Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

Bachelor's Thesis

RustiC, a Rust-Like Type System for Temporal Memory Safety in C

by
Yannick Schillo

submitted
July 14, 2020

Reviewers:

1. Prof. Dr. Sebastian Hack
2. Prof. Dr. Jan Reineke

Advisor:

Tina Jung, B.Sc.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

Abstract

In languages with manual memory management like C, temporal memory safety errors, such as uses after free and double frees, are often hard to track bugs that can lead to security vulnerabilities. Therefore it is desirable to enforce temporal memory safety at compile time. Since temporal memory safety is not decidable in general at compile time, solving this problem efficiently requires a combination of static analysis and run-time checks.

In this thesis, we define the type system "RustiC" for LLVM IR along with a static analysis to infer sound RustiC typings for LLVM IR. RustiC uses the concepts of ownership and borrowing inspired by the Rust programming language. These allow us to track origins of pointers locally and through function calls, and to infer the validity of pointers at compile time. This reduces the amount and complexity of run-time checks necessary to enforce temporal memory safety at run time.

In our evaluation, we tested bigger projects, including bzip2 and brotli, a data compression algorithm developed by Google. We were able to statically validate up to 83% of all dereferences of pointers, whereas in the remaining dereferences, the inferred RustiC pointer types are such that the required run-time checks can be implemented using only a few instructions.

Contents

List of Figures

1 Introduction

The C programming language is known for its manual memory management combined with unrestricted use of pointers. In C, dynamic memory has to be allocated and deallocated manually. Pointers can point to any object, including local variables on the stack and dynamically allocated objects, and can be stored in memory. This leads to possible memory safety violations, which can be categorized into *temporal* and *spatial memory safety* violations. Temporal memory safety violations include...

1. dereferencing a *dangling pointer*. A dangling pointer is a pointer to an object that no longer exists. If a dynamically allocated object is freed, all pointers to that object become dangling pointers. Dereferencing such a pointer is called a *use after free*.

   Dangling pointers can also be created by leaving a scope: If a variable goes out of scope, all pointers to that variable become dangling pointers.

2. Freeing a pointer to a dynamically allocated object after it was already freed. This is called a *double free*.

3. Freeing a pointer that does not point to a dynamically allocated object, e.g. a pointer with a non-zero offset or a pointer to a local variable. This is called an *invalid free*.

4. Using a pointer before it was initialized, e.g. when using an uninitialized variable or using uninitialized memory. We call such a pointer a *wild pointer*.

Figure 1.1 shows examples for each of these cases.

Spatial memory safety violations occur if pointers are out of bounds: Pointer arithmetic in C is unrestricted, i.e. there are no bounds checks when performing pointer arithmetic. This allows creating pointers that are out of bounds, which leads to spatial memory safety violations.

Because memory safety violations lead to undefined behaviour in C, they are bugs that are often hard to track and in some cases remain undetected for a very long time, leading to security vulnerabilties, e.g. Heartbleed [9]. Therefore, it is desirable to enforce memory safety at compile time. In our thesis we focus on temporal memory safety, therefore we assume that spatial memory safety is guaranteed by another approach.

In general, temporal memory safety is undecidable at compile time. Consider the following code snippet:

```
1 while (/* some condition */) {
2     /* some memory safe code */
```

```
}
/* any temporal memory safety violation here */
```

The code above is memory safe iff the while loop does not terminate. Therefore, deciding memory safety for this program implies deciding termination of the loop, which is undecidable in general [1].

```
void use_after_free() {
    int* buf = (int*)malloc(16 * sizeof(int));
    free(buf);
    *buf = 10; //use after free
}

int* return_local() {
    int x = 10;
    return &x;
    //After the function returns, x will no longer exists.
    //Therefore, this function always returns a dangling pointer.
}

void double_free() {
    int* buf = (int*)malloc(41 * sizeof(int));
    free(buf);
    free(buf); //double free
}

void invalid_free_local() {
    int x = 10;
    free(&x); //attempt to free a local variable
}

void invalid_free_offset() {
    int* buf = (int*)malloc(971 * sizeof(int));
    int* alias = buf + 10;
    free(alias);
    //alias does not point to the beginning of buf,
    //therefore this is an invalid free
}

void use_wild() {
    int* x;
    *x = 10; //dereferencing a wild pointer
}
```

Figure 1.1: Examples for temporal memory unsafety.

Because it is undecidable, enforcing temporal memory safety at compile time requires instrumenting the code with run-time checks. Our approach is a static analysis with the goal of allowing a code instrumentation to insert fewer and simpler run-time checks, where simpler refers to the number of instructions needed to implement this check. While we define the static analysis, we do not implement the code instrumentation, which is therefore considered future work.

In our thesis, we define the RustiC type system along with a static analysis to infer sound RustiC typings. The idea of RustiC is to track origins of pointers locally and

[1] We can use the same argument to show that spatial memory safety aswell as memory safety in general is undecidable.

through function calls, and using the ideas of ownership and borrowing from the Rust programming language to infer their validity at compile time.

Our type system computes for each node in the Control-Flow Graph (CFG) of the program a set of pointers that may be used at this node. We say that these pointers are *alive* at this node. We define that two pointers *alias* at a node if they are both alive and both pointing to the same, possibly deleted object.

The RustiC type system distinguishes different kinds of pointers:

- An *unsafe pointer* is a pretty much unrestricted pointer: it can be null, a dangling pointer, a wild pointer or pointing to a valid object. The only guarantee of unsafe pointers is that they only alias with unsafe pointers.

  As opposed to all other pointer types in RustiC, which we also call *safe pointers*, code may depend on the memory layout of unsafe pointers. This is because they allow arbitrary casts, e.g. casting between a pointer and an integer, while casting is more restricted for other pointer types.

- A *shared pointer* is a pointer that may alias with other shared pointers. Therefore, if a shared pointer is alive, it is either null, pointing to a valid object or pointing to a deleted object. In the latter case, it must have been freed using any of the aliasing shared pointer.

- A *unique pointer* is a pointer that does not alias with any unique, shared or unsafe pointer. If a unique pointer is alive, it is either null or pointing to a valid object.

- A *reference* is a pointer annotated with a *lifetime identifier*. Intuitively, a lifetime identifier identifies the origin of a pointer, e.g. by doing pointer arithmetic on a safe pointer, one can derive a reference annotated with the lifetime identifier of that safe pointer. If a reference is alive, it is either null or pointing to a valid object, while the object identified by its lifetime identifier must also be alive.

  This allows our type system to infer that references may have been invalidated, e.g. if a unique pointer is freed, it is killed, i.e. it is not alive in all successor nodes. Therefore, all references annotated with the lifetime identifier of that unique pointer are also killed, such that all references derived from that unique pointer cannot be used anymore.

RustiC allows typing function parameters and the return type of a function as references. Since the lifetime identifiers will be different for each call site in general, we abstract from all call sites using "generic lifetime identifiers". In each call to the function, we can instantiate these generic lifetime identifiers with concrete ones. The idea here is to propagate lifetime identifiers through function calls without analyzing the function at each call site. This allows our static analysis to perform inter-procedual analysis.

To further generalize over different call sites, RustiC allows "overloading" function typings, i.e. each function can have multiple typings, which we also call "function variants", and each call site can potentially use each of these variants.

If a program can be typed only using safe pointers, we can use a straight-forward instrumentation to guarantee temporal memory safety:

- If a unique pointer or a reference is used, it is either null or pointing to a valid object. Therefore, using this pointer only requires a null-pointer check.

- If a shared pointers is used, it is either null, pointing to a valid object or pointing to an object that was already freed. Therefore, using this pointer requires checking if the object, the pointer points to, still exists.

  Since code cannot depend on the memory layout of a shared pointer, this run-time check can be implemented by changing the memory layout of shared pointers, e.g. by adding metadata like a unique identifier for each allocation.

  Another approach, which is also discussed in this thesis, is using indirect pointers: For each allocated object pointed to by shared pointers, we create exactly one intermediate object, which contains a pointer to the allocated object. Shared pointers then point to the intermediate objects instead of the real object. If a shared pointer is freed, the pointer in the intermediate object is set to null. Therefore, when dereferencing a shared pointer, we load the pointer from the intermediate object. If it is null, the object has been freed. Else, it is a valid pointer to the real object.

Because unsafe pointers never alias with safe pointers, this instrumentation guarantees temporal memory safety for all uses of safe pointers even if the program contains unsafe pointers. Therefore, we can use this proposed, straight-forward instrumentation for all safe pointers and use a different approach for the remaining unsafe pointers.

If an unsafe pointer is used, it can be null, a dangling pointer, a wild pointer or pointing to a valid object. Because code can possibly depend on the memory layout of unsafe pointers, we recommend an approach that does not require changing the memory layout of pointers, e.g. CETS [13].

Our thesis is based on the LLVM compiler infrastructure. The full workflow of our approach, which is depicted in Figure 1.2, is as follows:

1. Each C source is compiled to LLVM IR without any optimization, which is done with a command like `clang -O0`.

2. We apply the `mem2reg` pass to each of the resulting IRs.

3. The resulting IR files are linked together to a single IR file, to which we apply the RustiClarify pass. This is a preparation pass for the static analysis which we define in our thesis to simplify the type system. The resulting IR of the RustiClarify pass is called the *clarified IR*.

4. The clarified IR can be analyzed with the RustiC analysis pass to get a valid RustiC typing. An instrumentation pass can use this type information to transform the clarified IR into a temporal memory safe IR.

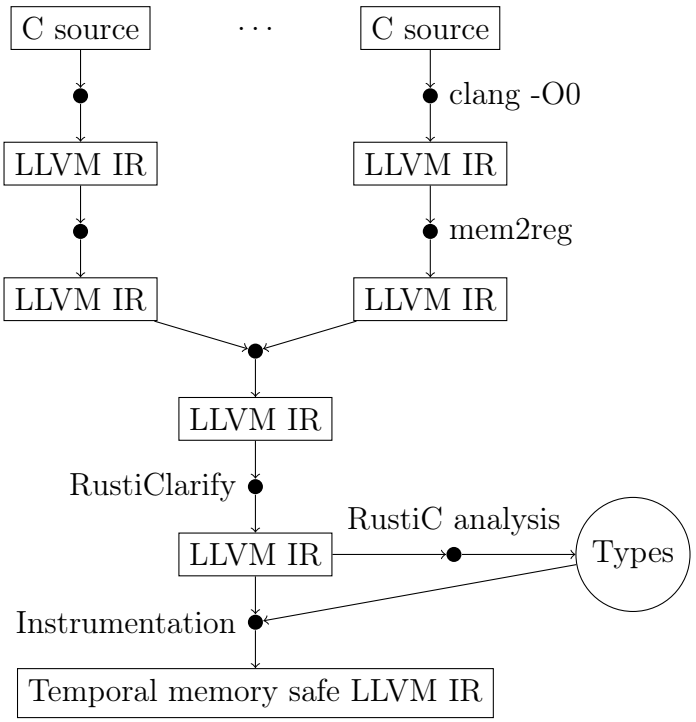

Figure 1.2: Workflow of our approach.

2 Background

2.1 LLVM

LLVM [12] is a compiler backend designed to support low-level program analysis and transformation for programs written in higher-level programming languages. It is used by compilers for a variety of languages, such as C, C++, Swift or Rust. Higher level languages are compiled into an intermediate representation defined by LLVM, called *LLVM IR*, which can be analyzed, transformed and compiled into machine code.

A compilation unit is called a *module*. It contains function declarations, function definitions, struct types and global variables. Function definitions contain *basic blocks*, which are lists of instructions that always end with a terminator instruction. A terminator instruction defines how the program is continued when the basic block was executed. It can be a (conditional) jump or a return instruction. Each function has one entry block, which is entered when the function is called.

The basic blocks of a function and their terminator instructions induce a Control-Flow Graph (CFG) for that function: Each basic block is a node and its successors are all basic blocks its terminator instruction may jump to. In our thesis, we consider each instruction in a basic block as a single node, where the successor of each instruction is the next instruction in the basic block and the successors of the terminator instruction are the first instructions of the successors of the basic block.

LLVM IR is in Static Single Assignment (SSA) form [8], which means that each variable is defined exactly once and each use of a variable is strictly dominated by its definition. To implement SSA, each basic block starts with (possibly zero) phi instructions. A phi instruction specifies one value for each predecessor block. If a basic block is entered, its phi instructions are "executed" by copying the corresponding values depending on the previous basic block.

Figure 2.1 shows a C function and an equivalent LLVM IR. The basic block `%entry` contains an unconditional jump to `%while.cond`, while `%while.cond` contains a conditional jump to either `%while.body` or `%while.end`. The example contains two phi instructions: When `%while.cond` is entered initially from `%entry`, `%a.0` is assigned `%a` and `%s.0` is assigned 0. Later, when `%while.cond` is entered from `while.body`, i.e. after the while loop repeats, `%a.0` is assigned `%inc` and `%s.0` is assigned `%add`, both of which were defined in `%while.body`.

When compiling the example above with `clang` without any optimizations, the generated LLVM IR does not contain any phi instructions. Instead, each local variable is an `alloca` instruction, which allocates a slot on the stack and returns a pointer to it. Reading and writing these local variables is done using the `load` and `store` instructions.

```
int sum(int a, int b) {
    int s = 0;
    while(a < b) {
        s += a;
        a++;
    }
    return s;
}
```

```
define i32 @sum(i32 %a, i32 %b) {
entry:
  br label %while.cond

while.cond:                                       ; preds = %while.body, %entry
  %a.0 = phi i32 [ %a, %entry ], [ %inc, %while.body ]
  %s.0 = phi i32 [ 0, %entry ], [ %add, %while.body ]
  %cmp = icmp slt i32 %a.0, %b
  br i1 %cmp, label %while.body, label %while.end

while.body:                                       ; preds = %while.cond
  %add = add nsw i32 %s.0, %a.0
  %inc = add nsw i32 %a.0, 1
  br label %while.cond

while.end:                                        ; preds = %while.cond
  ret i32 %sum.0
}
```

Figure 2.1: Example LLVM IR for a C program

The IR can be transformed into the IR from the example using the `mem2reg` pass, which tries to eliminate `alloca` instructions and replace them by local varibales and `phi` instructions. This is usually done very early in the compilation, because most analyses do overapproximations on `load` and `store` instructions, therefore eliminating `alloca`s and replacing them by local variables often results in better analysis information.

2.2 Ownership Semantics

In this section we explain the concepts of ownership and borrowing used in the Rust programming language. By now the Rust developers have not published a formalized memory model for their language. Therefore, we try to summarize the ideas given in The Rust Programming Language [11] and The Rustonomicon [6].

2.2.1 Ownership

Ownership is a relation between variables and values during the execution of a program that satisfies the following rules:

- Each variable owns at most one value.
- Each value is owned by at most one variable. This variable is also called the owner of this value.

- A variable can only be used if it owns a value.

In a language implementing ownership semantics, an assignment like `y := x` can have two meanings:

- The value that x owns is **moved** to y, which means that after executing the assignment, x no longer owns this value, but y does. This implies that this value can no longer be used by x.
- The value that x owns is **copied** to y, which means a new value "equal" to the value owned by x is created and this value is now owned by y.

The most obvious goal of ownership semantics is to prevent use after frees and double frees: Assuming that freeing a value is a move operation, a value can never be used after it has been freed and therefore it cannot be freed twice. Ownership semantics can also prevent bugs related to aliasing: consider the C snippet in Figure 2.2. This code contains a temporal memory violation: In line 3, we assign `buf` to `buf2`. The `realloc` in line 5 invalidates the pointer currently stored in `buf` and since `buf == buf2`, `buf2` is now a dangling pointer. This leads to a read from a dangling pointer in line 7.

```
1 char* buf = (char*)malloc(...);
2 //...
3 char* buf2 = buf;
4 //...
5 buf = (char*)realloc(buf, ...);
6 //...
7 printf("%s\n", buf2);
```

Figure 2.2: Example of use after free in C.

The Rust programming language implements ownership semantics as a static analysis, i.e. it ensures that each execution of the program satisfies the ownership rules. In Rust, an assignment is by default a move operation. If the type of the value implements Copy, the assignment is a copy operation. This is typically the case for scalar types like integers. A value whose type does not implement Copy can be copied with `(...).clone()` if it implements Clone (although some types neither implement Copy nor Clone).

In Rust, the example above can be written as in Figure 2.3. The Rust compiler does not compile this program because `String` is not a Copy type, therefore the assignment in line 5 is a move operation. This implies that we cannot use `buf` in line 7 because its value is now owned by `buf2`. The message „value borrowed" implies that the value is used. We discuss the term borrowing in the next section.

Ownership is not a static property: Consider the code in Figure 2.4. Whether `x` is moved depends on whether `cond` is true. Since the Rust compiler uses a static analysis for ownership semantics, it overapproximates and reject programs that would otherwise be valid, which is also the case in the example above: Assuming `cond` is immutable, `x` is never used after being moved. The analysis would overapproximate that after the

```
//the `mut` keyword makes a variable mutable
//(variables in Rust are immutable by default)
let mut buf = String::new();
//...
let mut buf2 = buf;
//...
buf.push_str("...");
//...
println!("{}", buf2);
```

(a) Rust code

```
error[E0382]: borrow of moved value: `buf`
 --> src/main.rs:6:5
  |
3 |     let mut buf = String::new();
  |         ------- move occurs because `buf` has type `std::string::String`, which
does not implement the `Copy` trait
4 |     //...
5 |     let mut buf2 = buf;
  |                    --- value moved here
6 |     //...
7 |     buf.push_str("...");
  |     ^^^ value borrowed here after move
```

(b) Error message

Figure 2.3: Erroneous Rust code due to ownership semantics.

first if statement, `x` may have been moved and therefore the compiler does not allow using it afterwards.

```
let x = ...;
if cond {
    //move x here
}
//do not use x here
if !cond {
    //use x here
}
```

Figure 2.4: Ownership is a dynamic property.

2.2.2 Borrowing

Borrowing is the term Rust uses for "creating a reference". References are pointers that have some restrictions. Rust has two types of references:

- Immutable references, which are created by `&value`
- Mutable references, which are created by `&mut value`

In both cases, we call `value` the *referent* of that reference. The Rustonomicon formulates the restrictions to references as follows:

- A reference cannot outlive its referent.
- A mutable reference cannot be aliased.

We interpret these rules as follows: We call a variable *alive* iff it owns a value. If a variable loses ownership, we say that it was *killed*. With that, we can formulate the rules above as follows:

- If a reference is alive, its referent is also alive.

  This implies that whenever a variable is killed, all references to that variable must also be killed.

- If a mutable reference is alive, it must be the only reference to its referent that is alive.

  This implies that whenever a reference is created or used, all mutable references to the same referent must be killed. If a mutable reference is created or used, all references to the same referent must be killed.

Similar to ownership semantics, these properties are not static. The Rust compiler uses a static analysis called "lifetime analysis" to ensure these properties. We briefly explain the concept of lifetimes in the next section.

To show some interesting implications of these rules, consider Figure 2.5. This is another way to implement our initial C example in Rust. Here, ownership semantics are fine, because we do not move `buf`. However, we violate the referencing rules: In line 3 we create a mutable reference to `buf`. In line 7 we read from `buf2`, which means that `buf2` has to be alive until this statement. This also implies that `buf2` is alive in line 5. But there, we implicitly create another mutable reference to `buf`, because `push_str` needs to mutate the object. This means that at this point, we have an implicit mutable reference to `buf` and the mutable reference `buf2` to `buf`, both of which are alive. This violates the referencing rules and the compiler rejects the program.

```
let mut buf = String::new();
//...
let buf2 = &mut buf;
//...
buf.push_str("...");
//...
println!("{}", buf2);
```

(a) Rust code

```
error[E0499]: cannot borrow `buf` as mutable more than once at a time
 --> src/main.rs:5:5
  |
3 |     let buf2 = &mut buf;
  |                ------- first mutable borrow occurs here
4 |     //...
5 |     buf.push_str("...");
  |     ^^^ second mutable borrow occurs here
6 |     //...
7 |     println!("{}", buf2);
  |                    ---- first borrow later used here
```

(b) Error message

Figure 2.5: Erroneous Rust code due to borrowing rules.

