FROM C
TOWARDS IDIOMATIC & SAFER RUST
THROUGH CONSTRAINTS-GUIDED REFACTORING

TAN YAO HONG, BRYAN

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2022

Supervisor:

Associate Professor Ilya Sergey

Examiners:

Professor Abhik Roychoudhury

Assistant Professor Umang Mathur

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

March 8, 2022	Tan Yao Hong, Bryan
.....	.....
Date	*Name*

Acknowledgements

First and foremost, this thesis would not have been possible without the guidance and kind mentorship of Prof. Ilya Sergey. As a sophomore new to the field of Computer Science, it was in an introductory data structures and algorithms course that Prof. Ilya would spark, and continue to instill, a love for tackling hard problems. It has been almost four years since, and I am eternally grateful for the numerous opportunities and learning experiences that he has provided me with to better myself.

Next, thank you to everyone at the VERSE research group for the warm and insightful discussions. To Kiran Gopinathan and Andreea Costea: thank you so much for being kind and patient with me in the process of penning this thesis. Much of this work is owed to you.

Finally, to my parents and Chau – thank you for always believing in me, even when I lacked the courage to do the same for myself.

Contents

Summary

The Rust programming language is a statically-typed programming language designed for *both* performance and safety, currently under development by Mozilla Research and the Rust community. Rust not only supports modern programming constructs such as macros, traits, ownership, and lifetimes in a syntax similar to C++, but also provides low-cost abstractions and memory safety without garbage collection. Consequently, it has found wide adoption as a "systems programming" language, and has been increasingly used in systems-level applications traditionally programmed in C code.

A plethora of legacy C code exist dormant in codebases that would benefit in terms of performance and safety if they were rewritten in Rust. However, manually porting these codebases is daunting and non-trivial because of Rust's ownership type system and its strict discipline on memory access.

While tools exist to automatically translate C into Rust, the translations are purely syntactic, mimicking the original C code and bypassing the safety checks of the Rust compiler through Rust's `unsafe` annotation. Thus, while the code is Rust in text, it is not Rust in spirit – the safety properties of Rust that are so desirable are elided.

This thesis describes a framework, CHRusty, for translating C to *safer* and *more idiomatic* Rust. Our theoretical contribution is this general framework for refactoring automatically translated Rust. We accomplish this by collecting constraints; a solution to the constraints point us towards the necessary changes, as the collected constraints inform our semantics-preserving rewrites. We provide an instantiation of this framework that removes conservative `unsafe` blocks and lifts raw pointers to arrays. Our practical contribution is an implementation of the array-rewrite instantiation of CHRusty as a tool written in Rust.

List of Figures

1 Introduction

Rust is a modern programming language currently developed by Mozilla Research with a thriving open source community. It began as a personal project by a disgruntled programmer because of a malfunctioning elevator in 2006,[1] before releasing its official version 1.0 in 2015, and is now supported by the Rust foundation. Rust is specifically designed with *both* safety and performance in mind, providing efficient yet reliable systems through low-level control, low-cost abstractions, and memory safety combined with the convenience and guarantees of modern and higher level constructs such as ownership, lifetimes, macros, generics, and traits [SCP17]. To this end, Rust provides strong static guarantees about memory and thread safety while avoiding garbage collection *and* still allowing for low-level manipulations.

Because of these qualities, Rust is often referred to as a "systems programming" language, and has seen adoption in building operating systems [Lev+15], garbage collectors [Lin+16], web browsers [And+15]. Furthermore, large and complex projects with both legacy and production codebases written in performant languages like C/C++ are increasingly being ported over to Rust: Firefox [Bry16], Android [SH21], and the Linux kernel [Cor21; Elh20]. More recently, Amazon has released an open-source Rust implementation of the QUIC protocol for HTTP/3 in their AWS encryption open-source library, citing "security [and] high performance" [Kam22].

By enforcing an ownership system that governs the capabilities to read and modify memory locations through borrowing and lifetimes, Rust guarantees

[1] https://twitter.com/mostlygeek/status/1492770712150413319

memory safety and thread safety as dangling pointers, iterator invalidation, concurrent data races, and so on, are automatically prevented. Furthermore, these guarantees are provided "almost for free". That is, they are statically checked at compile-time. Thus, existing codebases written in C/C++ can benefit from a Rust port, taking a small, possibly neglible hit (or even gain!) in performance in exchange for guarantees about memory and thread safety. For instance, a study performed on `cURL`, a data transfer utility library written in C, found that a Rust port would eliminate 53 of the 95 known `cURL` security flaws, at compile-time [Hut21]!

1.1 Why Automated Translation to Safe Rust?

Here is a disconcerting fact: there are plenty of codebases, known or unknown, with critical bugs and security flaws [Dur+14], potentially at great human and monetary costs [Dur+14; LT93; Dow97; ABC17]. Formally verifying these complex codebases is highly non-trivial and typically requires human intervention.

Even within Rust codebases, about 44.6% of unsafe function definitions were bindings for foreign functions used for linking against C libraries, suggesting that porting these C libraries to safe Rust would reduce the overall unsafety in the Rust ecosystem [Ast+20]. Porting existing codebases to Rust would not eliminate *all* bugs and flaws, but Rust's memory and thread safety guarantees provided almost for-free is a step in the right direction in reducing errors and attendant costs.

However, porting existing codebases to a new language is obviously an arduous task. With that, an *automated* translation of C into *safe* Rust, if possible, would eliminate a whole class of bugs automatically while preserving the semantics of the original C code, and can also present a method for discovering potential vulnerabilities. In fact, beyond eliminating bugs, it is possible that the resultant Rust code is more performant: by ruling out pointer aliasing, of which information is difficult to obtain [Hor97], program transformations that include but are not limited to reordering of statements can be performed as part of optimizations by a compiler [GLS01; WL95].

But *safe* Rust is not the only quality we can aspire towards – the automatically translated Rust code should also be *idiomatic*. We want the outcome of automatically translating a C codebase to be readable, and to begin to resemble something a human would write rather than code that has been coerced to abide by the Rust compiler's rules. Having such a property would be extremely desirable, as the tool would not be the final step of a translation pipeline; rather, programmers would be able to use the tool, *and then* carry on coding in Rust, using the safe and idiomatic Rust code as a starting point.

This thesis therefore investigates such an automated translation of C into *safer* and *more idiomatic* Rust. However, it is exactly Rust's expressive type system which guarantees memory and thread safety that presents a significant hurdle to automated translation: translating *unsafe* C to *safe* Rust code involves reasoning about properties of the C code in order to craft a semantically equivalent Rust program, whose *safe*-ness is captured in the Rust program being well-typed.

1.2 Contributions and Overview

In the following chapters, we describe a framework for automated translation of C into safer and more idiomatic Rust. We present several key contributions:

1. A translation pipeline from C to safer Rust: we use an industry-supported tool, `c2rust`,[2] to go from C to **`unsafe`** Rust, before running our tool to refactor **`unsafe`** Rust to safer and more idiomatic Rust.

2. A framework, CHRusty, for rewriting unsafe Rust into safer and more idiomatic Rust. The framework is based on constraint collection, solving the constraint system, and finally using the constraint system to inform a set of semantics-preserving rewrite rules.

3. An instantiation of this framework for lifting raw pointers to arrays.

4. A practical implementation of this instantiation of CHRusty in the form of a tool written in Rust.

5. An evaluation of CHRusty on several case studies.

[2] https://c2rust.com/

The thesis is structured as follows. We begin by providing some background in Ch. 2 about Rust, and how its sophisticated type system allows it to statically provide memory and thread safety guarantees. Then, in Ch. 3, we discuss existing work around automated translation from C to safe Rust, including an industry-backed tool called `c2rust` that we employ. We also explore related work in refactoring using type constraints, as well as closely-related efforts in refactoring Rust. Next, in Ch. 4, we describe the design of our framework as an algorithm that performs a principled translation via semantic-preserving rewrites informed by a solved constraint system. In particular, we look into an instantiation of this framework for lifting raw pointers to arrays. Ch. 5 details our practical implementation of the tool in Rust, alongside engineering to overcome several nontrivialities in writing our refactoring tool in Rust. Ch. 6 provides an evaluation of our tool on case studies. Finally, Ch. 7 concludes by comparing our results to closely-related efforts, discussing other instantiations of our framework, and exploring future work on the project.

2 Background

2.1 The Rust Programming Language

Rust is a programming language that provides static guarantees on memory safety, with no need for garbage collection, via its advanced type system features like affine types and regions. It has an interface with C and C++, features numerous object-oriented programming idioms like traits and generics as well as functional programming idioms like pattern matching and closures, and presents itself in a syntax similar to C++. Most of these features are zero-cost abstractions, lending to performance on par with C and C++ while being memory-safe.

2.1.1 Ownership in Rust

The key defining feature of Rust is how it ensures memory safety. Rust uses an ownership model to statically reason about references, mutability, lifetimes. The goal of Rust's memory management is simple: unreachable memory is deallocated memory, and no uninitialized memory should ever be read.

This is achieved via maintaining two invariants. Firstly, all allocated memory has a unique owner that is responsible for eventually deallocating it. Secondly, no memory can be simultaneously aliased, and mutable. For the Rust programmer, this is governed by a set of rules referred to as *ownership*; memory is managed through a system of ownership, and violations of the rules of ownership are caught at compile-time [KN21a]. These ownership rules preserve the two invariants of Rust's memory management [KN21a], and are presented as follows:

1. Each value in Rust has a variable that is its *owner*.

2. There can only be one owner at a time.

3. When the owner goes out of scope, the owned value is dropped.

The compiler uses an affine type system to track such ownership. Consider the following code snippet:

```
fn main() {
  let first_owner = "Hello, World!".to_string();
  let second_owner = first_owner;
  println!("{}", first_owner);
}
```

We first assign the string "Hello, World!" to `first_owner`, and then assign `first_owner` to `second_owner`. According to the rules of ownership, each value in Rust has only one owner at a time, and `first_owner` is no longer considered as valid. When `first_owner` is used in the `print` statement, the Rust compiler raises an error to prevent use of the invalidated reference:

```
error[E0382]: borrow of moved value: ‘first_owner‘
--> src/main.rs:4:20
|
|     let first_owner = "Hello, World!".to_string();
|         ----------- move occurs because ‘first_owner‘ has type ‘String‘,
|     which does not implement the ‘Copy‘ trait
|     let second_owner = first_owner;
|                        ----------- value moved here
|     println!("{}", first_owner);
|                    ^^^^^^^^^^^^ value borrowed here after move
```

This essentially statically rules out *double-free* errors, amongst numerous others common memory bugs.

The ownership of a variable abides by the same pattern every time: assigning a value to another variable moves it, and when a variable that includes data on the heap goes out of scope, the value will be automatically dropped.

However, one can observe that this quickly gets tedious: what if we just want a function to use a value, but not take ownership of it? Would Rust programmers always have to pass back anything that was passed into a function?

2.1.2 Borrowing and References

Luckily, Rust provides a notion of references. The key insight is that memory bugs occur when aliasing is mixed with mutability, i.e., when some memory is

accessible via multiple different paths, and is mutated. With this in mind, Rust can provide *references* to data, and these references can even be mutable, so long as they abide by what is known as the *exclusion* principle.

A reference is an address that can be followed to access data stored at that address (like a pointer) that is owned by another variable. A reference can either be immutable (`let x = &y`), or mutable (`let mut x = &mut y`). These references are governed by the *exclusion* principle: data can *either* be mutated through exactly one reference, *or* it can be immutably shared amongst many references [Jun+19]. This ensures memory safety – consider the following example:

```
1 fn main() {
2   let mut my_vec = vec![1, 2];
3   let int_ptr = &mut my_vec[1];              // int_ptr points into v.
4   my_vec.push(3);                            // May reallocate initial memory
5   println!("my_vec[1] = {}", *int_ptr);      // Compiler raises an error
6 }
```

We have a heap-allocated array of vectors `my_vec` with type `Vec<i32>`. `int_ptr` is a mutable reference of type `&mut i32` pointing into the address where the data of `my_vec` is stored. This is commonly referred to as an *interior pointer*. However, observe in line 4, that when we attempt to push the value 3 onto `my_vec`, new space on the heap may have to be allocated for `my_vec` if there is not enough space for the value 3 in the original position. Consequently, mutating `my_vec` may actually have a side effect of deallocating the memory on the heap that `my_vec` initially pointed to. This is a bug known as "iterator invalidation", and the compiler flags this with the following error:

```
1    error[E0499]: cannot borrow ‘my_vec‘ as mutable more than once at a time
2   --> src/main.rs:4:5
3    |
4  3 |     let int_ptr = &mut my_vec[1];
5    |                       ------ first mutable borrow occurs here
6  4 |     my_vec.push(3);
7    |     ^^^^^^^^^^^^^^ second mutable borrow occurs here
8  5 |     println!("my_vec[1] = {}", *int_ptr);
9    |                                -------- first borrow later used here
```

The error arises because we performed a mutable borrow via `my_vec.push(3)` in line 4, since `my_vec.push(3)` desugars to `Vec::push(&mut my_vec,3)`. But in

line 3, we also performed a mutable borrow when we assigned `int_ptr` as `&mut my_vec[1]`. Consequently, the exclusion principle is violated; the compiler complains, and we have avoided the "iterator invalidation" bug. This analysis performed by the compiler is known as "borrow checking", and relies on a notion of "lifetime":

```
1 fn main() {
2   let mut my_vec = vec![1, 2];
3   let int_ptr = &mut my_vec[1];          // Init: Lifetime 'a
4   Vec::push(&mut my_vec, 3);             // Init: Lifetime 'b
5   println!("my_vec[1] = {}", *int_ptr);
6 }
```

The compiler infers the following lifetimes for the borrows in lines 2 and 3. Then it checks that

1. The reference can only be used while its lifetime is ongoing.
2. The original referent does not get used until the lifetime of its loan gets expired.

However, in line 4, `int_ptr` is used, so the lifetime `'a` has to last at least as long as line 4 (rule 1). But, in line 3, `my_vec` is used while the lifetime of its loan is not expired, violating rule 2, leading to a compilation error.

On the other hand, many immutable references to the same memory location may exist together at the same time, but may not mutate. That is, immutable references permit aliasing but *not* mutation. Immutable references abide by the same rules as mutable references, but rule 2. is weakened: instead of checking that the original referent does not get *used*, the compiler checks that the original referent does not get *mutated*:

```
1 fn main() {
2   let mut my_vec = vec![1, 2];
3   let int_ptr1 = &my_vec[1];             // Init: Lifetime 'a
4   let int_ptr2 = &my_vec[1];             // Init: Lifetime 'b
5   println!("my_vec[1] = {}", *int_ptr1);
6   println!("my_vec[1] = {}", *int_ptr2);
7 }
```

2.1.3 Provenance of Unsafety

While the Rust ownership type system allows the compiler to statically guarantee memory safety, sometimes the discipline it enforces on access and sharing of memory locations is strict to the point that it limits expressiveness. That is, the restrictions can make it difficult, or even impossible, to implement certain common designs. For instance, data structures where the pointers form a cycle, such as a doubly-linked list that requires aliasing, are extremely challenging to implement in Rust [Ast+20]. In fact, there is even an unofficial guide to learning Rust by *writing different variants of linked lists*.[1]

Rust allows programmers to selectively determine where restrictions of the type system can be loosened by wrapping such code blocks as an **`unsafe`** block. However, that is the extent of what **`unsafe`** permits: the code should still be "correct". That is, the behavior of the resulting program (and interaction of **`unsafe`** and `safe` code) should not produce undefined behavior [KN21b].[2] Ensuring that the **`unsafe`** block is correct is the programmer's responsibility, and any **`unsafe`** code should be hidden behind a safe abstraction. In fact, recent work has attempted to verify such a property for a subset of Rust, and can even explicate the verification condition necessary for a Rust library that uses **`unsafe`** features to be considered safe [Jun+17].

The **`unsafe`** annotation grants access to the following features that may violate the memory safety guarantees of Rust's static semantics [KN21c]:

1. Dereferencing a raw pointer. A raw pointer is a variable that stores the address of an object in memory, but whose lifetime is not controlled by a smart pointer.[3]

2. Reading or writing a mutable or external static variable.

3. Accessing a field of a union, other than to assign to it.

4. Calling an unsafe function (including an intrinsic or foreign function).

[1] `https://rust-unofficial.github.io/too-many-lists/`

[2] For an unexhaustive list of undefined behavior, see: `https://doc.rust-lang.org/stable/reference/behavior-considered-undefined.html`

[3] Rust also provides *smart* pointers, which encapsulate raw pointers with additional metadata and capabilities. Some examples of smart pointers in Rust include `Box<T>` for allocating values on the heap, and `String` for string allocation on the heap.

5. Implementing an unsafe trait (a trait with one or more unsafe methods).

```
1 fn main(){
2   let mut v = 1;
3   let raw_ptr1 = &mut v as *mut i32;
4   let raw_ptr2 = raw_ptr1;
5   unsafe { *raw_ptr1 = 2; }
6   unsafe { println !("{}", *raw_ptr2); } // Prints "2".
7 }
```

Listing 2.1: Aliasing and Mutating in unsafe Rust

Therefore, we can actually alias *and* mutate in Rust, as in Listing 2.1. The onus is simply on us to reason and ensure that this is correct.

Note that the dereference of raw pointers that occur in lines 5 & 6 have to be wrapped in **unsafe** blocks. Since the borrow checker does not track raw pointers, it cannot guarantee that will not violate memory safety – therefore, the **unsafe** block is a promise that we have sufficiently ensured memory safety even when we dereference the raw pointers.

2.1.4 Discussion

For a more indepth introduction to Rust, one can visit the official introductory book to Rust, "The Rust Programming Language" [KN21a]. For a detailed reference to the Rust programming language, one can refer to the official reference [KN21d].

3 Related Work

In this chapter, we establish existing and related work upon which our framework builds upon. We discuss program transformation, program transformation from C to *unsafe* Rust via `c2rust`, the differences in sources of unsafety between automatically translated Rust and idiomatic Rust, refactoring using type constraints, and finally, closely-related efforts.

3.1 Program Transformation

Program transformation refers to changing a program's source code into a different source code. Program transformation is typically categorized into the categories of (1) translation and (2) rephrasing. Translation refers to the transformation from a particular language, for instance the C language, to another language like Rust. This is akin to what `c2rust` does when it performs a syntactic translation of C code to unsafe Rust (Sec. 3.2). On the other hand, rephrasing is similar to transpilation — it is a transformation of source code of a language to the same language. The category of rephrasing can be further refined: program refactoring is a sub-category of program rephrasing that targets *improving source code quality*. Program refactoring should not change the behavior of the program.

For our purposes, we are interested in a program transformation from C to safer and more idiomatic Rust. We achieve this by a program *translation* from C to unsafe Rust, then a program *rephrasing* to remove unsafety from unsafe Rust, bringing us closer to safe Rust, and finally a program *refactoring* to lift raw pointers to arrays, bringing us closer to idiomatic safe Rust. Nevertheless, in this

thesis, the terms *rewrite* and *refactor* are conflated, since our rewrites are intended to be semantics-preserving.

3.2 Automatic Translation with `c2rust`

`c2rust` is an industry-backed tool to automatically translate C programs to Rust [inc20a], and is the successor to earlier tools like Citrus [Dev17] and Corrode [Sha17].

