공학 학사 학위논문

Rust MIR Escape Analysis: An Application to Protecting Dynamic Pointer Metadata

2024년 2월

서울대학교 공과대학

전기·정보공학부

이 재 호

Rust MIR Escape Analysis: An Application to Protecting Dynamic Pointer Metadata

지도교수 백윤흥

이 논문을 공학 학사학위 논문으로 제출함.

서울대학교 공과대학

전기·정보공학부

이 재 호

이재호의 학사 학위 논문을 인준함.

2023년 12월 15일

지도교수 백 윤 흥 (인)

Abstract

This thesis introduces a sound escape analysis for Rust Mid-Level Intermediate Representation (MIR), aimed at optimizing Rust programs by safely eliding heap allocations. Grounded in the abstract interpretation framework, this escape analysis extends the MIRAI abstract interpreter and identifies potential escaping allocations leveraging the strong static type system of Rust. The analyzer has been evaluated using sample code and the publically available `aho-corasick` crate, demonstrating a low false positive rate. This work contributes to the memory management optimization in Rust.

Keywords: escape analysis, memory optimization, static analysis, abstract interpretation, Rust

Contents

List of Figures

Chapter 1

Introduction

Systems programming routinely interacts with low-level hardware resources with a relatively thin layer of abstractions, compared to typical application programming. This interaction demands high performance, as it directly manages time-critical hardware operations. Therefore, systems programming requires a more granular control over computing resources, which are typically abstracted away in high-level application programming. This level of control ensures efficient utilization of system resources, making it a fundamental aspect of operating systems, embedded systems, and other critical software infrastructures.

The C programming language has predominantly been used for systems programming, as it was the language devised to write the UNIX operating system [1]. It was initially designed to be a portable, high-level assembly language, and has been used as a *de facto* standard systems programming language for decades. As a high-level assembly language, C provides a fine control over low-level resources as well as the readability and structures of high-level languages. This balance, combined with its strightforward syntax, led to a widespread adoption of C in systems programming.

However, C puts the burden of resource management on the programmer, which, inevitably, led to numerous errors causing security vulnerabilities. In particular, C programmers are responsible for correctly allocating and freeing memory. It is well-known that even the most experienced programmers make mistakes: Microsoft reported that 70% of security vulnerability fixes are caused by memory safety violations [2]. As such, memory safety is a critical concern in systems programming.

Rust is a new systems programming language that aims to provide the same level of control over low-level resources as C, while preventing memory-safety bugs at compile time using a strong static type system [3]. Unlike previous approaches to manage resources which relied on best-practices, e.g., *Resource Acquisition Is Initialization* or RAII in C++, Rust enforces the safe usage of resources—it refuses to compile if it detects potential memory-safety violations. This proactive stance towards safety is a cornerstone of Rust's design philosophy.

Manual memory management of the data stored on the heap is one of the most common sources of memory-safety bugs in C/C++ programs. Such bugs often arise due to improper allocation, deallocation, or access to memory, leading to issues like memory leaks, dangling pointers, and buffer overflows.

Rust allows for safe heap allocation using the `Box` type, which is a smart pointer that *owns* the data it points to. This ownership implies that a `Box` is responsible for freeing the memory when it is no longer needed. Unlike manual memory management in C, where developers must explicitly allocate and free memory, Rust's `Box` type automates this process[1]. The use of `Box` (and other smart pointers) encapsulates the details of memory management, ensuring that resources are correctly managed, and prevent many of the common pitfalls encountered in traditional systems programming using C.

While it is always safe to allocate data on the heap, in the sense that the program will not lead to memory-safety bugs, allocating data on the heap is more costly than putting the data on the stack. Heap allocation involves complex bookkeeping machinery involving memory management systems, which track allocated and deallocated spaces to efficiently reuse memory. In contrast, stack allocation is much faster and more efficient due to its simple allocation strategy—it is essentially a form of pointer arithmetic. Therefore, it is preferable to avoid heap allocation when the data can simply be stored on the stack.

To this end, this thesis presents a sound analysis that detects heap allocations that can be safely elided. The analysis is based on the abstract interpretation, which is a generalized framework for soundly summarizing concrete program execution behaviors without calculating the exact values. This approach allows us to safely approximate the behavior of Rust programs guided by the semantics of the Rust itself.