2.2.3 Rust Lifetimes

The lifetime analysis of the Rust compiler works by annotating each reference with a set of nodes in the CFG of the program where the reference may be used, i.e. all nodes where the reference is life. This set is called the lifetime of a reference. The Rust compiler checks that there are no conflicts in overlapping lifetimes.

Consider the code in Figure 2.6. In Rust, `Vec<T>` is an implementation of an array list. The function `push_ref` pushes a new string to a `Vec<T>` and returns a mutable refernece to it. The function is annotated with lifetime identifiers, such that the lifetime returned by the function is the same as the lifetime of the parameter `vec`.

The lifetime analysis will approximately do the following:

- `x` is used in lines 10 and 11, therefore, the lifetime of `x` is $\{10, 11\}$.
- Because of the lifetime annotations of `push_ref`, the lifetime of `&mut vec` in line 10 must also be $\{10, 11\}$.
- `y` is used in lines 13 and 14, therefore, the lifetime of `x` is $\{13, 14\}$.
- Because of the lifetime annotations of `push_ref`, the lifetime of `&mut vec` in line 13 must also be $\{13, 14\}$.
- The program is valid because the two borrows `&mut vec` do not overlap.

If line 16 were uncommented, the program would not compile, because then the lifetime of `x` and therefore also the lifetime of the first `&mut vec` would be $\{10, 11, 13, 14, 16\}$, overlapping with the lifetime of the second `&mut vec`.

```
fn push_ref<'a>(vec: &'a mut Vec<String>) -> &'a mut String
{
    let idx = vec.len();
    vec.push(String::new());
    return &mut vec[idx];
}

let mut vec = Vec::new();

let x: &mut String = push_ref(&mut vec);
x.push_str("hello");

let y: &mut String = push_ref(&mut vec);
y.push_str("world");

//x.push_str(y);
```

Figure 2.6: Propagating lifetimes through function calls.

3 Related Work

CETS [13] is a code transformation to detect all temporal memory safety violations in C programs at run time without changing the memory layout of pointers. This approach mainly focuses on efficient run-time checks while using some static analysis to eliminate unnecessary and redundant checks: checking an access to a pointer which points to a local variable on the stack or a global variable is considered an unnecessary check. A redundant check is a check of a pointer which was already checked earlier and there was no function call since the earlier check. Our approach focuses on gaining information to reduce the amount of run-time checks before they are inserted and also allowing simpler run-time checks for certain pointer types. Another feature that our type system allows is inter-procedual analysis using generic lifetime identifiers. The approach described in the CETS paper could be applied in an instrumentation pass for RustiC to check unsafe pointers at run time, since CETS does not change the memory layout of pointers.

CCured [7,14] is a type system which, similar to our approach, distinguishes different pointer types, where some require more expensive run-time checks than others. The paper also describes a type-inference algorithm. The idea of the CCured type system is to verify properties of pointers statically, which allows typing these pointers such that they only need cheaper run-time checks, where the best case is, like in our thesis, only a null-pointer check. CCured focuses on spatial safety and type safety, including checking pointer casts and dynamic downcasts using RTTI (run-time type information). However, CCured does not incooperate temporal memory safety properties, i.e. it uses a garbage collector and ignores manual deallocations to ensure temporal memory safety. While this eliminates all temporal safety issues, it may have an unpredictable impact on run-time performance and memory consumption.

Cyclone [10] is a dialect of C that, amongst other features, introduces annotations for pointers, which they call pointer qualifiers. The Cyclone compiler performs a static analysis on the code to verify temporal memory safety at compile time using the given pointer qualifiers. Cyclone heavily influenced the type system of the Rust programming language: Cyclones region analysis to prevent dangling pointers lead to Rusts lifetime analysis, which inspired the lifetime analysis of our type system. Since Cyclone is a dialect of C, using the safety features of Cyclone requires porting the code from C to Cyclone. One goal of our approach is to analyze existing C code without any modifications.

4 RustiC

In this chapter, we define the RustiC type system for LLVM modules. We call a typing of an LLVM module a *RustiC type assignment*. In type assignments, values are assigned to types from the RustiC type system. To distinguish these types from LLVM's type system, we use the term "type" for RustiC types and the term "LLVM type" for types from LLVM's type system.

One component of the RustiC type system is the *lifetime analysis*, which implements ownership semantics: it computes which values are alive at each instruction. The lifetime analysis depends on the assigned types, but the type system also depends on the result of the lifetime analysis, because it requires that the operands used in instructions are alive.

We formalize our type system as follows: Each typing rule for an instrucion defines a set of values that are moved in this instruction and a set of *post conditions*. Given a set of typing rules the lifetime analysis uses the given sets of moved values to compute for each instruction which values are alive at that instruction. After performing lifetime analysis, we check the post conditions of the used typing rules, which may use the results of lifetime analysis. For most of the typing rules, the post conditions require that the operands used in the instruction are alive at that instruction.

4.1 Types

Figure 4.1 shows the types of the RustiC type system that are assigned to LLVM values. It includes `!`, the *never type*, which is never used in type assignments. It is only used to formalize the static analysis. The type system defines different kinds of pointers (where α is a lifetime idenifier):

- `*`T is an *unsafe pointer* to T.
- `Shared<`T`>` is a *shared pointer* to T.
- `Unique<`T`>` is a *unique pointer* to T.
- `&`α T is a *strong reference* to T annotated with α.
- `&`α `mut` T is a *strong mutable reference* to T annotated with α.
- `?`α T is a *weak reference* to T annotated with α.

In all cases, we call T the *target* of the pointer. We define the following notations:

$$
\begin{aligned}
\mathit{base} &::= \mathrm{i}n \qquad (n \in \mathbb{N}) \\
&\mid \text{void} \\
&\mid \text{struct } \mathit{name} \\
&\mid \texttt{!} \\
\mathit{unsafe} &::= \texttt{*}\ \mathit{unsafe} \\
&\mid \mathit{base} \\
\mathit{ptr} &::= \texttt{Shared<}\mathit{ptr}\texttt{>} \\
&\mid \texttt{Unique<}\mathit{ptr}\texttt{>} \\
&\mid \mathit{unsafe} \\
\mathit{type} &::= \texttt{\&}\ \alpha\ \mathit{ptr} \mid \texttt{\&}\ \alpha\ \texttt{mut}\ \mathit{ptr} \\
&\mid \texttt{?}\ \alpha\ \mathit{ptr} \\
&\mid \mathit{ptr}
\end{aligned}
$$

Figure 4.1: Types in the RustiC type system. α is a lifetime identifier.

- $\texttt{*}T$, $\texttt{Shared<}T\texttt{>}$ and $\texttt{Unique<}T\texttt{>}$ are *owned pointers*.
- $\texttt{\&}\alpha\ T$, $\texttt{\&}\alpha\ \texttt{mut}\ T$ and $\texttt{?}\alpha\ T$ are *references* annotated with α.
- $\texttt{*}T$ is an *unsafe pointer*, all other pointers are *safe pointers*.
- All non-pointer types are called *base types*.

The grammar does not specify the syntax of lifetime identifiers. We use greek letters to denote general lifetime identifiers and for practical examples with explicit lifetime identifiers, we name them with a leading apostrophe to differentiate them from other identifiers. We assume the existence of a lifetime identifier `'static`, which is used for global variables.

We start off by giving an informal description of the properties of pointers. We use the ideas of ownership given in Chapter 2. Further, we say that two pointers alias during the execution of the program iff they are both alive and they both point to the same object. We consider each pointer in memory as alive, such that when a pointer is loaded from memory using the `load` instruction, it always aliases with the pointer in memory (except if it is a null pointer).

Owned pointers can be used for deallocation, i.e. they can be passed to a deallocation function like `free`. Besides that, owned pointers have the following properties:

- A unique pointer is either null or pointing to a valid object as long as it is alive. It cannot be implicitly copied, therefore it does not alias with any other owned pointer.

- A shared pointer is either null, pointing to a valid object or pointing to a deleted object. It can be implicitly copied and if it aliases with an owned pointer, that pointer must be a shared pointer. Therefore, if a shared pointer points to a deleted object, it must have been freed using any of the aliasing shared pointers.

- An unsafe pointer is pretty much unrestricted: it can be null, a dangling pointer, a wild pointer or pointing to a valid object. It can be copied implicitly and it cannot alias with safe pointers.

  As opposed to all other pointer types, code may depend on the memory layout of unsafe pointers. This is because they allow arbitrary casts, e.g. casting between a pointer and an integer, while casting is more restricted for safe pointers.

Note that until now, we considered ownership, which defines whether a value is alive, as well as aliasing as dynamic properties. Since these are not decidable in general, we define a static notion of being alive that underapproximates the dynamic notion: whenever a value is statically alive, it must be dynamically alive in all executions. Therefore, if a value is statically alive, all properties related to aliasing stated before hold in all executions of the program. Since we only use the static notion in the rest of the thesis, when we say that a value is alive, we mean that it is statically alive.

In RustiC, all values have a unique lifetime identifier. The lifetime analysis computes which lifetime identifiers are alive at each instruction. We say that a value is alive at an instruction iff its lifetime identifier is alive at that instruction. We use this to implement ownership semantics and Rust-like referencing rules: If a value is moved to another value, the lifetime identifier of the old value is killed. If a reference has to be invalidated to satisfy referencing rules, the lifetime identifier of that reference is killed. Intuitively, we do the opposite of what is done in Rust: In Rust, references are annotated with a set of nodes in the CFG where the reference may be used, in RustiC, each node is annotated with a set of values that may be used at that node.

References are always derived from other safe pointers, e.g. by doing pointer arithmetic with a `getelementptr` instruction on a safe pointer, one can derive a reference that is annotated with the lifetime identifier of the safe pointer. Therefore, the annotated lifetime identifier of a reference identifies its origin. Note that there are two different lifetime identifiers for each reference: if a value v is typed as a reference, v has a unique lifetime identifier and the reference type is annotated with a different lifetime identifier. References cannot be stored in memory, but can be loaded from memory, e.g. we can load a `&`α T from a value v of type `&`β `Unique<`T`>`, where α is the lifetime identifier of v.

If a reference is alive, it is either null or pointing to a valid object, and its annotated lifetime identifier must be alive. This allows our type system to infer that references may have been invalidated, e.g. if a unique pointer is freed, its lifetime identifier is killed. Therefore, all references annotated with the lifetime identifier of that unique pointer are also killed, such that all references derived from that unique pointer cannot be used anymore. This is implemented as a must analysis, i.e. if a value is alive, it

must be valid and if it is not alive, it may be invalid. The different kinds of references have the following restrictions:

- A strong reference is always derived or loaded from a unique pointer or a strong (mutable) reference.
- A strong mutable reference is always derived or loaded from a unique pointer or a strong mutable reference. Also, if a `&`α `mut` T is alive, there is no other strong (mutable) reference to α that is alive.
- A weak reference can be derived and loaded from all safe pointers.

Whenever a reference is derived from an owned pointer, we create a reference that aliases with an owned pointer. We generalize this concept as *implicit downcasting*: Whenever an instruction creates an alias of a pointer and the resulting pointer is a different kind of pointer, we say that this instruction performs an implicit downcast. Implicit downcasts are relevant for code instrumentation, since they may require additional run-time checks. They happen right before the instruction, e.g. if a `getelementptr` instruction on a shared pointer returns a weak reference, the shared pointer is implicitly downcasted to a weak reference, to which the `getelementptr` instruction is applied. This is further discussed in Section 4.8.

4.2 Type Assignments

Types assigned to LLVM values have to "match" the LLVM types of these values. Formally, we define *compatibility* from types to LLVM types: Let T be a type and L be an LLVM type.

- For all $n \in \mathbb{N}$, `i`n and `void` are compatible to the syntactically identical LLVM types.
- `struct` *name* is compatible to the LLVM type *name*, if *name* is defined as a struct in the module.
- `*`T, `Shared<`T`>`, `Unique<`T`>`, `&`α T, `&`α `mut` T and `?`α T are compatible to $L*$ if T is compatible to L.
- `*`T, `Shared<`T`>` and `Unique<`T`>` are compatible to `[`n `x` L`]` [1] if T is compatible to L.
- `!` is not compatible to any LLVM type.

A RustiC type assignment $\mathcal{T}$ for an LLVM module fulfills the following properties:

[1] In LLVM, `[`n `x` L`]` is an array of n elements of type L

- For each global variable g, $\mathcal{T}(g)$ is either a shared or an unsafe pointer that is compatible to the LLVM type of g [2].

- For each struct s whose members have LLVM types $L_1, \ldots, L_n$, $\mathcal{T}(s)$ is a tuple $(T_1, \ldots, T_n)$ such that for each $i \in \{1, \ldots, n\}$, T_i is a type compatible to L_i and if T_i is a pointer type, it must be an owned pointer.

- For each function f, $\mathcal{T}(f)$ is a set of function variants.

A *function variant* for a function f is a tuple (G, I, t, R, A), where:

- G is a finite set of lifetime identifiers. We call them *generic lifetimes identifiers*.

- I is a mapping from all values in f to non-generic lifetime identifers, such that:
  - $I(v) = \texttt{'static} \Leftrightarrow v$ is a constant value
  - For all non-constant values $v, v' : v \neq v' \Rightarrow I(v) \neq I(v')$

  These conditions ensure that if $I(v) = \alpha$ for some $\alpha \neq \texttt{'static}$, v is the only value with $I(v) = \alpha$. In that case, we call v *the value associated to* α. We call each lifetime identifier that has an associated value a *local lifetime identifier*.

- t is a mapping from all values in f to compatible types. For all function parameters p of f, all lifetime identifiers in $t(p)$ must be `'static` or a generic lifetime. For all instructions i in f, if $t(i)$ contains a lifetime identifier α, one of the following conditions must hold:
  - $\alpha = \texttt{'static}$,
  - α is a generic lifetime identifier,
  - the value associated to α is an instruction that strictly dominates i. We assume that function parameters dominate instructions, therefore the value associated to α may also be a function parameter.

  We treat each use of a constant value as a different value, e.g. each time the null pointer constant is used, it can have a different type.

- R is a type that is compatible to the return LLVM type of f. All lifetime identifiers in R are either `'static` or must be generic lifetimes that are used by the type of at least one function parameter.

- $A \subseteq \{\textit{freeunsafe}\}$ is a set of attributes. If $\textit{freeunsafe} \in A$, we say that the variant is *freeunsafe*, else it is *freesafe*.

[2] Note that in LLVM, all global variables are pointers

Given a type assignment $\mathcal{T}$ and a function variant $\tilde{f} = (G, I, t, R, A) \in \mathcal{T}(f)$, we define the following notations:

$$\mathcal{T}_{\tilde{f}}(v) := \begin{cases} t(v), & \text{if } v \text{ is a value defined in } f. \\ \mathcal{T}(v), & \text{else} \end{cases}$$

$$\mathcal{I}_{\tilde{f}}(v) := \begin{cases} I(v), & \text{if } v \text{ is a value defined in } f, \\ \texttt{'static}, & \text{else} \end{cases}$$

Note that for each function variant, there are three disjunct kinds of lifetime identifiers:

- `'static`, which is the lifetime identifier of all global variables and constants,
- Generic lifetime idenfifiers, defined by G, and
- Local lifetime identifiers, which are associated to a value.

The type system is defined for function variants, i.e. we check each function variant independently. Therefore, in the rest of this chapter, let f be a function and $\tilde{f} = (G, I, t, R, A) \in \mathcal{T}(f)$. As a shorthand, we write $\mathcal{T}(v)$ for $\mathcal{T}_{\tilde{f}}(v)$ and $\mathcal{I}(v)$ for $\mathcal{I}_{\tilde{f}}(v)$.

We now give a running example on which we demonstrate different aspects of our type system. Figure 4.2 shows a C program, the corresponding clarified LLVM IR and a valid type assignment. Note that the *clarified* IR is the IR resulting from the RustiClarify pass, which is discussed in Chapter 5.

The syntax `fn idx<'A>(...) -> ?'A i32` describes a function variant of `idx`, where `'A` is a generic lifetime identifier and the return type is `?'A i32`. For each value, its local lifetime identifier is annotated on the left and its type is annotated on the right, e.g. the function parameter `%arr` has the lifetime identifier `'a` and the type `?'A i32`. To make the example more clear, it only includes the local lifetime identifiers of pointer typed values.

In the function `idx` we perform pointer arithmetic on `%arr` with the `getelementptr` instruction. The resulting value `%add.ptr` is a `?'a i32`, because the lifetime identifier of `%arr` is `'a`. After that, we return `%add.ptr`, which is annotated with the local `'a`, whereas the return type of this variant is annotated with `'A`. This is only possible if `'a` can be "backtracked" to `'A`, which is the case here: `'a` is associated to the function parameter `%arr` and `%arr` is annotated with `'A`. We also say that `'A` is the *origin* of `'a`. This is further discussed in Section 4.3.

In `f`, there are calls to RustiC intrinsics like `rustic.alloc`, which are inserted by the RustiClarfy pass. They are functions known to the RustiC type system, which means they have special typing rules. We discuss them in more detail in section 4.5. The instrinsics `rustic.alloc` and `rustic.free` wrap calls to `malloc` and `free` combined with a `bitcast`. Therefore, `rustic.alloc` can return any owned pointer and `rustic.free` can take any owned pointer as an argument. The `rustic.weaken` intrinsic performs an implicit downcast in the type system. In this example, the `rustic.weaken` instruction casts a `Unique<i32>` to a `?'a` $i32$. When calling `idx` in `f`, we instantiate the function variant of `idx` by replacing the generic lifetime identifier `'A` with the local lifetime identifier `'a`.

```c
int* idx(int* arr, int idx) {
    return arr + idx;
}

void f() {
    int* x = (int*)malloc(10 * sizeof(*x));
    int* alias = idx(x, 4);
    *alias = 10;
    free(x);
}
```

(a) C source

```
define i32* @idx(i32* %arr, i32 %idx) {
entry:
  %idx.ext = sext i32 %idx to i64
  %add.ptr = getelementptr inbounds i32, i32* %arr, i64 %idx.ext
  ret i32* %add.ptr
}

define void @f() {
entry:
  %x = call i32* @rustic.alloc.0(i64 40)
  %x1 = call i32* @rustic.weaken.0(i32* %x)
  %alias = call i32* @idx(i32* %x1, i32 4)
  store i32 10, i32* %alias, align 4
  call void @rustic.free.0(i32* %x)
  ret void
}
```

(b) Clarified IR

```
fn idx<'A>(
'a:     i32* %arr: ?'A i32
        i32 %idx: i32
) -> ?'A i32 {
    entry:
        %idx.ext = sext i32 %idx to i64: i64
'b:     %add.ptr = getelementptr inbounds i32, i32* %arr, i64 %idx.ext: ?'a i32
        ret i32* %add.ptr: void
}

fn f() -> void {
    entry:
'a:     %x = call i32* @rustic.alloc.0(i64 40): Unique<i32>
'b:     %x1 = call i32* @rustic.weaken.0(i32* %x): ?'a i32
'c:     %alias = call i32* @idx(i32* %x1, i32 4): ?'a i32 (?'a i32, i32)
        store i32 10, i32* %alias, align 4: void
        call void @rustic.free.0(i32* %x): void
        ret void: void
}
```

(c) Type assignment

Figure 4.2: Example for type assignment.

Note that unique pointers cannot be copied, therefore the call to `rustic.free` moves `%x`, such that `%x` and all references derived from it cannot be used anymore. This includes the values `%x1` and `%alias`. If one of these values would be used after the call to `rustic.free`, we could not type them as references. This is guaranteed by our lifetime analysis, which is described in Section 4.4.

4.3 Origin Ordering

We define $\preceq$ to be the smallest partial ordering such that for all values v:

- if $\mathcal{T}(v)$ is a reference annotated with some α, then $\mathcal{I}(v) \preceq \alpha$.