The `c2rust` transpiler translates a C program into an *unsafe* Rust program that mirrors the C code. However, these translations are purely syntactic, and produce memory and thread-unsafe Rust code by explicitly marking all translated code as **`unsafe`** code blocks [inc20c]. Furthermore, `c2rust` does not provide any formal guarantees that the resulting Rust code preserves the semantics of the original C code, and so they also provide a *cross-check tool* (Sec. 7.3.1) that compares the execution traces of two programs on a test input and validates that the semantics of the original program is indeed preserved [inc20b].

We leverage `c2rust` in our translation pipeline to take the original C program, and produce an initially **`unsafe`** Rust program. Then, we use this **`unsafe`** Rust program to perform our semantics-preserving incremental rewrites, as we will describe in Ch. 4.

3.3 Unsafety in Automatically Translated Rust code

Before we dive into the translation framework, we first investigate sources of unsafety in *automatically translated* Rust code.

Evans et al. [ECS20] observe that the use of **`unsafe`** blocks may actually result from an internal **`unsafe`** block in another function. Thus, they keep track of potentially unsafe functions by examining the callgraph of Rust programs, and found that 89.2% of potentially **`unsafe`** functions were because of **`unsafe`** propagation from calling other **`unsafe`** functions. For our framework, this points towards an approach where we begin from the "leaf" functions in the callgraph of a program, and work our way upwards towards the "root", i.e., the `main` function.

Further, Emre et al. [Emr+21] note that the sources of unsafety in Rust code that is automatically translated significantly differs from the sources of unsafety in idiomatic Rust code. Primarily, `c2rust` translates C-style pointers into raw pointers, and in Rust, the dereference of a raw pointer necessitates being wrapped in an **`unsafe`** block (Sec. 2.1.3). Thus, legitimate sources of unsafety in automatically translated Rust code stem prevalently from this conversion of pointers from C.

3.4 Refactoring using Type Constraints

Refactoring is the process of modifying a program's source code without altering the behavior of the program, with the goal of improving the quality of the source code [Fow18]. Existing work has looked into automated refactoring of Rust programs for syntactic clarity (variable renaming, inlining, lifetime elission from function signatures) [SCP17], as well as idiomatic refactoring for the `corrode` C-to-Rust translator [Zbo], which has since been superseded by `c2rust`.

Type constraints are a formalism to express the constraints that must be satisfied for a program to be type-correct [PS94; PS91]. While initially used to tackle type-checking and type-inference of object-oriented languages, it has recently been observed that source code modifications can be informed by a system of type constraints [TKB03; Tip+11]. By constructing a system of type constraints over the original program, a solution to the constraint system not only asserts the type safety of the proposed transformation, but also indicates possible refactorings.

3.5 An Existing Translator: Laertes

Emre et al. [Emr+21; ES21] present the first technique for automatically removing some sources of unsafety in translated Rust programs. Their technique generates safer Rust programs by optimistically converting a *subset* of raw pointers into safe references, and then using the Rust compiler as an oracle to iteratively refine the program.

Their technique hinges on the observation that raw pointers form the largest

source of unsafety in automatically translated Rust code. Thus, raw pointers are optimistically rewritten, regardless of what they were initially *intended* to be used for. Consequently, by treating raw pointers as ground truth for unsafety, a uniform rewrite is applied to all instances of raw pointers iteratively, using the compiler as oracle. However, this leads to unwieldy and unidiomatic Rust code (see: Listing 7.1), which goes against the principles of code refactoring: code quality should monotonically increase [Fow18]. Readability of automatically translated code is of paramount importance – while this may be a non-issue if one were just translating legacy code for the sake of verification, ongoing projects for which automated translation serves as a foundation for subsequent ports *require* readable code, so that developers can interact with the code base.

3.6 Our Approach

We approach the problem of rewriting `unsafe` code into safe code differently from Laertes, since we want to also have *idiomatic* code. Instead of rewriting from bottom-up – from raw pointers in the function upwards – we bubble unsafety downwards, starting from the function's signature, by applying specially designed semantics-preserving rewrites. These rewrites are informed by a system of type constraints that are constructed over the `unsafe` `c2rust`-translated Rust program. By collecting constraints regarding the use of raw pointers as well as type information, the solution to the type constraint system indicates when raw pointers are intended to be used as arrays, allowing us to "lift" raw pointers to arrays in a semantics-preserving fashion. Thus, not only do we eliminate code conservatively marked as `unsafe`, but we also refactor the source code to be more idiomatic by performing a rewrite in keeping with the intention of the original program instead of simply a syntactic refactoring, resulting in higher-quality code.

4 A Framework For Semantics-Preserving Translations

> *Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure.*
>
> — M. Fowler [Fow18]

We describe, in this chapter, our framework for automated translation of C code to *safer* Rust. The framework (Fig. 4.1) is intended to be *generic*, and is designed to take in different "rewrite passes" as parameters to iteratively refactor the Rust code until it resembles what the user wants. That is, specific instantiations of the system refer to parameterizing the system with a specific choice of constraint collection rules, constraint solver rules, and the transformation rules. This generic framework is referred to as CHRusty,[1] and we present an implementation of the array-rewrite instantiation of the CHRusty framework which refactors `c2rust`-translated, **`unsafe`** code into safer, more idiomatic Rust code by lifting raw pointers to arrays.

4.1 The Translation Pipeline

4.1.1 The Scope of the Translation

While we provide a generic framework, we limit the scope of this thesis to tackling specifically the problem of lifting raw pointers to arrays. As rewriting the *entirety* of the C language may be too ambitious, we restrict our target domain of *safe* Rust to the subset of trivially *safe* Rust that includes non-heap manipulating code, together with raw pointers that are semantically intended to

[1] CHRusty: C to Rust via CHR.

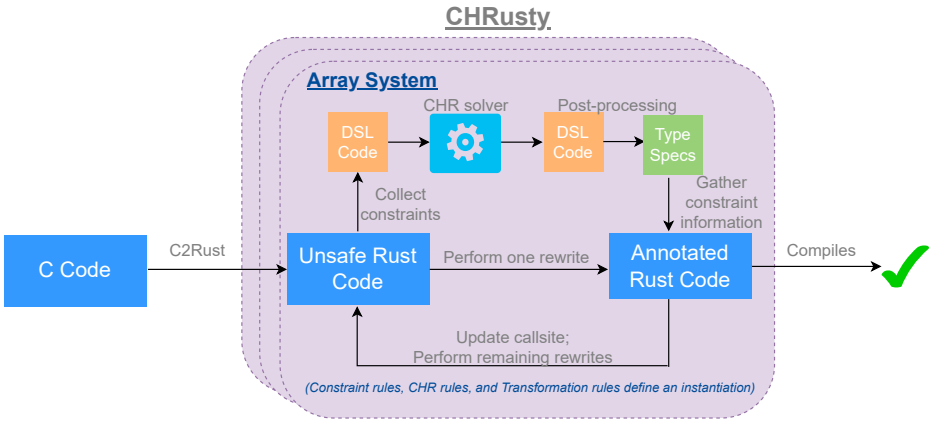


Figure 4.1: Framework for Automatic Translation from C to safer Rust

be used as static or dynamic arrays. Hence, our goal is to remove as much of the conservatively-labelled `unsafe` annotations, and to lift a subset of raw pointers to arrays. In particular, we aim to perform the rewrites in an idiomatic fashion that represents the intent of the original code, so that the rewritten Rust code can be further expanded upon by a developer.

4.1.2 Pipeline

We now present the framework, as illustrated in Fig. 4.1. Firstly, given a code base written in C, we transform the code base into `unsafe` Rust by leveraging the `c2rust` tool. As described in Sec. 3.2, `c2rust` is an industry-backed tool that allows for automatic translation of C programs into Rust. However, as a by-product of the syntactic translation, the resulting code is conservatively marked as `unsafe`. Furthermore, the sources of unsafety in automatically translated code differ significantly from those in idiomatic Rust code as explored in Sec. 3.3. Since `c2rust` merely performs a syntactic rewrite, it does not cross the hurdle of reasoning about properties of the C and resulting Rust code to remove unsafety. Thus, a large chunk of the translated Rust code *need not actually be* marked `unsafe`, but is done so out of convenience.

Our first observation is that rewriting C code into safe Rust directly is incredibly challenging and non-trivial. One would have to design and implement a series of static analyses on the original C code that can reason about lifetimes, ownership, and aliasing in keeping with Rust's sophisticated type system, and then construct a set of rewrite rules based on the analyses. Instead of trying to tackle this problem as a whole, we opt to "hill-climb": by using `c2rust` to generate Rust code that is conservatively marked as `unsafe`, we can incrementally

remove unsafety by traversing through the code and bubbling unsafety down from the function level, to the statement level, and so on.

Consider the motivating example in Listing 4.1. The `reset` function, written in C, takes as input `arr`, which is a pointer to an `int`, and `size` which is an `int`.

```
1 void reset(int* arr, int size) {
2   int i = 0;
3   while (i<size) {
4     *(arr + i) = 0;
5     i + +;
6   }
7 }
```

Listing 4.1: Motivating Example: Resetting Arrays in C

Then, starting from the `i`th offset from the memory address of `arr`, it sets the value at this address to 0, before incrementing `i`. This repeats until `i = size`.

Instead of rewriting `reset` immediately into safe Rust, we employ `c2rust` to automatically rewrite this into Rust code. `c2rust` then conservatively labels the function as **`unsafe`**:

```
1 pub unsafe extern "C" fn reset(mut arr: *mut libc::c_int,
2    mut size: libc::c_int) {
3         let mut i: libc::c_int = 0 as libc::c_int;
4         while i < size {
5                 *arr.offset(i as isize) = 0 as libc::c_int;
6                  i += 1
7           };
8 }
```

Listing 4.2: Resetting Arrays, `c2rust` output

Given the **`unsafe`** Rust code emitted from the `c2rust` tool, we retrieve its AST and give each of the nodes a unique label (Sec. 4.3.1). Then, the labels are assigned some constraints depending on what the nodes represent (Sec. 4.3.3, Sec. 4.3.6). These constraints are collected, parsed, and handed over to our Constraint Handling Rules (CHR) system (Sec. 4.3.4), a declarative, rule-based constraint solver, of which we use a Prolog implementation of the KU Leuven CHR system [Frü94; SD04].[2] We parse the output and collect the resolved

[2]Available at https://www.swi-prolog.org/man/chr.html

```rust
fn reset<T: IndexMut<usize, Output = i32>>(mut arr: &mut T, size: usize) {
        let mut i: usize = 0;
        while i < size {
                arr[i]= 0;
                i += 1
        };
}
```

Listing 4.3: Resetting Arrays, safe Rust

constraints from the CHR system, performing some necessary engineering like storing the results in a union-find data structure to allow easy access to the equivalence classes of labels (Sec. 5.3.1). Next, we perform our rewrite pass on the **unsafe** Rust code, which queries the resolved constraints for information about the program so that it can perform the refactoring (Sec. 4.4). Ultimately, we should end up with `reset` in safe Rust as in Listing 4.3, though the details about how we lift the raw pointer into arrays are left for later.

Finally, we want to propagate the changes made locally in a function across the whole program, so we update the call sites of the function to reflect this (Sec. 4.4.2). This takes place for every `rewrite` pass we have in our system. As discussed in Sec. 4.1.1, for the purposes of this thesis, we only provide a pass to handle conservative **unsafe** blocks, and the lifting of raw pointers to arrays.

4.2 Description of the Algorithm

4.2.1 Generic Algorithm for Rewrite Passes

Algorithm 4.2.1: A general algorithm for rewrites

```
1  Function Main(src):
2      call_graph ← build_callgraph(src);
3      rewrite_passes ← {arr_pass};
4      for (curr_fn, callers) in call_graph do
5          for rewrite_pass in rewrite_passes do
6              rewrite_pass(curr_fn);
7          end
8          for caller_fn in callers do
9              update_callsite(caller_fn);
10         end
11     end
```

Here, algorithm 4.2.1 describes the top-level algorithm for performing rewrites, while algorithm 4.2.3 describes an instantiation of our framework:

a *particular rewrite for arrays* implemented by CHRusty, which we call `arr_pass`. Firstly, algorithm 4.2.1 takes as input a "source" file. We will go into more details about the practical implementation in Ch. 5, but for now this can be understood as either the `lib.rs` file generated as part of the Rust project translated from `c2rust` that provides information about all the constituent files and functions in the project, or the `main.rs` file.

Since our rewrite pass `arr_pass` operates on the level of *functions*, we have to take care to perform our rewrite in the direction `callee` $\underset{\texttt{rewrite}}{\longrightarrow}$ `caller` instead of `caller` $\underset{\texttt{rewrite}}{\longrightarrow}$ `callee` so that we can propagate the necessary information and changes. Thus, we first construct a callgraph from `lib.rs`, which is a control-flow graph that represents the calling relationship between the functions (line 2). The callgraph is a set of pairs (*curr_fn*, *callers*), where *curr_fn* is a fully-qualified function name, and *callers* = $\{f' \mid f'$ calls *curr_fn*$\}$ is the set of functions that call *curr_fn*. To retrieve the "leaves" of the callgraph, i.e., the functions that have no callees, we perform a topological sort on our callgraph to obtain a list of (*curr_fn*, *callers*), ordered from the bottom of the calling hierarchy (the leaves of the callgraph) to the top (the `main` function), taking care to handle cycles. We then work our way from the leaves – the bottom of the calling hierarchy – up to the `main` function.

Having constructed our callgraph `call_graph`, we define our set of rewrite passes `rewrite_passes` (line 3). In this case, we collect only our particular array rewrite pass, `arr_pass`, which we describe in further detail in algorithm 4.2.3 and later on.

Next, for every pair of fully-qualified function name and caller set (*curr_fn*, *callers*) in our callgraph `call_graph` ordered by calling hierarchy (line 4), we iterate through our set of rewrite passes `rewrite_passes` (in this case we only have one!), and perform the rewrite pass on *curr_fn* (line 5-6). After performing all our rewrites on *curr_fn*, we want to propagate the updates that were made, so we look up all the functions `caller_fn` in *callers* that call *curr_fn*, and update the call sites to reflect the changes (line 8-9).

4.2.2 A Rewrite Pass for Arrays: `arr_pass`

Algorithm 4.2.2: Rewrite pass structure

```
1 Function rewrite_pass(curr_fn):
2     ast ← ast_of(curr_fn);
3     constraints ← collect_constraints(ast);
4     solved_constraints ← chr_solve_constraints(constraints);
5     transformed_ast ← rewrite(ast, solved_constraints);
6     surgery(curr_fn, transformed_ast);
```

Here, we describe the structure of a rewrite pass. `rewrite_pass` takes as input a fully-qualified function name `curr_fn` (line 1). Then, we build the AST of the function represented by `curr_fn` (line 2). Next, we label each node in the AST, assign and collect constraints corresponding to the labels (line 3), and solve them using our Prolog CHR constraint solver system (line 4), discussed in Sec. 4.3 and in Sec. 5.3. The resulting solved constraints, together with the AST, are then passed to a helper function `rewrite` that implements a particular set of rewrite rules to transform the AST (line 5). Finally, we perform "surgery" (Sec. 5.4): with the fully qualified fuction name, we have the absolute location of the file containing the function, so we `read` in the source file containing the function, transform the AST (Sec. 4.4), and finally, `write` the transformed AST back to the file (line 6).

Observe that this structure is generic: the functionality of the pass depends on the particular constraint system that we use, and the particular rewrite rules that we use to transform the AST. In particular, we can instantiate the rewrite pass, `arr_pass`, that was used in `rewrite_passes` in algorithm 4.2.1 (line 3), as follows:

Algorithm 4.2.3: Example rewrite pass: `arr_pass`

```
1 Function arr_pass(curr_fn):
2     ast ← ast_of(curr_fn);
3     constraints ← collect_constraints(ast, arr_pass);
4     solved_constraints ← chr_solve_constraints(constraints);
5     transformed_ast ← rewrite(ast, solved_constraints, arr_pass);
6     surgery(curr_fn, transformed_ast);
```

giving us a rewrite pass that targets non-heap manipulating code that was conservatively marked as **`unsafe`**, as well as the lifting of raw pointers to arrays. We describe the constraint system and rewrite rules used to instantiate `arr_pass`

in the following sections.

4.3 Constraint Collection

Type constraints are a formalism for expressing the subtyping relationships between the types of program elements that must be satisfied in order to determine whether a program is type-correct [PS94]. For our purposes, we define a set of type constraints that are expressive enough to handle our use-case of lifting pointers to arrays. Then, a solution to the constraint system not only asserts that the proposed transformation preserves the program behavior, but also indicates possible "fixes". These "fixes", for our purposes, reflect the transformations for lifting pointers to arrays. Note that these type constraints do not express the complete set of correctness constraints for any arbitrary transformation of Rust code. Rather, they are chosen specifically to preserve the program semantics for our subset of rewrites.

In the subsequent subsections, we will describe our constraint system for a *particular* rewrite pass, `arr_pass`, that is an instantiation of our framework. The constraint system for `arr_pass` collects and solves constraints pertaining to the refactoring of arrays.

4.3.1 Collecting Constraints from our Motivating Example

Recall that we began by taking a well-typed, but **`unsafe`**, Rust program that was naively translated from `c2rust`. Next, we construct its AST (line 2-3 in algorithm 4.2.2), and uniquely label every node, as we have done for our motivating example, `reset`, from Listing 4.1.

```
1  pub unsafe extern "C" fn reset(mut arr/*1*/: *mut libc::c_int,
2  mut size/*2*/: libc::c_int) {
3    let mut i/*3*/: libc::c_int = (0/*4*/ as libc::c_int)/*5*/;/*6*/
4    while (i/*7*/ < size/*8*/)/*9*/ {
5      (*(arr/*10*/.offset((i/*11*/ as isize)/*12*/))/*13*/)/*14*/ =
6      (0/*15*/ as libc::c_int)/*16*/;/*17*/
7      i/*18*/ += 1/*19*/;   /*20*/
8    }; /*21*/
```

```
9   }
```

Listing 4.4: `reset` labelled

Then, we collect the following type constraints, using the constraint collection rules that we will describe later in Sec. 4.3.6

$M = \{\texttt{arr} : \alpha_1, \texttt{size} : \alpha_2, \texttt{i} : \alpha_3\}$

$\Sigma = \{\alpha_4 \simeq \alpha_5, \alpha_3 \simeq \alpha_5, \alpha_7 \simeq \alpha_8, \alpha_7 \simeq \alpha_3, \alpha_3 \simeq \alpha_8, \alpha_8 \simeq \alpha_2, \alpha_1 \simeq \alpha_{10}, \alpha_3 \simeq \alpha_{11}, \alpha_{14} \simeq \alpha_{16}, \alpha_{15} \simeq \alpha_{16}, \alpha_{18} \simeq \alpha_{19}, \alpha_3 \simeq \alpha_{18},$

$Compat(\alpha_2, \texttt{c\_int}),$

$Compat(\alpha_3, \texttt{c\_int}),$

$Compat(\alpha_5, \texttt{c\_int}),$

$Compat(\alpha_{11}, \texttt{isize}),$

$Compat(\alpha_{15}, \texttt{c\_int})$

$Deref(\alpha_{13}, \alpha_{14}),$

$Mut(\alpha_{14}),$

$Mut(\alpha_{18}),$

$Offset(\alpha_{10}, \alpha_{12}, \alpha_{13})\}.$

This system of constraints is then solved to produce a solution set of type constraints:

$\Sigma' = \{$

$\{\alpha_4 \simeq \alpha_5, \alpha_3 \simeq \alpha_5, \alpha_7 \simeq \alpha_8, \alpha_7 \simeq \alpha_3, \alpha_3 \simeq \alpha_8,$

$\alpha_8 \simeq \alpha_2, \alpha_3 \simeq \alpha_{11}, \alpha_3 \simeq \alpha_{18}, \alpha_{18} \simeq \alpha_{19}\},$

$\{\alpha_{14} \simeq \alpha_{16}, \alpha_{15} \simeq \alpha_{16}\},$

$\{\alpha_1 \simeq \alpha_{10}\},$

$Compat(\alpha_2, \texttt{c\_int}),$

$Compat(\alpha_3, \texttt{c\_int}),$

$Compat(\alpha_5, \texttt{c\_int}),$

$Compat(\alpha_{11}, \texttt{isize}),$

$Compat(\alpha_{15}, \texttt{c\_int})$

$Mut(\alpha_{14}),$

$Mut(\alpha_{18}),$

$Index(\alpha_{10}, \alpha_{12}, \alpha_{14})\}.$

We abuse notation to represent the equivalence classes of $\simeq$. From this solution set, we can query and retrieve any necessary type constraint information. Notably, observe that the solution set has the type constraint $Index(\alpha_{10}, \alpha_{12}, \alpha_{14})$, which was inferred from $Offset(\alpha_{10}, \alpha_{12}, \alpha_{13}), Deref(\alpha_{13}, \alpha_{14}) \implies Index(\alpha_{10}, \alpha_{12}, \alpha_{14})$. In essence, this describes how we can deduce that the node labelled as α_{10}, with index node labelled as α_{12} and output node labelled as α_{13}, is in fact an *Index* constraint with index node α_{12} and output node α_{14}: the output node α_{13} is also used as part of a constraint $Deref(\alpha_{13}, \alpha_{14})$ which indicates that the output of the node labelled α_{13} when dereferenced is the node labelled α_{14}.

