = 𝕆_{(*q_1).next}
3  (*p).next = (*q).next;
4              //  𝕆_{p_3} = 𝕆_{p_2} ∧ 𝕆_{q_3} = 𝕆_{q_2}  ∧
5              //  𝕆_{(*p_2).next} = 0 ∧ 𝕆_{(*p_3).next} + 𝕆_{(*q_3).next} = 𝕆_{(*q_2).next}
```

Besides generating ownership constraints for assignments, we must model the ownership information for commonly used C standard function like `malloc`, `calloc`, `realloc`, `free`, `strcmp`, `memset`, etc. Due to space constraints, more details about these, as well as the rules for ownership monotonicity and interprocedural ownership analysis are provided in the extended version [41].

Handling Conditionals and Loops. As mentioned in Sect. 3.2, we only analyse the body of loops once as it is sufficient to expose all the required ownership variables. For inductively defined data structures, while further unrolling of loop bodies increases the length of access paths, it does not expose any new struct fields (struct fields do not change ownership between loop iterations).

To handle join points of control paths, we apply a variant of the SSA construction algorithm [6], where different paths are merged via ϕ nodes. The value of each ownership variable must be the same on all joined paths, or otherwise the analysis fails.

5.3   Solving Ownership Constraints

The ownership constraint system consists of a set of 3-variable linear constraints of the form $O_v = O_w + O_u$, and 1-variable equality constraints $O_v = 0$ and $O_v = 1$.

Definition 2 (Ownership constraint system). *An* ownership constraint system $(P, \Delta, \Sigma, \Sigma_\neg)$ *consists of a set of ownership variables P that can have either value 0 or 1, a set of 3-variable equality constraints $\Delta \subseteq P \times P \times P$, and two sets of 1-variable equality constraints, $\Sigma, \Sigma_\neg \subseteq P$. The equalities in Σ are of the form $x = 1$, whereas the equalities in $\Sigma_\neg$ are of the form $x = 0$.*

Theorem 3 (Complexity of the ownership constraint solving). *Deciding the satisfiability of the ownership constraint system in Definition 2 is NP-complete. (proof in the extended version [41]).*

We solve the ownership constraints by calling a SAT solver. The ownership constraints may have no solution. This happens when there is no ownership scheme that obeys the Rust ownership model and the ownership monotonicity property (which is stricter than the Rust model for some cases), or the original C program has a memory leak. In the case where the ownership constraints have more than one solution, we consider the first assignment returned by the SAT solver.

Due to the complex Rust semantics, we do not formally prove that a satisfying assignment obeys the Rust ownership model. Instead, this check is performed after the translation by running the Rust compiler.

5.4 Discussion on Ownership Monotonicity

As mentioned earlier in Sect. 5, ownership monotonicity is stricter than the Rust semantics, causing our analysis to potentially reject some ownership schemes that would otherwise be accepted by the Rust compiler. We identified two such scenarios:

(i) Reference output parameter: This denotes a reference passed as a function parameter, which acts as an output as it can be accessed from outside the function (e.g. `list` in Fig. 1a). For such parameters, the base variable is non-owning (as it is a reference) and mutable, whereas the pointers reachable from it may be owning (see example in Fig. 1c, where `(*node).head` gets assigned a pointer to a newly allocated node). We detect such situations and explicitly enable them. In particular, we explicitly convert owning pointers `p` to `&mut(*p)` at the translation stage.

(ii) Local borrows: The code below involving a mutable local borrow is not considered valid by CROWN as it disobeys the ownership monotonicity: after the assignment, `local_borrow` is non-owning, whereas `*local_borrow` is owning.

```
1   let local_borrow = &mut n;
2   *local_borrow = Box::new(1);
```

While we could explicitly handle the translation to local borrows, in order to do so soundly, we would have to reason about lifetime information (e.g. CROWN would have to check that there is no overlap between the lifetimes of different mutable references to the same object). In this work, we chose not to do this and instead leave it as future work (as also mentioned under limitations in Sect. 7). It was observed in [13] that scenario (i) is much more prevalent than scenario (ii). Additionally, we observed in our benchmarks that output parameter accounts for 93% of mutable references (hence the inclusion of a special case enabling the translation of scenario (i) in CROWN).