Let α be a local lifetime identifier and $v_1, \ldots, v_n$ a maximal chain of values, such that

- $\mathcal{I}(v_1) = \alpha$,
- v_i is a reference annotated with $\mathcal{I}(v_{i+1})$ for $1 \leq i < n$,

Note that for $i \in \{1, \ldots, n\}$, v_i has to be strictly dominated by v_{i+1}, therefore there exists such a maximal chain. This chain allows us to compute the set $\{\beta \mid \alpha \preceq \beta\}$. There are two cases:

- v_n is an owned pointer. In that case,

$$\{\beta \mid \alpha \preceq \beta\} = \{\mathcal{I}(v_1), \ldots, \mathcal{I}(v_n)\}.$$

- v_n is a reference annotated with a lifetime identifier β. In that case,

$$\{\beta \mid \alpha \preceq \beta\} = \{\mathcal{I}(v_1), \ldots, \mathcal{I}(v_n), \beta\}.$$

  Since the chain $v_1, \ldots, v_n$ is maximal, there is no value associated to β. Therefore, β must be a generic lifetime identifier or `'static`.

Note that in both cases, the set $\{\beta \mid \alpha \preceq \beta\}$ is totally ordered. We call the maximum of this set the *origin* of α:

$$\mathit{origin}(\alpha) := \max\{\beta \mid \alpha \preceq \beta\}.$$

The origin of a lifetime can either be the lifetime identifier of an owned pointer, a generic lifetime identifier, or `'static`.

In our running example in Figure 4.2, we have `'b` $\prec$ `'a` $\prec$ `'A` in `idx`, therefore origin(`'b`) = `'A`. In `f`, we have `'c` $\prec$ `'a`, where `'a` is associated to an owned pointer, therefore origin(`'c`) = `'a`.

4.4 Lifetime Analysis

The lifetime analysis is based on the CFG of f, where we consider each instruction in a basic block as a single node. Informally, the analysis works as follows:

- If a value is defined, its lifetime identifier is instantiated at this node, i.e. it is alive in all successors.
- The lifetime of each value in $move_i$ is *killed* at i, i.e. it is removed from the lifetime set of all successors.
- If a node instantiates a $\texttt{\&}\alpha\ T$, all strong mutable references annotated with α are killed at that node.
- If a node instantiates a $\texttt{\&}\alpha\ \texttt{mut}\ T$, all strong references and strong mutable references annotated with α are killed at that node.
- If a lifetime is killed at a node, each strong and weak reference annotated with that lifetime is also killed at that node.
- All other lifetimes are preserved iff they are in the lifetime set of all predecessors.

We denote the set of all instructions in f by $inst(f)$. For all $i \in inst(f)$, we define $kill_i$ as follows:

$$kill_i := \bigcup_{j \in move'_i} \{\alpha \mid \alpha \preceq \mathcal{I}(j)\},$$

where

$$move'_i := \begin{cases} move_i \cup use_{\texttt{\&mut}}(\alpha) \cup use_{\texttt{\&}}(\alpha), & \text{if } \mathcal{T}(i) = \texttt{\&}\alpha\ \texttt{mut}\ T \\ move_i \cup use_{\texttt{\&mut}}(\alpha), & \text{if } \mathcal{T}(i) = \texttt{\&}\alpha\ T \\ move_i, & \text{else,} \end{cases}$$

$move_i$ is a set given by the typing rule of the instruction and

$$use_{\texttt{\&}}(\alpha) := \{i \in inst(f) \mid \mathcal{T}(i) = \texttt{\&}\alpha\ T \text{ for some type } T\}$$
$$use_{\texttt{\&mut}}(\alpha) := \{i \in inst(f) \mid \mathcal{T}(i) = \texttt{\&}\alpha\ \texttt{mut}\ T \text{ for some type } T\}$$

The lattice for the lifetime analysis is the powerset of all lifetime identifiers with the partial ordering $S \sqsubseteq T :\Leftrightarrow S \supseteq T$, such that the most precise information is the set of all lifetime identifiers. For each $i \in inst(f)$ we define:

$$f_i(L) := (L \setminus kill_i) \cup \{I(i)\},$$

where L is a set of lifetime identifiers. Our analysis computes the least fixpoint, i.e. the greatest subset of lifetime identifiers, of the following system of equations:

$$\mathcal{L}(i_0) \subseteq \{\texttt{'static}\} \cup G \cup \{I(a) \mid a \in args(f)\}$$
$$\mathcal{L}(i) \subseteq \bigcap_{j \in pred(i)} f_j(\mathcal{L}(j)) \qquad \text{for all } i \in inst(f) \setminus \{i_0\}$$

```
1  fn idx<'A>(
2  'a:      i32* %arr: ?'A i32
3           i32 %idx: i32
4  ) -> ?'A i32 {
5      entry:
6           {'a}
7           %idx.ext = sext i32 %idx to i64: i64
8           {'a}
9  'b:      %add.ptr = getelementptr inbounds i32, i32* %arr, i64 %idx.ext: ?'a i32
10          {'a, 'b}
11          ret i32* %add.ptr: void
12 }
13
14 fn f() -> void {
15     entry:
16          {}
17 'a:      %x = call i32* @rustic.alloc.0(i64 40): Unique<i32>
18          {'a}
19 'b:      %x1 = call i32* @rustic.weaken.0(i32* %x): ?'a i32
20          {'a, 'b}
21 'c:      %alias = call i32* @idx(i32* %x1, i32 4): ?'a i32 (?'a i32, i32)
22          {'a, 'b, 'c}
23          store i32 10, i32* %alias, align 4: void
24          {'a, 'b, 'c}
25          call void @rustic.free.0(i32* %x): void
26          {}
27          ret void: void
28 }
```

Figure 4.3: Example for lifetime analysis.

where $pred(i)$ denotes the set of predecessors of i, i_0 is the first instruction in the entry block of f and $args(f)$ are the arguments of f. We say that a value v is *alive* at an instruction i iff $\mathcal{I}(v) \in \mathcal{L}(i)$.

The result of the lifetime analysis on our running example are shown in Figure 4.3. We omit the local lifetime identifiers of all non-pointer values because we only care about pointer-typed values [3]. The lifetime set for each instruction is written above the instruction. Note that the call to `rustic.free` kills `'a`, `'b` and `'c`. This is because freeing a unique pointer "moves" that pointer, killing its lifetime identifier `'a` and therefore also `'b` and `'c` because they both originate from `'a`. This prevents using any of the pointers that were just freed after the call to `rustic.free`, i.e. if any of these would be used after the call, this would not be a valid typing.

4.5 Type System

In this section we define typing rules for all instructions that potentially use or do arithmetic with pointers and whose types can be represented by our type system (i.e. that have compatible types). We refer to them as *relevant* instructions. Each typing rule contains:

[3] Formally, non-pointer typed values are never killed by our type system and their aliveness is never checked by the typing rules, making it unnecessary to consider whether their lifetime identifiers.

- a type for the value if the instruction defines a value,
- a set of conditions,
- a set of values that are moved, i.e. $move_i$,
- another set of conditions, which we call *post conditions*.

A function variant is *valid* iff there exists a matching typing rule for each relevant instruction that satisfies all conditions and after performing lifetime analysis using those typing rules, satisfy all post conditions. A type assignment is *valid* iff all its variants are valid.

In the following, let T and U be types, $L, L_1, L_2, \ldots$ be LLVM types and $\alpha, \beta, \beta_1, \beta_2, \ldots$ be lifetimes. Note that the typing rules do not give a relation between the types T, U and the LLVM types $L, L_1, L_2, \ldots$. This is because this relation is implicitly given by the fact that the types assigned to values must be compatible to their LLVM types, as stated in the definition of function variants in Section 4.2.

4.5.1 Alloca Instruction

$$val = \texttt{alloca}\ L$$

1. $\mathcal{T}(val) = \texttt{*}T$.
2. $\mathcal{T}(val) = \texttt{Shared<}T\texttt{>}$.
3. $\mathcal{T}(val) = \texttt{Unique<}T\texttt{>}$.

   Post condition: *val* is never moved.

4.5.2 Load Instruction

$$val = \texttt{load}\ L,\ L\texttt{*}\ ptr$$

1. $\mathcal{T}(val) = \texttt{i}n$ for some $n \in \mathbb{N}$.

   Post condition: *ptr* is alive.
2. $\mathcal{T}(val) = \texttt{*}T$.

   Condition: $\mathcal{T}(ptr)$ is a pointer to $\texttt{*}T$.

   Post condition: *ptr* is alive.
3. $\mathcal{T}(val) = \texttt{Shared<}T\texttt{>}$.

   Condition: $\mathcal{T}(ptr)$ is a safe pointer to $\texttt{Shared<}T\texttt{>}$.

   Post condition: *ptr* is alive.

4. $\mathcal{T}(val) = \texttt{\&}\alpha\ \texttt{mut}\ T$.

   Conditions:

   - $\alpha \succeq \mathcal{I}(ptr)$.
   - $\mathcal{T}(ptr)$ is one of `Unique<Unique<`T`>>` or `&`β `mut Unique<`T`>`.

   Post condition: *ptr* is alive.

5. $\mathcal{T}(val) = \texttt{\&}\alpha\ T$.

   Conditions:

   - $\alpha \succeq \mathcal{I}(ptr)$.
   - $\mathcal{T}(ptr)$ is one of `Unique<Unique<`T`>>`, `&`β `mut Unique<`T`>` or `&`β `Unique<`T`>`.

   Post condition: *ptr* is alive.

6. $\mathcal{T}(val) = \texttt{?}\alpha\ T$.

   Conditions:

   - $\alpha \succeq \mathcal{I}(ptr)$.
   - $\mathcal{T}(ptr)$ is a safe pointer to a safe pointer to T.

   Post condition: *ptr* is alive.

4.5.3 Store Instruction

`store` L *val*, L`*` *ptr*

1. Condition: $\mathcal{T}(val) = \texttt{i}n$ for some $n \in \mathbb{N}$.

   Post condition: *ptr* is alive.

2. Conditions:

   - $\mathcal{T}(ptr)$ is a pointer to $*T$.
   - $\mathcal{T}(val) = *T$.

   Post condition: *ptr* and *val* are alive.

3. Conditions:

   - $\mathcal{T}(ptr)$ is a safe pointer to `Shared<`T`>`.
   - $\mathcal{T}(val)$ is one of `Shared<`T`>` or `Unique<`T`>`.

   Moves: *val* if it is a unique pointer.

   Post condition: *ptr* and *val* are alive.

4. Conditions:

   - $\mathcal{T}(ptr)$ is a safe pointer to `Unique<`T`>`.

- $\mathcal{T}(val) = \texttt{Unique<}T\texttt{>}$.

Moves: *val*.

Post condition: *ptr* and *val* are alive.

4.5.4 GetElementPtr Instruction

$$res = \texttt{getelementptr}\ L,\ L\texttt{*}\ ptr,\ idx_0,\ idx_1,\ \ldots,\ idx_n$$

where $idx_0, \ldots, idx_n$ are integer values. We define a series of types $T_0, \ldots, T_n$:

- Because the LLVM type of *ptr* is always a pointer, $\mathcal{T}(ptr)$ must be a pointer. We define T_0 to be the target type of that pointer.
- For $1 \leq i \leq n$:
  - If $\mathcal{T}(T_{i-1}) = (t_1, \ldots, t_n)$ is a struct type, then $T_i := t_{idx_i}$. Note that this is well-defined because LLVM requires idx_i to be an integer constant when indexing struct members.
  - If $\mathcal{T}(T_{i-1})$ is a pointer, we define T_i to be the target type of that pointer. This is the case if the corresponding LLVM type is an array, which must be represented by a pointer type, because these are the only ones compatible to arrays in LLVM.

T_n is the target type of all pointers returned by the following typing rules.

1. $\mathcal{T}(res) = \texttt{*}T_n$.

   Conditions:

   - $\mathcal{T}(ptr) = \texttt{*}T_0$.

   Post condition: *ptr* is alive.

2. $\mathcal{T}(res) = \texttt{Shared<}T_n\texttt{>}$.

   Conditions:

   - $\mathcal{T}(ptr)$ is one of $\texttt{Shared<}T_0\texttt{>}$ or $\texttt{Unique<}T_0\texttt{>}$.

   Moves: *ptr* if it is a unique pointer.

   Post condition: *ptr* is alive.

3. $\mathcal{T}(res) = \texttt{Unique<}T_n\texttt{>}$.

   Conditions:

   - $\mathcal{T}(ptr) = \texttt{Unique<}T_0\texttt{>}$.

   Moves: *ptr*.

   Post condition: *ptr* is alive.

4. $\mathcal{T}(res) = \texttt{\&}\alpha\ \texttt{mut}\ T_n$.

   Conditions:

   - $\alpha \succeq \mathcal{I}(ptr)$.
   - $\mathcal{T}(ptr)$ is one of $\texttt{Unique<}T_0\texttt{>}$ or $\texttt{\&}\beta\ \texttt{mut}\ T_0$.

   Post condition: *ptr* is alive.

5. $\mathcal{T}(res) = \texttt{\&}\alpha\ T_n$.

   Conditions:

   - $\alpha \succeq \mathcal{I}(ptr)$.
   - $\mathcal{T}(ptr)$ is one of $\texttt{Unique<}T_0\texttt{>}$, $\texttt{\&}\beta\ \texttt{mut}\ T_0$ or $\texttt{\&}\beta\ T_0$.

   Post condition: *ptr* is alive.

6. $\mathcal{T}(res) = \texttt{?}\alpha\ T_n$.

   Conditions:

   - $\alpha \succeq \mathcal{I}(ptr)$.
   - $\mathcal{T}(ptr)$ is a safe pointer to T_0.

   Post condition: *ptr* is alive.

4.5.5 Bitcast Instruction

$$res = \texttt{bitcast}\ L\ val\ \texttt{to}\ L'$$

1. $\mathcal{T}(res) = *T$.

   Condition: $\mathcal{T}(val) = *U$.

   Post condition: *val* is alive.

   We call a bitcast an *unsafe bitcast* if it uses this typing rule. Any other bitcast is called a *safe bitcast*.

2. $\mathcal{T}(res) = T$.

   Condition: $\mathcal{T}(val) = T$.

   Moves: *val* if $\mathcal{T}(val)$ is a unique pointer.

   Post condition: *val* is alive.

3. $\mathcal{T}(res)$ is a safe pointer.

   Condition:

   - $\mathcal{T}(val)$ is safe pointer of the same kind as $\mathcal{T}(res)$, which means that they are equal up to their target type.
   - L and L' are pointers to types which contain no pointer.

   Moves: *val* if $\mathcal{T}(val)$ is a unique pointer.

   Post condition: *val* is alive.

4.5.6 Inttoptr Instruction

$$res = \texttt{inttoptr i}n\ val\ \texttt{to}\ L$$

1. $\mathcal{T}(res)$ is an unsafe pointer.

4.5.7 Ptrtoint Instruction

$$res = \texttt{ptrtoint}\ L\ ptr\ \texttt{to i}n$$

1. $\mathcal{T}(ptr) = \texttt{i}n$.

   Condition: $\mathcal{T}(ptr)$ is an unsafe pointer.

   Post condition: *ptr* is alive.

2. $\mathcal{T}(ptr) = \texttt{i}n$.

   Conditions:

   - $\mathcal{T}(ptr)$ is a safe pointer.
   - The LLVM module does not contain any unsafe bitcast.
   - The LLVM module does not contain any `inttoptr` instruction.

   Post condition: *ptr* is alive.

4.5.8 Return Instruction

$$\texttt{ret}\ L\ val$$

Recall that R is the return type of the current function variant $\tilde{f}$, as defined at the end of Section 4.2.

1. Condition: $\mathcal{T}(val) = \texttt{i}n$ for some $n \in \mathbb{N}$.

2. Conditions:

   - $R = *T$.
   - $\mathcal{T}(val) = *T$.

   Post condition: *val* is alive.

3. Conditions:

   - $R = \texttt{Shared<}T\texttt{>}$.
   - $\mathcal{T}(val)$ is one of $\texttt{Shared<}T\texttt{>}$ or $\texttt{Unique<}T\texttt{>}$.

   Moves: *val* if it is a unique pointer.

   Post condition: *val* is alive.

4. Conditions:

- $R =$ `Unique<`T`>`.
- $\mathcal{T}(val) =$ `Unique<`T`>`.

Moves: val.

Post condition: val is alive.

5. Conditions:
   - $R =$ `&`α `mut` T.
   - $\mathcal{T}(val) =$ `&`β `mut` T.
   - $origin(\beta) = \alpha$.

   Post condition: val is alive.

6. Conditions:
   - $R =$ `&`α T.
   - $\mathcal{T}(val)$ is one of `&`β `mut` T or `&`β T.
   - $origin(\beta) = \alpha$.

   Post condition: val is alive.

7. Conditions:
   - $R =$ `?`α T.
   - $\mathcal{T}(val)$ is one of `&`β `mut` T, `&`β T or `?`β T.
   - $origin(\beta) = \alpha$.

   Post condition: val is alive.

4.5.9 Phi Instruction

$$res = \texttt{phi}\ L\ [val_1, lbl_1], \ldots, [val_n, lbl_n]$$

where for $1 \leq i \leq n$, val_i is an LLVM value and lbl_i is the label of a basic block in f. For $1 \leq i \leq n$, let br_i be the branch instruction in lbl_i.

1. $\mathcal{T}(res) =$ `i`n for some $n \in \mathbb{N}$.
2. $\mathcal{T}(res) =$ `*`T.

   Conditions:
   - $\forall i, 1 \leq i \leq n : \mathcal{T}(val_i) =$ `*`T.

   Post condition: $\forall i, 1 \leq i \leq n : val_i$ is alive at br_i.

3. $\mathcal{T}(res) =$ `Shared<`T`>`.

   Conditions:

- $\forall i, 1 \leq i \leq n : \mathcal{T}(val_i)$ is one of `Shared<`T`>` or `Unique<`T`>`.

Moves: $\forall i, 1 \leq i \leq n : val_i$ if it is a unique pointer.

Post condition: $\forall i, 1 \leq i \leq n : val_i$ is alive at br_i.

4. $\mathcal{T}(res) =$ `Unique<`T`>`.

   Conditions:

   - $\forall i, 1 \leq i \leq n : \mathcal{T}(val_i) =$ `Unique<`T`>`.

   Moves: $\forall i, 1 \leq i \leq n : val_i$.

   Post condition: $\forall i, 1 \leq i \leq n : val_i$ is alive at br_i.

5. $\mathcal{T}(res) =$ `&`α `mut` T.

   Conditions:

   - $\forall i, 1 \leq i \leq n : \mathcal{T}(val_i) =$ `&`β_i `mut` T.
   - $\forall i, 1 \leq i \leq n : origin(\beta_i) \neq$ `'static` $\Rightarrow \alpha \succeq \beta_i$

   Post condition: $\forall i, 1 \leq i \leq n : val_i$ is alive at br_i.

6. $\mathcal{T}(res) =$ `&`α T.

   Conditions:

   - $\forall i, 1 \leq i \leq n : \mathcal{T}(val_i)$ is one of `&`β_i `mut` T or `&`β_i T.
   - $\forall i, 1 \leq i \leq n : origin(\beta_i) \neq$ `'static` $\Rightarrow \alpha \succeq \beta_i$

   Post condition: $\forall i, 1 \leq i \leq n : val_i$ is alive at br_i.

7. $\mathcal{T}(res) =$ `?`α T.

   Conditions:

   - $\forall i, 1 \leq i \leq n : \mathcal{T}(val_i)$ is one of `&`β_i `mut` T, `&`β_i T or `?`β_i T.
   - $\forall i, 1 \leq i \leq n : origin(\beta_i) \neq$ `'static` $\Rightarrow \alpha \succeq \beta_i$

   Post condition: $\forall i, 1 \leq i \leq n : val_i$ is alive at br_i.

The intuitive idea of the typing rules for phi instructions returning a reference is to find a common origin of all incoming references. However, the rules exclude references which originate from `'static`, which breaks the intuition of origins, but not the properties of references: The key property of references is that whenever they are alive, their annotated lifetime must be alive. Therefore, the reference returned by the phi instruction must be annotated such that whenever any lifetime identifier is killed after the phi instruction that would kill any incoming reference if it was alive, the returned reference must also be killed.