Finally, with the solved system of constraints, we have enough information to rewrite `*arr.offset(i as isize) = 0` as `arr[i] = 0`, and know that `arr` should implement the trait `T: IndexMut<usize, Output = i32>`:

```
fn reset<T: IndexMut<usize, Output = i32>>(mut arr: &mut T, size: usize) {
        let mut i: usize = 0;
        while i < size {
                arr[i]= 0;
                i += 1
        };
}
```

Listing 4.5: Resetting Arrays, refactored into safe Rust

Even though these constraint terms have not yet been properly introduced or defined, this example serves to demonstrate how we give each node a unique label, collect the constraints associated with each label, and then use the solved system of constraints to inform our rewrites.

4.3.2 Language of Rust Types

Before we describe the syntax of our constraints, we first define a compact representation of the language of the subset of Rust's types that we are concerned with, namely its primitive types and its pointer constructs

If t is a primitive Rust type or a Rust data structure,[3] then t is also a τ_b. If t is a τ_b, or $t \in TVars$, or t is a raw pointer of type τ, or a reference of type τ, or a smart pointer of type τ, then t is a τ.

[3] https://doc.rust-lang.org/reference/types.html

$$
\begin{array}{lll}
(\textit{Basic Types}) & \tau_b & := \quad \textit{prim\_type} \mid \textit{data\_struct} \\
 & \tau & := \quad \tau_b \mid T \mid \textit{Raw}(\tau) \mid \textit{Ref}(\tau) \mid \textit{Box}(\tau) \\
(\textit{Capability}) & c & := \quad \textit{Mut} \mid \textit{Imm}
\end{array}
$$

$$T \in \textit{TVars}$$

Figure 4.2: Language Syntax

Furthermore, *Mut* and *Imm* are both capabilities that reflect whether a variable is mutable or immutable.

4.3.3 Constraint Syntax

Domain

$$
\begin{array}{lrcl}
(\textit{Constraints Store}) & \Sigma & \triangleq & \textit{Set}(t^c) \\
(\textit{Label}) & \alpha \in \textit{NodeLabels} & \triangleq & \textit{GhostVars} \\
(\textit{Label Map}) & M : \textit{Vars} & \longrightarrow & \textit{NodeLabels}
\end{array}
$$

Constraints Syntax

$$
\begin{array}{lcl}
(\textit{Type Constraints Terms}) \quad t & \triangleq & \tau \mid \alpha \\
(\textit{Type Constraints}) \quad t^c & \triangleq & t \simeq t \mid \textit{Compat}(\alpha, \tau) \mid \textit{Offset}(\alpha, \alpha, \alpha) \mid \textit{Deref}(\alpha, t) \mid \\
 & & \textit{Index}(\alpha, \alpha, \alpha) \mid \textit{ShouldIndex}(\alpha, \alpha, \alpha) \mid \textit{Malloc}(\alpha, \tau) \mid \\
 & & \textit{Vec}(\alpha, \alpha, \alpha) \mid \textit{Mut}(\alpha)
\end{array}
$$

Figure 4.3: Constraints Syntax

A type constraint term t is either a Rust type τ as defined in Sec. 4.3.2, or a unique label α. A *type constraint* t^c is either an unary relation, or a relation between two or more terms. We define our constraints store Σ as a set containing our type constraints t^c. As a convenient shorthand, we define $\neg t^c$ to mean that $t^c \notin \Sigma$. To assist in collecting constraints, we also define a map M from program variables to labels. In this thesis, we define the following relations:

1. $t_1 \simeq t_2$ which indicates that t_1 is equivalent to t_2,
2. $\textit{Compat}(\alpha, \tau)$ which indicates that the node labelled α is compatible with the type τ,
3. $\textit{Offset}(\alpha, \alpha_{ind}, \alpha_{out})$ which indicates that offsetting the node labelled α by the node labelled α_{ind} will retrieve as an output the node labelled α_{out}.

4. $\mathit{Deref}(\alpha, \alpha_{out})$ which indicates that dereferencing the node labelled α will retrieve as an output the node labelled α_{out}.

5. $\mathit{Index}(\alpha, \alpha_{ind}, \alpha_{out})$ which indicates that indexing the node labelled α by the node labelled α_{ind} will retrieve as an output the node labelled α_{out}.

6. $\mathit{ShouldIndex}(\alpha, \alpha_{ind}, \alpha_{out})$ which indicates that the node labelled α *should* be a collection, that when indexed by the node labelled α_{ind}, will retrieve as an output the node labelled α_{out}.

7. $\mathit{Malloc}(\alpha, \tau)$ which indicates that the node with label α has memory allocated corresponding to the size of type τ,

8. $\mathit{Vec}(\alpha, \alpha_{ind}, \alpha_{out})$ which indicates that the node labelled α is a dynamic array, that when indexed by the node labelled α_{ind}, will retrieve as an output the node labelled α_{out}.

9. And finally, $\mathit{Mut}(\alpha)$, which indicates that the node labelled α is mutable.

4.3.4 The Constraint System

Constraints System

(*Offset Collapse*)	$\mathit{Offset}(t_1, t_2, t_3), \mathit{Offset}(t_3, t_4, t_5)$ $\iff t_2 \simeq t_4, t_3 \simeq t_5, \mathit{Offset}(t_1, t_2, t_5)$
(*Array Reconstruction*)	$\mathit{Offset}(t_1, t_2, t_3), \mathit{Deref}(t_3, t_4)$ $\iff \mathit{Index}(t_1, t_2, t_4)$
(*Implicit Deref*)	$\mathit{Index}(t_1, \_, t_3), \mathit{Deref}(t_1, t_4)$ $\iff t_3 \simeq t_4$
(*Vec Reconstruction*)	$\mathit{ShouldIndex}(t_1, t_2, t_3), \mathit{Malloc}(t_1, t_3), \neg \mathit{Compat}(t_1, \mathit{Raw}(\_))$ $\implies \mathit{Vec}(t_1, t_2, t_3)$
(*Indexed Subsumption*)	$\mathit{Vec}(t_1, t_2, t_3)$ $\implies \mathit{Index}(t_1, t_2, t_3)$
(*Dereference Reference*)	$\mathit{Deref}(t_1, t_2), \mathit{Ref}(t_1, t_3)$ $\iff t_2 \simeq t_3$
(*Indexed Collapse*)	$\mathit{Index}(t_1, \_, t_3) \setminus \mathit{ShouldIndex}(t_1, \_, t_4)$ $\iff t_3 \simeq t_4$

Figure 4.4: Constraints System

To solve our constraint system, we use the Constraint Handling Rules (CHR) system, which is a declarative, rule-based constraint solver [Frü94; SD04]. The

rules of our constraint system are defined above in Fig. 4.4, and are written in the syntax of CHR rules.[4]

There are three types of rules in CHR, each with their own semantics. Firstly, the `simplification`: LHS $\iff$ RHS rule, given a constraints store Σ, removes the constraints in the head (LHS) from Σ and adds the constraints in the body (RHS) to Σ. Secondly, the `simpagation`: LHS \ LHS' $\iff$ RHS rule, given a constraints store Σ, removes the constraints in the head *after* the \, i.e., LHS', from the Σ and adds the constraints in the body (RHS) to Σ. Finally, the `propagation`: LHS $\implies$ RHS rule, given a constraints store Σ, adds the constraints in the body (RHS) to Σ *exactly once* for the constraints in the head (LHS).

For this thesis, we have defined seven rules so that the solved constraint system can inform the lifting of raw pointers to arrays, and these rules generalize to n-dimensional arrays.

- **Offset Collapse.** This rule states that, given the constraints $\mathit{Offset}(t_1, t_2, t_3)$, $\mathit{Offset}(t_3, t_4, t_5)$, remove them from Σ and add in the constraints $t_2 \simeq t_4, t_3 \simeq t_5, \mathit{Offset}(t_1, t_2, t_5)$. The **Offset Collapse** rule reflects the transitive closure of *Offset* and reduces the occurences of *Offset* in our constraint store by collapsing two *Offset*s together.

- **Array Reconstruction.** This rule states that, given the constraints $\mathit{Offset}(t_1, t_2, t_3)$, $\mathit{Deref}(t_3, t_4)$, remove them from Σ and add in the constraints $\mathit{Index}(t_1, t_2, t_4)$. The **Array Reconstruction** rule reflects that we can deduce that if the output t_3 (from when t_1 is offset by t_2) can be dereferenced to produce t_4, then it must be that t_1 is an array that can be indexed by t_2 to produce t_4.

- **Implicit Deref.** This rule states that, given the constraints $\mathit{Index}(t_1, t_2, t_3)$, $\mathit{Deref}(t_1, t_4)$, remove them from Σ and add in the constraint $t_3 \simeq t_4$. The **Implicit Deref** rule reflects that if dereferencing an array t_1 produces t_4, then it must be the case that indexing that array should also produce t_4.

- **Vec Reconstruction.** This rule states that, given the constraints $\mathit{ShouldIndex}(t_1, t_2, t_3)$, $\mathit{Malloc}(t_1, t_3)$, $\neg \mathit{Compat}(t_1, \mathit{Raw}(_))$, remove them from Σ and add in the constraint $\mathit{Vec}(t_1, t_2, t_3)$. The **Vec Reconstruction** rule reflects that, if t_1 *should*

[4] https://www.swi-prolog.org/pldoc/man?section=chr-syntaxandsemantics

be a collection that can be indexed by t_2, and t_1 has memory allocated to it corresponding to the size of t_3, and t_1 is *not compatible* with a raw pointer type, then we can deduce that it is a dynamic array (vector) indexed by t_2 and with an output of t_3.

- **Indexed Subsumption.** This rule states that, given the constraint $Vec(t_1, t_2, t_3)$, we can add in the constraint $Index(t_1, t_2, t_3)$. The **Indexed Subsumption** rule reflects that vectors in Rust implement the `Index` trait.

- **Dereference Reference.** This rule states that, given the constraints $Deref(t_1, t_2)$ and $Ref(t_1, t_3)$, remove them from Σ and add in the constraint $t_2 \simeq t_3$. The **Dereference Reference** rule reflects that, if t_1 is a reference to t_3, and dereferencing t_1 returns t_2, then it must be that t_2 and t_3 are equivalent.

- **Indexed Collapse.** This rule states that, given the constraints $Index(t_1, \_, t_3)$ and $ShouldIndex(t_1, \_, t_4)$, remove $ShouldIndex(t_1, \_, t_4)$ from Σ and add in the constraint $t_3 \simeq t_4$. The **Indexed Collapse** rule reflects that, if t_1 can be indexed into to retrieve an output t_3, but one *should* also be able to index into t_1 to retrieve some other output t_4, then it must be that t_3 and t_4 are equivalent.

We present the operational semantics of CHR informally. The CHR constraint solver iterates over every constraint in the constraint store Σ. Then, the CHR solver tries to apply rules in sequential order. If the current constraint matches the head of a rule, then the remaining constraints are searched for in Σ. If all the constraints in the head are matched, then the rule is executed. If the current constraint does not match the head of any rule, then it is suspended. This process ends when all constraints have been iterated through, and the final set of constraints is considered as the solved constraint system.

4.3.5 Solver as Oracle

We use the solved system of constraints as an oracle to answer queries pertaining to a particular labelled AST node in the program. The rules for querying the solved system of constraints are provided in Fig. 4.5. We provide two ways to query the oracle. Firstly,

$$\Sigma \vdash t^c$$

$$mut(\alpha, \Sigma) = \begin{cases} Imm & \Sigma \nvdash Mut(\alpha) \\ Mut & \Sigma \vdash Mut(\alpha) \end{cases}$$

$$\textsc{s-BaseType}\ \frac{\Sigma \vdash \alpha \simeq \tau \qquad c = mut(\alpha, \Sigma)}{\Sigma \vdash_\alpha \tau, c, \{\}, \Sigma}$$

$$\textsc{s-Default}\ \frac{\Sigma \nvdash \alpha \simeq \tau \qquad \Sigma \vdash Compat(\alpha, \tau) \qquad c = mut(\alpha, \Sigma)}{\Sigma \vdash_\alpha \tau, c, \{\}, \Sigma}$$

$$\textsc{s-Indexed}\ \frac{\Sigma \vdash Index(\alpha, \alpha_{ind}, \alpha_o) \qquad \Sigma \vdash_{\alpha_o} \tau_o, Imm, G_o, \Sigma_o \qquad \Sigma_o \nvdash Compat(\alpha, \_) \qquad fresh\ \mathtt{T} \\ G' = \{\mathtt{T} : \mathtt{Index}\langle \mathsf{usize}, \mathtt{Output} = \tau_o \rangle\} \cup G_o \qquad \Sigma' = \{Compat(\alpha, \mathtt{T})\} \cup \Sigma_o \qquad c = mut(\alpha, \Sigma')}{\Sigma \vdash_\alpha \mathtt{T}, c, G', \Sigma'}$$

$$\textsc{s-IndexedMut}\ \frac{\Sigma \vdash Index(\alpha, \alpha_{ind}, \alpha_o) \qquad \Sigma \vdash_{\alpha_o} \tau_o, Mut, G_o, \Sigma_o \qquad \Sigma_o \nvdash Compat(\alpha, \_) \qquad fresh\ \mathtt{T} \\ G' = \{\mathtt{T} : \mathtt{IndexMut}\langle \mathsf{usize}, \mathtt{Output} = \tau_o \rangle\} \cup G_o \qquad \Sigma' = \{Compat(\alpha, \mathtt{T})\} \cup \Sigma_o \qquad c = mut(\alpha, \Sigma')}{\Sigma \vdash_\alpha \mathtt{T}, c, G', \Sigma'}$$

Figure 4.5: Rules for Querying the Solver as an Oracle

reads as "the constraint t^c can be deduced from the given set of constraints Σ". Secondly,

$$\Sigma \vdash_\alpha \tau, c, G, \Sigma'$$

reads as "given a set of constraints Σ and a label α, the solver deduces that α maps to a type τ with capability c, and in the solving process it generates a set of traits G and updated set of constraints Σ'. We will use the notation $\Sigma \vdash_\alpha \tau, c$ when G is empty, and $\Sigma = \Sigma'$.

4.3.6 Collecting the Constraints

Now, we discuss how we collect the constraints from our program to construct our system of constraints. Recall our motivating example:

```
1  pub unsafe extern "C" fn reset(mut arr: *mut libc::c_int,
2  mut size: libc::c_int) {
3    let mut i: libc::c_int = 0 as libc::c_int;
4    while i < size {
5      *arr.offset(i as isize) = 0 as libc::c_int;
6      i += 1
7    };
8  }
```

Listing 4.6: Motivating Example: Resetting Arrays

We constructed the AST of this program, and gave every node in the AST a unique label:

```
pub unsafe extern "C" fn reset(mut arr/*1*/: *mut libc::c_int,
mut size/*2*/: libc::c_int) {
  let mut i/*3*/: libc::c_int = (0/*4*/ as libc::c_int)/*5*/;/*6*/
  while (i/*7*/ < size/*8*/)/*9*/ {
    (*(arr/*10*/.offset((i/*11*/ as isize)/*12*/))/*13*/)/*14*/ =
    (0/*15*/ as libc::c_int)/*16*/;/*17*/
    i/*18*/ += 1/*19*/;   /*20*/
  }; /*21*/
}
```

Listing 4.7: `reset` labelled

We present a selected set of constraint collection rules in Fig. 4.6. While we have omitted some of the rules for brevity, the entire Zoo of constraint collection rules, separated into general and array-related constraint collection rules, can be found at Appendix A.

1. Firstly, `c-FUN` constructs our label map M by iterating through every argument in the function declaration, giving each parameter a unique label, and then mapping each parameter to its unique label. This gives us $M = \{\texttt{arr} : \alpha_1, \texttt{size} : \alpha_2, i : \alpha_3\}$.

2. Secondly, `c-SEQ` iterates through each statement in the program, collects the constraints in each statement, and returns the set of collected constraints. This is essentially the "workhorse" that propagates the collection of rules. Different expressions are then handled by pattern-matching on what type of expressions they are: for instance, in our example, `while i < size` would invoke the constraint collection rule for `while` statements, which in turn invokes the constraint collection rule for binary operations, which returns that the label of `i` is equivalent to the label of `size`.

3. Next, the `c-ASSIGNMENT` rule targets statements of the form $\texttt{e}_1 := \texttt{e}_2$. We not only collect the constraints corresponding to the labels of $\texttt{e}_1$ and $\texttt{e}_2$, but also observe that, because $\texttt{e}_1$ is being assigned to, it must be *mutable*. Hence, we also add the constraint that the label of $\texttt{e}_1$ is *Mut*.