6 C to Rust Translation

CROWN uses the results of the ownership, mutability and fatness analyses to perform the actual translation, which consists of retyping pointers (Sect. 6.1) and rewriting pointer uses (Sect. 6.2).

6.1 Retyping Pointers

As mentioned in Sect. 2.2, we do not attempt to translate array pointers to safe pointers. In the rest of the section, we focus on mutable, non-array pointers.

The translation requires a global view of pointers' ownership, whereas information inferred by the ownership analysis refers to individual program locations. For the purpose of translation, given that we refactor owning pointers into box pointers, a pointer is considered (globally) owning if it owns a memory object at any program location within its scope. Otherwise, it is (globally) non-owning.

When retyping pointer fields of structs, we must consider the scope of the struct declaration, which generally transcends the whole program. Within this scope, each field is usually accessed from several base variables, which must all be taken into consideration. For instance, given the `List` declaration in Fig. 1b and two variables `l1` and `l2` of type `*mut List`. Then, in order to determine the ownership status of field `next`, we have to consider all the access paths to `next` originating from both base variables `l1` and `l2`.

The next table shows the retyping rules for mutable, non-array pointers, where we wrap safe pointer types into `Option` to account for null pointer values:

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

The non-owning pointers that are kept as raw pointers `*mut T` correspond to mutable local borrows. As explained in Sects. 5.4 and 7, CROWN doesn't currently handle the translation to mutable local borrows due to the fact that we do not have a lifetime analysis. Notably, this restriction does not apply to output parameters (which covers the majority of mutable references), where we translate to mutable references. The lack of a lifetime analysis means that we also can't handle immutable local borrows, hence our translation's focus on mutable pointers.

6.2   Rewriting Pointer Uses

The rewrite of a pointer expression depends on its new type and the context in which it is used. For example, when rewriting `q` in `p = q`, the context will depend on the new type of `p`. Based on this new type, we can have four contexts: BoxCtxt which requires `Box` pointers, MutCtxt which requires `&mut` references, ConstCtxt which requires `&` references, and RawCtxt which requires raw pointers. For example, if `p` above is a `Box` pointer, then we rewrite `q` in a BoxCtxt.

Then, the rewrite takes place according to the following table, where columns correspond to the new type of the pointer to be rewritten, and rows represent possible contexts[1].

	`Option<Box<T>>`	`Option<&mut T>`	`*mut T`
BoxCtxt	`p.take()`	$\perp$	`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`

Our translation uses functions from the Rust standard library, as follows:

1. When `Option<Box<T>>` is passed to a BoxCtxt, we expect a move, and consequently we use `take` to replace the value inside the option with None;
2. We use `as_deref` and `as_deref_mut` in order to not consume the original option, and we create new options with references to the original ones;

---

[1] The cell marked as $\perp$ is not applicable due to our treatment of output parameter.

3. `as_mut` and `as_ref` converts raw pointers to references;
4. `Box::from_raw` converts raw pointers into `Box` pointers.

We also define the helper function `to_raw` that transform safe pointers into raw pointers:

```
fn to_raw<T>(b: &mut Option<Box<T>>) -> *mut T {
    b.as_deref_mut().map(|b| b as *mut T).unwrap_or(null_mut())
}
```

Here, we explain `to_raw` for a `Box` argument (the explanation for `&mut` is the same because of the polymorphic nature of `as_deref_mut`):

1. To convert `Option<Box<T>>`, we first mutably borrow the entire option as denoted by the mutable borrow argument of the helper function. This is needed because `Option` is not copyable, and it would be otherwise consumed;
2. `as_deref_mut` converts `&mut Option<Box<T>>` to `Option<&mut T>`;
3. `map` converts the optional part of the reference into an option of raw pointers;
4. Finally, `unwrap_or` returns the `Some` value of the option, or a null pointer `std::ptr::null_mut()` if the value is `None`.

Dereferences: When a pointer `p` is dereferenced as part of a larger expression (e.g. `(*p).next`), we need an additional `unwrap()`.

   Box pointers check: Rust disallows the use of `Box` pointers after they lost their ownership. As this rule cannot be captured by the ownership analysis, such situations are detected at translation stage, and the culpable `Box` pointers are reverted back to raw pointers.

   For brevity, we omitted the slightly different treatment of struct fields that are not of pointer type.