A strong reference originating from `'static` is only possible if the origin is a null pointer constant, since the only other way to get a reference to `'static` is using global variables, which are all either shared or unsafe pointers from which no strong references

can be derived. Since the null pointer constant is never killed, we can exclude strong references originating from `'static`.

Weak references are only killed by freeunsafe instructions or if their annotated lifetime identifier is killed. Freeunsafe instructions, which are discussed later in this section, kill all weak references, including the one returned by the phi instruction. Therefore, the only other way a weak reference is killed is when its origin is killed, which does not happen for `'static` because global variables and constants are never killed. Therefore, we can also exclude weak references annotated with `'static`.

4.5.10 Call Instruction

The RustiC type system defines some intrinsic functions with known semantics and special type rules. Each intrinsic `rustic.name` can be overloaded, i.e. it can have different *instantiations*, which have different types, different numbers of arguments or even different behaviours. The insantiations are named like `rustic.name.n`, where n is a unique suffix, which can be a natural number or some string that represents the types and behaviour of the intrinsic. They are inserted by the RustiClarify pass, which is discussed in Chapter 5.

RustiC intrinsics are meant to replace certain instructions in the program to simplify the type system and also to get more precise type information:

- `rustic.weaken` is an identity function, i.e. it takes one argument of any type and returns it. This intrinsic can perform implicit downcasts, e.g. it can be called with a `Shared<`T`>` and return a `?`α T. It is inserted by RustiClarify in many places with the idea to make implicit downcasts explicit, which allows us to omit many implicit downcasts in the type system. As an example, RustiClarify inserts a `rustic.weaken` call to each pointer-typed argument of a call instruction. This allows us to call a function variant with a weak reference downcasted from a shared pointer without implementing downcasting in the typing rule for the call instruction.

- `rustic.alloc` is semantically equivalent to a call to an allocation function like `malloc` followed by a bitcast. It has an arbitrary amount of non-pointer parameters which are passed to the allocation function. Therefore, a `rustic.alloc` instantiation can use different allocation functions, e.g. `malloc` or `calloc`.

  In C, a typical usage of an allocation function is a call to a function like `malloc`, which returns a `void*`, which is then casted to another pointer type. In LLVM, this corresponds to a call to `malloc`, which returns a `i8*`, followed by a bitcast. Because bitcasts on safe pointers are quite restricted in RustiC, we define this intrinsic to allow allocation of safe pointers without bitcasts.

- `rustic.free` is semantically equivalent to a call to a deallocation function like `free` preceded by a bitcast. It has exactly one pointer parameter and an arbitary amount of non-pointer parameters.

- `rustic.realloc` is semantically equivalent to a bitcast, followed by a call to a reallocation function like `realloc`, followed by a bitcast. It has exactly one pointer parameter and an arbitrary amount of non-pointer parameters.

- `rustic.ptrswap` is an intrinsic function that represents a very specific usage of a load and a store instruction. For any type T, `rustic.ptrswap` has the following semantics:

```
T* ptrswap(T** ptr, T* val) {
    T* old = *ptr;
    *ptr = val;
    return old;
}
```

  Although a call to `rustic.ptrswap` is always semantically equivalent to a load followed by a store, `rustic.ptrswap` allows typings that are not possible with an equivalent load and a store: RustiC does not allow a load instruction to return a unique pointer, because unique pointers are non-aliasing and loading a unique pointer from memory would create an alias. With `rustic.ptrswap`, we can load a unique pointer from memory, e.g. by calling it with a `Unique<Unique<`T`>>` and a `Unique<`T`>`, returning a `Unique<`T`>`. This is allowed because while loading a unique pointer from memory creates an alias, the value in memory is immediatly overwritten with another unique pointer, which is moved. Therefore, the returned value as well as the pointer in memory are non-aliasing after the call to `rustic.ptrswap`.

- `rustic.memset0` is semantically equivalent to a bitcast followed by a `memset` to the value 0. It has exactly one pointer parameter and one integer parameter, which determines the number of bytes that are overwritten by 0. We define this intrinsic because a `memset` to 0 is often used to initialize memory.

- `rustic.memcpy` is semantically equivalent to two bitcasts followed by a call to `memcpy`. It has two pointer parameters of the same LLVM type and one integer parameter, which are passed to either `memcpy` or `memmove`.

RustiC imposes the following restrictions on call instructions, which are partially eliminated by the RustiClarify pass:

- The callee must be a RustiC intrinsic, a function for which a typing rule exists or a function defined in the module. This implicitly forbids calling function pointers, which are therefore not supported by the RustiC type system.

- If the callee is not a RustiC intrinsic, all pointer arguments $p_1, \ldots, p_n$ must be a call to a `rustic.weaken` intrinsic. Also, p_i must be the direct predcessor of p_{i+1} for all $1 \leq i < n$, and p_n must be the direct predecessor of the call instruction. We call this restriction the *weaken restriction*. It is always eliminated by the RustiClarify pass.

- The weaken restriction is applied to the first argument of a call to `rustic.ptrswap`, i.e. the first argument of a call to `rustic.ptrswap` must be a call to `rustic.weaken` and it must be the direct predecessor of the call to `rustic.ptrswap`.

Depending on the used typing rule, some call instructions have to invalidate all weak references that are currently alive. An example for this is when freeing a shared pointer: since shared pointers can potentially alias with any other shared pointer, they can also potentially alias with any weak reference and therefore freeing a shared pointer may potentially invalidate any weak reference. We call those instructions *freesunafe*. An instruction i that is freeunsafe has two rules:

- All weak references that are alive are killed. We formalize this by defining $move_i$ to contain all instructions that dominate i whose type is a weak reference.
- $\tilde{f}$, the current function variant, must be freeunsafe.

rustic.weaken

$$res = \texttt{call}\ L\ \texttt{@rustic.weaken.n}(L\ val)$$

1. $\mathcal{T}(res) = \texttt{i}n$ for some $n \in \mathbb{N}$.
2. $\mathcal{T}(res) = \texttt{*}T$.

   Conditions:

   - $\mathcal{T}(val) = \texttt{*}T$.

   Post condition: val is alive.
3. $\mathcal{T}(res) = \texttt{Shared<}T\texttt{>}$.

   Conditions:

   - $\mathcal{T}(val)$ is one of `Shared<`T`>` or `Unique<`T`>`.

   Moves: val if it is a unique pointer.

   Post condition: val is alive.
4. $\mathcal{T}(res) = \texttt{Unique<}T\texttt{>}$.

   Conditions:

   - $\mathcal{T}(val) = \texttt{Unique<}T\texttt{>}$.

   Moves: val.

   Post condition: val is alive.
5. $\mathcal{T}(res) = \texttt{\&}\alpha\ \texttt{mut}\ T$.

   Conditions:

   - $\alpha \succeq \mathcal{I}(val)$.

- $\mathcal{T}(val)$ is one of `Unique<`T`>` or `&`β `mut` T.

Post condition: *val* is alive.

6. $\mathcal{T}(res) =$ `&`α T.

   Conditions:

   - $\alpha \succeq \mathcal{I}(val)$.
   - $\mathcal{T}(val)$ is one of `Unique<`T`>`, `&`β `mut` T or `&`β T.

   Post condition: *val* is alive.

7. $\mathcal{T}(res) =$ `?`α T.

   Conditions:

   - $\alpha \succeq \mathcal{I}(val)$.
   - $\mathcal{T}(val)$ is a safe pointer to T.

   Post condition: *val* is alive.

rustic.alloc

$$res = \texttt{call } L\texttt{* @rustic.alloc.n}(\ldots)$$

1. $\mathcal{T}(res)$ is an owned pointer to T.

rustic.free

$$\texttt{call void @rustic.free.n}(L\texttt{* } ptr, \ldots)$$

1. Condition: $\mathcal{T}(ptr) =$ `*`T.

   Post condition: *ptr* is alive.

2. Condition: $\mathcal{T}(ptr) =$ `Shared<`T`>`.

   Is freeunsafe.

   Post condition: *ptr* is alive.

3. Condition: $\mathcal{T}(ptr) =$ `Unique<`T`>`.

   Moves: *ptr*.

   Post condition: *ptr* is alive.

rustic.realloc

$$res = \texttt{call } L\texttt{*} \texttt{ @rustic.realloc.n}(L\texttt{*} \ ptr, \ldots)$$

1. $\mathcal{T}(res)$ is an owned pointer to T.
   Condition: $\mathcal{T}(ptr) = \texttt{*}T$.
   Post condition: ptr is alive.
2. $\mathcal{T}(res)$ is an owned pointer to T.
   Condition: $\mathcal{T}(ptr) = \texttt{Shared<}T\texttt{>}$.
   Is freeunsafe.
   Post condition: ptr is alive.
3. $\mathcal{T}(res)$ is an owned pointer to T.
   Condition: $\mathcal{T}(ptr) = \texttt{Unique<}T\texttt{>}$.
   Moves: ptr.
   Post condition: ptr is alive.

rustic.ptrswap

$$res = \texttt{call } L\texttt{*} \texttt{ @rustic.ptrswap.n}(L\texttt{**} \ ptr, L\texttt{*} \ val)$$

1. $\mathcal{T}(res) = \texttt{*}T$
   Conditions:
   - $\mathcal{T}(ptr)$ is a pointer to $\texttt{*}T$
   - $\mathcal{T}(val) = \texttt{*}T$

   Post condition: ptr and val are alive.
2. $\mathcal{T}(res) = \texttt{Shared<}T\texttt{>}$
   Conditions:
   - $\mathcal{T}(ptr)$ is a safe pointer to $\texttt{Shared<}T\texttt{>}$
   - $\mathcal{T}(val)$ is one of $\texttt{Shared<}T\texttt{>}$ or $\texttt{Unique<}T\texttt{>}$

   Moves: val if it is a unique pointer
   Post condition: ptr and val are alive.
3. $\mathcal{T}(res) = \texttt{Unique<}T\texttt{>}$
   Conditions:
   - $\mathcal{T}(ptr)$ is one of
     - $\texttt{Shared<Unique<}T\texttt{>>}$

  - `&`α `mut Unique<`T`>`
- $\mathcal{T}(val) =$ `Unique<`T`>`

Moves: *val*

Is freeunsafe.

Post condition: *ptr* and *val* are alive.

rustic.memset0

`call void @rustic.memset0.n(`L`* `$ptr, \ldots$`)`

1. Post condition: *ptr* is alive.

rustic.memcpy

`call void @rustic.memcpy.n(`L`* `$dest$`, `L`* `src`, i`n` `$count$`)`

1. Conditions:
   - If $\mathcal{T}(dest)$ and $\mathcal{T}(src)$ are pointers to a pointer, the target pointers must be equal and they must be a shared or an unsafe pointer.
   - If $\mathcal{T}(dest)$ and $\mathcal{T}(src)$ are pointers to a struct, each pointer in the struct must be a shared or an unsafe pointer.

   Post condition: *dest* and *src* is alive.

Function Call

$$res = \texttt{call}\ L\ callee(L_1\ val_1, \ldots, L_n\ val_n)$$

1. $\mathcal{T}(res) = \tilde{R}'$.

   Conditions:
   - Let $\widetilde{callee} := (G', I', t', R', A') \in \mathcal{T}(callee)$.
   - Let φ be a mapping from G' to lifetime identifiers.
   - Let $p_1, \ldots, p_n$ be the LLVM values of the function arguments in the definition of *callee*.
   - Define T_i by replacing every $\alpha \in G'$ in $t'(p_i)$ by $\varphi(\alpha)$ for each $1 \leq i \leq n$.
   - Define $\tilde{R}'$ by replacing every $\alpha \in G'$ in R' by $\varphi(\alpha)$.
   - $\forall i, 1 \leq i \leq n : \mathcal{T}(val_i) = T_i$

   Is freeunsafe if $\widetilde{callee}$ is freeunsafe.

   Moves: $\forall i, 1 \leq i \leq n : val_i$ if it is a unique pointer.

   Post condition: $\forall i, 1 \leq i \leq n : val_i$ is alive.

4.5.11 C Standard Library

To allow type assignments of programs that use C standard library functions, one either needs to include implementations of the library functions in the IR or introduce type rules for each library function. Because analyzing implementations of these function may lead to more inaccurate typings, we choose the second approach.

We found some main categories of C standard library functions:

- A function that takes no pointer parameter and does not return a pointer. Their typing rule is trivial, i.e. it has no conditions and just returns a compatible type. Examples for these functions are mathematical functions like `sin` or `cos` or character classification functions like `isalnum` or `iscntrl`.
- A function that takes some pointer parameters and does not return a pointer. Their typing rule is like the typing rule of `memset0`, i.e. its post conditions require that the pointer arguments are alive and it returns a compatible type. Examples for these functions are string functions like `strlen` or `strcmp`.
- A function that takes some pointer parameters and always returns a pointer derived from exactly one of these arguments. Their typing rule is like the typing rule of `rustic.weaken`, i.e. its post conditions require that the pointer arguments are alive and it returns a downcasted pointer from the corresponding argument. Examples for these functions are string functions like `strcpy` or `strchr`, that always return a pointer derived from their first argument.

Note that these typing rules can only be used for functions whose side effects are limited to the given pointers, i.e. a function may not store an argument pointer into a global variable or load a pointer from a global variable.

4.6 Weaken Restriction

In this section we explain why the weaken restriction is necessary for our type system correctly implementing ownership semantics. The reason is that passing an argument to a function may kill other arguments: Let *ptr* is a unique pointer and *alias* be some reference derived from *ptr*. In that case, we cannot call a function with both *ptr* and *alias*, because passing *ptr* to the function moves *ptr*, killing *alias*. However, our type system without the weaken restriction would allow that, because *ptr* and *alias* are only killed after the call instruction. With the weaken restriction, when calling a function with *ptr* and *alias*, we actually have three calls: one `rustic.weaken` call to *ptr*, one to *alias* and the actual function call. In that case, the first call would kill *alias* and make the typing invalid.

In general, the weaken restriction ensures that after all move operations on the arguments are done, the arguments are still alive. Technically, the typing rule of the call instruction could be defined such that the weaken restriction is not necessary. However, this makes type checking and also the RustiC analysis more complicated, which is why we defined it using the weaken restriction.

```
struct S {
    int x;
    long y;
};

struct T {
    struct S s;
    double z;
};

struct U {
    int x;
    long y;
    double z;
};
```

Figure 4.4: Subtyping in C: `T` and `U` are subtypes of `S`.

4.7 Extensions to the Type System

In this section, we briefly describe two straightforward extensions of the type system.

4.7.1 Safe Bitcast for Subtypes

C allows certain casts between pointers which can be used to implement subtyping: Figure 4.4 shows three structs `S`, `T` and `U`. A `T*` can always be casted to a `S*`, because `S` is the first member of `T`. Also, `U*` can always be casted to `S*`, because the members of `S` are a prefix of the members of `U`. In those cases, we say that `T` and `U` are *subtypes* of `S` and we call the casts above *upcasts*. Casts in the other direction are called *downcasts* and they are only valid if the object in memory is a subtype of the type resulting from the cast. In general, checking the validity of downcasts requires run-time checks. CCured [7] describes an approach to implement those run-time checks, while providing a more general definition of type equivalence and subtyping. Assuming such a mechanism exists during the instrumentation of the program, we can add a typing rule that allows a safe bitcast between safe pointers to T and U if T is a subtype of U or U is a subtype of T.

4.7.2 Derived Pointers

Derived pointers are relevant for efficiently implementing run-time checks on safe pointers. The `getelementptr` instruction can return a unique or a shared pointer that has a non-zero offset, i.e. it does not point to the beginning of the object. However, these pointers can be passed to `rustic.free`, which requires the pointer to have an offset of zero. Therefore, when calling `rustic.free`, we have to ensure that the pointer has an offset of zero.

In many cases we can omit this run-time check by incorporating this property into the type system: we define two variants of safe pointers: the *derived unique pointer* and the *derived shared pointer*. The `getelementptr` must return derived unique or shared

pointers instead of a unique or shared pointers. Also, derived pointers can never be implicitly casted to non-derived pointers, i.e. the "derivedness" of an owned pointer must be preserved in all instructions.

4.8 Properties and Run-Time Checks

In this section we describe properties of pointer types at run-time and which run-time checks are needed to ensure temporal memory safety. The most important property of safe pointers is that the type system prevents code that depends on their memory layout. This allows us to change the memory layout of safe pointers, e.g. by replacing the pointer by a struct containing the pointer and some metadata. To distinguish these representation from raw pointer-typed values in LLVM, we call the latter ones *raw pointers*.

- If a unique pointer or reference is alive, it is either null or pointing to a valid object. This means that using such a pointer in a load or store instruction only requires a null-pointer check.

  On a `getelementptr` instruction on such a pointer, we must check if the pointer is null and in that case, return null instead of performing arithmetic.

  On a call to `rustic.free` with a derived unique pointer, we must check that the pointer has an offset of zero. Note that if `rustic.free` is called with a unique pointer, it is either a null pointer or points to an object returned by `rustic.alloc`. `rustic.free` can never be called with a unique pointer returned by an `alloca` instruction because the type system does not allow moving unique pointers returned by an `alloca`.

  Implementing the zero-offset check on derived unique pointers can be done by modifying their memory layout to include an offset, which can be compared to zero.

  Note that run-time checks for non-derived unique pointers and references can be implemented without changing their memory layout.

- A shared pointer is either null, pointing to a valid object or pointing to a deleted object. In the latter case, it must have been deleted by any aliasing shared pointer. Loading from or storing into a shared pointer requires checking if the object, the pointer points to, was not deleted. In the next section, we briefly describe two approaches for implementing shared pointers that allow this check.

  For any instruction that implicitly downcast a shared pointer, which may occur in a `load`, `store`, `getelementptr` or `rustic.weaken`, we have to check the validity of the shared pointer and if it is invalid, i.e. it got deleted, the returned pointer must be null.

  Similar to the unique pointer, on a `getelementptr` instruction on a shared pointer we must preserve the nullity of that pointer and also the invalidity.

`rustic.alloc` must return a valid shared pointer or null. A call to `rustic.free` must invalidate a shared pointer if it is valid and was created by `rustic.alloc`. If the shared pointer is derived, we must check that its offset is zero. Note that not all shared pointers are created with `rustic.alloc`: Global variables can be shared pointers and `alloca` can return a shared pointer.

Similar to unique pointers, implementing the zero-offset check on derived shared pointers can be done by modifying their memory layout to include an offset, which can be compared to zero.

An `alloca` instruction returns a valid shared pointer that may not be deallocated with `rustic.free`. When the function returns, all shared pointers returned by `alloca` instructions must be invalidated.

- An unsafe pointer can be null, a dangling pointer, a wild pointer or pointing to a valid object. It is only guaranteed that it does not alias with a safe pointer.

  Describing a run-time check for these pointers is out of scope of our thesis. Because code can depend on the memory layout of unsafe pointers, we recommend an approach that does not require changing the memory layout of pointers, e.g. CETS [13].

If a standard library function is called and one of the arguments is an unsafe pointer or a safe pointer whose memory layout is changed, a wrapper is needed that checks the pointers for validity and converts them to raw pointers, which are passed to the function. Also, if the function returns a safe pointer, the raw pointer returned from the library functon must be converted into a safe pointer.

An example for this is the `strchr` function, which takes one pointer as an argument and returns either null or a pointer that is derived from the argument pointer. Assume a call to `strchr` with a shared pointer that returns a derived shared pointer. In that case, the shared pointer must be checked for validity and must be converted to a raw pointer to the object, which is essentially the same as downcasting it to a reference. The returned raw pointer is then converted back to a derived shared pointer, i.e. a derived shared pointer must be initialized that aliases with the given shared pointer.

A special case is the `rustic.memset0` intrinsic. If this intrinsic is called on a safe pointer to a type that contains a pointer, it may happen that this pointer is only partially overwritten with zeroes, e.g. if the C program contains a `memset` to `int**`, where the number of bytes is not a multiple of `sizeof(int*)`. In our thesis, we assume that partially overwriting a pointer with zeroes always results in an invalid pointer. Therefore, an implementation of `rustic.memset0` must ensure that if a pointer is partially overwritten, it is initialized to the null pointer.