[1]The `unique_ptr` type of C++ is similar to `Box` in that the heap data gets freed when the pointer is no longer needed, but `unique_ptr` can still be `nullptr` which leads to potential memory-safety bugs.

1.1 Heap allocation elision

In some scenarios, such as automatic code generation, data is wrapped in a `Box` and stored on the heap even when it is not necessary. Moreover, programmers may use `Box` to avoid the complexity of ownership semantics, which may lead to unnecessary heap allocations. In such cases, it is desirable to elide heap allocations and store data on the stack.

1.1.1 Optimization perspective

There are specific kinds of data that are stored on the heap. These include dynamic data structures that may grow or shrink in size or have undetermined size at compile time. Additionally, large objects are often allocated on the heap, as the limited size of stack may not be appropriate for them. Furthermore, *escaping* objects that outlive the lifetime of a function should be stored on the heap. These objects need storage that is not bound to the scope of a function, unlike the stack.

However, the cost of heap allocation is high compared to stack allocation, as mentioned above. The heap allocation comes with an overhead to track allocated and deallocated spaces to efficiently reuse memory. This additional complexity leads to increased memory and runtime overhead, especially in scenarios with frequent allocations and deallocations.

To optimize the overhead associated with heap allocation, we can elide heap allocations that are not necessary; short-lived objects that do not escape a function can be safely allocated on the stack.

Memory-managed languages like Go perform *escape analysis* to determine the most efficient allocation strategy for each object [4]. This analysis ensures that only the objects that may "escape," or persist beyond the scope of a function, are placed on the heap.

In Rust, we can apply a similar approach for `Box`ed types. By analyzing the usage and lifetime of `Box`ed objects, it is possible to determine whether they can be safely allocated on the stack rather than on the heap. This analysis can be performed statically, leveraging the strong type system of Rust.

1.1.2 Safety perspective

Rust's type system incorporates a concept of *ownership* in order to prevent memory-safety bugs at compile time. This system is central to Rust's approach to ensuring safe memory management and preventing concurrency issues.

In some circumstances, however, this strict enforcement of safety can sometimes prevent efficient implementations of certain data structures, such as doubly-linked lists. This is undesirable for any programming language, and even more so for a *systems* programming language like Rust.

To address these limitations, Rust introduces `unsafe` blocks, allowing programmers to perform operations that are typically prohibited by the safety guarantees of Rust. Within these `unsafe` blocks, programmers can freely dereference raw pointers, read and modify mutable static variables, and perform other operations that are normally deemed unsafe.

While this feature provides the flexibility needed for certain low-level operations, it also reintroduces many of the memory safety issues commonly found in "unsafe" languages like C and C++. The use of `unsafe` code can potentially compromise the memory safety of the entire codebase, making it vulnerable to the same kinds of bugs and exploits that Rust aims to prevent.

To mitigate the risks associated with `unsafe` Rust, recent studies such as [5] suggest in-process isolation of safe and unsafe memory regions. This approach involves segregating safe objects in a designated safe memory region, while placing other objects in an unsafe region, similar to heap-like allocation.

However, this naïve segregation approach can hinder performance, as previously discussed in the context of heap allocations. As such, the same escape analysis used for heap allocation elision as described in this thesis can be applied to optimize the performance of safe and unsafe memory regions.

1.2 Overview

The remainder of the thesis is structured as follows. In chapter 2, we briefly overview the distinctive features of the Rust programming language. In chapter 3, we present a sound analysis that detects heap allocations that can be safely elided. In chapter 4, we evaluate the analysis on some sample code, and discuss future work. Finally, we conclude in chapter 5.

Chapter 2

Rust 101

Rust brings the *safety* of high-level programming languages and the *low-level* control of systems programming languages:

- It is a *safe* language in the sense that it prevents memory safety bugs at compile time. In C/C++, use-after-free, double-free, etc. are considered undefined behavior, and the compiler does not prevent such bugs—it is the programmer's responsibility to prevent such behaviors to occur in a program. In contrast, Rust's type system does not allow such code to be compiled in the first place.
- It is a *low-level* language in the sense that it provides facilities to control low-level resources. In Java or Python, the programmer does not have a direct control over the memory management system. Memory is automatically allocated and freed by the garbage collector in these languages. On the other hand, Rust programmers can choose the allocation strategy of each object, and can even directly interact with the pointer to the object.