C-FUN

$$\frac{\alpha = \mathit{label}(\mathtt{e}) \qquad \alpha_i = \mathit{label}(\mathtt{v_i}), i \in 1..n \qquad M = \bigcup_{i=1}^{n}\{v_i : \alpha_i\} \qquad C = \bigcup_{i=1}^{n}\{\mathit{Compat}(\alpha_i, \tau_i)\}}{\Sigma(\mathtt{fn\ fname(v_1 : \tau_1, .., v_n : \tau_n) \to \tau_{out}\{e\}}, []) \triangleq \Sigma(e, M) \cup C}$$

C-SEQ

$$\frac{\alpha_2 = \mathit{label}(\mathtt{e_2}) \qquad \alpha = \mathit{label}(\mathtt{e_1; e_2})}{\Sigma(\mathtt{e_1; e_2}, M) \triangleq \{\alpha \simeq \alpha_2\} \cup \Sigma(e_1, M) \cup \Sigma(e_2, M)}$$

C-ASSIGNMENT

$$\frac{\alpha_1 = \mathit{label}(\mathtt{e_1}) \qquad \alpha_2 = \mathit{label}(\mathtt{e_2})}{\Sigma(\mathtt{e_1 := e_2}, M) \triangleq \{\alpha_1 \simeq \alpha_2, \mathit{Mut}(\alpha_1)\} \cup \Sigma(e_1, M) \cup \Sigma(e_2, M)}$$

C-INDEXED

$$\frac{\alpha = \mathit{label}(\mathtt{e_1.offset(e_2)}) \qquad \alpha_1 = \mathit{label}(\mathtt{e_1}) \qquad \alpha_2 = \mathit{label}(\mathtt{e_2}) \qquad t_1^c := \mathit{Offset}(\alpha_1, \alpha_2, \alpha) \qquad t_2^c := \mathit{Compat}(\alpha_2, \mathtt{usize})}{\Sigma(\mathtt{e_1.offset(e_2)}, M) \triangleq \{t_1^c, t_2^c\} \cup \Sigma(e_1, M) \cup \Sigma(e_2, M)}$$

C-DEREF

$$\frac{\alpha = \mathit{label}(\mathtt{*e}) \qquad \alpha' = \mathit{label}(\mathtt{e})}{\Sigma(\mathtt{*e}, M) \triangleq \{\mathit{Deref}(\alpha', \alpha)\} \cup \Sigma(e, M)}$$

C-MALLOC

$$\frac{\alpha = \mathit{label}(\mathtt{malloc(N \cdot sizeof(\tau))}) \qquad \Sigma' = \Sigma(\mathtt{e}, M)}{\Sigma(\mathtt{malloc(N \cdot sizeof(\tau))}, M) \triangleq \{\mathit{Malloc}(\alpha, \tau)\} \cup \{\mathit{Compat}(\alpha, \tau)\} \cup \Sigma'}$$

C-VEC

$$\frac{\alpha = \mathit{label}(\mathtt{e}) \qquad \Sigma' = \Sigma(\mathtt{e}, M) \cup \Sigma(\tau_{out}, M) \qquad \alpha' = M(e)}{\Sigma(\mathtt{IndexMutWrapper(e, \tau_{ind}, \tau_{out})}, M) \triangleq \{\mathit{ShouldIndex}(\alpha, \tau_{ind}, \tau_{out}), \alpha \simeq \alpha'\} \cup \Sigma'}$$

Figure 4.6: A Selection of Constraint Collection Rules

4. We also want to collect constraint information for pointers that can be lifted to arrays. Hence, when considering statements of the form `*arr.offset(i)`, which is how `c2rust` syntactically translates arrays in C to unsafe Rust, we first invoke `c-INDEXED` to capture that `arr.offset(i)` indicates that `arr`, when offset by `i`, returns `arr.offset(i)`. Then, we invoke `c-DEREF` to collect the constraint that the expression `arr.offset(i)` dereferences to `*arr.offset(i)`. These two constraints, $\mathit{Offset}(\alpha_{10}, \alpha_{12}, \alpha_{13}), \mathit{Deref}(\alpha_{13}, \alpha_{14})$, will be targeted by the *Array Reconstruction* rule in Fig. 4.4 to produce $\mathit{Index}(t_1, t_2, t_4)$, which indicates that `*arr.offset(i)` can be refactored as indexing an array in safe Rust, i.e., `arr[i]`.

The above constraint collection rules provides enough information to refactor the indexing of arrays, but we want to also handle the allocation of arrays.

Consider the following extension of our motivating example, where we have some parent function that allocates arrays and calls `reset` on them.

```
1   unsafe fn main_0() -> libc::c_int {
2     let mut N: libc::c_int = 3 as libc::c_int;
3     /*  static array  allocation */
4     let mut static_arr: [libc::c_int; 3] =
5     [1 as libc::c_int, 2 as libc::c_int, 3 as libc::c_int];
6     reset(static_arr.as_mut_ptr(), N);
7     /*  dynamic array  allocation */
8     let mut dyn_arr /*1*/: *mut libc::c_int =
9     malloc((N/*2*/ as libc::c_ulong)/*3*/.wrapping_mul(
10    (::std::mem::size_of::<libc::c_int>()/*4*/
11    as libc::c_ulong)/*5*/)/*6*/)/*7*/ as *mut libc::c_int/*8*/;
12    reset(IndexMutWrapper(dyn_arr, usize, libc::c_int)/*9*/, N);
13    free(dyn_arr as *mut libc::c_void);
14    return 0 as libc::c_int;
15  }
16
```

Listing 4.8: Dynamic Array Allocation

In the example above, we have only labelled the section of the code pertaining to dynamic array allocation for brevity. In line 12, we have

```
reset(IndexMutWrapper(dyn_arr, usize, libc::c_int)/*9*/, N);
```

This is because the example is a *caller* of `reset`; hence, after performing a rewrite pass on `reset`, we update the call sites of `reset` with our `update_callsite` pass (algorithm 4.2.1). As we will describe later in Sec. 4.4.2, we will be able to infer when an array is dynamic or static. If it is dynamic, then in the call site, we wrap the array in a `wrapper` (in this case, `IndexMutWrapper`) to pass along the information to subsequent passes that the array should be treated as such.

If the array is statically declared, i.e., `int static_arr[] = {1,2,3}` in C, then `c2rust` performs a direct syntactic translation into the idiomatic static declaration of an array in Rust. Thus, we only need to handle the case where an array is dynamically declared, i.e., it is allocated some memory via `malloc`. This is handled by the `c-MALLOC` and `c-VEC` rules:

```
unsafe fn main_0() -> libc::c_int {
  ...
  /*  dynamic array  allocation */
  let mut dyn_arr /*1*/: *mut libc::c_int =
  malloc((N/*2*/ as libc::c_ulong)/*3*/.wrapping_mul(
  (::std::mem::size_of::<libc::c_int>()/*4*/
  as libc::c_ulong)/*5*/)/*6*/)/*7*/ as *mut libc::c_int/*8*/;
  reset(IndexMutWrapper(dyn_arr, usize, libc::c_int)/*9*/, N);
  ...
}

```

Listing 4.9: Dynamic Array Allocation

```
fn main_0() -> libc::c_int {
  ...
  /*  dynamic array  allocation */
  let mut dyn_arr = vec![Default::default(); N.try_into().unwrap()];
  reset(&mut dyn_arr, N);
  ...
}

```

Listing 4.10: Dynamic Array Allocation, refactored into safe Rust

1. The c-VEC rule targets occurences of the form `IndexMutWrapper`$(\mathtt{e}, \tau_{\mathtt{ind}}, \tau_{\mathtt{out}})$. Since we define a specific wrapper so that subsequent passes can pick up that an expression has a type that implements the `IndexedMut` trait, we add the constraint that the expression *e should* be a type that implements the `IndexedMut` trait. That is, in subsequent passes, it *should* be refactored into a dynamic array. So, we add to Σ the constraint that $\mathit{ShouldIndex}(\alpha_1, \mathtt{usize}, \mathtt{c\_int})$.

2. Then, the c-MALLOC rule picks up that line 9-11 is a `malloc` call of the form $\mathtt{malloc}(\mathtt{N} \cdot \mathtt{sizeof}(\tau))$, so it adds to Σ the constraint that $\mathit{Malloc}(\alpha_7, \mathtt{c\_int})$ and $\mathit{Compat}(\alpha_7, \mathtt{c\_int})$.

3. Finally, the CHR solver will first observe that $\alpha_1 \simeq \alpha_7$ by the c-LET and c-CAST rule. Then, it will apply the *Vec Reconstruction* rule on $\mathit{ShouldIndex}(\alpha_1, \mathtt{usize}, \mathtt{c\_int})$, $\mathit{Malloc}(\alpha_1, \mathtt{c\_int})$ and the fact that $\neg\mathit{Compat}(\alpha_1, \mathit{Raw}(\_))$ to obtain the constraint that $\mathit{Vec}(\alpha_1, \mathtt{usize}, \mathtt{c\_int})$.

With the solved system of constraints informing us that $\mathit{Vec}(\alpha_1, \mathtt{usize}, \mathtt{c\_int})$, this indicates that we are able to refactor Listing 4.9 into Listing 4.10, as we shall see in the next section.

4.4 Rewrite Rules

In this section, we will discuss the rewrite rules for `arr_pass` in algorithm 4.2.3 as well as for `update_callsite` in algorithm 4.2.1. To reiterate, our motivation is to target unsafety in code that is conservatively labelled **`unsafe`** by `c2rust`, as well as to perform an idiomatic refactoring of a subset of raw pointers that were intended to be used as arrays.

Again, consider our motivating example. We had previously labelled the AST of the code, collected the constraints, and solved the system of constraints.

```
1  pub unsafe extern "C" fn reset(mut arr/*1*/: *mut libc::c_int,
2  mut size/*2*/: libc::c_int) {
3    let mut i/*3*/: libc::c_int = (0/*4*/ as libc::c_int)/*5*/;/*6*/
4    while (i/*7*/ < size/*8*/)/*9*/ {
5      (*(arr/*10*/.offset((i/*11*/ as isize)/*12*/))/*13*/)/*14*/ =
6      (0/*15*/ as libc::c_int)/*16*/;/*17*/
7      i/*18*/ += 1/*19*/;   /*20*/
8    }; /*21*/
9  }
```

Listing 4.11: `reset` labelled

The solved system of constraints gives us the following solution set: $\Sigma' = \{$

$\{\alpha_4 \simeq \alpha_5, \alpha_3 \simeq \alpha_5, \alpha_7 \simeq \alpha_8, \alpha_7 \simeq \alpha_3, \alpha_3 \simeq \alpha_8,$

$\alpha_8 \simeq \alpha_2, \alpha_3 \simeq \alpha_{11}, \alpha_3 \simeq \alpha_{18}, \alpha_{18} \simeq \alpha_{19}\},$

$\{\alpha_{14} \simeq \alpha_{16}, \alpha_{15} \simeq \alpha_{16}\},$

$\{\alpha_1 \simeq \alpha_{10}\},$

$Compat(\alpha_2, \texttt{c\_int}),$

$Compat(\alpha_3, \texttt{c\_int}),$

$Compat(\alpha_5, \texttt{c\_int}),$

$Compat(\alpha_{11}, \texttt{isize}),$

$Compat(\alpha_{15}, \texttt{c\_int})$

$Mut(\alpha_{14}),$

$Mut(\alpha_{18}),$

$Index(\alpha_{10}, \alpha_{12}, \alpha_{14})\}.$

$$\frac{\alpha_i = \mathit{label}(\mathtt{v_i}) \quad \Sigma \vdash_{\alpha_i} \tau_i', c_i', G_i, \Sigma_i \text{ for } i \in 1..n \quad \Sigma' = \cup_{i=1}^{n} \Sigma_i \quad \alpha = \mathit{label}(\mathtt{e}) \quad \Sigma' \vdash_{\alpha} \tau', c', G', \Sigma'' \quad G = \cup_{i=1}^{n} G_i \cup G' \quad \Sigma''; \texttt{unsafe\{e\}} \rightarrow \mathtt{e'}}{\begin{array}{c}\Sigma; \texttt{unsafe fn fname}(c_1\ \mathtt{v_1} : \tau_1, .., c_n\ \mathtt{v_n} : \tau_n) \rightarrow c\ \tau\ \{\mathtt{e}\} \rightarrow \\ \texttt{fn fname}\langle \mathtt{G} \rangle (c_1'\ \mathtt{v_1} : \tau_1', .., c_n'\ \mathtt{v_n} : \tau_n') \rightarrow c'\ \tau'\ \{\mathtt{e'}\}\end{array}}\ \textsc{u-Fun}$$

$$\frac{\Sigma; \texttt{unsafe\{e}_1\texttt{\}} \rightarrow \mathtt{e}_1' \qquad \Sigma; \texttt{unsafe\{e}_2\texttt{\}} \rightarrow \mathtt{e}_2'}{\Sigma; \texttt{unsafe\{e}_1\texttt{;e}_2\texttt{\}} \rightarrow \mathtt{e}_1'\texttt{;}\mathtt{e}_2'}\ \textsc{u-Seq}$$

$$\frac{\Sigma \vdash \mathit{Index}(\mathit{label}(\mathtt{e}_1), \_, \_) \qquad \Sigma; \texttt{unsafe\{e}_1\texttt{\}} \rightarrow \mathtt{e}_1' \qquad \Sigma; \texttt{unsafe\{e}_2\texttt{\}} \rightarrow \mathtt{e}_2'}{\Sigma; \texttt{unsafe\{*e}_1.\mathit{offset}(\mathtt{e}_2)\texttt{\}} \rightarrow \mathtt{e}_1'[\mathtt{e}_2']}\ \textsc{u-ArrayIndex}$$

Figure 4.7: A Selection of Rewrite Rules for `arr_pass`

4.4.1 Rewrite Rules for `arr_pass`

We present a selection of rewrite rules in Fig. 4.7 to describe the rewrite of our motivating example, `reset`. The general rewrite rule for instructions is as follows:

$$\mathcal{P}; \Sigma; \ \mathtt{e} \rightarrow \mathtt{e'}$$

and it reads as "given the global environment of the program $\mathcal{P}$, (the AST of) expression `e` is transformed into (the AST corresponding to) program `e'`, in the presence of type constraints Σ ". For brevity, however, we will not explicitly state $\mathcal{P}$ in the rules, but will assume that it always exists. The entire Zoo of rewrite rules, separated into general and array-related transformation rules, for `arr_pass` can be found in Appendix B.

Firstly, we begin by applying the `u-FUN` rule. The `u-FUN` rule targets an **`unsafe`** function declaration and performs two actions:

1. It pushes **`unsafe`** function declaration inwards, so that instead of the function being labelled **`unsafe`**, it is the body of the function that is labelled **`unsafe`**.

2. It updates the function signature using the information from the collected constraints. Besides updating the parameters and their types in the function signature, if the solved system of constraints can infer the presence of an array, then we include the trait `IndexMut` or `Index` into the trait bound of the function as such: $\mathtt{T0} : \mathtt{IndexMut} < \tau_1, \mathtt{Output} = \tau_2 >$, where `T0` is fresh. The function parameters which use these traits are then updated to have the type `T0`.

This gives us the following program code:

```
1  fn reset<T0: IndexMut<usize, Output = c_int>>(mut arr: &mut T0,
2  mut size: c_int) {
3    unsafe {
4      let mut i: libc::c_int = 0 as libc::c_int;
5      while i < size {
6        *arr.offset(i as isize) = 0 as libc::c_int;
7        i += 1
8      };
9    }
10 }
```

Listing 4.12: `reset` after `u-FUN`

Observe that the function signature has been rewritten to reflect that `arr` is a mutable reference to `T0`, which in turn implements the trait `IndexMut<usize, Output = c_int>`. Furthermore, the function is *no longer conservatively marked as* `unsafe` – instead, the unsafety has been bubbled inwards to the body of the function!

Now we know, from Rust's provenance of unsafety (Sec. 2.1.3), that the remaining statements are safe with the exception of line 6. By applying the other rules in Appendix B that traverse the different types of expressions in conjunction with `u-SEQ`, we can further remove every `unsafe` block with the exception of the raw pointer dereference in line 6:

```
unsafe { *arr.offset(i as isize) } = 0 as libc::c_int;
```

Here, `u-ARRAYINDEX` comes into play – given `unsafe{*e1.offset(e2)}`, if our solved system of constraints indicate that e_1 is in fact an array, i.e., $Index(label(e_1), \_, \_)$, and e_1 and e_2 are both expressions that can be rewritten into safe expressions e_1', e_2', then we can rewrite the dereference operation and `offset` method call on `arr` as simply an array indexing (which is safe), giving us $e_1'[e_2']$. Since $label(arr) = \alpha_{10}$, and Σ' indeed indicates that $Index(\alpha_{10}, \alpha_{12}, \alpha_{14})$, we can complete the rewrite of our motivating example:

```
1  fn reset<T0: IndexMut<usize, Output = c_int>>(mut arr: &mut T0,
2  mut size: c_int) {
3    let mut i: libc::c_int = 0 as libc::c_int;
```

```
    while i < size {
      arr[i as usize] = 0 as libc::c_int;
      i += 1
    };
  }
```

Listing 4.13: `reset` after `arr_pass`

Observe that the final code has no **`unsafe`** code blocks – we have successfully removed the conservative **`unsafe`** annotations! The key to this was first to observe that the function was *conservatively* labelled **`unsafe`**, and so if the function body was in fact safe, we could rewrite the entire function as safe. Secondly, while it is **`unsafe`** to dereference a pointer, if we can deduce that the raw pointer is *intended* to be used as an array, then the raw pointer dereference *should not* be **`unsafe`**, since the semantics of the dereference (when used in conjunction with an `offset`) is to index the array, which is safe.

4.4.2 Updating Call Sites

Having rewritten `reset`, we now want to update all of the functions that call it. Let's take a look at our parent function `main` from before, when we collected constraints for dynamic array allocation:

```
    unsafe fn main_0() -> libc::c_int {
      let mut N: libc::c_int = 3 as libc::c_int;
      /*  static array  allocation */
      let mut static_arr: [libc::c_int; 3] =
      [1 as libc::c_int, 2 as libc::c_int, 3 as libc::c_int];
      reset(static_arr.as_mut_ptr(), N);
      /*  dynamic array  allocation */
      let mut dyn_arr /*1*/: *mut libc::c_int =
      malloc((N/*2*/ as libc::c_ulong)/*3*/.wrapping_mul(
      (::std::mem::size_of::<libc::c_int>()/*4*/
      as libc::c_ulong)/*5*/)/*6*/)/*7*/ as *mut libc::c_int/*8*/;
      reset(dyn_arr /*9*/, N);
      free(dyn_arr as *mut libc::c_void);
      return 0 as libc::c_int;
    }

```

$$\frac{\texttt{type}(e) = \mathit{Raw}(\tau) \qquad \texttt{type}(\rho) = \texttt{T} : \texttt{IndexMut}\langle \tau_1, \texttt{Output} = \tau \rangle}{\Sigma;\ (\texttt{e}, \rho) \rightsquigarrow \texttt{IndexMutWrapper(e, usize}, \tau)}\ \textsc{u-UpdateVec}$$

$$\frac{\texttt{type}(e) = \mathit{Mut\ Array}(\tau) \qquad \texttt{type}(\rho) = \texttt{T} : \texttt{IndexMut}\langle \tau_1, \texttt{Output} = \tau \rangle}{\Sigma;\ (\texttt{e.as\_mut\_ptr()}, \rho) \rightsquigarrow \texttt{Mut Ref e}}\ \textsc{u-UpdateStaticArrMut}$$

$$\frac{\texttt{type}(e) = \mathit{Array}(\tau) \qquad \texttt{type}(\rho) = \texttt{T} : \texttt{Index}\langle \tau_1, \texttt{Output} = \tau \rangle}{\Sigma;\ (\texttt{e.as\_mut\_ptr()}, \rho) \rightsquigarrow \texttt{Ref e}}\ \textsc{u-UpdateStaticArrImm}$$

$$\frac{\texttt{fname} = \texttt{callname} \qquad \texttt{fn callname}(\rho_1, \ldots, \rho_n) \qquad \Sigma;\ (\texttt{e}_i, \rho_i) \rightsquigarrow \texttt{e}'_i}{\Sigma;\ \texttt{fname}(\texttt{e}_1, .., \texttt{e}_n) \underset{\text{callname}}{\rightarrow} \texttt{fname}(\texttt{e}'_1, .., \texttt{e}'_n)}\ \textsc{u-UpdateCallsite}$$

Figure 4.8: Rewrite Rules for Updating Call Sites

The rewrite rules for updating call sites are presented in Fig. 4.8. Before we begin updating the call sites, we require the aid of an oracle that can provide us with the `type` of a variable in the program environment. A query to this oracle is represented as `type(e)`, and returns a Rust type τ. In practice, this is implemented as the local type context of the caller function that we are about to update.