Another special case is the `rustic.memcpy` intrinsic, which has the same issue as `rustic.memset0`, i.e. it may only partially copy a pointer. This issue can be handled similarly to the `rustic.memset0` instruction. Also note that `rustic.memcpy` may copy shared or unsafe pointers, e.g. when called with a pointer to a pointer or a pointer to a struct. Depending on the implementation of shared and unsafe pointers, this copy operation must be handled.

4.9 Implementing Shared Pointers

We briefly describe two approaches to implement shared pointers. In each of them we describe how shared pointers are represented, how allocating and freeing shared pointers is handled, and how shared pointer dereferences are checked at run-time.

We say that a *shared pointer dereference* occurs whenever a shared pointer is loaded from or stored to or a shared pointer is implicitly downcasted. By that definition, a `load` instruction can have two shared pointer dereferences: if we load a `?`α T from a `Shared<Shared<`T`>>`, we first dereference the `Shared<Shared<`T`>>`, resulting in a `Shared<`T`>`, which has to be downcasted into `?`α T, which is again a dereference.

Note that shared pointers may be allocated by `rustic.alloc`, but also each global variable that is a shared pointer is allocated when the program starts and each alloca allocates a new shared pointer, both of which may not be freed by `rustic.free`.

4.9.1 Identifier-Based approach

The idea of this approach is to associate a unique identifier to each allocation, while identifiers are never reused. In our setting, it is sufficient to use integers as identifiers [4]. To implement this approach, a mechanism is needed to track which identifiers are valid and which of them may be freed. A simple approach for this is to use a hash table that maps each valid identifier to the base address of the allocation. Non-freeable identifiers are mapped to null.

A shared pointer is represented as a struct containing a pointer and a corresponding identifier. When a shared pointer is allocated, a new identifier is chosen which is marked as valid and stored in the shared pointer. When dereferencing a shared pointer, we check whether its identifier is valid. When calling `rustic.free` on a shared pointer, we check whether its identifier is valid and if the pointer is equal to the base address mapped to the identifier.

Note that this approach does not differentiate between shared and derived pointers. This is because we use the base address for two reasons: To check whether the (derived) shared pointer has a zero offset and to check whether the identifier is freeable.

4.9.2 Proxy Objects

The idea of this approach is to represent shared pointers in a doubly indirect way: A shared pointer points to a *proxy object*, which is a dynamically allocated object containing the pointer to the actual object. Whenever a shared pointer is freed, the pointer in the proxy object is set to null.

A proxy object is a struct containing a pointer and some additional information, e.g. whether it may be freed. A shared pointer is represented as a pointer to the proxy

[4] In theory, this limits the number of allocations by the maximal representable integer. In practice, using a 64 bit unsigned integer is sufficient, because even assuming that a shared pointer allocation occurs on every CPU cycle on a 4 GHz CPU, it would take about 146 years until all identifiers were used.

object. A derived shared pointer is represented as two pointers: a pointer to the proxy object and the actual pointer.

When a shared pointer is allocated, a new proxy object is allocated and the returned shared pointer points to this proxy object. When a shared pointer is dereferenced, we load the pointer from the proxy object and check if is not null. For a non-derived shared pointer, this load also returns the pointer to the actual object. When calling `rustic.free` on a shared pointer, we load the pointer from the proxy object and check if is not null and we check whether the proxy object may be freed. In that case, we deallocate the actual object and write null into the proxy object.

Note that each shared pointer allocation allocates a new proxy object but freeing shared pointers only writes a null pointer into the proxy object, leaving the proxy object alive. We can only free the proxy object itself if we know that there are no other shared pointers pointing to that proxy object, which may be the case in temporally unsafe code. Implementing this requires a garbage collection mechanism, which may introduce additional overhead when loading or storing shared pointers or even when implicitly copying shared pointers.

While this is a huge disadvantage compared to the identifier-based approach, this approach has the advantage that dereferencing has much less overhead: instead of a hash table lookup to check the validity of a pointer, here we only need an additional load and a null-pointer check.

4.10 Optimizations

4.10.1 Redundant Check Elimination

In some cases, we can omit run-time checks of shared pointers:

- Global variable typed as a shared pointer do not need to be checked at run-time, because they cannot be freed during the program.

- Shared pointers returned by an `alloca` instruction also do not need a run-time check, because they are alive until the function returns.

- Assume two instructions i and j that dereference the same shared pointer, such that i strictly dominates j. If there is no freeunsafe instructions "between" i and j, we can omit the run-time check for the dereference of j. More formally, there must be no freeunsafe instruction k such that i dominates k and k dominates j.

These optimizations are pretty similar to those applied in CETS [13], which they call unnecessary and redundant check elimination. In CETS, a redundant check occurs if a pointer is checked that has been checked before and the check is not killed by a call to free. Conceptually, in our case we replaced "killed by a call to free" with "killed by

a freeunsafe instruction", which permits function calls of functions that do not free a shared pointer [5].

4.10.2 Null-Pointer Checks

Assuming that dereferencing a null pointer always fails and aborts the program, which is a reasonable assumption for general purpose operating systems, we can omit null-pointer checks:

- We can omit the null-pointer check for loads and stores on unique pointers and references, as they are either null or valid. With this, dereferencing unique pointers or references requires no run-time checks at all.
- If shared pointers are implemented using proxy objects, dereferencing non-derived shared pointers can be implemented using only a load instructions, because the pointer in the proxy object is either null or valid.

  Note that this does not work for derived shared pointers, because the proxy object only contains the base pointer and not the derived pointer, therefore if a shared pointer was freed, the base pointer is null, but the derived pointer is invalid. This means that we need a null-pointer check such that dereferencing a freed derived shared pointer yields a null pointer instead of an invalid pointer.

[5] Formally, an instruction being freeunsafe does not imply that a shared pointer is freed, but the inverse is true. Therefore, being freeunsafe is an overapproximation for a shared pointer being freed.

5 RustiClarify

The RustiClarify pass transforms an LLVM module such that it can be analyzed by the RustiC analysis. While RustiClarify is necessary for the analysis to handle the weaken restriction on call instructions, RustiClarify also transforms the module to get more precise analysis results. It does that by recognizing certain patterns and replacing them by RustiC intrinsics. We refer to the IR resulting from the RustiClarify transformation as the *clarified* IR.

5.1 RustiC Intrinsics

When replacing code with a RustiC intrinsic `rustic.name`, RustiClarify has to choose an instantiation `rustic.name.n` of that intrinsic that has the appropriate type and behaviour. If no such instantiation exists, it is defined by RustiClarify.

The most straightforward way to instantiate an intrinsic is to choose an unused number `n`, define a function `rustic.name.n`, and put the replaced code inside this function. It is however recommended to implement a lookup to prevent instantiating the same intrinsic multiple times. This should be done at least for `rustic.weaken` because this intrinsic will be used quite often.

5.2 Patterns

In RustiClarify, a *pattern* is described by a set of instructions with certain properties, and how they are replaced. A necessary condition for all patterns is that their replacement preserves the semantics of the program. One common pattern is the "malloc+bitcast pattern": It consists of a call to `malloc` and a `bitcast`, where the `bitcast` is the only instruction that uses the result of the call.

In this section, we describe the patterns we used in our implementation of RustiClarify. RustiClarify can always be extended by adding new patterns, which leaves room for future work.

Note that in general patterns can be overlapping: There may be a situation in which two patterns are matched, but if one is replaced, the other one can no longer be matched. In that case, one would need a tie-break between overlapping patterns. While the patterns provided in this thesis cannot overlap, extending RustiClarify by more patterns can lead to such a situation.

Also note that replacing a pattern can lead to more patterns being matched, because the replacement of one pattern can introduce code that matches another pattern. This

may lead to a cycle in the pattern replacement, e.g. when a pattern A is matched and replaced by code that matches pattern B, but pattern B is replaced by code that matches pattern A. Therefore, it is important to ensure termination when implementing new patterns.

5.2.1 malloc+bitcast Pattern

Pattern

$$\begin{aligned} tmp &= \texttt{call i8* @malloc}(size) \\ val &= \texttt{bitcast i8* } tmp \texttt{ to } L\texttt{*} \end{aligned}$$

tmp may only be used by *val*.

Replacement

$$val = \texttt{call } L\texttt{* @rustic.alloc.n}(size)$$

5.2.2 calloc+bitcast pattern

Pattern

$$\begin{aligned} tmp &= \texttt{call i8* @calloc}(m, size) \\ val &= \texttt{bitcast i8* } tmp \texttt{ to } L\texttt{*} \end{aligned}$$

tmp may only be used by *val*.

Replacement

$$val = \texttt{call } L\texttt{* @rustic.alloc.n}(m, size)$$

5.2.3 realloc+bitcast pattern

Pattern

$$\begin{aligned} tmp &= \texttt{bitcast } L\texttt{* } ptr \texttt{ to i8*} \\ tmp2 &= \texttt{call i8* @realloc}(\texttt{i8* } tmp,\ size) \\ val &= \texttt{bitcast i8* } tmp2 \texttt{ to } L\texttt{*} \end{aligned}$$

tmp2 may only be used by *val*.

Replacement

$$val = \texttt{call } L\texttt{*} \texttt{ @rustic.realloc.n}(L\texttt{*} \ ptr, size)$$

5.2.4 free+bitcast pattern

Pattern

$$tmp = \texttt{bitcast } L\texttt{*} \ ptr \texttt{ to i8*}$$
$$\texttt{call void @free}(\texttt{i8*} \ tmp)$$

Replacement

$$\texttt{call void @rustic.free.n}(L\texttt{*} \ ptr)$$

5.2.5 memset0 pattern

Pattern

$$tmp = \texttt{bitcast } L\texttt{*} \ ptr \texttt{ to i8*}$$
$$\texttt{call void @llvm.memset.*}(\texttt{i8*} \ tmp, \ \texttt{i8 } 0, \ len, \ volatile)$$

Note that LLVM defines two versions of their memset intrinsic, which differ in their name and also in the type of *len*.

Replacement

$$\texttt{call void @rustic.memset0.n}(L\texttt{*} \ ptr, len)$$

Note that the *volatile* parameter is not passed to the `rustic.memset` intrinsic. This is because when calling the `llvm.memset.*` intrinsic, the *volatile* parameter has to be a constant. Therefore, the instantiation `rustic.memset0.n` has to pass this parameter as a constant.

5.2.6 load+store pattern

Pattern

$$val = \texttt{load } L\texttt{*}, \ L\texttt{**} \ ptr$$
$$\texttt{store } L\texttt{*} \ val2, \ L\texttt{**} \ ptr$$

There must be no other `store` or `call` instruction between the two instructions and each use of *val* must be strictly dominated by the `store` instruction. Also, *val* must be a different value than *val2*.

Replacement

$$val = \texttt{call } L* \texttt{ @rustic.ptrswap.n}(L** \ ptr,\ L* \ val2)$$

The instruction is inserted at the previous position of the `store` instruction.

5.2.7 load+free pattern

Pattern

$$val = \texttt{load } L*,\ L** \ ptr$$
$$\texttt{call void @rustic.free.n}(L* \ val)$$

There must be no other `store` or `call` instruction between the two instructions and each use of *val* must be dominated by the `call` instruction.

Replacement

$$val = \texttt{call } L* \texttt{ @rustic.ptrswap.n}(L** \ ptr,\ L* \texttt{ null})$$
$$\texttt{call void @rustic.free.n}(L* \ val)$$

The call to `ptrswap` must be inserted directly before the call to `rustic.free`.

5.2.8 load+realloc pattern

Pattern

$$val = \texttt{load } L*,\ L** \ ptr$$
$$\texttt{call void @rustic.realloc.n}(L* \ val, size)$$

There must be no other `store` or `call` instruction between the two instructions and each use of *val* must be dominated by the `call` instruction.

Replacement

$$val = \texttt{call } L* \texttt{ @rustic.ptrswap.n}(L** \ ptr,\ L* \texttt{ null})$$
$$\texttt{call void @rustic.realloc.n}(L* \ val, size)$$

The call to `ptrswap` must be inserted directly before the call to `rustic.realloc`.

Note that this replacement is only semantics preserving if `realloc` never fails. This is because if `realloc` fails, it does not free the old pointer, but the replacement sets the old pointer to null. Therefore, to make this replacement valid, an instrumentation pass must abort the program if a `realloc` fails.

5.3 Lowering of `bitcast` and `getelementptr` instructions

Lowering is defined as follows: For each instruction i that is not a `bitcast` or `getelementptr` instruction, consider each of its operands v_0 that is a `bitcast` or `getelementptr` instruction. Construct a maximal chain $v_1, \ldots, v_n$, such that for all $i \geq 0$:

- v_i is a `bitcast` or `getelementptr` instruction,
- v_i uses v_{i+1} as an operand.

Now duplicate this chain into $v'_0, \ldots, v'_n$, where the uses of $v_0, \ldots, v_n$ in the duplicated chain are replaced by their duplicates $v'_0, \ldots, v'_n$. This duplicated chain is inserted before i and v_0 is replaced by v'_0 in i.

The idea of lowering is to duplicate `bitcast` and `getelementptr` instructions before all of their uses:

```
%a = getelementptr ...
%b = bitcast %a to ...
...
some instruction that uses %b
...
some instruction that uses %b
```

is transformed into

```
%a1 = getelementptr ...
%b1 = bitcast %a1 to ...
some instruction that uses %b1
...
%a2 = getelementptr ...
%b2 = bitcast %a2 to ...
some instruction that uses %b2
```

This is a sound transformation because LLVM is in SSA form and `bitcast` and `getelementptr` instructions have no side effects, therefore the duplicated instructions will compute the same value.

5.4 Main Algorithm

The main algorithm of RustiClarify is quite straightforward: For each defined function f in the LLVM module:

1. While there is a pattern that can be matched, replace that pattern accordingly.
2. Replace each remaining call to `malloc` or `calloc` by a call to `rustic.alloc`.
3. Replace each remaining call to `realloc` by a call to `rustic.realloc`.
4. Replace each remaining call to `free` by a call to `rustic.free`.

5. For each phi instruction in a basic block b that has a predecessor block b', where b' has multiple successor blocks, split the edge between b' and b: Create a new basic block c that branches to b and in b', replace the branch to b by a branch to c.

6. For each phi instruction that returns a pointer, replace each incoming value by a `rustic.weaken` call to that value, which is inserted before the terminator instruction of the incoming block.

7. Perform lowering of `bitcast` and `getelementptr` instructions.

8. For each call instruction that does not call RustiC intrinsic, replace each pointer parameter by a `rustic.weaken` call to that parameter.

9. For each call to `rustic.ptrswap`, replace the first pointer parameter by a `rustic.weaken` call to that parameter.

10. Replace the operand of each `bitcast` by a `rustic.weaken` call to that operand.

5.5 Examples

5.5.1 Running example

Figure 5.1 shows the IR of our running example resulting from `clang -O0` and the `mem2reg` pass, and the LLVM IR after the RustiClarify pass including the instantiations of the RustiC intrinsics. In `idx`, nothing is done by RustiClarify. In `f`, the malloc+bitcast and the free+bitcast patterns were replaced. Also, a weaken call was inserted for the argument `%x` of the call to `idx`.

5.5.2 The Relevance of `rustic.ptrswap`

In this example, we demonstrate how inserting calls to `rustic.ptrswap` can lead to more accurate typings. Figure 5.2 shows C code that deallocates a matrix represented as an indirect two-dimensional array.

The line `free(matrix[i]);` is compiled into a load, a bitcast and a free instruction. If we ignore the patterns that insert a `rustic.ptrswap`, we end up with the IR shown in Figure 5.3. The call to `rustic.free` requires an owned pointer, therefore the load instruction has to load an owned pointer. Because a load instruction cannot load a unique pointer, it must load a shared pointer, thus `%matrix` must be a `Unique<Shared<i32>>`.

If we perform RustiClarify with all patterns, we get the IR shown in Figure 5.4. Here, the load followed by the free instruction was replaced with a `rustic.ptrswap` followed by a free instruction. Here we can type `%matrix` as a `Unique<Unique<i32>>` because `rustic.ptrswap` allows loading a unique pointer from memory.

The relevance of `rustic.ptrswap` comes from the fact that it is the only way to load a unique pointer from memory. If any program contains a dynamically allocated data

```
define i32* @idx(i32* %arr, i32 %idx) {
entry:
  %idx.ext = sext i32 %idx to i64
  %add.ptr = getelementptr inbounds i32, i32* %arr, i64 %idx.ext
  ret i32* %add.ptr
}

define void @f() {
entry:
  %tmp = call noalias i8* @malloc(i64 40)
  %x = bitcast i8* %tmp to i32*
  %alias = call i32* @idx(i32* %x, i32 4)
  store i32 10, i32* %alias, align 4
  %tmp2 = bitcast i32* %x to i8*
  call void @free(i8* %tmp2)
  ret void
}
```

(a) IR of running example from Figure 4.2.

```
define i32* @idx(i32* %arr, i32 %idx) {
entry:
  %idx.ext = sext i32 %idx to i64
  %add.ptr = getelementptr inbounds i32, i32* %arr, i64 %idx.ext
  ret i32* %add.ptr
}

define void @f() {
entry:
  %x = call i32* @rustic.alloc.0(i64 40)
  %x1 = call i32* @rustic.weaken.0(i32* %x)
  %alias = call i32* @idx(i32* %x1, i32 4)
  store i32 10, i32* %alias, align 4
  call void @rustic.free.0(i32* %x)
  ret void
}

define i32* @rustic.alloc.0(i64) {
entry:
  %1 = call i8* @malloc(i64 %0)
  %2 = bitcast i8* %1 to i32*
  ret i32* %2
}

define void @rustic.free.0(i32*) {
entry:
  %1 = bitcast i32* %0 to i8*
  call void @free(i8* %1)
  ret void
}

define i32* @rustic.weaken.0(i32*) {
entry:
  ret i32* %0
}
```

(b) Clarified IR including RustiC instrinsics.