7   Challenges of Handling Real-World Code

We designed CROWN to be able to analyse and translate real-world code, which poses significant challenges. In this section, we discuss some of the engineering challenges of CROWN and its current limitations.

7.1   Preprocessing

During the transpilation of C libraries, `c2rust` treats each file as a separate compilation unit, which gets translated into a separate Rust module. Consequently, struct definitions are duplicated, and available function definitions are put in `extern` blocks [17]. We apply a preprocessing step similar to the resolve-imports tool of Laertes [17] that links those definitions across files.

7.2   Limitations of the Ownership Analysis

There are a few C constructs and idioms that are not fully supported by our implementation, for which CROWN generates partial ownership constraints. CROWN's translation will attempt to rewrite a variable as long as there exists a constraint involving it. As a result, the translation is in theory neither *sound* nor *complete*: it may generate code that does not compile (though we have not observed this in practice for the benchmarks where CROWN produces a result – see Sect. 8) and it may leave some pointers as raw pointers resulting in a less than optimal translation. We list below the cases when such a scenario may happen.

Certain Unsafe C Constructs. For type casts, we only generate ownership transfer constraints for head pointers; for unions we assume that they contain no pointer fields and consequently, we generate no constraints; similarly, we generate no constraints for variadic arguments. We noticed that unions and variadic arguments may cause our tool to crash (e.g. three of the benchmarks in [17], as mentioned in Sect. 8). Those crashes happen when analysing access paths that contain dereferences of union fields (where we assumed no pointer fields), and when analysing calls to functions with variadic arguments where a pointer is passed as argument.

Function Pointers. CROWN does not generate any constraints for them.

Non-standard Memory Management in C Libraries. Certain C libraries wrap `malloc` and `free`, often with static function pointers (pointers to allocator/deallocator are stored in static variables), or function pointers in structs. CROWN does not generate any constraints in such scenarios. In C, it is also possible to use `malloc` to allocate a large piece of memory, and then split it into several sub-regions assigned to different pointers. In our ownership analysis, only one pointer can gain ownership of the memory allocated by a call to `malloc`. Another C idiom that we don't fully support occurs when certain pointers can point to either heap allocated objects, or statically allocated stack arrays. CROWN generates ownership constraints only for the heap and, consequently, those variables will be left under-constrained.

7.3   Other Limitations of *CROWN*

Array Pointers. For array pointers, although CROWN infers the correct ownership information, it does not generate the meta data required to synthesise Rust code.

Mutable Local Borrows. As explained in the last paragraph of Sect. 6.1, CROWN does not translate mutable non-owning pointers to local mutable references as this requires dedicated analysis of lifetimes. Note that CROWN does however generate mutable references for output parameters.

Access Paths that Break Ownership Monotonicity. As discussed in Sect. 5.4, ownership monotonicity may be stricter in certain cases than Rust's semantics.

8   Experimental Evaluation

We implement CROWN on top of the Rust compiler, version `nightly-2023-01-26`. We use `c2rust` with version 0.16.0. For the SAT solver, we rely on a Rust-binding of `z3` [20] with version 0.11.2. We run all our experiments on a MacBook Pro with an Apple M1 chip, with 8 cores (4 performance and 4 efficiency). The computer has 16 GB RAM and runs macOS Monterey 12.5.1.

Benchmark Selection. To evaluate the utility of CROWN, we collected a benchmark suite of 20 programs (Table 1). These include benchmarks from Laertes [17]'s accompanying artifact [16] (marked by * in Table 1)[2], and additionally 8 real-world projects (`binn`, `brotli`, `buffer`, `heman`, `json.h`, `libtree`, `lodepng`, `rgba`) together with 4 commonly used data structure libraries (`avl`, `bst`, `ht`, `quadtree`).

Functional and Non-functional Guarantees. With respect to functional properties, we want the original program and the refactored program to be observationally equivalent, i.e. for each input they produce the same output. We empirically validated this using all the available test suites (i.e. for `libtree`, `rgba`, `quadtree`, `urlparser`, `genann`, `buffer` in Table 1). All the test suites continue to pass after the translation. For nonfunctional properties, we intend to preserve memory usage and CPU time, i.e. we don't want our translation to introduce runtime overhead. We also validated this using the test suites.