Here we overview the core concepts of the ownership system of Rust, which is the key to the memory safety guarantees. We then take a look at the escape hatch, `unsafe` Rust, which allows programmers to bypass some of the safety disciplines. Finally, the compilation pipeline of Rust is briefly discussed.

2.1 Ownership and borrowing

Every value in Rust has a *unique* owner variable. When a variable gets passed to a function, the ownership of the value is transferred to the function, and the variable is no longer valid in the caller; the ownership has been *moved* to the callee.

Consider the following code:

```rust
fn main() {
    let mut v = vec![3, 1, 4];
    f(v);
    println!("{}", v[3]); // error[E0382]: borrow of moved value: `v`
}
fn f(mut v: Vec<i32>) {
    v.push(1);
}
```

This is a compile-time error, as the ownership of `v` has been moved to `f`, and `v` is no longer valid in `main`.

This raises a necessity of passing a value to a function without transferring the ownership: *borrowing*. A borrowed value is passed by *reference*, and the owner of the value is still valid. Thus the above code can be fixed as follows:

```rust
fn main() {
    let mut v = vec![3, 1, 4];
    f(&mut v); // borrow `v`
    println!("{}", v[3]); // 1
}
fn f(v: &mut Vec<i32>) { // `v` is a mutable reference to `Vec<i32>`
    v.push(1);
}
```

However, borrowing should be done with care, as it can lead to uncontrolled *aliasing*. Uncontrolled ownership is the key source of memory safety bugs in C/C++.

The following demonstrates the aliasing problem:

```rust
fn f(x: &mut i32, y: &mut i32) -> i32 {
    *x = 42;
    *y = 0;
    return *x;
}
```

Without any assumptions on the ownership of `x` and `y`, the return value of `f` can be either `42` or `0`. It can be `0` if `x` and `y` are aliases to the same object. Fortunately, in Rust, the ownership of `x` and `y` are guaranteed to be distinct, and the above code is guaranteed to return `42`.

Rust introduces aliasing disciplines of references to prevent such aliasing. The *borrow checker* enforces such disciplines:

- At any given time, you can have *either* one mutable reference *or* any number of immutable references.
- References must always be valid.

Therefore, in the above code, the mutable references `x` and `y` are guaranteed to point to distinct objects.

2.2 The escape hatch: `unsafe` Rust

While the ownership system of Rust soundly prevents memory safety bugs, it is sometimes too restrictive. In the case of doubly-linked lists, for example, each node is pointed by its two neighbors, and the ownership of the node cannot be uniquely determined. This is a fundamental limitation of the ownership system, and Rust provides an escape hatch, `unsafe` Rust, to relax some of the restrictions.

Basic smart pointer types like `Arc`, a reference-counted pointer, are implemented using `unsafe` Rust. Without `unsafe` Rust, it is cumbersome or even impossible to implement such types, as they do not have clear ownership semantics.

Generally speaking, `unsafe` Rust is necessary for efficient manipulation of and direct interaction with low-level resources without clear ownership.

In the scope of `unsafe` blocks, the aliasing disciplines are lifted:

- dereferencing a raw pointer,
- calling an `unsafe` function or methods,
- accessing or modifying a mutable static variables, etc.

However, the use of `unsafe` Rust, as its name implies, introduces certain risks. We revisit the aliasing problem in `unsafe` Rust:

```rust
fn f(x: &mut i32, y: &mut i32) -> i32 {
    *x = 42;
    *y = 0;
    return *x; // NOT guaranteed to be 42
}
fn main() {
    let mut v = 13;
    let p = &mut v as *mut i32;
    let x = unsafe {
        f(&mut *p, &mut *p)
    };
    println!("{}", x); // 0
}
```

Now with a presence of the `unsafe` block, the return value of `f` is not guaranteed to be `42`; it is actually `0` in this case. Without the aliasing disciplines not enforced in the `unsafe` block, `p` is being aliased and then passed to `f`.

Luckily, it is possible to confirm the correctness of Rust programs with `unsafe` blocks. This means that a programmer can indeed encapsulate `unsafe` logic in a safe interface, and the users of the interface will not be affected by its `unsafe` implementation.

This is done by formal verification of Rust programs in the Iris framework. The Iris framework provides a theoretical foundation to prove the correctness of `unsafe` Rust programs. Using the notion of *semantic type soundness*, it is possible to prove the correctness of `unsafe` Rust programs [6].