Figure 5.1: Example for RustiClarify transformation.

structure whose owned pointer is a unique pointer stored in memory, it can only be freed using a `rustic.ptrswap`.

```
void free_matrix(int** matrix, unsigned long long n) {
    for(unsigned long long i = 0; i < n; i++)
        free(matrix[i]);
    free(matrix);
}
```

```
define void @free_matrix(i32** %matrix, i64 %n) {
entry:
  br label %for.cond

for.cond:                                         ; preds = %for.body, %entry
  %i.0 = phi i64 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp ult i64 %i.0, %n
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %arrayidx = getelementptr inbounds i32*, i32** %matrix, i64 %i.0
  %0 = load i32*, i32** %arrayidx, align 8
  %1 = bitcast i32* %0 to i8*
  call void @free(i8* %1)
  %inc = add i64 %i.0, 1
  br label %for.cond

for.end:                                          ; preds = %for.cond
  %2 = bitcast i32** %matrix to i8*
  call void @free(i8* %2)
  ret void
}
```

Figure 5.2: Deallocating a two-dimensional array.

```
define void @free_matrix(i32** %matrix, i64 %n) #0 {
entry:
  br label %for.cond

for.cond:                                         ; preds = %for.body, %entry
  %i.0 = phi i64 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp ult i64 %i.0, %n
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %arrayidx = getelementptr inbounds i32*, i32** %matrix, i64 %i.0
  %0 = load i32*, i32** %arrayidx, align 8
  call void @rustic.free.0(i32* %0)
  %inc = add i64 %i.0, 1
  br label %for.cond

for.end:                                          ; preds = %for.cond
  call void @rustic.free.1(i32** %matrix)
  ret void
}
```

```
fn free_matrix(
'a:     i32** %matrix: Unique<Shared<i32>>
        i64 %n: i64
) -> void @freeunsafe {
    entry:
        br label %for.cond: void

    for.cond:
        %i.0 = phi i64 [ 0, %entry ], [ %inc, %for.inc ]: i64
        %cmp = icmp ult i64 %i.0, %n: i1
        br i1 %cmp, label %for.body, label %for.end: void

    for.body:
'b:     %arrayidx = getelementptr inbounds i32*,
            i32** %matrix, i64 %i.0: ?'a Shared<i32>
'c:     %0 = load i32*, i32** %arrayidx: Shared<i32>
        call void @rustic.free.0(i32* %0): void @freeunsafe
        %inc = add i64 %i.0, 1: i64
        br label %for.cond: void

    for.end:
        call void @rustic.free.1(i32** %matrix): void
        ret void: void
}
```

Figure 5.3: Code from Figure 5.2 clarified without ptrswap.

```
define void @free_matrix(i32** %matrix, i64 %n) #0 {
entry:
  br label %for.cond

for.cond:                                         ; preds = %for.body, %entry
  %i.0 = phi i64 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp ult i64 %i.0, %n
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %arrayidx = getelementptr inbounds i32*, i32** %matrix, i64 %i.0
  %0 = call i32* @rustic.ptrswap.0(i32** %arrayidx, i32* null)
  call void @rustic.free.0(i32* %0)
  %inc = add i64 %i.0, 1
  br label %for.cond

for.end:                                          ; preds = %for.cond
  call void @rustic.free.1(i32** %matrix)
  ret void
}
```

```
fn free_matrix(
'a:     i32** %matrix: Unique<Unique<i32>>
        i64 %n: i64
) -> void @freeunsafe {
    entry:
        br label %for.cond: void

    for.cond:
        %i.0 = phi i64 [ 0, %entry ], [ %inc, %for.inc ]: i64
        %cmp = icmp ult i64 %i.0, %n: i1
        br i1 %cmp, label %for.body, label %for.end: void

    for.body:
'b:     %arrayidx = getelementptr inbounds i32*,
            i32** %matrix, i64 %i.0: &mut 'a Unique<i32>
'c:     %1 = call i32* @rustic.ptrswap.0
            (i32** %arrayidx, i32* null): Unique<i32> @freeunsafe
'd:     %2 = call i32* @rustic.weaken.1(i32* %1): Unique<i32>
        call void @rustic.free.0(i32* %2): void
        %inc = add i64 %i.0, 1: i64
        br label %for.cond: void

    for.end:
        call void @rustic.free.1(i32** %matrix2): void
        ret void: void
}
```

Figure 5.4: Code from Figure 5.2 clarified with ptrswap.

6 Analysis

In this chapter we describe a static analysis that computes a valid RustiC type assignment for a given clarified module. We implement this analysis as a nested worklist algorithm: The inner worklist algorithm analyzes a function variant by changing the types of values until the function variant has a valid typing. The outer worklist algorithm instantiates function variants and calls the inner worklist algorithm.

6.1 Lattice

For the inner worklist algorithm, we need a lattice for types. We define $\sqsubseteq$ to be the smallest partial ordering such that for all types T and U and all lifetime identifiers α:

- `!` $\sqsubseteq T$
- `?'static` $T \sqsubseteq$ `?`α T
- `?`α $T \sqsubseteq$ `&`α T
- `&`α $T \sqsubseteq$ `&`α `mut` T
- `&`α `mut` $T \sqsubseteq$ `Unique<`T`>`
- `Unique<`T`>` $\sqsubseteq$ `Shared<`T`>`
- `Shared<`T`>` $\sqsubseteq$ `*`T
- `?`α $T \sqsubseteq$ `?`α U if $T \sqsubseteq U$
- `&`α $T \sqsubseteq$ `&`α U if $T \sqsubseteq U$
- `&`α `mut` $T \sqsubseteq$ `&`α `mut` U if $T \sqsubseteq U$
- `Unique<`T`>` $\sqsubseteq$ `Unique<`U`>` if $T \sqsubseteq U$
- `Shared<`T`>` $\sqsubseteq$ `Shared<`U`>` if $T \sqsubseteq U$
- `*`$T \sqsubseteq$ `*`U if $T \sqsubseteq U$

Figure 6.1 show the lattice for pointer types as a Hasse diagram. If $T \sqsubseteq U$, we say that T is *weaker* than U. We define $T \sqcup U$ to be the least upper bound of T and U regarding $\sqsubseteq$.

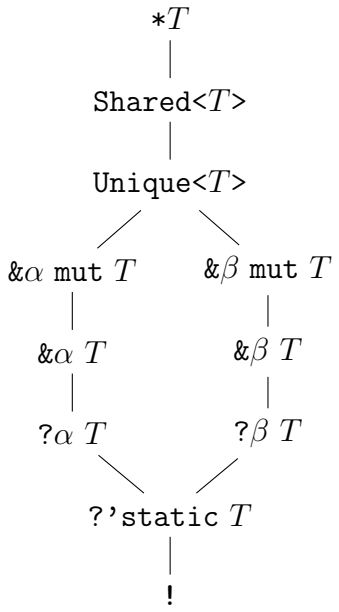

Figure 6.1: Lattice for pointer types.

Note that formally, this is not a lattice, e.g. there does not exists an element for $\texttt{i8} \sqcup \texttt{i16}$. However, our analysis only does three kinds of joins: $\texttt{!} \sqcup T$, $T \sqcup \texttt{!}$ or $T \sqcup U$ where T and U are compatible to the same LLVM type L. In these cases, the least upper bound always exists. Also note that the join of two references with different, non-static lifetime identifiers is always a unique pointer.

6.2 Data Structures

The RustiC analysis manages a list of generic lifetime identifiers in an array. We denote the n-th generic lifetime identifier as GLT_n. They are initialized lazily, i.e. whenever the analysis needs GLT_n, the array is extended until GLT_n is defined. When giving examples, we use capital letters to denote these identifiers, i.e. `'A`, `'B`, `'C`, ... denote $\text{GLT}_0, \text{GLT}_1, \text{GLT}_2, \ldots$

A *signature* is a tuple $(f, args, ret, attr)$ used to identify function variants, where

- f is a function defined in the LLVM module,
- $args$ is a tuple of types, which represent the types of the function arguments,
- ret is a type that represents the return type of a function and
- $attr$ is a set of attributes.

Let

$$s = (f, (s_1, \ldots, s_n), r_s, A_s), \qquad g = (f, (g_1, \ldots, g_n), r_g, A_g)$$

be signatures. We say that s is *weaker* than g iff $s_i \sqsubseteq g_i$ for all $i \in \{1, \ldots, n\}$, $r_s \sqsubseteq r_g$ and $A_s \subseteq A_g$. We define the following operations on signatures:

- *Normalization* is an operation that has the following property: Let s_1, s_2 be signatures. If there exists a bijection of non-static lifetime identfiers such that replacing all lifetime identifiers in s_1 using that bijection yields s_2, then the normalized signatures of s_1 and s_2 must be equal.

  Normalization is done by iterating through the argument types and maintaining a mapping φ. For each argument type that is a reference annotated with $\alpha \neq$ `'static`:

  - If $\varphi(\alpha)$ is undefined, define $\varphi(\alpha) := \mathrm{GLT}_{|\varphi|}$ and replace α with $\varphi(\alpha)$ in the argument type.
  - If $\varphi(\alpha)$ is defined, replace α with $\varphi(\alpha)$ in the argument type.

  After that, if the return type is a reference annotated with $\alpha \neq$ `'static`, replace α with $\varphi(\alpha)$ in the return type. Note that $\varphi(\alpha)$ must be defined because α is annotated on the return type, therefore there must be a function argument that uses α.

  To give an example,

  $$(f, (\texttt{?'d i32}, \texttt{?'d i32}, \texttt{\&'f mut Unique<i8>}, \texttt{?'static i8}), \texttt{?'d i32}, \{\})$$

  is normalized to

  $$(f, (\texttt{?'A i32}, \texttt{?'A i32}, \texttt{\&'B mut Unique<i8>}, \texttt{?'static i8}), \texttt{?'A i32}, \{\}),$$

  where $\varphi(\texttt{'d}) = \texttt{'A}$ and $\varphi(\texttt{'f}) = \texttt{'B}$. Note that signatures with generic lifetime identifiers can also be normalized:

  $$(f, (\texttt{?'A i32}, \texttt{Shared<i32>}, \texttt{?'C i32}), \texttt{?'C i32}, \{\})$$

  is normalized to

  $$(f, (\texttt{?'A i32}, \texttt{Shared<i32>}, \texttt{?'B i32}), \texttt{?'B i32}, \{\}),$$

  where $\varphi(\texttt{'A}) = \texttt{'A}$ and $\varphi(\texttt{'C}) = \texttt{'B}$.

- A *generalized* signature is a normalized signature after removing all attributes.

- *Instantiation* is intuitively the inverse operation of normalization. We define it as an operation on two signatures: Let

  $$g = (f, (g_1, \ldots, g_n), r_g, A)$$

  and

  $$s = (f, (s_1, \ldots, s_n), r_s, \{\})$$

  be signatures. To instantiate g with s:

1. Compute a mapping φ such that for each $i \in \{1, \ldots, n\}$: If g_i and s_i are references annotated with α and β then either $\alpha = \beta =$ `'static` or $\varphi(\alpha) = \beta$.
2. For each $i \in \{1, \ldots, n\}$, define t_i by replacing all lifetime identifiers in g_i using φ.
3. Define r by replacing all lifetime identifiers in r_g using φ.
4. The returned signature is:

$$(f, (t_1 \sqcup s_1, \ldots, t_n \sqcup s_n), r \sqcup r_s, A)$$

Note that instantiation is only partially defined, because such a mapping φ may not exist or because of the partially defined joins. However, whenever our analysis does an instantiation, it is defined. This is due to the following property: Let s be a signature and g be the generalized signature. Then, every signature stronger than g can be instantiated with s.

A *variant* is a mutable data structure that implements a function variant from the RustiC type system. It contains a function f defined in the LLVM module, a mapping from all pointer-typed values in f to local lifetime identifiers, a mapping from all values in f to types, a return type and a set of attributes. A variant is initialized with a signature $(f, args, ret, attr)$ where the types in $args$ must be compatible to the LLVM type of the arguments of f and ret is either `!` or compatible to the return LLVM type of f. Each pointer-typed value is assigned a unique local lifetime identifier, the argument values are initialized with the types given in $args$, the return type is initialized with ret and the type of all other values is set to `!`.

A *worklist* is a list of items that need to be processed. Items can be pushed into the worklist and popped from the worklist. In our case, we consider a worklist as a set, i.e. if an item is pushed that already is in the worklist, it is not added again. The inner worklist algorithm uses a worklist of instructions while the outer worklist algorithm uses a worklist of function variants.

6.3 Inner Worklist Algorithm

Figure 6.2 shows pseudocode for the inner worklist algorithm. We start with an informal explanation of the algorithm: It consists of a nested while loop, where the inner while loop visits each instruction in the worklist and checks if its typing is valid, i.e. if there exists a typing rule for that instruction. If this is not the case, the types of the instruction and its operands are adjusted until its typing is valid. Everytime the type of an instruction is changed, it itself and all instructions that use it as an operand are pushed to the worklist.

In `visit`, we disregard all post conditions, meaning that when the inner while loop finishes, the function variant is correctly typed up to the post conditions. Checking post conditions is done after the inner while loop: we perform lifetime analysis and visit each instruction to check if its typing is valid, this time including post conditions.

```
function analyze_variant(v: Variant)
    sig ← v.get_signature();
    while not v.worklist.empty() do
        while not v.worklist.empty() do
            i ← v.worklist.pop();
            visit(v, i);
        perform lifetime analysis;
        foreach instruction i do
            if not visit_post(v, i) then
                break;
    return v.get_signature() ≠ sig;
```

Figure 6.2: Inner worklist algorithm.

If this is not the case for an instruction, its typing is adjusted, and the outer while loop continues.

Note that after the lifetime analysis, the type of only one instruction is adjusted. The reason for this and also the reason for the nested while loop is to perform lifetime analysis less often: If the type of an instruction changes, the results of lifetime analysis may also change, therefore the lifetime analysis must be re-done. Also, if the type of one instruction is changed, most likely some instructions are not well-typed anymore, even disregarding post conditions. Therefore, the idea is to propagate the one change done after the lifetime analysis using the inner while loop before redoing the lifetime analysis.

We now discuss important aspects of the algorithm in more detail. Whenever a type is changed, which may be the type of an instruction, the type of a function parameter, the return type of the function variant, the type of a global variable or the type of a struct member, the following conditions must be met:

- If T is changed to T', then $T \sqsubset T'$.
- If the type of an instruction is changed, that instruction and all of its uses must be pushed to the worklist.
- If the type of a function parameter is changed, all uses of that parameter must be pushed to the worklist.
- If the return type is changed, all return instructions must be pushed to the worklist.
- If the type of a global variable or a struct is changed, all uses in all function variants of that variable or struct must be pushed to their worklist. We refer to those types as *global types*.

- If the type of a local value v is changed from a reference annotated with α to either a reference annotated with $\beta \neq \alpha$ or to an owned pointer, the following instructions must be pushed to the worklist:
  - All return instructions strictly dominated by v.
  - All phi instructions that have a predecessor block whose branch instruction is strictly dominated by v.
  - All instruction strictly dominated by v, whose call to `visit` uses the origin relation.

  This is necessary because changing the annotation of a reference changes the origin relation.

We assume a function `update_type` that changes a type and implements the conditions above.

The function `visit` ensures that the type of an instruction and its operands can be justified by a typing rule, disregarding its post conditions. For most instructions this is done in the following, canonical way: First, the weakest possible types for the instruction is determined, depending on the current type of its operands. This type is then joined with the current type of the instruction. After that, the types are adjusted as needed.

An important special case is visiting a call instruction that calls a function defined in the module. Whether the typing of such a call instruction is valid depends on whether a variant exists that can be instantiated with the operands of the call instruction. Our analysis does this lazily: When a call instruction is visited, it "queries" the signature derived from the operands of the call instruction and its current type from the outer worklist algorithm. The outer worklist algorithm either returns a function variant or it returns that no such variant exists. In the latter case, we return from `visit` without changing any types. Note that this may result in the type of the instruction to be `!`. This is only temporary, because as soon as the queried variant is analyzed, the outer worklist algorithm will add our call instruction to the worklist and call the inner worklist algorithm. We discuss visiting call instructions in Section 6.6.

For all instruction except phi instructions, if an operand of the visited instruction is `!`, we can return from `visit` without changing any types. This may be the case after visiting a call instruction, as discussed before. This may also be the case if the operand is an instruction that has not yet been visited. Phi instructions are discussed in Section 6.5.

6.4 Outer Worklist Algorithm

Figure 6.3 shows pseudocode for the outer worklist algorithm. The algorithm is called with a non-empty set of signatures, which is used to initialize the outer worklist. We call the variants initialized from the given signatures the *initial* variants. In practice, if an instrumentation pass uses the analysis, it can call the algorithm with a signature of the

```
function analyze(init: Set<Signature>)
    worklist ← new Worklist();
    lookup ← new Map<Signature, Variant>();
    queries ← new Map<Instruction, Signature>();
    initialize global types;
    foreach sig ∈ init do
        sig′ ← sig.generalize();
        v ← new Variant(sig');
        lookup[sig′] ← v;
        worklist.push(v);
    while not worklist.empty() do
        v ← worklist.pop();
        sig_changed ← analyze_variant(v);
        if sig_changed then
            sig ← v.get_signature().generalize();
            lookup[sig] ← v;
            foreach variant v′ do
                foreach call instruction i ∈ v′ that calls the function of v do
                    v′.worklist.push(i);
                    worklist.push(v′);
        foreach sig ∈ queries do
            if sig ∉ lookup then
                v′ ← new Variant(sig);
                lookup[sig] ← v′;
                worklist.push(v′);
        queries.clear();
    function query(i: Instruction, sig: Signature)
        sig′ ← sig.generalize();
        if sig′ ∈ lookup then
            return lookup[sig′];
        else
            queries[i] ← sig′;
            return null;
```

Figure 6.3: Outer worklist algorithm.

main function. `lookup` is a map from normalized signatures to variants. This is used to implement `query`, which is used in the inner worklist algorithm. The unresolved queries are stored in `queries`, which is a map from call instructions to normalized signatures.

In the while loop, we analyze all function variants in the worklist using the inner worklist algorithm. When the signature of a variant changed, we normalize the new signature, map it to the variant and we push every potential use of the variant to the worklist. After that, we process all queries that occured in the inner worklist algorithm by instantiating the generalized signatures as variants. Note that the signatures inserted into `queries` are already generalized.

Global types are stored in a map accessible from the inner worklist algorithm, where each global variable or struct is initialized with its weakest possible type. When a global type is changed in the inner worklist algorithm, it may update the worklist of other function variants. In that case, these function variant must be pushed to the outer worklist. This must be done in `update_type` from the inner worklist algorithm.

The reason why `queries` is a map is to reduce the number of initialized function variants: In the inner worklist algorithm, the same call instruction may be visited multiple times, querying different signatures. In the end, we only need the last signature queried by that instruction.

Note that when the signature changed after the inner worklist algorithm, the new signature may already be mapped in `lookup`. In that case, one can either keep the old variant or overwrite it with the new one, which is done in the given pseudocode. In both cases, the now unused variant may still be in the worklist. While this does not harm the correctness of the algorithm, it can harm its performance, therefore we recommend removing the unused variant from the worklist.

We define the call graph of function variants as follows: whenever a variant contains a function call to a defined function, there is an edge from that variant to the variant resulting from the query of that call instruction. After executing the algorithm, we can drop every variant that is not reachable from one of the initial variants.

6.5 Phi Instructions

In this section, we describe how phi instructions are visited in the inner worklist algorithm. Conceptually, this is done in the canonical way as described before: First, we compute the weakest possible type depending on the types of the incoming values. This type is then joined with the current type of the phi instruction and after that, the types of the operands are adjusted to fit the typing rules.

The weakest possible type is computed by joining all incoming values, while using a slightly modified join operator: Recall that $\preceq$ refers to the origin ordering, defined in Section 4.3. Let $\sqsubseteq_\preceq$ be the smallest partial ordering, such that:

- $T \sqsubseteq_\preceq U$ if $T \sqsubseteq U$,
- $?\alpha\ T \sqsubseteq_\preceq ?\beta\ T$ if $\alpha \preceq \beta$,

- `&`α $T \sqsubseteq_\preceq$ `&`β T if $\alpha \preceq \beta$,
- `&`α `mut` $T \sqsubseteq_\preceq$ `&`β `mut` T if $\alpha \preceq \beta$.

With that, we define $T \sqcup_\preceq U$ to be the least upper bound of T and U regarding $\sqsubseteq_\preceq$.

When visiting a phi instruction, we compute the weakest possible type by joining the type of all incoming values using $\sqcup_\preceq$. If the joined type is a reference to T annotated with α, we replace α by a lifetime identifier $\beta \preceq \alpha$ such that the value associated to β strictly dominates the phi instruction. If no such β exists, we change the weakest possible type to a unique pointer to T.

6.6 Call Instructions

In this section, we describe how function calls are visited in the inner worklist algorithm. Assume we visit a call to a function f defined in the LLVM module with the arguments $v_1, \ldots, v_n$. If one of $\mathcal{T}(v_1), \ldots, \mathcal{T}(v_n)$ is `!`, we return from `visit`.

Let t be the current type of the call instruction. We define the signature

$$s := (f, (\mathcal{T}(v_1), \ldots, \mathcal{T}(v_n)), t, \{\}).$$

This signature is queried from the outer worklist algorithm. If a variant $\tilde{f}$ is returned, we instantiate the signature of $\tilde{f}$ with s, resulting in a signature s':

$$s' = (f, (t_1, \ldots, t_n), t', A).$$

Note that s is always weaker than the signature of $\tilde{f}$. Therefore, the instantiation is always defined and because of the way it is defined, we know that $\mathcal{T}(v_i) \sqsubseteq t_i$ for all $i \in \{1, \ldots, n\}$, and also $t \sqsubseteq t'$. Therefore, we update the type of all v_i to t_i and we update the type of the call instruction to t'. If *freeunsafe* $\in A$, we annotate the call instruction as well as the current function variant as freeunsafe.