Table 1. Benchmarks information

Benchmark	Files	Structs	Functions	LOC	Benchmark	Files	Structs	Functions	LOC
Avl	1	2	11	229	libcsv*	1	6	23	976
binn	1	5	165	4426	libtree	1	18	32	2610
brotli	30	237	867	537723	libzahl*	49	65	108	4655
bst	1	1	6	154	lil*	2	9	136	5670
buffer	2	3	42	1207	lodepng	1	19	236	14153
bzip2*	9	39	126	14829	quadtree	5	14	31	1216
genann*	6	10	27	2410	rgba	2	3	19	1855
heman	24	52	302	13762	robotfindskitten*	1	8	18	1508
ht	1	3	10	264	tulipindicators*	111	18	229	22363
json.h	1	13	53	3860	urlparser*	1	1	21	1379

8.1   Research Questions

We aim at answering the following research questions.

---

[2] We excluded `json-c`, `optipng`, `tinycc` where CROWN crashes because of the uses of unions and variadic arguments as discussed in Sect. 7. Additional programs (`qsort`, `grabc`, `xzoom`, `snudown`, `tmux`, `libxml2`) are mentioned in the paper [17] but are either missing or incomplete in the artifact [16].

RQ1. How many raw pointers/pointer uses can CROWN translate to safe pointers/pointer uses?

RQ2. How does CROWN's result compare with the state-of-the-art [17]?

RQ3. What is the runtime performance of CROWN?

RQ 1: Unsafe pointer reduction. In order to judge CROWN's efficacy, we measure the reduction rate of raw pointer declarations and uses. This is a direct indicative of the improvement in safety, as safe pointers are always checked by the Rust compiler (even inside `unsafe` regions). As previously mentioned, we focus on mutable non-array pointers. The results are presented in Table 2, where #ptrs counts the number of raw pointer declarations in a given benchmark, #uses counts the number of times raw pointers are being used, and the Laertes and Crown headers denote the reduction rates of the number of raw pointers and raw pointer uses achieved by the two tools, respectively. For instance, for benchmark `avl`, the rate of 100% means that all raw pointer declarations and their uses are translated into safe ones. Note that the "-" symbols on the row corresponding to `robotfindskitten` are due to the fact that the benchmark contains 0 raw pointer uses.

The median reduction rates achieved by CROWN for raw pointers and raw pointer uses are 37.3% and 62.1%, respectively. CROWN achieves a 100% reduction rate for many non-trivial data structures (`avl`, `bst`, `buffer`, `ht`), as well as for `rgba`. For `brotli`, a lossless data compression algorithm developed by Google, which is our largest benchmark, CROWN achieves reduction rates of 21.4% and 20.9%, respectively. The relatively low reduction rates for `brotli` and a few other benchmarks (`tulipindicators`, `lodepng`, `bzip2`, `genann`, `libzahl`) is due to their use of non-standard memory management strategies (discussed in detail in Sect. 7).

Notably, all the translated benchmarks compile under the aforementioned Rust compiler version. As a check of semantics preservation, for the benchmarks that provide test suites (`libtree`, `rgba`, `quadtree`, `urlparser`, `genann`, `buffer`), our translated benchmarks pass all the provided tests.

RQ 2: Comparing with state-of-the-art. The comparison of CROWN with Laertes [17] is also shown in Table 2, with bold font highlighting better results. The data on Laertes is either directly extracted from the artifact [16] or has been confirmed by the authors through private correspondence. We can see that CROWN outperforms the state-of-the-art (often by a significant degree) in most cases, with `lodepng` being the only exception, where we suspect that the reason also lies with non-standard memory management strategies mentioned before. Laertes is less affected by this as it does not rely on ownership analysis.

RQ 3: Runtime performance. Although our analysis relies on solving a constraint satisfaction problem that is proven to be NP-complete, in practice the runtime performance of CROWN is consistently high. The execution time of the analysis and the rewrite for the whole benchmark suite is within 60 s (where the execution time for our largest benchmark, `brotli`, is under 10 s).