In `main`, there are two calls to `reset` that we want to update. The `u-UPDATECALLSITE` rule ($\underset{callname}{\rightarrow}$) is parameterized by the name of the callee, in this case `reset`. Then, when it fires, it ensures that the call site's function name matches the callee's name. If the names match, then it `zip`s the parameters $\texttt{e}_1, \ldots, \texttt{e}_n$ of the call site function and the parameters $\rho_1, \ldots, \rho_n$ of the callee function, and iterates over the pair $(\texttt{e}_i, \texttt{p}_i)$, applying one of `u-UPDATEVEC`, `u-UPDATESTATICARRMUT`, or `u-UPDATESTATICARRIMM`. This is represented by the $\rightsquigarrow$ symbol in the premise of `u-UPDATECALLSITE` ($\Sigma;\ (\texttt{e}_i, \rho_i) \rightsquigarrow \texttt{e}'_i$).

The `u-UPDATECALLSITE` rule first applies to the application of `reset` on our statically declared array (line 6):

```
reset(static_arr.as_mut_ptr(), N);
```

To begin, we observe that a mutable pointer to `static_arr` is being passed

to `reset`, in keeping with `reset`'s function signature. This is because `c2rust` explicitly casts idiomatic uses of arrays in C to a decayed raw pointer in unsafe Rust. As we keep track of the local type context, our oracle returns that `type(e) = Mut Array(c_int)`. Since the type signature for the first parameter, `arr`, in `reset` is `&mut T0` where `T0` is bound as `T0: IndexMut<usize, Output = c_int>`, the `u-UPDATESTATICARRMUT` rule fires and returns `Mut Ref e` to `u-UPDATECALLSITE`, which performs the rewrite at the call site to update the first parameter of `reset` in line 6 from `static_arr.as_mut_ptr()` to `&mut static_arr`. This gives us:

```
reset(&mut static_arr, N);
```

Then, the `u-UPDATECALLSITE` rule applies to the application of `reset` on our dynamically declared array (line 12):

```
reset(dyn_arr, N);
```

Since `dyn_arr` is initialized as a mutable pointer via memory allocation on line 9, it is being passed directly to the `reset` call. Our oracle returns that `type(e) = Raw(c_int)`, and since the type signature for the first parameter, `arr`, in `reset` is `&mut T0` where `T0` is bound as `T0: IndexMut<usize, Output = c_int>`, the `u-UPDATEVEC` rule fires and returns `IndexMutWrapper(dyn_arr, usize, c_int)` to `u-UPDATECALLSITE`.

`IndexMutWrapper(dyn_arr, usize, c_int)` is a wrapper which encapsulates information about `dyn_arr`, so that on the subsequent `arr_pass` on `main`, the constraint collector can gather that `dyn_arr` *should be* implementing the `IndexMut` trait (see: item 1 describing the `c-VEC` constraint collection rule). Then, `u-UPDATECALLSITE` performs the rewrite at the call site to update the first parameter of `reset` in line 12 from `dyn_arr` to `IndexMutWrapper(dyn_arr, usize, c_int)`, giving us:

```
reset(IndexMutWrapper(dyn_arr, usize, c_int), N);
```

4.4.3 Rewriting After Call Site Update

To complete the code transformation, it remains to rewrite `main` after updating the call sites:

```
1     unsafe fn main_0() -> libc::c_int {
2       let mut N: libc::c_int = 3 as libc::c_int;
3       /*  static array  allocation */
4       let mut static_arr: [libc::c_int; 3] =
5       [1 as libc::c_int, 2 as libc::c_int, 3 as libc::c_int];
6       reset(&mut static_arr, N);
7       /*  dynamic array  allocation */
8       let mut dyn_arr: *mut libc::c_int =
9       malloc((N as libc::c_ulong).wrapping_mul(
10      (::std::mem::size_of::<libc::c_int>()
11      as libc::c_ulong))) as *mut libc::c_int;
12      reset(IndexMutWrapper(dyn_arr, usize, cint), N);
13      free(dyn_arr as *mut libc::c_void);
14      return 0 as libc::c_int;
15    }
16
```

Listing 4.14: Dynamic Array Allocation, Call Sites Updated

We will focus on our selected rewrite rules in Fig. 4.9 that specifically target dynamically declared arrays. Again, the complete set of rules can be found in Appendix B.

As we did before in Sec. 4.4.1, we begin by applying the `u-FUN` rule to push the **`unsafe`** function declaration inwards to the body of the function. Then, via Rust's provenance of unsafety (Sec. 2.1.3), we can traverse the statements with `u-SEQ` and remove unsafety from expressions that are conservatively marked as **`unsafe`**. We focus on lines 8-13 in particular, since they perform the allocation, use, and deallocation of the dynamically allocated array.

Firstly, `dyn_arr` is allocated on the heap with the C `malloc` function (lines 8-11). In Rust, the idiomatic way to dynamically allocate an array is to use the `vec!()` macro. Thus, our aim here is to rewrite the lengthy `malloc` call that was automatically translated by `c2rust` into a use of the `vec!()` macro. This is handled by the `u-VECMALLOC` rule in Fig. 4.9, which targets expressions

$$\frac{\alpha = \mathit{label}(\texttt{IndexMutWrapper(e, }\tau_{\texttt{ind}}\texttt{, }\tau_{\texttt{out}}\texttt{)}) \qquad \Sigma \vdash \mathit{Index}(\alpha, \_, \tau_{out})}{\Sigma;\ \texttt{IndexMutWrapper(e, }\tau_{\texttt{ind}}\texttt{, }\tau_{\texttt{out}}\texttt{)} \longrightarrow \texttt{Mut Ref e}}\ \textsc{u-VecWrapper}$$

$$\frac{\alpha = \mathit{label}(\texttt{malloc(N} \cdot \texttt{sizeof}(\tau)\texttt{)}) \quad \Sigma \vdash \mathit{Vec}(\alpha, \alpha_{ind}, \alpha_{out}) \qquad \Sigma \vdash \mathit{Malloc}(\alpha, \_) \qquad \Sigma \vdash \alpha_{out} \simeq \tau_{out} \qquad \tau = \tau_{out}}{\Sigma;\ \texttt{unsafe \{malloc(N} \cdot \texttt{sizeof}(\tau)\texttt{)\}} \longrightarrow \texttt{vec![}\tau_{\texttt{out}}\texttt{;N]}}\ \textsc{u-VecMalloc}$$

$$\frac{\alpha = \mathit{label}(\texttt{e}) \qquad \Sigma \vdash \mathit{Vec}(\alpha, \_, \_) \qquad \Sigma \vdash \mathit{Malloc}(\alpha, \_)}{\Sigma;\ \texttt{unsafe \{free(e as *mut } \tau\texttt{)\}} \longrightarrow ()}\ \textsc{u-VecFree}$$

Figure 4.9: Rewrite Rules for Dynamic Arrays

of the form `malloc(N * sizeof(e))`. If our solved system of constraints, given the label α of `malloc(N * sizeof(e))`, indicate that $\mathit{Vec}(\alpha, \alpha_{\texttt{ind}}, \alpha_{\texttt{out}}) \in \Sigma$ and $\mathit{Malloc}(\alpha, \tau) \in \Sigma$, and that the type of α_{out} matches τ, then we know that we can rewrite `malloc(N * sizeof(e))` into `vec![`$\tau_{\texttt{out}}$`;N]`. This is because our constraint system informs us that `dyn_arr` is being used as a dynamic array, and has also been allocated memory. Thus, we know that it is semantics-preserving to use the Rust macro `vec![]` to dynamically allocate memory for `dyn_arr`, and we can rewrite

```
let mut dyn_arr: *mut libc::c_int =
malloc((N as libc::c_ulong).wrapping_mul(
(::std::mem::size_of::<libc::c_int>()
as libc::c_ulong))) as *mut libc::c_int;
```

into

```
let mut dyn_arr: *mut libc::c_int = vec![Default::default(); N.try_into().unwrap()];
```

Then, the u-LET rule transforms the type signature in the declaration of `dyn_arr` to give us

```
let mut dyn_arr: Vec<c_int> = vec![Default::default(); N.try_into().unwrap()];
```

Next, we want to get rid of the wrapper in line 12 that was constructed in the call site update pass to propagate information that `dyn_arr` implements the `IndexMut` trait to the constraint collector. This is handled by u-VECWRAPPER, which simply checks that we have successfully collected this information, and then removes the wrapper, returning just `&mut dyn_arr`. Observe that, if in fact `dyn_arr`

were determined to *not be* compatible with an `Index` trait, then it must be a raw pointer by the premise of `u-UPDATEVEC`. In this case, we do not remove the wrapper. However, this is still valid, compilable Rust code: we just have to ensure that the implementation of `IndexMutWrapper` returns an array when given a pointer. This scenario is fairly common, and highlights the gradual, iterative "hill-climbing" nature of CHRusty wherein intermediate rewrites should preserve compilability – suppose a function `f` takes in an array, and another function `g` calls `f` with a variable `x` that may have found use as a pointer *after* `f` is called.

Finally, consider line 13:

```
free(dyn_arr as *mut libc::c_void);
```

Since Rust automatically frees memory for us, the deallocation of the raw pointer using C's `free` is unnecessary, and the `u-VECFREE` rule handles it by removing the line of code, ensuring that the expression has been determined by the constraints to be a dynamically allocated array.

With all these changes, the code transformation for `main` is complete, and we can now present the fully-transformed, safer, and more idiomatic, `reset` and `main` together:

```
1  fn reset<T0: IndexMut<usize, Output = c_int>>(arr: &mut T0, size: c_int) {
2    let mut i: libc::c_int = 0 as c_int;
3    while i < size {
4      arr[i as usize] = 0 as c_int;
5      i += 1
6    }
7  }
8  fn main_0() -> libc::c_int {
9    let mut N: libc::c_int = 3 as c_int;
10   let mut static_arr: [libc::c_int; 3] = [1 as c_int, 2 as c_int, 3 as c_int];
11   reset(&mut static_arr, N);
12   let mut dyn_arr = vec![Default::default(); N.try_into().unwrap()];
13   reset(&mut dyn_arr, N);
14   return 0 as c_int;
15 }
```

Listing 4.15: Transformed Code

4.5 Discussion

This concludes the program rephrasing and refactoring algorithm as part of the pipeline from C code to safer and more idiomatic Rust code.

In this chapter, we have presented a generic framework for automated translation of C code to *safer* Rust (Sec. 4.1, Sec. 4.2), working through `reset` and its caller `main` as a motivating example. In particular, we focused on the problem of lifting raw pointers to arrays: `c2rust`, during their syntactic translation of C to **`unsafe`** Rust, naively translates code that was intended to be used as arrays into raw pointers. By using a constraint collector, the solved system of constraints can indicate whether a raw pointer was actually intended to be used as an array (Sec. 4.3). Then, our rewrite system uses information from the solved constraints to perform these rewrites (Sec. 4.4). In the process, conservative **`unsafe`** blocks are removed, and the subset of raw pointers intended to be used as arrays are refactored to Rust idioms.

5 Implementation of CHRusty

In this chapter, we detail several aspects of CHRusty, which implements the array-refactoring instantiation of our framework, `arr_pass`. We implemented CHRusty as described in Ch. 4 in around 7,000 LOC of Rust, with the exception of the CHR system, which was written in Prolog. A snapshot of the implementation of CHRusty is available as an artifact[1] for this thesis, but on the whole is still undergoing development.

5.1 Working with Rust's AST and IR

Rust exposes several source code representations. Those that are easily available are the AST of the source code [KN22a], a high-level intermediate representation called HIR [KN22b], and the mid-level intermediate representation called MIR [KN22c]. There is also a typed high-level intermediate representation called THIR, but it is mostly for the purposes of constructing the MIR. For more information, one can refer to the official Rust compiler guide [KN22d].

5.1.1 HIR

The high-level intermediate representation (HIR) used by the Rust compiler is a compiler-friendly representation of the AST generated after parsing, macro expansion, and name resolution [KN22b].

Many aspects of HIR resemble Rust language syntax closely, except that some expressions have been desugared. For instance, **`for`**-loops are eliminated, and are instead represented using the **`loop`** construct.

[1] https://zenodo.org/record/6334872

5.1.2 MIR

Where HIR abides by the same structure as the original source code, the mid-level intermediate representation (MIR) is based on a control-flow graph, and more closely resembles LLVM IR. MIR is intended to reduce Rust down to a simple core language, removing most of the Rust syntax by converting them to a small set of primitives [Mat16].

5.1.3 Our IR of Choice: The `syn crate`

For our purposes, however, the above source code representations are not amenable to both the analysis pass *and* the refactoring. This is because our constraint collection and rewrite pass occurs at the level of the AST of the program. The exception to this is when we use the HIR to more easily collect the calling relationships for our call graph generation.

The syn[2] crate is a parsing library for parsing a stream of Rust tokens into a syntax tree of Rust source code. The two pertinent features that it provides are

1. A complete syntax tree for representing any valid Rust source code, and

2. Syntax tree traversal to transform the nodes[3].

These features enable us to collect constraints, as well as perform the refactoring, using a single representation. Recall our rewrite pass structure, which we present here again for easy reference:

Algorithm 5.1.1: Rewrite pass structure

```
1 Function rewrite_pass(curr_fn):
2     ast ← ast_of(curr_fn);
3     constraints ← collect_constraints(ast);
4     solved_constraints ← chr_solve_constraints(constraints);
5     transformed_ast ← rewrite(ast, solved_constraints);
6     surgery(curr_fn, transformed_ast);
```

In line 2, we use syn to generate our AST. We can then traverse the AST in line 3 to collect constraints, pass them to our CHR solver to solve them, and then perform the rewrites directly to the AST in line 5. Finally, we write

[2] https://crates.io/crates/syn
[3] https://docs.rs/syn/1.0.86/syn/fold/index.html

the transformed AST back to the file in line 6. But this describes the process for refactoring an individual function. What then, is the entry point for our algorithm to operate on?

5.2 Generating Callgraphs

We leverage the power of the Rust compiler to provide us with information about all the functions that we need to operate on. When we generate our call graph in the main algorithm (algorithm 4.2.1), we pass the algorithm either a `main.rs` or `lib.rs` file, depending on whether the program is a single-file program, or a large project spanning across multiple files.

This file is passed to `build_callgraph`, which runs the Rust compiler on it, with the provision that we perform some additional analysis relevant to generating the callgraph at the HIR/MIR linting phase, which runs just before the translation to LLVM IR. Since generating callgraphs using static analysis is undecidable (reachability analysis is undecidable), our callgraph is in fact an overapproximation of the actual program callgraph, though the precision of the callgraph (when it is an overapproximation) does not affect our analysis.

Because the `main.rs` or `lib.rs` file that we pass to this analysis contains all the necessary information regarding the functions that are used in the program, our callgraph analysis first operates on the level of the function representation in HIR before coverting it into a fully-qualified function name, which is an unambiguous function name that specifies the absolute path of the file where the function resides.

Thus, the final output of our callgraph analysis is a list of functions and their callers, ordered from the bottom of the calling hierarchy to the top (the `main` function) via a topological sort. We eliminate any self-referential calls by ignoring cycles. Since each such function has a fully-qualified name, when we pass the function to the rewrite pass function algorithm 5.1.1, it knows exactly where to retrieve the source code from.

5.3 CHR system

As discussed in Sec. 4.1.2, our constraint solver of choice is the Constraint Handling Rules (CHR) system, which is a declarative, rule-based constraint solver. The particular implementation of the CHR system that we use is the Prolog implementation of the K.U.Leuven CHR system [Frü94; SD04], implemented as a library in Prolog.[4]

5.3.1 Interfacing with CHR

The details of our constraints (Sec. 4.3.3), how we collect them (Appendix A), and the rules we apply to them (Sec. 4.3.4), have all been documented extensively previously. Here, we discuss the engineering behind how our implementation interfaces with the CHR system.

When we collect constraints, we begin by traversing the AST obtained from the `syn` crate (Sec. 5.1.3) and storing the constraints in our own internal representation. Then, to pass it to the CHR system, we translate the internal representation into the Prolog CHR syntax. CHRusty then spawns a child process to run the CHR system, pipes our collected constraint to it, retrieves the output from the solver, and parses it back into our internal representation. Finally, to represent "equality", or the $\simeq$ relation, we use a union-find data structure to allow for easy access to the equivalence classes of labels.

5.4 Surgery

The final step of our rewrite pass is to perform "surgery": with the absolute path of the function, we write the transformed AST back to the file (line 6 in algorithm 5.1.1) that the function belongs to.

Once every function has been re-written, and the files written to, we complete the "surgery" of the code by resolving all the necessary imports. For instance, if we had to bound a function that uses raw pointers as arrays during our refactoring with the `IndexMut` trait, then we have to import the `std::ops::IndexMut` trait from

[4] Available at `https://www.swi-prolog.org/man/chr.html`

the standard library. Furthermore, in the `u-VECMALLOC` rule (Appendix B), when we rewrite a use of `malloc(N · sizeof(`τ`))` as `vec![`τ_{out}`;N]` where $\tau_{out} = \tau$, as a convenient shorthand we leverage Rust's `Default` trait for giving a type a useful default value instead of coming up with a default value ourselves for every Rust type. The reason for this is, when we initialize an array, we don't really want to be bogged down by the choice of initial element, and selecting an arbitrary initial element might affect program behavior. Thus, we import the `std::default::Default` trait to use the `Default::default()` value for every primitive type.

6 Evaluation and Case Studies

In this chapter, we explore two case studies: 1. allocating a matrix, and 2. implementing Quicksort. We demonstrate how our framework transforms the initial C program into `unsafe` Rust code using `c2rust`, and then how CHRusty refactors the `unsafe` Rust code into safer and more idiomatic Rust code. Finally, we run a randomized testing suite to validate that the behavior of the refactored code matches that of the original code in C.[1]

Again, our implementation of CHRusty is available as an artifact[2], and the case studies here can be accessed, and ran, via the artifact.

6.1 Multi-dimensional Arrays

We demonstrate how our framework for lifting raw pointers to arrays scales to n-dimensional arrays.

6.1.1 Original C Program

Consider the following program, written in C, that allocates memory for a $N \times M$ matrix, and sets every value in the matrix to 0. We have omitted static arrays for brevity in the code.

```
1 void resetMatrix(int * arr, int N, int M){
2   int i, j;
3   int * mat = arr;
4   for (i = 0; i < N; i++) {
5     for (j = 0; j < M; j++) {
```

[1] We envision more robust testing by interfacing with `c2rust`'s cross-checking system (Sec. 7.3.1) in the future.