6.7 Examples

6.7.1 Running Example

In this section, we demonstrate the RustiC analysis on our running example given in Figure 4.2. We initialize the analysis with the function `f`. The outer worklist algorithm starts by analyzing `f` using the inner worklist algorithm. The initial state of `f` is shown in Figure 6.4. `%x` is typed `Unique<i32>` and `%x2` is typed `?'a i32`, because the lifetime identifier of `%x` is `'a`. When visiting `%alias`, the signature

$$(\texttt{idx}, (\texttt{?'a i32}, \texttt{i32}), \texttt{!}, \{\})$$

is queried, which is normalized to

$$(\texttt{idx}, (\texttt{?'A i32}, \texttt{i32}), \texttt{!}, \{\}).$$

Since such a variant does not yet exist, the signature is be inserted into `queries` and the query returns null, such that no types are changed. Therefore, visiting the store instruction does nothing because its operand `%alias` is `!`. Visiting `rustic.free` only changes its type to `void`, since its operand `%alias` is already an owned pointer. Visiting the return instruction changes its type to `void` and updates the return type of `f` to `void`. This finishes the inner worklist algorithm for `f`.

In the outer worklist algorithm, the query from before is instantiated into a new function variant for `idx`. The state of the analysis after that is shown in Figure 6.5. Since `idx` was added to the outer worklist, it is analyzed next. `%idx.ext` is typed `i64` and `%add.ptr` is typed `?'a i32`. When visiting the return instruction, the origin of the operand `%add.ptr` is determined to be `'A`, therefore the return type of the function is updated to `?'A i32`, changing the signature of the variant. This finishes the inner worklist algorithm for `idx`. After that, since the signature of `idx` changed, the call to `idx` from `f` is pushed into the worklist of `f` and `f` is pushed into the outer worklist.

Since `f` is now in the worklist, it is analyzed again. The inner worklist algorithm visits `%alias`, which again queries the signature

$$(\texttt{idx}, (\texttt{?'a i32}, \texttt{i32}), \texttt{!}, \{\}),$$

which is generalized to

$$(\texttt{idx}, (\texttt{?'A i32}, \texttt{i32}), \texttt{!}, \{\}).$$

This signature is now mapped to the variant of `idx` from before, whose signature is

$$(\texttt{idx}, (\texttt{?'A i32}, \texttt{i32}), \texttt{?'A i32}, \{\}),$$

which is instantiated into

$$(\texttt{idx}, (\texttt{?'a i32}, \texttt{i32}), \texttt{?'a i32}, \{\}).$$

Therefore, the type of `%alias` is updated to `?'a i32`. This pushes the store instruction to the inner worklist, because it uses the operand `%alias`. Analyzing the store instruction only changes its type to `void`.

This finishes the inner worklist algorithm and also the outer worklist algorithm. The final result of the analysis is shown in Figure 6.6. We did not mention the lifetime analysis here, because during the analysis of this example, it never happens that any post conditions are violated. In the next section, we see an example where this happens.

```
fn f() -> ! {
    entry:
'a:     %x = call i32* @rustic.alloc.0(i64 40): !
'b:     %x1 = call i32* @rustic.weaken.0(i32* %x): !
'c:     %alias = call i32* @idx(i32* %x1, i32 4): ! (!, i32)
        store i32 10, i32* %alias, align 4: !
        call void @rustic.free.0(i32* %x): !
        ret void: !
}
```

Figure 6.4: Analysis of running example from Figure 4.2.

```
fn f() -> void {
    entry:
'a:     %x = call i32* @rustic.alloc.0(i64 40): Unique<i32>
'b:     %x1 = call i32* @rustic.weaken.0(i32* %x): ?'a i32
'c:     %alias = call i32* @idx(i32* %x1, i32 4): ! (?'a i32, i32)
        store i32 10, i32* %alias, align 4: !
        call void @rustic.free.0(i32* %x): void
        ret void: void
}

fn idx<'A>(
'a:     i32* %arr: ?'A i32
        i32 %idx: i32
) -> ! {
    entry:
        %idx.ext = sext i32 %idx to i64: !
'b:     %add.ptr = getelementptr inbounds i32,
            i32* %arr, i64 %idx.ext: !
        ret i32* %add.ptr: !
}
```

Figure 6.5: Analysis of running example from Figure 4.2, continued.

```
fn f() -> void {
    entry:
'a:     %x = call i32* @rustic.alloc.0(i64 40): Unique<i32>
'b:     %x1 = call i32* @rustic.weaken.0(i32* %x): ?'a i32
'c:     %alias = call i32* @idx(i32* %x1, i32 4): ?'a i32 (?'a i32, i32)
        store i32 10, i32* %alias, align 4: void
        call void @rustic.free.0(i32* %x): void
        ret void: void
}

fn idx<'A>(
'a:     i32* %arr: ?'A i32
        i32 %idx: i32
) -> ?'A i32 {
    entry:
        %idx.ext = sext i32 %idx to i64: i64
'b:     %add.ptr = getelementptr inbounds i32,
            i32* %arr, i64 %idx.ext: ?'a i32
        ret i32* %add.ptr: void
}
```

Figure 6.6: Analysis of running example from Figure 4.2, final.

6.7.2 Modified Running Example

We modify our running example by moving the assignment after the call to free, which is shown in Figure 6.7. This introduces a use after free, because the memory referenced by `alias` is written to after it is freed.

We again initialize our analysis with the function `f`. The analysis starts very similar to our previous example. When we reach the point where that analysis ended, there is a violation of a post condition: Figure 6.8 shows the state of the analysis at that point, including lifetime the results of the lifetime analysis. The store instruction uses `%alias`, which is not alive. Therefore, `%alias` is updated to `Unique<i32>`, which results in a new query

$$(\texttt{idx}, (\texttt{?'a i32}, \texttt{i32}), \texttt{Unique<i32>}, \{\}).$$

The outer worklist algorithm later initializes a function variant for `idx`, whose signature after its analysis ends up being

$$(\texttt{idx}, (\texttt{Unique<i32>}, \texttt{i32}), \texttt{Unique<i32>}, \{\}).$$

Therefore, the type of `%x1` is updated to `Unique<i32>`.

After performing lifetime analysis, which is shown in 6.9, we have another violation of a post condition: In the call to `free`, the operand `%x` is not alive. Therefore, the type of `%x` is updated to `Shared<i32>`. This annotates the call to free and therefore the variant of `f` as freeunsafe. The type of `%x1` is updated to `Shared<i32>`, which results in a new query

$$(\texttt{idx}, (\texttt{Unique<i32>}, \texttt{i32}), \texttt{Shared<i32>}, \{\}),$$

The outer worklist algorithm later initializes a function variant for `idx`, whose signature after its analysis ends up being

$$(\texttt{idx}, (\texttt{Shared<i32>}, \texttt{i32}), \texttt{Shared<i32>}, \{\}).$$

Therefore, the type of `%alias` is updated to `Shared<i32>`.

This finishes the inner worklist algorithm and also the outer worklist algorithm. The final result of the analysis is shown in Figure 6.10. Note that `%x`, `%x2` and `%alias` are now shared pointers. This means that when the program is instrumented, writing to `%alias` will execute a run-time check, which will abort the program because the object it pointed to will have been freed by the call to `rustic.free`.

```
int* idx(int* arr, int idx) {
    return arr + idx;
}

void f() {
    int* x = (int*)malloc(10 * sizeof(*x));
    int* alias = idx(x, 4);
    free(x);
    *alias = 10;
}
```

(a) C source.

```
define i32* @idx(i32* %arr, i32 %idx) {
entry:
  %idx.ext = sext i32 %idx to i64
  %add.ptr = getelementptr inbounds i32, i32* %arr, i64 %idx.ext
  ret i32* %add.ptr
}

define void @f() {
entry:
  %x = call i32* @rustic.alloc.0(i64 40)
  %x1 = call i32* @rustic.weaken.0(i32* %x)
  %alias = call i32* @idx(i32* %x1, i32 4)
  call void @rustic.free.0(i32* %x2)
  store i32 10, i32* %alias, align 4
  %x2 = call i32* @rustic.weaken.0(i32* %x)
  ret void
}
```

(b) Clarified IR.

Figure 6.7: Modified running example.

```
fn f() -> void {
    entry:
        {}
'a:     %x = call i32* @rustic.alloc.0(i64 40): Unique<i32>
        {'a}
'b:     %x1 = call i32* @rustic.weaken.0(i32* %x): ?'a i32
        {'a, 'b}
'c:     %alias = call i32* @idx(i32* %x1, i32 4): ?'a i32 (?'a i32, i32)
        {'a, 'b, 'c}
        call void @rustic.free.0(i32* %x): void
        {}
        store i32 10, i32* %alias, align 4: void
        {}
        ret void: void
}

fn idx<'A>(?'A i32, i32) -> ?'A i32 { ... }
```

Figure 6.8: Analysis of modified running example.

```
fn f() -> void {
    entry:
        {}
'a:     %x = call i32* @rustic.alloc.0(i64 40): Unique<i32>
        {'a}
'b:     %x1 = call i32* @rustic.weaken.0(i32* %x): Unique<i32>
        {'b}
'c:     %alias = call i32* @idx(i32* %x1, i32 4): Unique<i32> (Unique<i32>, i32)
        {'c}
        call void @rustic.free.0(i32* %x): void
        {}
        store i32 10, i32* %alias, align 4: void
        {}
        ret void: void
}

fn idx<'A>(?'A i32, i32) -> ?'A i32 { ... }
fn idx(Unique<i32>, i32) -> Unique<i32> { ... }
```

Figure 6.9: Analysis of modified running example, continued.

```
fn f() -> void @freeunsafe {
    entry:
'a:     %x = call i32* @rustic.alloc.0(i64 40): Shared<i32>
'b:     %x1 = call i32* @rustic.weaken.0(i32* %x): Shared<i32>
'c:     %alias = call i32* @idx(i32* %x1, i32 4): Shared<i32> (Shared<i32>, i32)
        call void @rustic.free.0(i32* %x): void @freeunsafe
        store i32 10, i32* %alias, align 4: void
        ret void: void
}

fn idx<'A>(?'A i32, i32) -> ?'A i32 { ... }
fn idx(Unique<i32>, i32) -> Unique<i32> { ... }
fn idx(Shared<i32>, i32) -> Shared<i32> { ... }
```

Figure 6.10: Analysis of modified running example, final.

7 Evaluation

To evaluate our thesis, we implemented RustiClarify and the RustiC analysis and ran them on different test cases. In the following, we describe some implementation details relevent for the evaluation, the evaluation approach, i.e. how the results of the RustiC analysis were evaluated, and the results.

7.1 Implementation

We implemented RustiClarify and the RustiC analysis as LLVM passes. To analyze a set of C sources, we did the following:

1. Each C source is compiled into an LLVM module without any optimization, to which we apply the two passes `mem2reg` and `dce`.
2. The modules are linked together into a single module.
3. We run RustiClarify and the RustiC analysis on the resulting module, outputting a RustiC type assignment and some statistics. The analysis is initialized with the weakest signature of every function defined in the module.

Figure 7.1 shows the used commands. For technical reasons, our implementation extends the weaken restriction to all calls that do not call `rustic.weaken`, i.e. calls to all other RustiC intrinsics are prepended by `rustic.weaken` calls. This results in a very high amount of inserted `rustic.weaken` calls, e.g. whenever a pattern is replaced that inserts a call to `rustic.ptrswap`, three `rustic.weaken` calls are inserted, whereas the methods described in our thesis would only insert one `rustic.weaken` call. We mention this detail here because this influences the generated statistics for our evaluation, which are described in the next section.

7.2 Approach

We evaluate our thesis in two ways: we generate statistics by counting the number of pointer types and the number of certain operations, and we manually inspect the type assignments to see which patterns our analysis can handle well and which ones cause problems.

In some test cases, we had to modify the sources such that we could analyze them. This is mainly because of function pointers: some of the test cases were libraries that support custom allocators by passing function pointers to allocation functions that are

```
# Compiling .c to LLVM IR:
clang -S -emit-llvm -fno-discard-value-names -Xclang -disable-O0-optnone \
    -O0 -o source.ll source.c
opt -S -mem2reg -dce -o source.opt.ll source.ll

# Linking:
llvm-link -S -o output.ll source1.opt.ll source2.opt.ll ...

# Analyzing:
opt -load lib/LLVMRustiClarify.so -load lib/LLVMRustiC.so -analyze -stats \
    -rusticlarify -rustic output.ll
```

Figure 7.1: Commands used for evaluation.

called instead of `malloc` and `free`. Before we analyzed those cases, we had to perform "manual function pointer inlining", i.e. we had to remove all function pointers and replace the calls of them by calls to `malloc` and `free`.

We collected the following statistics for each test case:

- The lines of code in the original sources, how many of them we analyzed, and how many we needed to modify.
- Number of instructions, defined functions, initialized variants and variants reachable from the initial variants.
- How many global variables are shared or unsafe pointers.
- For global variables, which are either shared or unsafe pointers, we count the target pointer type and all recursive target pointer types. For struct members which are pointers, we count the pointer type and all recursive target pointer types. We refer to all these pointers as *in-memory* pointers.

  Note that array types in LLVM are represented by owned pointers in RustiC. Because they are no actual pointers, we excluded those cases, i.e. a pointer mapped to a LLVM array is not counted as an in-memory pointer.
- The amount of pointer types of all instructions, function parameters and function return types.
- For unsafe, shared and unique pointers: The amount of pointer dereferences, loads and stores. A pointer dereference occurs everytime a pointer is loaded from or stored to, i.e. the pointer is used as the pointer operand in `load` or `store`. For shared pointers, implicit downcast are also counted as a dereference.

  A pointer load occurs when a pointer results from a `load` instruction, i.e. when a pointer is loaded from memory and the resulting pointer is an owned pointer, we count that as a pointer load of that owned pointer. Similarly, when an owned pointer is stored in memory, we count the stored pointer as a pointer store.

- The amount of reference dereferences. We combined this number with the number of unique pointer dereferences because dereferencing references, just like unique pointers, needs no run-time checks.

7.3 Results

We tested the following projects:

- `bzip2-1.0.8` [4], which is a widely used data compression tool. Here, we analyzed everything necessary to compile `libbz2.a`. The code included use of function pointers to allow custom allocators, which were replaced.

  Here, we analyzed all 3571 lines of code and modified 19 lines of code.

- `brotli-1.0.7` [2], which is a data compression tool developed by Google. Here, we analyzed everything necessary to compile all C sources in `c/dec`, which is the code for decompressing data. It also included function pointers for custom allocators, which were replaced.

  Here, we analyzed all 10980 lines of code and modified 37 lines of code.

- `bstring-0.1.1` [3], which is a small library defining a data structure for strings. It included some variadic functions, which we could not analyze and therefore removed from the code.

  Here, we analyzed 2708 out of the 4273 line of code and modified 0 lines of code.

- `aiger-1.9.9` [1], which is a project to work with And-Inverter Graphs. We analyzed the AIGER C library, which consists of only one source: `aiger.c`. It included function pointers for custom allocators, which were replaced. It also included some functions using file I/O, which we removed because our implementation could not handle it.

  Here, we analyzed 950 out of the 2262 line of code and modified 30 lines of code.

- The Olden benchmark [5], which consists of 10 test cases. For this test case, we had to do some technical changes and some manual function pointer inlining.

  Here, we analyzed all 5026 lines of code and modified 109 lines of code.

The generated statistics are shown in Figures 7.2, 7.3, 7.4 and 7.5. As seen in Figure 7.4, the first four test cases contain a lot of unsafe pointers. We were able to eliminate all of them with very minor changes to the C sources, which are discussed in Section 7.6. The modified test cases are named with the suffix `-fixed` and are shown in the last four rows of each table. Because our further discussion only focuses on the modified test cases, we omit the `-fixed` prefix, e.g. when mentioning `aiger`, we actually refer to `aiger-fixed`.

Test case	Instructions	Functions	Variants	Reachable variants
aiger	2485	44	54	51
brotli-dec	17030	56	145	103
bstring	5321	77	107	90
bzip2	15037	45	85	81
olden-bh	2126	35	51	45
olden-bisort	303	13	21	15
olden-em3d	769	29	33	30
olden-health	624	13	44	18
olden-mst	478	16	28	18
olden-perimeter	427	12	23	13
olden-power	1097	17	20	17
olden-treeadd	76	4	4	4
olden-tsp	527	14	50	18
olden-voronoi	1903	45	92	57
aiger-fixed	2343	44	50	49
brotli-dec-fixed	17032	56	69	63
bstring-fixed	5320	77	98	81
bzip2-fixed	15039	45	47	47

Figure 7.2: Size of test cases and the number of function variants.

	Globals		In-Memory		
Test case	`Unsafe`	`Shared`	`Unsafe`	`Shared`	`Unique`
aiger	0	55	11	1	3
brotli-dec	3	41	24	0	0
bstring	0	4	1	1	2
bzip2	0	4	18	0	0
olden-bh	0	14	0	12	0
olden-bisort	0	11	0	3	0
olden-em3d	0	28	0	10	3
olden-health	0	12	0	6	0
olden-mst	0	18	9	0	0
olden-perimeter	0	4	0	5	0
olden-power	0	9	0	0	5
olden-treeadd	0	5	0	0	2
olden-tsp	0	11	0	4	0
olden-voronoi	0	31	9	0	3
aiger-fixed	0	55	0	11	4
brotli-dec-fixed	0	44	0	18	6
bstring-fixed	0	4	0	2	2
bzip2-fixed	0	4	0	15	3

Figure 7.3: Statistics about global variables and structs.

Test case	`Unsafe`	`Shared`	`Unique`	`&`	`&mut`	`?`
aiger	311	4	105	62	83	1431
brotli-dec	10117	0	297	26	23	7966
bstring	703	22	306	24	104	987
bzip2	6599	0	47	1	6	2983
olden-bh	0	130	62	0	0	784
olden-bisort	0	40	14	0	0	134
olden-em3d	0	22	17	0	0	352
olden-health	0	110	42	0	0	372
olden-mst	170	0	10	0	0	127
olden-perimeter	0	22	2	0	0	153
olden-power	0	0	36	0	0	368
olden-treeadd	0	0	8	0	0	33
olden-tsp	0	197	26	0	0	297
olden-voronoi	1031	0	62	0	2	448
aiger-fixed	0	62	128	68	83	1526
brotli-dec-fixed	0	419	168	26	23	9083
bstring-fixed	0	38	316	24	106	1486
bzip2-fixed	0	660	67	1	6	5498

Figure 7.4: Statistics for pointer types of instructions.

	`Unsafe`			`Shared`			`Unique`/reference		
Test case	deref	load	store	deref	load	store	deref	load	store
aiger	21	83	29	208	1	4	339	5	8
brotli-dec	4320	901	393	441	0	0	4720	0	1
bstring	143	305	23	51	1	0	800	4	7
bzip2	3493	701	98	43	0	0	1778	0	0
olden-bh	0	0	0	150	30	29	276	0	0
olden-bisort	0	0	0	42	8	12	56	0	0
olden-em3d	0	0	0	105	9	18	112	0	3
olden-health	0	0	0	141	15	21	128	0	0
olden-mst	49	41	19	37	0	0	28	0	0
olden-perimeter	0	0	0	65	0	10	41	0	0
olden-power	0	0	0	38	0	0	201	0	5
olden-treeadd	0	0	0	7	0	0	7	0	2
olden-tsp	0	0	0	225	37	82	150	0	0
olden-voronoi	189	160	37	45	0	0	132	1	2
aiger-fixed	0	0	0	249	27	28	348	8	13
brotli-dec-fixed	0	0	0	950	203	216	4751	8	12
bstring-fixed	0	0	0	327	8	16	885	4	7
bzip2-fixed	0	0	0	1641	73	79	3407	7	10

Figure 7.5: Amount of dereferences, loads and stores.

As seen in Figures 7.4 and 7.5, in most test cases that do not contain unsafe pointers, the majority of pointers are typed as references and also most dereferences are of unique pointers or references, all of which require no run-time checks.

The best case in that regard is `brotli-dec`: 96% of all pointers are typed as unique pointers or references. Also, 83% of all dereferences are of unique pointers or references.