Table 2. Reduction of (mutable, non-array) raw pointer declarations and uses

Benchmark	#ptrs	Laertes	Crown	#uses	Laertes	Crown
avl	8	0.0%	**100.0%**	41	0.0%	**100.0%**
binn	103	46.6%	**65.0%**	247	62.3%	**71.3%**
brotli	846	0.0%	**21.4%**	3686	0.0%	**20.9%**
bst	5	0.0%	**100.0%**	22	0.0%	**100.0%**
buffer	38	0.0%	**100.0%**	56	0.0%	**100.0%**
bzip2*	126	14.3%	**26.2%**	2946	2.2%	**3.7%**
genann*	28	0.0%	**7.1%**	160	0.0%	**15.0%**
heman	360	30.3%	**35.0%**	926	50.2%	**60.2%**
ht	6	33.3%	**100.0%**	28	42.9%	**100.0%**
json.h	128	2.3%	**23.4%**	647	1.2%	**62.1%**
libcsv*	20	65.0%	**70.0%**	141	97.9%	97.9%
libtree	48	29.2%	**39.6%**	227	33.0%	**62.1%**
libzahl*	87	2.2%	**16.1%**	279	4.1%	**16.8%**
lil*	202	9.2%	**18.8%**	1018	51.4%	**69.4%**
lodepng	227	**46.3%**	44.9%	1232	40.4%	37.7%
quadtree	33	0.0%	**42.4%**	117	0.0%	**48.7%**
rgba	6	83.3%	83.3%	12	100.0%	100.0%
robotfindskitten*	1	0.0%	0.0%	0	–	–
tulipindicators*	134	0.0%	**0.7%**	625	0.0%	0.0%
urlparser*	9	0.0%	**11.1%**	40	0.0%	**45.0%**

9   Related Works

Ownership Discussion. Ownership has been used in OO programming to enable controlled aliasing by restricting object graphs underlying the runtime heap [11,12] with efforts made in the automatic inference of ownership information [1,4,39], and applications of ownership to memory management [5,42]. Similarly, the concept of ownership has also been applied to analyse C/C++ programs. Heine et al. [24] inferred pointer ownership information for detecting memory leaks. Ravitch et al. [37] apply static analysis to infer ownership for automatic library binding generation. Giving the different application domains, each of these works makes different assumptions. Heine et al. [24] assumes that indirectly-accessed pointers (i.e. any pointer accessed through a path, like (*p).next) cannot acquire ownership, whereas Ravitch et al. [37] assumes that all struct fields are owning unless explicitly annotated. We took from [24] its handling of flow sensitivity, but enhanced it with the analysis of nested pointers and inductively defined data structures, which we found to be essential for translating real-world code. The analysis in [24] assigns a default "non-owning" status to all indirectly accessed pointers. This rules out many interesting data structures such as linked lists, trees, hash tables, etc., and commonly used idioms such as passing by reference. Conversely, in our work, we rely on a strengthening assumption about the Rust ownership model, which allows handling the aforementioned scenarios and data structures. Lastly, the idea of ownership is also

broadly applied in concurrent separation logic [7–9,19,38]. However, these works are not aimed as automatic ownership inference systems.

Rust Verification. The separation logic based reasoning framework Iris [28] was used to formalise the Rust type system [27], and verify Rust programs [34]. While these works cover unsafe Rust fragments, they are not fully automatic. When restricting reasoning to only safe Rust, RustHorn [35] gives a first-order logic formulation of the behavior of Rust code, which is ameanable to fully automatic verification, while Prusti [3] leverages Rust compiler information to generate separation logic verification conditions that are discharged by Viper [36]. In the current work, we provide an automatic ownership analysis for unsafe Rust programs.

Type Qualifiers. Type qualifiers are a lightweight, practical mechanism for specifying and checking properties not captured by traditional type systems. A general flow-insensitive type qualifier framework has been proposed [21], with subsequent applications analysing `Java` reference mutability [22,25] and `C` array bounds [32]. We adapted these works to Rust for our mutability and fatness analyses, respectively.

C to Rust Translation. We have already discussed `c2rust` [26], which is an industrial strength tool that converts C to Rust syntax. `c2rust` does not attempt to fix unsafe features such as raw pointers and the programs it generates are always annotated as unsafe. Nevertheless it forms the bases of other translation efforts. CRustS [31] applies AST-based code transformations to remove superfluous unsafe labelling generated by `c2rust`. But it does not fix the unsafe features either. Laertes [17] is the first tool that is actually able to automatically reduce the presence of unsafe code. It uses the Rust compiler as a blackbox oracle and search for code changes that remove raw pointers, which is different from CROWN's approach (see Sect. 8 for an experimental comparison). The subsequent work [15] develops an evaluation methodology for studying the limitations of existing techniques that translate unsafe raw pointers to safe Rust references. The work adopts a new concept of 'pseudo safety', under which semantics preservation of the original programs is no longer guaranteed. As explained in Sect. 8, in our work, we aim to maintain semantic equivalence.