Nevertheless, not everyone has the luxury of formal verification of programs. The Iris framework requires a programmer to write a machine-checked proof in Coq proof assistant, which is, at best, a tedious task. This is why runtime safety checks are still helpful in the presence of `unsafe` Rust.

2.3 The compilation pipeline

The Rust compiler `rustc` is built on top of the LLVM compiler infrastructure. Thus, `rustc` only needs to perform the front-end tasks, such as parsing, type checking, and semantic analysis.

`rustc` first translates the Rust source code into the *abstract syntax tree* (AST). In this phase, macros are expanded, the AST is validated, and names are resolved.

The AST is then translated into the *high-level intermediate representation* (HIR). This is where type inference and checking, trait solving, and other semantic analysis are performed.

The HIR is then lowered into the *mid-level intermediate representation* (MIR). This is where borrow checking, pattern and exhaustiveness checking, and optimizations are performed.

Finally, the MIR is lowered into the LLVM intermediate representation (LLVM IR), which is then compiled into the machine code.

MIR, being the lowest Rust-specific intermediate representation, still contains high-level structure of the original program, while resolving complex statements into simple bites, making it a comfortable spot for performing the escape analysis. It also makes the dependencies explicit, which helps the analysis to be modularized.

Chapter 3

Safely eliding the heap

We now present a sound escape analysis that detects heap allocations that can be safely elided, based on the abstract interpretation framework. The target language for the analysis is the Rust MIR, and the analysis is implemented as a part of the MIRAI abstract interpreter [7].

3.1 Abstract interpretation of MIR

3.1.1 Abstract interpretation

Before we dive into the details of the analysis, we briefly overview the abstract interpretation framework.

Abstract interpretation is a generalized framework for *soundly* summarizing concrete program execution behaviors *without* calculating the exact values [8]. It has the benefit of being able to analyze programs by abstracting the concrete semantics of the language, and thus can be applied to virtually all programming languages.

To design an abstract interpreter, we need to define a semantic domain $\mathbb{D}$ of the target language. The semantic domain $\mathbb{D}$ should describe concrete program execution behaviors, and it should be a complete partial order set (CPO). An example of $\mathbb{D}$ is the powerset of the set of memory states, where each element would describe the set of possible memory states of the program.

The next step is to design an abstract domain $\mathbb{D}^\#$ that aptly summarizes the semantic values for the target property. The abstract domain $\mathbb{D}^\#$ should also be a CPO, and should have a Galois connection $\mathbb{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{D}^\#$. An example of $\mathbb{D}^\#$ is the set of abstract memory states, where each element would describe an abstract memory state of the program that soundly summarizes all possible concrete memory states.

With the concrete domain $\mathbb{D}$ and the abstract domain $\mathbb{D}^\sharp$ in our hands, we can calculate the least fixed point of an abstract semantic function. An abstract semantic function should describe a single step of abstract transition between two abstract states. Therefore, the least fixed point of an abstract semantic function would describe the abstract state of the program that summarizes all possible concrete states after the target program has terminated.

3.1.2 The core of MIR

Now we present the core of MIR in Figure 3.1, which is the target language of the analysis. Although the full MIR language is more complex than the core, the core captures the essence of how MIR is structured.

Function Bodies	F	::=	(L^*, B^*)	
Local Declarations	L	::=	(l, τ)	
Basic Blocks	B	::=	$\ell_B : S^*\texttt{;}\ T$	
Statements	S	::=	$\ell_S :\ p \mathrel{\texttt{=}} r$	assignment
Terminators	T	::=	$\texttt{goto}\ \ell_B$	jump
		|	$\texttt{switch[}\ell_B^*\texttt{][}o\texttt{]}$	switch
		|	$p \mathrel{\texttt{=}} o(o^*) \rightarrow \ell_B$	function call
Places	p	::=	l	local
		|	$\texttt{*}p$	dereference
		|	$p\texttt{.}f$	field index
		|	$p\texttt{[}i\texttt{]}$	array index
Rvalues	r	::=	o	operand
		|	$\texttt{\&}p$	reference
		|	$o \oplus o$	binary operation
		|	$A\texttt{\{}o^*\texttt{\}}$	aggregate
Operands	o	::=	$\texttt{copy}\ p$	copy
		|	$\texttt{move}\ p$	move
		|	k	constant

Figure 3.1: The abstract syntax of the core of MIR.