The worst case in that regard is `olden-tsp`: only 60% of all pointers are typed as unique pointers or references, and 40% of all dereferences are of unique pointers or references.

We can also see that our type system is pretty prone to propagating unsafe pointers: in most test cases, if there is an unsafe pointer, the majority or at least a great proportion of all pointers end up as unsafe pointers. In Section 7.6, we show how small changes to the code can eliminate a huge amount of unsafe pointers, e.g. in `brotli-dec`, we changed 5 lines of code, which eliminated all unsafe pointers in that test case.

One notable exception is the unmodified `aiger` test case. It does contain unsafe pointers, but only 16% of all pointers are unsafe pointers and only 4% of all dereferences are unsafe pointer dereferences. Note that this test case contains a lot of unsafe pointer loads and stores, as about 86% of all loads and stores are unsafe pointers, which may impose some run-time overhead.

7.4 Good Patterns

In this section we describe patterns for which our analysis is able to infer unique pointers or references.

- Functions that use temporarily allocated data structures, i.e. functions allocating memory, using it, and freeing it afterwards such that it is not used after the function returns. While this pattern works very well in smaller examples, the test cases we evaluated did not contain this pattern directly. This pattern works best if the pointers are local variables, but often times, the pointers are stored in structs.

  As an example, `bzip2` and `brotli-dec` define structs that represent the state of a compression or decompression in progress. This state is implemented as a struct and there are functions to initialize the state, to compress or decompress data and to finalize the operation and free allocated memory. When used correctly, e.g. by initializing a state, operating on some data and then finalizing the operation, this is an indirect instance of the temporary allocation pattern, where the temporary buffers are stored in the state struct. As it turns out, some of the pointers in these structs are typed as unique pointers.

- Functions that only use allocated memory, but do not allocate or free memory or shift pointers around in memory. Owned pointers are only needed when allocating memory, when pointers loaded from non-unique pointers are used after a freeunsafe instruction or when pointers are stored in memory. For functions where

none of this is the case, many of them can be typed only using weak references. This is the main reason why some test cases are dominated by weak references, as seen in Figure 7.4.

- In `aiger`, our analysis was able to identify in-memory pointers as unique pointers that were reallocated frequently: `aiger` implements array lists using macros, which use `realloc` to grow the array, while the pointer to the arrays are stored in a struct. Three of these array lists were typed as unique pointers.

- Another more specific pattern is when memory is allocated, operations are performed on that memory and then the pointer to this memory is stored in memory. Because our type system allows implicitly downcasting unique pointers to shared pointers, the pointer can be used as a unique pointer until it is stored in memory, even if the pointer in memory is a shared pointer.

7.5 Bad Patterns

In this section we discuss patterns that prevent typing pointers as references or unique pointers and result in shared pointers.

- One obvious case are pointers that are supposed to alias, e.g. when working with a graph structure whose nodes store pointers to adjacent nodes.

- One problem is that moving unique pointers out of memory is currently only possible with `rustic.ptrswap`, which is only inserted in very specific patterns by RustiClarify. This prevents pointers being typed as unique pointers if they are permuted in memory: Assume two unique pointers stored in memory and the program swaps these two pointers. Swapping them involves two loads and two stores, where one load and store may be replaced by RustiClarify into a `rustic.ptrswap`, but not both. This means that there is a load of an owned pointer, which is typed as a shared pointer because unique pointers can only be loaded using `rustic.ptrswap`. This problem occured e.g. in `olden-bisort`.

- Another related problem is an indirect load+free or load+realloc, e.g. if a pointer is loaded, the value is passed to a function and the free or the realloc is performed in that function. In those cases, RustiClarify does not detect the load+free or load+realloc pattern and therefore does not insert a `rustic.ptrswap`.

- Using a pointer after it was stored in memory. Whenever a pointer is stored in memory, it must be an owned pointer. If a unique pointer is stored in memory, it is moved, i.e. it cannot be used after the store. Therefore, if a pointer is used after it was stored, it must be at least a shared pointer. In some cases, it is possible to fix this problem by re-loading the pointer after the store, but such a transformation is currently not supported by RustiClarify.

- Loading a pointer from an object that is deleted afterwards. In that case, the loaded pointer will be at least a shared pointer. If the pointer in the object is never used again between the load and the deletion, one could replace the load by a `rustic.ptrswap`, allowing a load of a unique pointer. However, we did not find a good way of implementing such a pattern in RustiClarify.
- Some shared pointers result from the fact that our type system does not support storing references in memory or in structs. This occured in some cases where a function has multiple return values, which is either implemented by passing it a pointer to which the result is written or by returning a struct.
- Some shared pointers result from the lack of combined lifetimes, e.g. when code selects one of multiple pointers to different objects depending on some condition. This is further discussed in Chapter 8.

7.6 Eliminating Unsafe Pointers

In this section, we discuss how we eliminated all unsafe pointers in the test cases `aiger`, `brotli-dec`, `bstring` and `bzip2`. We also mention how many lines of code we changed in each of these cases.

7.6.1 aiger

The `aiger` C library uses array lists, whose dynamic growth is conceptually implemented with `realloc`. Because the library uses generic allocators (implemented with function pointers) which do not require a `realloc` method, `aiger` defines a macro that implements `realloc` using `malloc`, `memcpy` and `free`. The macro after the function pointer inlining is shown in Figure 7.6. The problem is that the pointer returned by `malloc` is used by `memcpy` and `memset` before being casted to the right pointer type, such that RustiClarify does not recognize the malloc+bitcast pattern. This leads to unsafe pointers in all cases where `p` is a pointer to a type that contains pointers. A straightforward fix is to use `realloc`, which is also shown in Figure 7.6. Here, we changed 5 lines of code.

7.6.2 brotli-dec

In `brotli-dec`, the code in the file `dec/state.c` contains an optimization to allocate two objects with one allocation, shown in Figure 7.7. In our analysis, the cast from `HuffmanCode**` to `HuffmanCode*` is an unsafe bitcast, resulting in unsafe pointers. Also, the library uses some `ptrtoint` instructions, which results in a lot of unsafe pointers because of the existence of an unsafe bitcast. To fix this issue, one needs to replace the one allocation by two distinct allocations, also shown in Figure 7.7. In the function that frees the allocated memory, we also had to add one more line of code to free the second allocated object. Here, we changed 5 lines of code.

```
#define REALLOCN(p,m,n) \
  do { \
    size_t mbytes = (m) * sizeof (*(p)); \
    size_t nbytes = (n) * sizeof (*(p)); \
    size_t minbytes = (mbytes < nbytes) ? mbytes : nbytes; \
    void * res = malloc (nbytes); \
    memcpy (res, (p), minbytes); \
    if (nbytes > mbytes) \
      memset ((p) + m, 0, nbytes - mbytes); \
    free ((p)); \
    (p) = res; \
  } while (0)
```

(a) Original code.

```
#define REALLOCN(p,m,n) \
  do { \
    size_t mbytes = (m) * sizeof (*(p)); \
    size_t nbytes = (n) * sizeof (*(p)); \
    (p) = realloc ((p), nbytes); \
    if (nbytes > mbytes) \
      memset ((p) + m, 0, nbytes - mbytes); \
  } while (0)
```

(b) Modified code.

Figure 7.6: Eliminating unsafe pointers in `aiger`.

```
BROTLI_BOOL BrotliDecoderHuffmanTreeGroupInit(BrotliDecoderState* s,
    HuffmanTreeGroup* group, uint32_t alphabet_size, uint32_t max_symbol,
    uint32_t ntrees) {
  /* Pack two allocations into one */
  const size_t max_table_size = kMaxHuffmanTableSize[(alphabet_size + 31) >> 5];
  const size_t code_size = sizeof(HuffmanCode) * ntrees * max_table_size;
  const size_t htree_size = sizeof(HuffmanCode*) * ntrees;
  /* Pointer alignment is, hopefully, wider than sizeof(HuffmanCode). */
  HuffmanCode** p = (HuffmanCode**)BROTLI_DECODER_ALLOC(s,
      code_size + htree_size);
  group->alphabet_size = (uint16_t)alphabet_size;
  group->max_symbol = (uint16_t)max_symbol;
  group->num_htrees = (uint16_t)ntrees;
  group->htrees = p;
  group->codes = (HuffmanCode*)(&p[ntrees]);
  return !!p;
}
```

(a) Original code.

```
BROTLI_BOOL BrotliDecoderHuffmanTreeGroupInit(BrotliDecoderState* s,
    HuffmanTreeGroup* group, uint32_t alphabet_size, uint32_t max_symbol,
    uint32_t ntrees) {
  const size_t max_table_size = kMaxHuffmanTableSize[(alphabet_size + 31) >> 5];
  const size_t code_size = sizeof(HuffmanCode) * ntrees * max_table_size;
  const size_t htree_size = sizeof(HuffmanCode*) * ntrees;
  HuffmanCode** p = (HuffmanCode**)BROTLI_DECODER_ALLOC(s, htree_size);
  HuffmanCode* c = (HuffmanCode*)BROTLI_DECODER_ALLOC(s, code_size);
  group->alphabet_size = (uint16_t)alphabet_size;
  group->max_symbol = (uint16_t)max_symbol;
  group->num_htrees = (uint16_t)ntrees;
  group->htrees = p;
  group->codes = c;
  return !!p && !! c;
}
```

(b) Modified code.

Figure 7.7: Eliminating unsafe pointers in `brotli-dec`.

7.6.3 bstring

In `bstring`, the function `bStr2NetStr`, defined in `bstraux.c`, contains the line `bcstrfree((char *)s);`. The function `bcstrfree` only frees the given pointer and `s` is a pointer to a struct containing a pointer to a string buffer. In LLVM, the argument of the call is a bitcast, which results in an unsafe bitcast. To fix this issue, one can simply replace the line above by `free(s);`, such that RustiClarify detects the free+bitcast pattern, preventing the unsafe bitcast and all unsafe pointers in this test case. Here, we changed 1 line of code.

7.6.4 bzip2

This project defines two structs `DState` and `EState` in `bzlip_private.h`. The struct `bz_stream`, defined in `bzlib.h`, contains a `void*`, which can store a pointer to either an instance of `DState` or `EState`. Since a `void*` in C is compiled into an `i8*` in LLVM, casting between `i8*` and `DState*` or `EState*` results in unsafe bitcasts. To fix this issue, we defined an empty struct `State`, which were added as the first member of `DState` and `EState`, making them subtypes of `struct State`. Further, the `void*` was replaced with a `struct State*`, and we replaced the implicit casts between `void*` and `DState*` or `EState*` by explicit casts between `struct State*` and `DState*` or `EState*`, which occured 7 times in the code. Because of the subtyping, these casts can be typed as safe bitcasts, eliminating all unsafe pointers.

This is a rather technical problem that results from the fact that a `void*` is compiled to a `i8*` in LLVM. If there was a `void*` in LLVM, we could define that each type is a subtype of `void*`, eliminating this issue. Another approach would be to extend the RustiC type system such that a pointer to void is compatible to the LLVM type `i8*`, while only allowing certain operations on these pointers. We consider this as future work. Here, we changed 11 lines of code.

7.7 Remaining Unsafe Pointers

In this section, we discuss which patterns resulting in unsafe pointers that are not trivially fixable as discussed in the last section.

- Custom allocator functions. This occured in `olden-mst`, which implemented a custom allocator that allocates chunks of memory using `malloc`, which are divided into objects that are returned by the allocator. The custom allocator returns a pointer into a `i8*` typed object, which is later casted to another pointer type. This results in an unsafe bitcast as soon as a type containing pointers is allocated using the custom allocator.

- Performing arithmetic on the integer values of pointers, i.e. casting between integers and pointers, while performing arithmetic on the integers. This occured in `olden-voronoi`.

- Unions where one of its members contain a pointer. When compiling C to LLVM, unions are compiled into a struct containing the largest member of the union. An access to a union member is compiled into a bitcast. Therefore, when accessing a member that contains a pointer, the corresponding bitcast will be an unsafe bitcast.
- Pointers that are stored as integers in memory, which are later used as pointers. This will result in unsafe pointers, because our type system only allows casting integers to unsafe pointers.

8 Conclusion and Future Work

8.1 Future Work

One major milestone would be to implement an instrumentation pass that instruments a program using the RustiC analysis. Since RustiC needs all defined functions, an instrumentation has to be applied to a program that is compiled to an executable program, i.e. a program that contains a `main` function.

In the following, we describe possible extensions to RustiClarify or the RustiC type system. RustiClarify can always be extended by new patterns. Besides that, one feature we thought of is function poiner inlining: In many cases, generic functions are implemented using function pointers and void pointers. An example for this is the C standard library function `qsort`. It takes a pointer to an array as a void pointer, the number of elements in the array, the size of each element in the array and a comparison function as a function pointer to a function getting two void pointers and returning an integer. To sort an array, the array, as a pointer, is casted to a void pointer and given to the sort function, which calls the comparison function, which itself casts the void pointer back to the original pointer type. In LLVM, these casts will be bitcasts between `i8*` and `L*` for some type `L`. For RustiC, this is not only problematic because of the non-supported function pointers, but also because of the bitcasts, which will be unsafe if `L` contains any pointers.

To eliminate function pointers, one idea is to inline function poiner arguments, i.e. each time a call instruction calls a function defined in a module with a function pointer to another function defined in the module, we clone the called function and replace the function pointer parameter with the argument of the call instruction. While this eliminates some uses of function pointers, it does not eliminate the remaining pointer casts, which we envision could be eliminated with further analysis: Generic function usually only do some pointer arithmetic on the void pointers and pass them to other generic functions or to the given function pointers. After function pointer inlining, the only operations performed on generic parameters are pointer arithmetic and bitcasts to another pointer type. Our idea here is a pass that detects this pattern and eliminates the bitcasts.

Even after all this effort, there still may be function pointers remaining that cannot be inlined, e.g. when function pointers are stored in memory and called later. We saw this in some test cases in the evaluation, which stored function pointers to allocation functions in a struct. To fully support function pointers, RustiC has to be extended to support function pointers.

For RustiC, one major extension is to annotate references with sets of lifetime identifiers instead of a single lifetime identifier. The idea of the annotation of a reference is to identify the origin of the reference, therefore if a reference is annotated with a set of lifetime identifiers, its origin may be any one of the lifetime identifiers in the set. This allows always joining lifetimes in phi instructions, not only if they have a common origin.

Another major extension is pointers to references and references in structs. Especially with the temporary data structure pattern in mind, as discussed in the evaluation, some code stores pointers in such data structures, which are not freed before the temporary data structure is freed. This is currently not supported by our type system, as only owned pointers can be stored in memory, therefore such pointers end up as shared pointers. Another problem are functions that return multiple values. In C, this is often implemented either by functions returning a struct or functions taking the return value by reference, which means they take a pointer to which the result is written. Because there are no pointers to references, all of these pointers must be owned pointers, resulting in a lot of shared pointers.

We also saw in the evaluation that in some cases, there are only a few function that require members of a struct to be shared or unsafe. This may lead to a situation where if a struct is used in many different places in the code independent from each other and only one of them uses one of those functions, the struct member will be a shared or unsafe pointer globally. Our idea here is to implement struct variants with the same idea as function variants, i.e. each time a struct type is used, it may be any of the defined struct variants.

Currently, if a shared pointer is freed, all weak references are invalidated and the function is annotated as freeunsafe, such that calling it also invalidates all weak references at the call site. This comes from the assumption that each shared reference may alias with any other shared reference. While this is true in general, it is a very conservative assumption. One consequence of this is the `barrier` function, which we used in smaller test cases and is shown in Figure 8.1. This function is annotated as freeunsafe and calling it will always invalidate all weak references, although the freed shared pointer clearly does not alias with anything outside the function. Our ideas here are to use some kind of alias analysis to allow omitting the freeunsafe annotation, and to extend freeunsafe by annotating it with a set of function parameters: Whenever a shared pointer is freed that may alias with a function parameter, the freeunsafe annotation is annotated with this parameter. This would allow restricting the amount of weak references which have to be invalidated when calling the function.

Another point is the performance of our RustiC analysis, mainly that it needs an IR that contains all defined functions. While a RustiC type assignments needs to contain all used functions in the end, it may be possible to do some analysis earlier. Our idea here is to infer some kind of abstract rules for each function, something like “this parameter must be at least a unique pointer” or “if this parameter is a shared poiner, the other parameter must be a strong reference”. One way we envisioned to implement this is by creating a small stub function for each function that admits the same typings as the original function. These stub functions are then used to infer all needed function

```
1  void barrier() {
2      int* x = malloc(16);
3      int a = *x;
4      if(a)
5          free(x);
6      if(!a)
7          free(x);
8      //only one free will be executed, but or analysis overapproximates
9      //and types x as a shared pointer, resulting in freeunsafe
10 }
```

Figure 8.1: Barrier function.

signatures. This would save a considerable amount of time especially for functions with a lot of instructions and would also allow more parallelization, as these stub functions could be generated for each compilation unit individually.

One last point is proving the correctness of our approach. This includes a formal proof for the correctness of our type system, i.e. that the properties we claim for each pointer type hold in every execution of the program.

8.2 Conclusion

In this thesis, we use the ideas of ownership and borrowing from the Rust programming language to define a type system that allows infering temporal properties of pointers at compile time. This allows us to reduce the amount as well as the complexity of run-time checks needed to instrument a C program to be temporally memory safe. While dereferncing unique pointers and references needs no run-time checks at all, we discussed the proxy objects approach for shared pointers, which allows a very simple run-time check, consisting of only one additional load instruction in the best case.

We were able to get promising results with only minor changes to the evaluated cases, e.g. we could analyze `brotli-dec`, which contains 10980 lines of code, after modifying only 37 lines of code. Further, in some test cases we could eliminate all unsafe pointers with even smaller changes to the code: in `brotli-dec`, where the majority of pointers were unsafe pointer, we could eliminate all of them while only changing 5 lines of code. In the end, only 2 of our 14 test cases contained unsafe pointers.

The majority of pointer dereferences are unique pointers or references, which do not need any run-time checks. The best case in that regard was `brotli-dec`, where 83% of all dereferences are unique pointers or references. The remaining dereferences that need run-time checks are for shared pointers. Because of that, we think that an instrumentation pass using the RustiC analysis could achive temporal memory safety with relatively low overhead.

Although we were able to get decent results, we still needed to remove some code from the original projects that we could not analyze. There are still some bits missing such that most full programs can be analyzed and instrumented. We mainly consider our thesis as an exploration of the idea of using a Rust-like type system to infer temporal

properties of C programs, that has shown some promising results and offers a wide range of future work for even further improvements.

Bibliography

[1] AIGER. http://fmv.jku.at/aiger/, 2020. Date accessed: 2020-07-09.

[2] Brotli. https://github.com/google/brotli/releases/tag/v1.0.7, 2020. Date accessed: 2020-07-09.

[3] bstring. https://github.com/msteinert/bstring/tree/eb3b7aed6a9dfdd4c3e2a36af9b395d773b3fec4, 2020. Date accessed: 2020-07-09.

[4] bzip2. https://sourceware.org/pub/bzip2/bzip2-1.0.8.tar.gz, 2020. Date accessed: 2020-07-09.

[5] Olden benchmark. https://github.com/llvm/llvm-test-suite/tree/a27411b851a42ba241235235881fd480a8049a94/MultiSource/Benchmarks/Olden, 2020. Date accessed: 2020-07-09.

[6] The Rustonomicon. https://doc.rust-lang.org/nomicon/, 2020. Date accessed: 2020-07-01.

[7] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the Real World. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, page 232–244, New York, NY, USA, 2003. Association for Computing Machinery.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[9] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, page 475–488, New York, NY, USA, 2014. Association for Computing Machinery.

[10] Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. Cyclone: A type-safe dialect of C. *C/C++ Users Journal*, 23(1):112–139, 2005.

[11] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, USA, 2018.

[12] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[13] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, page 31–40, New York, NY, USA, 2010. Association for Computing Machinery.

[14] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, page 128–139, New York, NY, USA, 2002. Association for Computing Machinery.