10   Conclusion

We devised an ownership analysis for Rust programs translated by `c2rust` that is scalable (handling half a million LOC in less than 10 s) and precise (handling inductive data structures) thanks to a strengthening of the Rust ownership model, which we call ownership monotonicity. Based on this new analysis, we prototyped a refactoring tool for translating C programs into Rust programs. Our experimental evaluation shows that the proposed approach handles real-world benchmarks and outperforms the state-of-the-art.

References

1. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA '02, pp. 311–330. Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/582419.582448

2. Astrauskas, V., Matheja, C., Poli, F., Müller, P., Summers, A.J.: How do programmers use unsafe rust? Proc. ACM Program. Lang. **4**(OOPSLA) (2020). https://doi.org/10.1145/3428204

3. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging rust types for modular specification and verification. Proc. ACM Program. Lang. **3**(OOPSLA) (oct 2019). https://doi.org/10.1145/3360573

4. Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '03, pp. 213–223. Association for Computing Machinery, New York, NY, USA (2003). https://doi.org/10.1145/604131.604156

5. Boyapati, C., Salcianu, A., Beebee, W., Rinard, M.: Ownership types for safe region-based memory management in real-time Java. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. PLDI '03, pp. 324–337. Association for Computing Machinery, New York, NY, USA (2003). https://doi.org/10.1145/781131.781168

6. Briggs, P., Cooper, K.D., Harvey, T.J., Simpson, L.T.: Practical improvements to the construction and destruction of static single assignment form. Softw. Pract. Exper. **28**(8), 859–881 (1998)

7. Brookes, S.: Variables as resource for shared-memory programs: semantics and soundness. Electron. Notes Theor. Comput. Sci. **158**, 123–150 (2006). https://doi.org/10.1016/j.entcs.2006.04.008

8. Brookes, S.: A semantics for concurrent separation logic. Theor. Comput. Sci. **375**(1–3), 227–270 (2007). https://doi.org/10.1016/j.tcs.2006.12.034

9. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science. p. 366–378. LICS '07, IEEE Computer Society, USA (2007). https://doi.org/10.1109/LICS.2007.30

10. Cheng, B.C., Hwu, W.M.W.: Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. PLDI '00, pp. 57–69. Association for Computing Machinery, New York, NY, USA (2000). https://doi.org/10.1145/349299.349311

11. Clarke, D., Östlund, J., Sergey, I., Wrigstad, T.: Ownership types: a survey. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. Types, Analysis and Verification. LNCS, vol. 7850, pp. 15–58. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36946-9_3

12. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA '98, pp. 48–64. Association for Computing Machinery, New York, NY, USA (1998). https://doi.org/10.1145/286936.286947

13. Das, M.: Unification-based pointer analysis with directional assignments. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language

Design and Implementation. PLDI '00, pp. 35–46. Association for Computing Machinery, New York, NY, USA (2000). https://doi.org/10.1145/349299.349309

14. De, A., D'Souza, D.: Scalable flow-sensitive pointer analysis for java with strong updates. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 665–687. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31057-7_29

15. Emre, M., Boyland, P., Parekh, A., Schroeder, R., Dewey, K., Hardekopf, B.: Aliasing limits on translating c to safe rust. Proc. ACM Program. Lang. **7**(OOPSLA1) (2023). https://doi.org/10.1145/3586046

16. Emre, M., Schroeder, R.: Artifact for "translating c to safer rust", September 2021. https://doi.org/10.5281/zenodo.5442253

17. Emre, M., Schroeder, R., Dewey, K., Hardekopf, B.: Translating C to safer rust. Proc. ACM Program. Lang. **5**(OOPSLA), 1–29 (2021). https://doi.org/10.1145/3485498

18. Evans, A.N., Campbell, B., Soffa, M.L.: Is rust used safely by software developers? In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. ICSE '20, pp. 246–257. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3377811.3380413

19. Feng, X.: Local rely-guarantee reasoning. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '09, pp. 315–327. Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1480881.1480922

20. Fitzgerald, N., Hoare, G., Mitchener, B., Puri, S.: Rust bindings to the z3 SMT solver. https://crates.io/crates/z3