[2] https://zenodo.org/record/6334872

```
6         *(*(mat + i) + j) = 0;
7       }
8     }
9 }
10
11 int main() {
12   int N = 3, M = 2;
13   /* dynamic matrix allocation */
14   int  * dynamic_matrix = (int  * ) malloc(N * sizeof(int *));
15   for (int i = 0; i < N; i+ +) {
16     dynamic_matrix[i] = (int *) malloc(M * sizeof(int));
17   }
18   resetMatrix((int * )dynamic_matrix, N, M);
19   free(*dynamic_matrix);
20   free(dynamic_matrix);
21
22   return 0;
23 }
```

Listing 6.1: Initializing a Matrix in C

The program does the following:

1. It declares N and M.
2. It allocates memory of the size $N \times$ `sizeof(c_int)` for the "rows", which is a pointer to the "columns".
3. For every $i \in \{0, \ldots, N-1\}$, it allocates memory of the size $M \times$ `sizeof(c_int)` for the "column" of the ith "row".
4. It calls `resetMatrix` to set every element in the matrix to 0.
5. It then frees the $N \times M$ matrix.

6.1.2 Translation Pipeline

As part of the translation pipeline Fig. 4.1, we first leverage `c2rust` to syntactically transform the above program into **`unsafe`** Rust. This automatically generates a `lib.rs` file containing function information and dependencies. Then, we pass `lib.rs` to our tool, which performs our rewrite algorithm (algorithm 4.2.1). We build a call graph to represent the calling relationship between the functions in

the program. The calling relationship to note here is that `main` calls `resetMatrix`, so we have to tackle `resetMatrix` first, and then `main`.

6.1.3 Refactoring `resetMatrix`

Consider the following `c2rust`-translated code for `resetMatrix`:

```
1  pub unsafe extern "C" fn resetMatrix(mut arr: *mut *mut libc::c_int,
2  mut N: libc::c_int, mut M: libc::c_int) {
3    let mut i: libc::c_int = 0;
4    let mut j: libc::c_int = 0;
5    let mut mat: *mut *mut libc::c_int = arr;
6    i = 0 as libc::c_int;
7    while i < N {
8      j = 0 as libc::c_int;
9      while j < M {
10       *(*mat.offset(i as isize)).offset(j as isize) = 0 as libc::c_int;
11       j += 1
12     }
13     i += 1
14 };
```

Listing 6.2: resetMatrix, translated from c2rust

As in our rewrite pass for arrays, `arr_pass` (algorithm 4.2.3), we begin by generating the AST of `resetMatrix`. Then, we collect the constraints from the AST (Sec. 4.3). In line 10,

```
*(*mat.offset(i as isize)).offset(j as isize) = 0 as libc::c_int;
```

we can gather from the `c-INDEXED` rule that `mat`, if offset by i, returns `mat.offset(i as isize)`. Furthermore, by the `c-DEREF` rule, `mat.offset(i as isize)` dereferences to `*mat.offset(i as isize)`. Thus, by the **Array Reconstruction** CHR rule, when we solve the system of constraints, we know that if we index `mat` by `i`, then we get `*mat.offset(i as isize)`. But `*mat.offset(i as isize)` itself is then offset, and dereferenced. Thus, we actually have a *nested* index: if we index into `*mat.offset(i as isize)` with `j`, we get `*(*mat.offset(i as isize)).offset(j as isize)`.

Denote the labels of `mat`, `*mat.offset(i as isize)`, `*(*mat.offset(i as isize)).offset(j as isize)` by $\alpha_1, \alpha_2, \alpha_3$ respectively. We also know, by the `c-ASSIGNMENT` rule, that

Compat(α_3, `c_int`). Consider then the following relevant collected constraints:

$$\textit{Index}(\alpha_1, \texttt{usize}, \alpha_2), \textit{Index}(\alpha_2, \texttt{usize}, \alpha_3), \textit{Compat}(\alpha_3, \texttt{c\_int}).$$

When we solve our constraints, we begin by looking at *Index*(α_2, `usize`, α_3), since applying the `s-INDEXEDMUT` rule to generate trait bounds via the solver (Fig. 4.5) for *Index*(α_1, `usize`, α_2) relies on knowing what α_2 is compatible with. By the `s-INDEXEDMUT` rule (Fig. 4.5), we come up with a fresh type variable `T0` that implements the `IndexMut<usize, Output=c_int>` trait to represent arrays. The `s-INDEXEDMUT` rule also adds *Compat*(α_2, `T0`) to our constraints.

Since we now know what α_2 is compatible with, we can apply the `s-INDEXEDMUT` rule for *Index*(α_1, `usize`, α_2). We come up with a fresh type variable `T1` that implements the `IndexMut<usize, Output=T0>` trait, and add *Compat*(α_1, `T1`) to our constraints. Finally, by the `c-ASSIGNMENT` rule, we know that $\alpha_1 \simeq \textit{label}(\textit{arr})$.

With the above information, we can refactor `resetMatrix` into safe Rust that uses the Rust idioms for dynamic allocation of arrays, since the constraints indicate that `arr` and `mat` are being used as arrays.

```
pub extern "C" fn resetMatrix<
    T1: IndexMut<usize, Output = T0>,
    T0: IndexMut<usize, Output = c_int>,
>(arr: &mut T1, mut N: c_int, mut M: c_int) {
    let mut i: libc::c_int = 0;
    let mut j: libc::c_int = 0;
    let mut mat: &mut T1 = arr;
    i = 0 as c_int;
    while i < N {
        j = 0 as c_int;
        while j < M {
            (mat[i as usize])[j as usize] = 0 as c_int;
            j += 1
        }
        i += 1
    }
}
```

Listing 6.3: resetMatrix, refactored

Firstly, observe that all `unsafe` code have been removed! Again, this is a property of our refactoring, where we bubble `unsafe`-ty inwards, and since the `resetMatrix` was conservatively marked `unsafe`, most of the refactoring into safe code followed immediately, except for the dereference of raw pointers (Sec. 2.1.3). For the dereferencing of raw pointers, because our solved system of constraints indicated that they could be lifted to arrays, by refactoring the dereferencing of raw pointers into array indexing, we could also remove `unsafe`-ty. `arr` and `mat`, instead of being a raw pointer to a raw pointer to an `c_int`, have been refactored to instead be type `T1`, where `T1` implements `IndexMut<usize, Output=T0>`, and where `T0` implements `IndexMut<usize, Output=c_int>`. This completes our refactoring of `resetMatrix`.

6.1.4 Refactoring `main`

Finally, we tackle `main`:

```
unsafe fn main_0() -> libc::c_int {
  let mut N: libc::c_int = 3 as libc::c_int;
  let mut M: libc::c_int = 2 as libc::c_int;
  ...
  /* dynamic matrix allocation */
  let mut dynamic_matrix: *mut *mut libc::c_int =
  malloc((N as libc::c_ulong)
  .wrapping_mul(::std::mem::size_of::<*mut libc::c_int>()
  as libc::c_ulong)) as *mut *mut libc::c_int;
  let mut i: libc::c_int = 0 as libc::c_int;
  while i < N {
    let ref mut fresh0 = *dynamic_matrix.offset(i as isize);
    *fresh0 =
    malloc((M as libc::c_ulong)
    .wrapping_mul(::std::mem::size_of::<libc::c_int>()
    as libc::c_ulong)) as *mut libc::c_int;
    i += 1
  }
  resetMatrix(IndexMutWrapper(dynamic_matrix, usize, T0), N, M);
  free(*dynamic_matrix as *mut libc::c_void);
  free(dynamic_matrix as *mut libc::c_void);
  return 0 as libc::c_int;
```

```
23 }
```

Listing 6.4: main, translated from c2rust, after call site update

The above code snippet is *after* we update the call site of `resetMatrix` – in line 19, note that we have `IndexMutWrapper(dynamic_matrix, usize, T0)`. Denote the label of `dynamic_matrix` as α. Then, during the `arr_pass` rewrite pass for `main`, the constraint collector picks up from `IndexMutWrapper(dynamic_matrix, usize, T0)` the constraint

$$\mathit{ShouldIndex}(\alpha, \texttt{usize}, \alpha_{out})$$

where α_{out} denotes the label of the output of indexing into `dynamic_matrix`.

By keeping track of type variables, we know that $\mathit{Compat}(\alpha_{out}, \texttt{T0})$, and that `T0` implements the `IndexMut<usize, Output = c_int>` trait too, so we also add the constraint

$$\mathit{ShouldIndex}(\alpha_{out}, \texttt{usize}, \texttt{c\_int})$$

From lines 6-9, we gather that $\mathit{Malloc}(\alpha, \mathit{Raw}(\texttt{c\_int}))$. Because we are performing a `malloc`, and we know that `T0` is an `IndexMut` which was lifted from a `Raw(c_int)`, we want to assume that in this case $\texttt{T0} \simeq \texttt{Raw(c\_int)}$), so we also have $\mathit{Malloc}(\alpha, \alpha_{out})$.

Then, by the **Vec Reconstruction** CHR rule, since we have that $\mathit{Malloc}(\alpha, \alpha_{out})$ and $\mathit{ShouldIndex}(\alpha, \texttt{usize}, \alpha_{out})$, we also have the constraint that $\mathit{Vec}(\alpha, \texttt{usize}, \alpha_{out})$. Thus, lines 6-9 can be refactored as:

```
let mut dynamic_matrix = vec![Default::default(); N.try_into().unwrap()];
```

Listing 6.5: main, lines 6-9 after refactor

Next, for lines 12-16, let us consider line 12 first:

```
let ref mut fresh0 = *dynamic_matrix.offset(i as isize);
```

We first observe that on the RHS of the assignment, we are indexing into `dynamic_matrix`. Recall that `dynamic_matrix` has label α, so we denote `*dynamic_matrix.offset(i as isize)` as having label α'. Then, the constraint collector collects:

$$\mathit{Index}(\alpha, \texttt{usize}, \alpha').$$

From before, our constraint collector collected that

$$\mathit{ShouldIndex}(\alpha, \texttt{usize}, \alpha_{out}), \mathit{ShouldIndex}(\alpha_{out}, \texttt{usize}, \texttt{c\_int}).$$

By the **Indexed Collapse** rule, since we know that $\mathit{Index}(\alpha, \texttt{usize}, \alpha')$ and $\mathit{ShouldIndex}(\alpha, \texttt{usize}, \alpha_{out})$, we have that $\alpha' \simeq \alpha_{out}$. Thus, we also have that $\mathit{Index}(\alpha, \texttt{usize}, \alpha_{out})$. The LHS of the assignment then states that `fresh0` is a mutable reference to the RHS: let the label of `fresh0` be α_f, and the label of `*fresh0` be α'_f. Then, via the `c-LETMUTREF` rule (Appendix A), we collect that

$$\mathit{Ref}(\alpha_f, \alpha'),$$

indicating that α_f is a reference to α'.

But since we know that $\alpha' \simeq \alpha_{out}$, we also have

$$\mathit{Ref}(\alpha_f, \alpha_{out}).$$

In line 13, `fresh0` is dereferenced. Thus, via the `c-DEREF` rule, we collect that

$$\mathit{Deref}(\alpha_f, \alpha'_f),$$

indicating that α_f should dereference into α'_f. Thus, by the **Dereference Reference** CHR rule, we can unify this with

$$\alpha'_f \simeq \alpha_{out}$$

From lines 14-16, we gather that $\mathit{Malloc}(\alpha'_f, \texttt{c\_int})$. But, as $\alpha'_f \simeq \alpha_{out}$, we also have that $\mathit{Malloc}(\alpha_{out}, \texttt{c\_int})$. Finally, with $\mathit{Malloc}(\alpha_{out}, \texttt{c\_int})$ and $\mathit{ShouldIndex}(\alpha_{out}, \texttt{usize}, \texttt{c\_int})$ from before, we can once again apply the **Vec Reconstruction** CHR rule to get

$$\mathit{Vec}(\alpha_{out}, \texttt{usize}, \texttt{c\_int}), \mathit{Vec}(\alpha'_f, \texttt{usize}, \texttt{c\_int})$$

Thus, we can rewrite lines 12-16 as:

```
let ref mut fresh0 = dynamic_matrix[i as usize];
*fresh0 = vec![Default::default(); M.try_into().unwrap()];
```

In conjunction with the rest of the rewrites that we omit for brevity, as they have been thoroughly covered in Ch. 4, this completes the refactoring of `main`:

```
fn main_0() -> libc::c_int {
  let mut N: libc::c_int = 3 as c_int;
  let mut M: libc::c_int = 2 as c_int;
  let mut dynamic_matrix = vec![Default::default(); N.try_into().unwrap()];
  let mut i: libc::c_int = 0 as c_int;
  while i < N {
    let ref mut fresh0 = dynamic_matrix[i as usize];
    *fresh0 = vec![Default::default(); M.try_into().unwrap()];
    i += 1
  }
  resetMatrix(&mut dynamic_matrix, N, M);
  return 0 as c_int;
}
```

Listing 6.6: main, refactored

6.1.5 Discussion

Since the refactoring is intended to be semantics-preserving, this refactoring maintains the behavior of the original C program. In particular, the major refactors were to

1. Replace usages of the calls to `malloc` in C that was intended to dynamically allocate memory for an array, with the Rust idiom `vec![]`,
2. Lift raw pointers back into arrays *iff* they satisfied the constraints with the help of a trait bound implementing the `IndexMut` trait, and finally,
3. Refactor the pointer arithmetic back into array indexing, which is syntactic sugar for pointer arithmetic anyway.

6.1.6 Randomized Testing

We verify that the refactoring preserves the behavior of the original C program by running a randomized unit test suite. For this program, we want to test three

properties of our refactored program: firstly, that our dynamic allocation for matrices is valid, and that `resetMatrix` correctly sets each element to 0; secondly, that the allocated matrix indeed has N rows; and lastly that the allocated matrix has M columns. To this end, we run the following unit test on our refactored Rust program using `cargo`'s built-in command to execute unit tests of a package: `cargo test`. The randomized unit test runs for $n = 1000$ iterations, and initializes the matrix with a random size from 1 to 100. Next, it resets the matrix with our refactored `resetMatrix` function. It then performs the checks described before.

```
#[test]
fn matrix_correct() {
  // Run test 1000 times
  for _ in 0..1000 {
    // Initialize matrix with random size from 1 to 100.
    let N = rand::thread_rng().gen_range(1..100);
    let M = rand::thread_rng().gen_range(1..100);
    let mut dynamic_matrix = vec![Default::default(); N.try_into().unwrap()];
    let mut i: libc::c_int = 0 as c_int;
    while i < N {
      let ref mut fresh0 = dynamic_matrix[i as usize];
      *fresh0 = vec![Default::default(); M.try_into().unwrap()];
      i += 1
    }
    resetMatrix(&mut dynamic_matrix, N, M);
    // First test: Check that every element in matrix is set to 0
    for i in 1..N{
      for j in 1..M {
        assert_eq!(dynamic_matrix[i as usize][j as usize], 0)
      }
    }
    // Second test: Check that we have N rows
    assert_eq!(dynamic_matrix.len() as i32, N);
    // Second test: Check that we have M columns
    assert_eq!(dynamic_matrix[0 as usize].len() as i32, M)
  }
}
```

Running `cargo test` verifies that our program behaves as expected.

```
Finished test [unoptimized + debuginfo] target(s) in 0.55s
```

```
    Running unittests (target/debug/deps/matrix-1bb6ae8ed49d39d9)

    running 1 test
    test tests::matrix_correct ... ok

    test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured;
    0 filtered out; finished in 0.31s
```

6.2 Quicksort

As another case study, we apply our framework to a recursive sorting algorithm, Quicksort.

6.2.1 Original C Program

The original C program performs a canonical implementation of *deterministic* Quicksort, with the choice of pivot being the right-most element in the array.

```c
void swap(int *a, int *b) {
  int t = *a;
  *a = *b;
  *b = t;
}

int partition(int array[], int low, int high) {
  int pivot = array[high];
  int i = (low - 1);
  for (int j = low; j < high; j+ +) {
    if (array[j] ≤ pivot) {
      i∪;
      swap(&array[i], &array[j]);
    }
  }
  swap(&array[i + 1], &array[high]);
  return (i + 1);
}

void quickSort(int array[], int low, int high) {
  if (low < high) {
    int pi = partition(array, low, high);
    quickSort(array, low, pi - 1);
    quickSort(array, pi + 1, high);
```

```c
  }
}

int main() {
  int data[] = {8, 7, 2, 1, 0, 9, 6};
  int n = sizeof(data) / sizeof(data[0]);
  quickSort(data, 0, n - 1);
```

Listing 6.7: Quicksort in C

6.2.2 Translation Pipeline

As before, we leverage `c2rust` to syntactically transform the above program into **`unsafe`** Rust. We then pass the generated `lib.rs` to our tool to perform the rewrite algorithm. This begins by building a callgraph to represent the calling relationship between the functions in the program, which in this case is: `main` calls `quicksort`, which calls `partition`, which calls `swap`. Note that for `quicksort`, which is recursive, we have handled cycles by eliminating them, since a single pass of the analysis is sufficient to perform the rewrites.

6.2.3 Refactoring Quicksort

Our tool takes the output of `c2rust`, which has conservatively marked the entire code as **`unsafe`** *and* has non-idiomatic uses of raw pointers as decayed arrays, and refactors it into *mostly*-safe Rust code with idiomatic uses of Rust arrays instead of raw pointers.

```rust
pub extern "C" fn swap(mut a: *mut libc::c_int, mut b: *mut libc::c_int) {
    let mut t: libc::c_int = unsafe { *a };
    unsafe { *a = unsafe { *b } };
    unsafe { *b = t };
}
pub extern "C" fn partition<T0: IndexMut<usize, Output = c_int>>(
    array: &mut T0,
    low: c_int,
    high: c_int,
) -> libc::c_int {
    let mut pivot: libc::c_int = array[high as usize];
    let mut i: libc::c_int = low - 1 as c_int;
    let mut j: libc::c_int = low;
    while j < high {
```

```
        if array[j as usize] ≤ pivot {
            i += 1;
            swap(&mut array[i as usize], &mut array[j as usize]);
        }
        j += 1
    }
    swap(
        &mut array[(i + 1 as libc::c_int) as usize],
        &mut array[high as usize],
    );
    return i + 1 as c_int;
}
pub extern "C" fn quickSort<T0: IndexMut<usize, Output = c_int>>(
    array: &mut T0,
    mut low: libc::c_int,
    mut high: libc::c_int,
) {
    if low < high {
        let mut pi: libc::c_int = partition(array, low, high);
        quickSort(array, low, pi - 1 as c_int);
        quickSort(array, pi + 1 as c_int, high);
    };
}
fn main_0() -> libc::c_int {
    let mut data: [libc::c_int; 7] = [
        8 as c_int,
        7 as c_int,
        2 as c_int,
        1 as c_int,
        0 as c_int,
        9 as c_int,
        6 as c_int,
    ];
    let mut n: libc::c_int = (::std::mem::size_of::<[libc::c_int; 7]>() as libc::c_ulong)
        .wrapping_div(::std::mem::size_of::<libc::c_int>() as libc::c_ulong)
        as c_int;
    quickSort(&mut data, 0 as c_int, n - 1 as c_int);
    return 0 as c_int;
}
```

Listing 6.8: Quicksort, translated from c2rust

6.2.4 Discussion

As discussed in Sec. 6.1.5, the refactoring maintains the behavior of the original C program. Further, observe that the refactored Rust code is *mostly*-safe, with idiomatic uses of Rust arrays replacing the non-idiomatic uses of raw pointers as decayed arrays. Furthermore, while Quicksort is a recursive algorithm, because our algorithm only needs a single pass across each function to perform the necessary refactoring, we can handle recursion by eliminating cycles from our call graph.