21. Foster, J.S., Johnson, R., Kodumal, J., Aiken, A.: Flow-insensitive type qualifiers. ACM Trans. Program. Lang. Syst. **28**(6), 1035–1087 (2006). https://doi.org/10.1145/1186632.1186635

22. Greenfieldboyce, D., Foster, J.S.: Type qualifier inference for java. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications. OOPSLA '07, pp. 321–336. Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1297027.1297051

23. Hackett, B., Aiken, A.: How is aliasing used in systems software? In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering. SIGSOFT '06/FSE-14, pp. 69–80. Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1181775.1181785

24. Heine, D.L., Lam, M.S.: A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In: Cytron, R., Gupta, R. (eds.) Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, 9–11 June 2003, pp. 168–181. ACM (2003). https://doi.org/10.1145/781131.781150

25. Huang, W., Milanova, A., Dietl, W., Ernst, M.D.: Reim & reiminfer: checking and inference of reference immutability and method purity. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA '12, pp. 879–896. Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2384616.2384680

26. inc., I.: c2rust. https://github.com/immunant/c2rust

27. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: Rustbelt: securing the foundations of the rust programming language. Proc. ACM Program. Lang. **2**(POPL) (2017). https://doi.org/10.1145/3158154

28. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: a modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**(e20) (2018). https://doi.org/10.1017/S0956796818000151, https://hal.science/hal-01945446

29. Lerch, J., Späth, J., Bodden, E., Mezini, M.: Access-path abstraction: scaling field-sensitive data-flow analysis with unbounded access paths (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 619–629 (2015). https://doi.org/10.1109/ASE.2015.9

30. Lhoták, O., Chung, K.C.A.: Points-to analysis with efficient strong updates. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '11, pp. 3–16. Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1926385.1926389

31. Ling, M., Yu, Y., Wu, H., Wang, Y., Cordy, J.R., Hassan, A.E.: In rust we trust - a transpiler from unsafe C to safer rust. In: 44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, 22–24 May 2022, pp. 354–355. ACM/IEEE (2022). https://doi.org/10.1145/3510454.3528640

32. Machiry, A., Kastner, J., McCutchen, M., Eline, A., Headley, K., Hicks, M.: C to checked c by 3c. Proc. ACM Program. Lang. **6**(OOPSLA1) (2022). https://doi.org/10.1145/3527322

33. Matsakis, N.D., Klock, F.S.: The rust language. In: Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology. HILT '14, pp. 103–104. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2663171.2663188

34. Matsushita, Y., Denis, X., Jourdan, J.H., Dreyer, D.: Rusthornbelt: a semantic foundation for functional verification of rust programs with unsafe code. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2022, pp. 841–856. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523704

35. Matsushita, Y., Tsukada, T., Kobayashi, N.: Rusthorn: CHC-based verification for rust programs. ACM Trans. Program. Lang. Syst. **43**(4) (2021). https://doi.org/10.1145/3462205

36. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2

37. Ravitch, T., Jackson, S., Aderhold, E., Liblit, B.: Automatic generation of library bindings using static analysis. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, 15–21 June 2009, pp. 352–362. ACM (2009). https://doi.org/10.1145/1542476.1542516

38. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '06, pp. 129–136. Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1122971.1122992

39. Wolff, F., Bílý, A., Matheja, C., Müller, P., Summers, A.J.: Modular specification and verification of closures in rust. Proc. ACM Program. Lang. **5**(OOPSLA) (2021). https://doi.org/10.1145/3485522

40. Wu, J., Hu, G., Tang, Y., Yang, J.: Effective dynamic detection of alias analysis errors. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2013, pp. 279–289. Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2491411.2491439
41. Zhang, H., David, C., Yu, Y., Wang, M.: Ownership guided c to rust translation (2023). https://doi.org/10.48550/arXiv.2303.10515
42. Zhao, T., Baker, J., Hunt, J., Noble, J., Vitek, J.: Implicit ownership types for memory management. Sci. Comput. Program. **71**(3), 213–241 (2008)