An interesting point to note is that the exception that makes this code *mostly*-safe instead of *completely*-safe lies within the `swap` function. Since the `swap` function works at the level of raw pointers and memory to swap the values at the memory locations, it is inherently **`unsafe`** according to Rust's provenance of unsafety (Sec. 2.1.3). Therefore, the analysis performed by our tool cannot determine a suitable refactoring that will remove this **`unsafe`**-ty. What our tool can do, is to bubble **`unsafe`** code so that it is self-contained, i.e., we minimize the scope of the **`unsafe`** block so that instead of encapsulating the entire function, it simply encapsulates the dereferencing of raw pointers. Again, this is achieved as a consequence of how our refactoring "bubbles" **`unsafe`**-ty inwards as it traverses the AST of the function.

6.2.5 Randomized Testing

We verify that the refactoring preserves the behavior of the original C program by running a randomized unit test suite. For this program, we want to test three properties of our refactored program: firstly, our vector should be sorted in monotonically increasing order; secondly that our vector should have the same length after sorting; and lastly that our sorted vector should contain the same elements. The randomized unit test runs for $n = 1000$ iterations. It initializes a vector of size 10000, and then assigns it with random elements in the range $0 - 10000$. Next, it performs our refactored Quicksort on the vector. It then performs the checks described before.

```
1   #[test]
```

```
fn quicksort_sorts() {
  // Run our test 1000 times
  for _ in 1..1000 {
    // Initialize array with 10000 random elements from 0-10000
    let mut rng = rand::thread_rng();
    let range = Uniform::new(0, 10000);
    let mut vals: Vec<i32> = (0..10000).map(|_| rng.sample(&range)).collect();
    let n = vals.len();
    let mut copy_of_vals = vals.clone();
    quickSort(&mut vals, 0 as c_int, (n - 1).try_into().unwrap());
    // Three tests for sorted-ness.
    // First test: sorted in monotonically increasing order
    for i in 0..n-1{
      assert!(vals[i] ≤ vals[i+1])
    }
    // Second test: vector has same length after sorting
    assert_eq!(n, vals.len());
    // Third test: sorted vector contains same elements as before
    copy_of_vals.sort();
    assert_eq!(vals, copy_of_vals);
  }

```

Running the tests with `cargo test` verifies that our program behaves as expected.

```
  Finished test [unoptimized + debuginfo] target(s) in 0.00s
  Running unittests (target/debug/deps/qsort-9f94a42ab344e435)

  running 1 test
  test tests::quicksort_sorts ... ok

  test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
  finished in 1.66s

```

7 Discussion and Conclusion

We conclude this work by comparing and contrasting our approach to Laertes, exploring future instantiations of the framework, and finally discussing future work.

7.1 Comparison with Laertes

As discussed in Sec. 3.5, Emre et al. [Emr+21] recently presented a first technique for automatically removing some sources of unsafety in translated Rust programs. Their tool, `Laertes`, is available as an artifact ([ES21]).[1]

Their technique relies on the iterative refinement of the `unsafe` Rust code, using the Rust compiler to guide their optimistic conversion of raw pointers into safe references. This results in unwieldy code, as they explicate lifetimes and *convert all raw pointer declarations into optional references*, with additional helper functions to assist in rewriting pointers to references and for borrowing.

Furthermore, Laertes deliberately *only* targets raw pointers that are obtained via custom memory allocation (via `malloc`), or if the raw pointer appears as part of the public signature of an API implemented by the program. Thus, they avoid raw pointers that are used in pointer arithmetic, which is what we target, and hence our results are complementary as their tool would not function on our array examples. Nevertheless, their motivating example in Listing 7.1, demonstrates the readability of their translated code [Emr+21]. Even a simple use-case like translating `swap`, as in Listing 7.2, requires optional references and helper functions to operate those optional references.

[1] https://zenodo.org/record/5442253

```
  pub fn find <'a1 , 'a2 , 'a3 , 'a4 , 'a5 , 'a6 >
  (mut value: c_int , mut node: Option <&'a1 mut node_t <'a2 , 'a3 >>)
  -> Option <&'a4 mut node_t <'a5 , 'a6 >> {

    if value < *(* node.as_ref().unwrap()).value.as_ref().unwrap() &&
    !(* node.as_ref().unwrap()).left.
    is_none() {
      return find(value , borrow_mut (&mut (*node.unwrap()).left))
    } else {
      if value > *(* node.as_ref().unwrap()).value.as_ref().unwrap() &&
      !(* node.as_ref().unwrap()).
      right.is_none() {
        return find(value , borrow_mut (&mut (*node.unwrap()).right))
      } else { if value = *(* node.as_mut().unwrap()).value.as_mut().unwrap()
        { return node } }
    }
    return None;

```

Listing 7.1: Laertes-translated code

```
fn swap(mut a: Option<&mut i32>, mut b: Option<&mut i32>) {
    let mut t: i32 = *borrow_mut(&mut a).unwrap();
    *borrow_mut(&mut a).unwrap() = *borrow_mut(&mut b).unwrap();
    *borrow_mut(&mut b).unwrap() = t;
}

unsafe fn f(mut a: *mut i32, mut b: *mut i32) {
    // ...
    swap(Some(&mut *a), Some(&mut *b))
}

pub fn borrow_mut<'a, 'b: 'a, T>(p: &'a mut Option<&'b mut T>) -> Option<&'a mut T> {
    p.as_mut().map(|x| &mut *x)
}

```

Listing 7.2: `swap` translated by Laertes

On the other hand, instead of *coercing* the compiler by using optional references, we use constraints to guide our refactoring so that we can discern the intent of the original code. By being able to determine when a raw pointer is being used as an array, we can structure our refactoring to be more *idiomatic* by using Rust idioms for arrays, rather than a brute-force wrapping with optional references. The remaining sources of `unsafe`-ty can be handled by our rewrite rules that target unnecessary conservative `unsafe` code.

While our approach only tackles lifting raw pointers to arrays for now, the resulting code is higher-quality and more readable, in line with our goal of making automatically translated code from C to Rust more palatable. To that end, we envision being able to target other specific uses of raw pointers in C

beyond arrays, such as `strings`, while leaving legitimate uses of raw pointers as `unsafe`.

One future direction for our work is to target the lifting of raw pointers to references or smart pointers using the framework in this thesis, instead of a brute-force optimistic rewrite with the compiler as oracle (Sec. 7.2.2). It is our hope that this will provide more readable, higher-quality code that is idiomatic as the constraints will guide our refactoring.

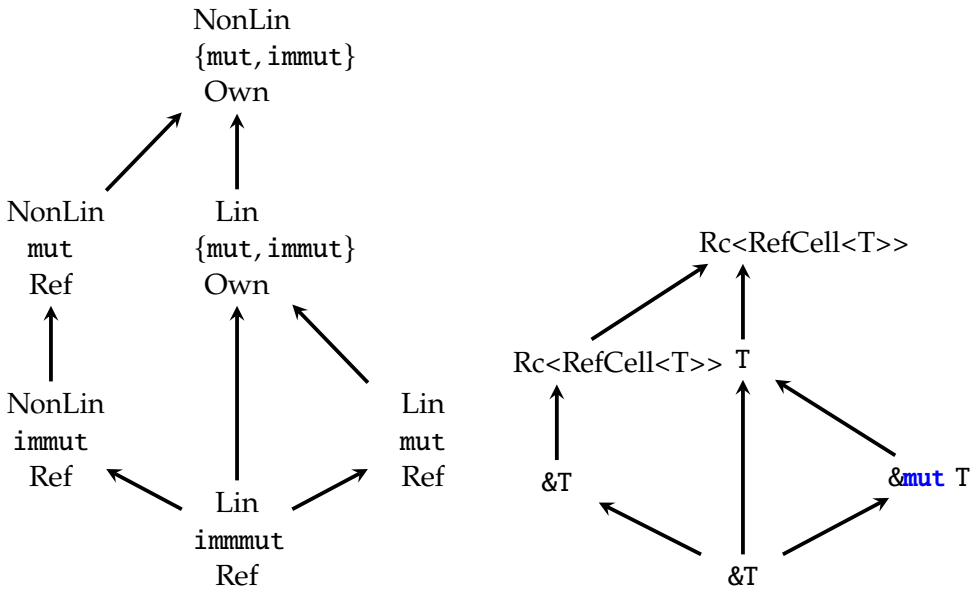
7.2 Extensibility: Other Instantiations of the Framework

7.2.1 `string_pass`

To expand the repertoire of rewrite passes in our framework beyond `arr_pass`, as a future direction we are exploring the refactoring of raw pointers that are used as `strings` into idiomatic Rust.

To this end, we introduce a new instantiation of our framework, `string_pass`. `string_pass` will require a separate constraint collection pass, constraint system rules, and rewrite rules, that are specific to handling `strings`. As described in Ch. 4, CHRusty is a generic framework – thus, while instantiations of CHRusty may share some common rules (Fig. A.1 in Appendix A and Fig. B.1 in Appendix B), extending CHRusty with the instantiation specific to `strings` will be to append an additional set of constraint collection rules, CHR rules, and rewrite rules on top of the general system.

`strings` are similar to arrays in that they both can be dynamically allocated and require the use of raw pointers, but have a safe abstraction in Rust. Rust provides two notions of strings: `String` is the dynamic heap type that can be modified, and `str` is an immutable sequence that is usually passed around as a slice, `&str`. This would be an interesting endeavour as Rust would eliminate potential buffer overflows in C that are possible via misuse of functions like `strcpy` rather than `strncpy` and so on; furthermore, the translation into either `String` or `str` idioms is technically interesting and would benefit from our constraint collection.



(a) Lattice of Lin/Mut/Own

(b) Lattice of Corresponding Rust Types

Figure 7.1: Lattice of Lin/Mut/Own and corresponding Rust types

7.2.2 `pointer_pass`

Another future direction for rewrite passes that we are exploring is a `pointer_pass` instantiation of our framework. We draw inspiration from the *Stacked Borrows* model that defines an aliasing discipline for Rust programs [Jun+19]. *Stacked Borrows* avoids the use of lifetimes by the borrow checker, instead using a dynamic analysis that does not use lifetimes. By supporting a variant of the *Stacked Borrows* model and capturing the aliasing discipline as a type constraint system according to their operational semantics, we will be able to extract ownership information and a happens-before relation as constraint information.

In essence, we would like to keep track of three orthogonal but integral properties of a Rust variable: whether it is an affine usage, whether it is mutable, and whether it is owned. These properties form a lattice as in Fig. 7.1a. Then, for each function signature and field, we want to assign the least point in the lattice that supports all the operations that are performed on that field. Every element in Fig. 7.1a then corresponds to a Rust type as in Fig. 7.1b.

Since our current constraint system conflates both a general array constraint system together with a mutability constraint system, we can separate our current implementation into different constraint collection passes. Then, we

can leverage the mutability constraint collection pass as part of the constraint system for `pointer_pass`, without having to rely on the array constraint system. The constraint system can then help justify certain reorderings and program transformations, such as indicating whether to lift a raw pointer into a reference, a smart pointer like a `Box` or `Rc`, or to not lift at all.

7.3 Future work

In this section, we discuss future work for the development of CHRusty, besides the other instantiations listed in Sec. 7.2.

7.3.1 Integration with `c2rust`'s cross-checking

The `c2rust` team has provided a cross-checking tool to automatically verify that their translated program behaves the same as the original C code.[2] This is done by comparing execution traces of the program before and after transformation. For more robust testing, instead of our randomized testing suite, future work would interface with this cross-checking tool to automatically verify that the code refactored by CHRusty preserves behavior.

7.3.2 Handling the rest of the C Language

As of now, CHRusty only handles a *small* subset of the C language. While our tool provides support for running on large Cargo projects by using the `lib.rs` file as an entry-point, it still does not handle a large number of C types. Furthermore, additional support will have to be implemented for constructs like **`struct`**s, **`union`**s, *etc.*, in order for CHRusty to work on any arbitrary project. While it may be trivial to lift all raw pointers to `Rc<RefCell<T>>`, this sacrifices performances and the static guarantees of Rust in favour of removing **`unsafe`**ty, which is contradictory as **`unsafe`**ty is a means provided by the Rust compiler to allow performant uses of memory *provided* the programmer can guarantee that the **`unsafe`** code does not lead to undefined behaviour.

Regardless, we have shown that CHRusty successfully refactors both static

[2] https://c2rust.com/manual/docs/cross-check-tutorial.html

and dynamic arrays regardless of type, so providing support for the rest of the C language is a matter of implementation rather than contribution in the context of array refactoring.

7.3.3 Survey on Idiomatic Code & Uses of Raw Pointers

While we claim that our refactoring leads to more idiomatic code by virtue of using the Rust idioms for dynamically allocating memory for arrays, and for array indexing, this claim can still be supported by a survey of what idiomatic code should look like.

Furthermore, while it seems obvious that arrays are extremely prevalent in C code, a survey of the usage of arrays in real-life C code, and the usage of raw pointers *for the purposes* of arrays, would be useful in illustrating the feasibility of CHRusty. This survey should also be expanded to other uses of raw pointers, such as for `strings`, so that we have a better sense of what proportion of raw pointers are intended for arrays, strings, and other uses, and thus what proportion of raw pointers our tool tackles.

7.4 Conclusion

This thesis presents a framework, CHRusty, for translating C to safer, and more idiomatic Rust. Our implementation is an instatiation of this framework that specifically targets the lifting of raw pointers to arrays, though we also suggest other instantiations like `string_pass` and `pointer_pass`. Existing automatic translators from C to Rust do not preserve the memory-safety property that makes Rust code desirable; and, even if they do, the resulting code is difficult to work with. Our tool provides an idiomatic refactoring from `unsafe` Rust to safer Rust by lifting raw pointers back to arrays, in accordance with the intent of the original program. The feasibility of this approach hints at a future where one can automatically translate C code into Rust, perhaps as part of a migration of a legacy C codebase, and then continue working with the newly-translated Rust codebase alongside the performance and safety guarantees of Rust.

Bibliography

[ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. "A survey of attacks on ethereum smart contracts (sok)". In: *International conference on principles of security and trust*. Springer. 2017, pp. 164–186 (cit. on p. 2).

[And+15] Brian Anderson, Lars Bergstrom, David Herman, Josh Matthews, Keegan McAllister, Manish Goregaokar, Jack Moffitt, and Simon Sapin. "Experience report: Developing the Servo web browser engine using Rust". In: *arXiv preprint arXiv:1505.07383* (2015) (cit. on p. 1).

[Ast+20] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J Summers. "How Do Programmers Use Unsafe Rust?" In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–27 (cit. on pp. 2, 9).

[Bry16] David Bryant. *A Quantum Leap for the Web*. 2016. URL: https://medium.com/mozilla-tech/a-quantum-leap-for-the-web-a3b7174b3c12 (visited on 02/16/2022) (cit. on p. 1).

[Cor21] Jonathan Corbet. *Rust Support hits linux-next*. 2021. URL: https://lwn.net/Articles/849849/ (visited on 02/16/2022) (cit. on p. 1).

[Dev17] Citrus Developers. *Citrus/Citrus*. 2017. URL: https://gitlab.com/citrus-rs/citrus (visited on 02/17/2022) (cit. on p. 12).

[Dow97] Mark Dowson. "The Ariane 5 software failure". In: *ACM SIGSOFT Software Engineering Notes* 22.2 (1997), p. 84 (cit. on p. 2).

[Dur+14] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. "The matter of heartbleed". In: *Proceedings of the*

2014 conference on internet measurement conference. 2014, pp. 475–488 (cit. on p. 2).

[ECS20] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. "Is Rust used safely by software developers?" In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE. 2020, pp. 246–257 (cit. on p. 12).

[Elh20] Nelson Elhage. *Supporting Linux kernel development in Rust*. 2020. URL: https://lwn.net/Articles/829858/ (visited on 02/16/2022) (cit. on p. 1).

[Emr+21] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. "Translating C to safer Rust". In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (2021), pp. 1–29 (cit. on pp. 13, 62).

[ES21] Mehmet Emre and Ryan Schroeder. *Artifact for "Translating C to Safer Rust"*. Sept. 2021. DOI: 10.5281/zenodo.5442253. URL: https://doi.org/10.5281/zenodo.5442253 (cit. on pp. 13, 62).

[Fow18] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018 (cit. on pp. 13–15).

[Frü94] Thom Frühwirth. "Constraint handling rules". In: *French School on Theoretical Computer Science*. Springer. 1994, pp. 90–107 (cit. on pp. 17, 25, 46).

[GLS01] Rakesh Ghiya, Daniel Lavery, and David Sehr. "On the importance of points-to analysis and other memory disambiguation methods for C programs". In: *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. 2001, pp. 47–58 (cit. on p. 2).

[Hor97] Susan Horwitz. "Precise flow-insensitive may-alias analysis is NP-hard". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19.1 (1997), pp. 1–6 (cit. on p. 2).

[Hut21] Tim Hutt. *Would Rust secure cURL?* 2021. URL: http://blog.timhutt.co.uk/curl-vulnerabilities-rust/ (visited on 02/16/2022) (cit. on p. 2).

[inc20a] Immunant inc. *immunant/c2rust*. 2020. URL: https://github.com/immunant/c2rust (visited on 02/17/2022) (cit. on p. 12).

[inc20b] Immunant inc. *immunant/c2rust/crosschecks*. 2020. URL: https://c2rust.com/manual/cross-checks/index.html (visited on 02/17/2022) (cit. on p. 12).

[inc20c] Immunant inc. *immunant/c2rust/transpiler*. 2020. URL: https://c2rust.com/manual/c2rust-transpile/index.html (visited on 02/17/2022) (cit. on p. 12).

[Jun+17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the foundations of the Rust programming language". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–34 (cit. on p. 9).

[Jun+19] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. "Stacked borrows: an aliasing model for Rust". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–32 (cit. on pp. 7, 65).

[Kam22] Panos Kampanakis. *Introducing s2n-quic, a new open-source QUIC protocol implementation in Rust*. 2022. URL: https://aws.amazon.com/blogs/security/introducing-s2n-quic-open-source-protocol-rust/ (visited on 02/21/2022) (cit. on p. 1).

[KN21a] S. Klabnik and C. Nichols. *The Rust Programming Language*. 2021. URL: https://doc.rust-lang.org/book/ (visited on 02/17/2022) (cit. on pp. 5, 10).

[KN21b] S. Klabnik and C. Nichols. *The Rust Programming Language Reference*. 2021. URL: https://doc.rust-lang.org/stable/reference/behavior-considered-undefined.html (visited on 02/17/2022) (cit. on p. 9).

[KN21c] S. Klabnik and C. Nichols. *The Rust Programming Language Reference*. 2021. URL: https://doc.rust-lang.org/stable/reference/unsafety.html (visited on 02/17/2022) (cit. on p. 9).

[KN21d] S. Klabnik and C. Nichols. *The Rust Programming Language Reference*. 2021. URL: https://doc.rust-lang.org/stable/reference/unsafety.html (visited on 02/17/2022) (cit. on p. 10).

[KN22a] S. Klabnik and C. Nichols. *Guide to Rustc Development*. 2022. URL: https://rustc-dev-guide.rust-lang.org/syntax-intro.html (visited on 03/04/2022) (cit. on p. 43).

[KN22b] S. Klabnik and C. Nichols. *Guide to Rustc Development*. 2022. URL: https://rustc-dev-guide.rust-lang.org/hir.html (visited on 03/04/2022) (cit. on p. 43).

[KN22c] S. Klabnik and C. Nichols. *Guide to Rustc Development*. 2022. URL: https://rustc-dev-guide.rust-lang.org/mir/index.html (visited on 03/04/2022) (cit. on p. 43).

[KN22d] S. Klabnik and C. Nichols. *Guide to Rustc Development*. 2022. URL: https://rustc-dev-guide.rust-lang.org/about-this-guide.html (visited on 03/04/2022) (cit. on p. 43).

[Lev+15] Amit Levy, Michael P Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. "Ownership is theft: Experiences building an embedded OS in Rust". In: *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. 2015, pp. 21–26 (cit. on p. 1).

[Lin+16] Yi Lin, Stephen M Blackburn, Antony L Hosking, and Michael Norrish. "Rust as a language for high performance GC implementation". In: *ACM SIGPLAN Notices* 51.11 (2016), pp. 89–98 (cit. on p. 1).

[LT93] Nancy G Leveson and Clark S Turner. "An investigation of the Therac-25 accidents". In: *Computer* 26.7 (1993), pp. 18–41 (cit. on p. 2).

[Mat16] Niko Matsakis. *Introducing MIR*. 2016. URL: https://blog.rust-lang.org/2016/04/19/MIR.html (visited on 03/04/2022) (cit. on p. 44).

[PS91] Jens Palsberg and Michael I Schwartzbach. "Object-oriented type inference". In: *ACM SIGPLAN Notices* 26.11 (1991), pp. 146–161 (cit. on p. 13).

[PS94] Jens Palsberg and Michael I Schwartzbach. *Object-oriented type systems*. John Wiley and Sons Ltd., 1994 (cit. on pp. 13, 21).

[SCP17] Garming Sam, Nick Cameron, and Alex Potanin. "Automated refactoring of rust programs". In: *Proceedings of the Australasian Computer Science Week Multiconference*. 2017, pp. 1–9 (cit. on pp. 1, 13).

[SD04] Tom Schrijvers and Bart Demoen. "The KU Leuven CHR system: Implementation and application". In: *First workshop on constraint handling rules: selected contributions*. 2004, pp. 1–5 (cit. on pp. 17, 25, 46).

[SH21] Jeff Vander Stoep and Stephen Hines. *Rust in the Android platform*. 2021. URL: https://security.googleblog.com/2021/04/rust-in-android-platform.html (visited on 02/16/2022) (cit. on p. 1).

[Sha17] Jamey Sharp. *Citrus/Citrus*. 2017. URL: https://github.com/jameysharp/corrode (visited on 02/17/2022) (cit. on p. 12).

[Tip+11] Frank Tip, Robert M Fuhrer, Adam Kieżun, Michael D Ernst, Ittai Balaban, and Bjorn De Sutter. "Refactoring using type constraints". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33.3 (2011), pp. 1–47 (cit. on p. 13).

[TKB03] Frank Tip, Adam Kiezun, and Dirk Bäumer. "Refactoring for generalization using type constraints". In: *ACM SIGPLAN Notices* 38.11 (2003), pp. 13–26 (cit. on p. 13).

[WL95] Robert P Wilson and Monica S Lam. "Efficient context-sensitive pointer analysis for C programs". In: *ACM Sigplan Notices* 30.6 (1995), pp. 1–12 (cit. on p. 2).

[Zbo] Adrian Zborowski. “Framework for Idiomatic Refactoring of Rust Programming Language Code”. In: () (cit. on p. 13).

A Constraint Collection Rules

C-Fun

$$\frac{\alpha = \mathit{label}(\mathtt{e}) \qquad \alpha_i = \mathit{label}(\mathtt{v_i}), i \in 1..n \qquad M = \bigcup_{i=1}^{n}\{v_i : \alpha_i\} \qquad C = \bigcup_{i=1}^{n}\{\mathit{Compat}(\alpha_i, \tau_i)\}}{\Sigma(\mathtt{fn\ fname}(\mathtt{v_1} : \tau_1, .., \mathtt{v_n} : \tau_n) \rightarrow \tau_{\mathtt{out}}\,\{\mathtt{e}\}, []) \triangleq \Sigma(e, M) \cup C}$$

C-Call

$$\frac{\alpha_i = \mathit{label}(\mathtt{e_i}), i \in 1..n \qquad \Sigma' = \bigcup_{i=1}^{n}\Sigma(\mathtt{e_i}, M)}{\Sigma(\mathtt{fname}(\mathtt{e_1}, \ldots, \mathtt{e_n}), M) \triangleq \Sigma'}$$

C-Var

$$\frac{\alpha = \mathit{label}(\mathtt{v}) \qquad \alpha' = M(\mathtt{v})}{\Sigma(\mathtt{v}, M) \triangleq \{\alpha \simeq \alpha'\}}$$

C-Let

$$\frac{\alpha = \mathit{label}(\mathtt{v}) \qquad \alpha_1 = \mathit{label}(\mathtt{e_1}) \qquad \alpha_2 = \mathit{label}(\mathtt{e_2}) \qquad \alpha_3 = \mathit{label}(\mathtt{e_1; e_2}) \qquad M' = \{\mathtt{v} : \alpha\} \cup M}{\Sigma(\mathtt{let\ v} : \tau\ =\ e_1; e_2, M) \triangleq \{\alpha \simeq \alpha_1, \alpha_2 \simeq \alpha_3\} \cup \Sigma(e_1, M') \cup \Sigma(e_2, M')}$$

C-Seq

$$\frac{\alpha_2 = \mathit{label}(\mathtt{e_2}) \qquad \alpha = \mathit{label}(\mathtt{e_1; e_2})}{\Sigma(\mathtt{e_1; e_2}, M) \triangleq \{\alpha \simeq \alpha_2\} \cup \Sigma(e_1, M) \cup \Sigma(e_2, M)}$$

C-While

$$\frac{\alpha = \mathit{label}(\mathtt{while\ e_1\ \{e_2\}}) \qquad \alpha_2 = \mathit{label}(\mathtt{e_2})}{\Sigma(\mathtt{while\ e_1\ \{e_2\}}, M) \triangleq \{\alpha \simeq \alpha_2\}\Sigma(e_1, M) \cup \Sigma(e_2, M)}$$

C-Assignment

$$\frac{\alpha_1 = \mathit{label}(\mathtt{e_1}) \qquad \alpha_2 = \mathit{label}(\mathtt{e_2})}{\Sigma(\mathtt{e_1} := \mathtt{e_2}, M) \triangleq \{\alpha_1 \simeq \alpha_2, \mathit{Mut}(\alpha_1)\} \cup \Sigma(e_1, M) \cup \Sigma(e_2, M)}$$

C-Constant

$$\frac{}{\Sigma(\mathtt{c}, M) \triangleq \{\}}$$

C-Cast

$$\frac{\alpha = \mathit{label}(\mathtt{e}) \qquad \alpha' = \mathit{label}(\mathtt{e\ as}\ \tau) \qquad \mathtt{e} = \mathtt{fname}(\mathtt{e_1}, \ldots, \mathtt{e_n}) \Rightarrow \mathtt{fname} \neq \mathtt{malloc}}{\Sigma(\mathtt{e\ as}\ \tau, M) \triangleq \{\alpha \simeq \alpha', \mathit{Compat}(\alpha, \tau)\}}$$

C-BinOp

$$\frac{\alpha_1 = \mathit{label}(\mathtt{e_1}) \qquad \alpha_2 = \mathit{label}(\mathtt{e_2})}{\Sigma(\mathtt{e_1\ bop\ e_2}, M) \triangleq \{\alpha_1 \simeq \alpha_2\} \cup \Sigma(e_1, M) \cup \Sigma(e_2, M)}$$

Figure A.1: General Constraint Collection Rules

C-MALLOC

$$\frac{\alpha = \mathit{label}(\texttt{malloc(N} \cdot \texttt{sizeof}(\tau))) \qquad \Sigma' = \Sigma(\texttt{e}, M)}{\Sigma(\texttt{malloc(N} \cdot \texttt{sizeof}(\tau))\texttt{)}, M) \triangleq \{\mathit{Malloc}(\alpha, \tau)\} \cup \{\mathit{Compat}(\alpha, \tau)\} \cup \Sigma'}$$

C-VEC

$$\frac{\alpha = \mathit{label}(\texttt{e}) \qquad \Sigma' = \Sigma(\texttt{e}, M) \cup \Sigma(\tau_{\texttt{out}}, M) \qquad \alpha' = M(e)}{\Sigma(\texttt{IndexMutWrapper(e,}\ \tau_{\texttt{ind}}\texttt{,}\ \tau_{\texttt{out}}\texttt{)}, M) \triangleq \{\mathit{ShouldIndex}(\alpha, \tau_{ind}, \tau_{out}), \alpha \simeq \alpha'\} \cup \Sigma'}$$

C-INDEXED

$$\frac{\alpha = \mathit{label}(\texttt{e}_1\texttt{.offset(e}_2\texttt{)}) \qquad \alpha_1 = \mathit{label}(\texttt{e}_1) \qquad \alpha_2 = \mathit{label}(\texttt{e}_2) \qquad t_1^c := \mathit{Offset}(\alpha_1, \alpha_2, \alpha) \qquad t_2^c := \mathit{Compat}(\alpha_2, \texttt{usize})}{\Sigma(\texttt{e}_1\texttt{.offset(e}_2\texttt{)}, M) \triangleq \{t_1^c, t_2^c\} \cup \Sigma(e_1, M) \cup \Sigma(e_2, M)}$$

C-DEREF

$$\frac{\alpha = \mathit{label}(*\texttt{e}) \qquad \alpha' = \mathit{label}(\texttt{e})}{\Sigma(*\texttt{e}, M) \triangleq \{\mathit{Deref}(\alpha', \alpha)\} \cup \Sigma(e, M)}$$

C-LETMUTREF

$$\frac{\alpha = \mathit{label}(\texttt{v}) \qquad \alpha_1 = \mathit{label}(\texttt{e}_1) \qquad \alpha_2 = \mathit{label}(\texttt{e}_2) \qquad \alpha_3 = \mathit{label}(\texttt{e}_1\texttt{;e}_2) \qquad M' = \{\texttt{v} : \alpha\} \cup M}{\Sigma(\texttt{let ref mut v} : \tau\ \texttt{=}\ e_1\texttt{;}e_2, M) \triangleq \{\alpha \simeq \alpha_1, \alpha_2 \simeq \alpha_3, \mathit{Ref}(\alpha, \alpha_1)\} \cup \Sigma(e_1, M') \cup \Sigma(e_2, M')}$$

Figure A.2: Array-related Constraint Collection Rules

B Rewrite Rules

U-FUN

$$\frac{\alpha_i = \mathit{label}(\mathtt{v_i}) \quad \Sigma \vdash_{\alpha_i} \tau_i', c_i', G_i, \Sigma_i \text{ for } i \in 1..n \quad \Sigma' = \cup_{i=1}^{n} \Sigma_i \quad \alpha = \mathit{label}(\mathtt{e}) \quad \Sigma' \vdash_{\alpha} \tau', c', G', \Sigma'' \quad G = \cup_{i=1}^{n} G_i \cup G' \quad \Sigma''; \mathtt{unsafe\{e\}} \longrightarrow \mathtt{e'}}{\begin{array}{c}\Sigma; \mathtt{unsafe\ fn\ fname}(c_1\ \mathtt{v_1} : \tau_1, .., c_n\ \mathtt{v_n} : \tau_n) \longrightarrow c\ \tau\ \{\mathtt{e}\} \longrightarrow \\ \mathtt{fn\ fname}\langle \mathtt{G} \rangle (c_1'\ \mathtt{v_1} : \tau_1', .., c_n'\ \mathtt{v_n} : \tau_n') \longrightarrow c'\ \tau'\ \{\mathtt{e'}\}\end{array}}$$

U-CALL

$$\frac{\Sigma; \mathtt{unsafe\{e_i\}} \longrightarrow \mathtt{e_i'} \text{ for } i \in 1..n}{\Sigma; \mathtt{unsafe}\ \{\mathtt{fname}(\mathtt{e_1}, .., \mathtt{e_n})\} \longrightarrow \mathtt{fname}(\mathtt{e_1'}, .., \mathtt{e_n'})}$$

U-LET

$$\frac{\alpha = \mathit{label}(\mathtt{v}) \quad \Sigma \vdash_{\alpha} \tau', c' \text{ for } i \in 1..n \quad \Sigma; \mathtt{unsafe\{e_1\}} \longrightarrow \mathtt{e_1'} \quad \Sigma; \mathtt{unsafe\{e_2\}} \longrightarrow \mathtt{e_2'}}{\Sigma; \mathtt{unsafe\{let}\ c\ \mathtt{v} : \tau\ =\ e_1; e_2\} \longrightarrow \mathtt{let}\ c'\ \mathtt{v} : \tau'\ =\ e_1'; e_2'}$$

U-SEQ

$$\frac{\Sigma; \mathtt{unsafe\{e_1\}} \longrightarrow \mathtt{e_1'} \quad \Sigma; \mathtt{unsafe\{e_2\}} \longrightarrow \mathtt{e_2'}}{\Sigma; \mathtt{unsafe\{e_1; e_2\}} \longrightarrow \mathtt{e_1'; e_2'}}$$

U-ASSIGNMENT

$$\frac{\Sigma; \mathtt{unsafe\{e_1\}} \longrightarrow \mathtt{e_1'} \quad \Sigma; \mathtt{unsafe\{e_2\}} \longrightarrow \mathtt{e_2'}}{\Sigma; \mathtt{unsafe\{e_1 := e_2\}} \longrightarrow \mathtt{e_1' := e_2'}}$$

U-CONSTANT

$$\frac{}{\Sigma; \mathtt{unsafe\{c\}} \longrightarrow \mathtt{c}}$$

U-VAR

$$\frac{}{\Sigma; \mathtt{unsafe\{v\}} \longrightarrow \mathtt{v}}$$

U-CAST

$$\frac{\alpha = \mathit{label}(\mathtt{e\ as}\ \tau) \quad \Sigma \vdash \alpha \simeq \tau' \quad \Sigma; \mathtt{unsafe\{e\}} \longrightarrow \mathtt{e'}}{\Sigma; \mathtt{unsafe\{e\ as}\ \tau\} \longrightarrow \mathtt{e'\ as}\ \tau'}$$

U-BINOP

$$\frac{\Sigma; \mathtt{unsafe\{e_1\}} \longrightarrow \mathtt{e_1'} \quad \Sigma; \mathtt{unsafe\{e_2\}} \longrightarrow \mathtt{e_2'}}{\Sigma; \mathtt{unsafe\{e_1\ bop\ e_2\}} \longrightarrow \mathtt{e_1'\ bop\ e_2'}}$$

U-WHILE

$$\frac{\Sigma; \mathtt{unsafe\{e_1\}} \longrightarrow \mathtt{e_1'} \quad \Sigma; \mathtt{unsafe\{e_2\}} \longrightarrow \mathtt{e_2'}}{\Sigma; \mathtt{unsafe\{while\ e_1\ \{e_2\}\}} \longrightarrow \mathtt{while\ e_1'\ \{e_2'\}}}$$

Figure B.1: General Transformation Rules

U-ArrayIndex

$$\frac{\Sigma \vdash \mathit{Index}(\mathit{label}(\mathtt{e}_1), \_, \_) \qquad \Sigma;\ \mathtt{unsafe}\{\mathtt{e}_1\} \longrightarrow \mathtt{e}_1' \qquad \Sigma;\ \mathtt{unsafe}\{\mathtt{e}_2\} \longrightarrow \mathtt{e}_2'}{\Sigma;\ \mathtt{unsafe}\{*\mathtt{e}_1.\mathit{offset}(\mathtt{e}_2)\} \longrightarrow \mathtt{e}_1'[\mathtt{e}_2']}$$

U-DerefArray

$$\frac{\Sigma \vdash \mathit{Index}(\mathit{label}(\mathtt{e}_1), \_, \_) \qquad \Sigma;\ \mathtt{unsafe}\{\mathtt{e}_1\} \longrightarrow \mathtt{e}_1'}{\Sigma;\ \mathtt{unsafe}\{*\mathtt{e}_1\} \longrightarrow \mathtt{e}_1'[\mathtt{0}]}$$

U-VecWrapper

$$\frac{\alpha = \mathit{label}(\mathtt{IndexMutWrapper}(\mathtt{e}, \tau_{\mathtt{ind}}, \tau_{\mathtt{out}})) \qquad \Sigma \vdash \mathit{Index}(\alpha, \_, \tau_{out})}{\Sigma;\ \mathtt{IndexMutWrapper}(\mathtt{e}, \tau_{\mathtt{ind}}, \tau_{\mathtt{out}}) \longrightarrow \mathtt{Mut\ Ref\ e}}$$

U-VecMalloc

$$\frac{\alpha = \mathit{label}(\mathtt{malloc}(\mathtt{N} \cdot \mathtt{sizeof}(\tau))) \qquad \Sigma \vdash \mathit{Vec}(\alpha, \alpha_{ind}, \alpha_{out}) \qquad \Sigma \vdash \mathit{Malloc}(\alpha, \_) \qquad \Sigma \vdash \alpha_{out} \simeq \tau_{out} \qquad \tau = \tau_{out}}{\Sigma;\ \mathtt{unsafe}\ \{\mathtt{malloc}(\mathtt{N} \cdot \mathtt{sizeof}(\tau))\} \longrightarrow \mathtt{vec!}[\tau_{\mathtt{out}}; \mathtt{N}]}$$

U-VecFree

$$\frac{\alpha = \mathit{label}(\mathtt{e}) \qquad \Sigma \vdash \mathit{Vec}(\alpha, \_, \_) \qquad \Sigma \vdash \mathit{Malloc}(\alpha, \_)}{\Sigma;\ \mathtt{unsafe}\ \{\mathtt{free(e\ as\ *mut}\ \tau)\} \longrightarrow ()}$$

Figure B.2: Array-related Transformation Rules

U-UpdateVec

$$\frac{\mathtt{type}(e) = \mathit{Raw}(\tau) \qquad \mathtt{type}(\rho) = \mathtt{T} : \mathtt{IndexMut}\langle \tau_1, \mathtt{Output} = \tau \rangle}{\Sigma;\ (\mathtt{e}, \rho) \rightsquigarrow \mathtt{IndexMutWrapper(e, usize}, \tau)}$$

U-UpdateStaticArrMut

$$\frac{\mathtt{type}(e) = \mathit{Mut\ Array}(\tau) \qquad \mathtt{type}(\rho) = \mathtt{T} : \mathtt{IndexMut}\langle \tau_1, \mathtt{Output} = \tau \rangle}{\Sigma;\ (\mathtt{e.as\_mut\_ptr()}, \rho) \rightsquigarrow \mathtt{Mut\ Ref\ e}}$$

U-UpdateStaticArrImm

$$\frac{\mathtt{type}(e) = \mathit{Array}(\tau) \qquad \mathtt{type}(\rho) = \mathtt{T} : \mathtt{Index}\langle \tau_1, \mathtt{Output} = \tau \rangle}{\Sigma;\ (\mathtt{e.as\_mut\_ptr()}, \rho) \rightsquigarrow \mathtt{Ref\ e}}$$

U-UpdateCallsite

$$\frac{\mathtt{fname} = \mathtt{callname} \qquad \mathtt{fn\ callname}(\rho_1, \ldots, \rho_n) \qquad \Sigma;\ (\mathtt{e}_i, \rho_i) \rightsquigarrow \mathtt{e}_i'}{\Sigma;\ \mathtt{fname}(\mathtt{e}_1, .., \mathtt{e}_n) \underset{\text{callname}}{\longrightarrow} \mathtt{fname}(\mathtt{e}_1', .., \mathtt{e}_n')}$$

Figure B.3: Array-related Call Site Update Rules