The MIR is basically a list of function bodies F. Each function body F consists of a list of local declarations L and a list of basic blocks B; Each basic block B consists of a list of statements S and a terminator T. A place p conceptually represents a memory location, and an operand o conceptually represents a value. An rvalue r is either a variation of operand o or a reference to a place p.

Notable constructs of MIR are the following:

- The assignment statement S assigns an rvalue r to a place p.
- The terminator T is the last statement of a basic block, and it determines the control flow of the program. It is either a jump to another basic block B labelled ℓ_B, a switch statement that jumps to ℓ_B based on an operand o, or a function call that jumps to ℓ_B.
- For a function call, the function name o and the arguments o^* are operands, and the result of the function call is assigned to a place p.

3.1.3 MIRAI

While it is a challenging task to formalize the semantics of a language as complex as Rust, Miri, an experimental interpreter of MIR, has been developed to provide a reference semantics of Rust [6]. It is used to detect undefined behaviors in Rust programs, as well as being used as a testbed for the formalization of Rust semantics.

Closely modeled after Miri, MIRAI is built as an abstract interpreter of MIR [7]. It uses the symbolic expressions abstract domain to summarize MIR programs, and runs over the MIR control flow graph. It then employs the Z3 SMT solver to resolve the constraints of symbolic expressions, subsequently concretizing these expressions into concrete values. This process is repeated until the abstract interpreter reaches a fixed point.

When MIRAI reaches a function call, it generates a summary for the callee function. A summary for a function is a list of (abstract) side-effects to the heap and the mutable reference parameters. From the summary, MIRAI specializes it and determines the side-effects and the abstract return value of the function call.

3.2 Analyzing heap allocations

We give a semi-formal definition of the escape analysis in Definition 1.

Definition 1 (Escape Analysis)**.** Given the set of algebraic data types (ADTs) $\mathscr{T}$ of our interest, construct the set of fields $\mathscr{F}_t$ for each $t \in \mathscr{T}$, where each $f \in \mathscr{F}_t$ has a boxed type $\tau(f) =$ `Box<_>`. Assume that a (context-sensitive) type resolving oracle τ exists. Find the multi-map of program locations to addresses $\mathscr{M}$ where each $\ell \in \operatorname{dom} \mathscr{M}$ is where escaping objects $\mathscr{M}(\ell)$ of type $\tau(f)$ are allocated.

To provide an example of the notation given in Definition 1, consider the following code:

```
struct Person {
    name: String,
    age: Box<i32>,
}
```

Here, struct `Person` has a boxed field `age`, while the field `name` is not boxed. Then if `Person` is included in the analysis (`Person` $\in \mathscr{T}$), then $\mathscr{F}_{\texttt{Person}} = \{\texttt{age}\}$.

We also give a notion of *type containment* operator $\Subset$ in Definition 2.

Definition 2 (Type Containment). Given two types τ and τ', we say that τ contains τ', denoted as $\tau' \Subset \tau$, if $\tau' = \tau$ or $\tau' \Subset \tau''$ for any generic type paramter τ'' of τ.

For example, `Option<Box<i32>>` contains `Box<i32>`, which in turn contains `i32`:

$$\texttt{i32} \Subset \texttt{Box<i32>} \Subset \texttt{Option<Box<i32>>}.$$

Finally, we give a notion of *rootedness* operator $\rightsquigarrow$ in Definition 3.

Definition 3 (Rootedness). Given two paths p and p', we say that p is rooted by p', denoted as $p' \rightsquigarrow p$, if p' is a prefix of p.

As an example, `x.y.z` is rooted by `x.y` and `x.y` is rooted by `x`:

$$\texttt{x} \rightsquigarrow \texttt{x.y} \rightsquigarrow \texttt{x.y.z}.$$

Algorithms shown in Figure 3.2 and Figure 3.3 are run after the abstract interpreter reaches the boxed patterns, to track allocations in $\mathscr{A}_B$ for each body B. For the assignment construct $T ::= p \coloneqq o_f(o_a^*) \rightarrow \ell_B$ (Figure 3.2), the analyzer gathers all the boxed types from the arguments and the return path, and then searches the environment σ for the values for which the paths are rooted by the gathered paths. For the ADT construct $r ::= A\{o_a^*\}$ (Figure 3.3), the analyzer gathers all the boxed types from the arguments, and then searches the environment σ in a similar manner.

```
$\mathscr{P} \leftarrow \emptyset$
foreach $o \in o_a^*$ do
    if $\exists t \in \mathscr{T} \cdot \tau(o) \Subset t$ then
        $\mathscr{P} \leftarrow \mathscr{P} \cup \{o.p\}$
    end
end
if $\exists t \in \mathscr{T} \cdot \tau(p) \Subset t$ then
    $\mathscr{P} \leftarrow \mathscr{P} \cup \{p\}$
end
foreach $p' \in \mathscr{P}$ do
    foreach $(p_\sigma, v_\sigma) \in \sigma$ do
        if $p' \rightsquigarrow p_\sigma$ then
            $\mathscr{A}_B(\ell_B) \leftarrow \mathscr{A}_B(\ell_B) \sqcup v_\sigma$
        end
    end
end
```

Figure 3.2: Tracking allocations for the assignment construct $T ::= p \coloneqq o_f(o_a^*) \rightarrow \ell_B$.

```
if $\exists t \in \mathscr{T} \cdot A \Subset t$ then
    foreach $o \in o_a^*$ do
        if $o = \texttt{copy}(p) \vee o = \texttt{move}(p)$ then
            foreach $(p_\sigma, v_\sigma) \in \sigma$ do
                if $p \rightsquigarrow p_\sigma$ then
                    $\mathscr{A}_B(\ell_S) \leftarrow \mathscr{A}_B(\ell_S) \sqcup v_\sigma$
                end
            end
        end
    end
end
```

Figure 3.3: Tracking allocations for the ADT construct $r ::= A\{o_a^*\}$.

The algorithm for building the label-wise escaping allocations $\mathscr{M}$ is shown in Figure 3.4. After the end of the body of a function is reached, using the gathered allocations $\mathscr{A}_B$, the analyzer consults the environment σ to build the label-wise escaping allocations. At the end of the function body, σ is a side-effects list of the summary.

```
Loop: foreach $(\ell, v) \in \mathscr{A}_B$ do
    foreach $a \in \gamma(v)$ do
        if $a = \top$ then
            $\mathscr{M} \leftarrow \mathscr{M}[\ell \mapsto a]$
            continue Loop
        else if $a = \text{Heap(\_)}$ then
            foreach $(p_\sigma, v_\sigma) \in \sigma$ do
                if $\exists t \in \mathscr{T} \cdot \big(\tau(p_\sigma) \Subset t \lor \exists f \in \mathscr{F}_t \cdot \tau(p_\sigma) \Subset \tau(f)\big)$ then
                    if $a \in \gamma(v_\sigma) \lor \top \in \gamma(v_\sigma)$ then
                        $\mathscr{M} \leftarrow \mathscr{M}[\ell \mapsto a]$
                        continue Loop
                    end
                end
            end
        else
            halt!
        end
    end
end
```

Figure 3.4: Label-wise escaping allocations for the function construct $F ::= (L^*, B^*)$.

Chapter 4

Implementation and evaluation

The escape analysis algorithm described in chapter 3 is implemented on top of MIRAI [7][1]. It has been evaluated on a sample test code and identified escaping and non-escaping allocations. Moreover, the analysis on the `aho-corasick` crate [9] successfully found eight escaping allocations.

4.1 Implementation

The escape analysis targets to optimize a custom `Box`-like type, `OptBox`, on ADTs implementing a custom trait, `EscapeAnalysis`. That is, unlike the original definition of the escape analysis in Definition 1, $\mathscr{F}_t$ of an ADT $t \in \mathscr{T}$ need to be modified to hold the fields of type `OptBox<_>`, not `Box<_>`. This is to reduce the false positive rate of the analysis, and to make the analysis opt-in.

Central data structures to the algorithms presented in section 3.2, such as the tracked allocations $\mathscr{A}_B$ and the label-wise escaping allocations $\mathscr{M}$, are augmented into the visitor data structures.

Also worth mentioning is that the `rustc` compiler has been slightly extended to easily identify the analysis targets. It is simply augmented with two more `rustc_span::symbol::Symbols`, `OptBox` and `EscapeAnalysis`. These two symbols simplify the identification of the `OptBox` struct and the `EscapeAnalysis` trait while carrying out the escape analysis.

[1]The implementation can be found in `https://www.github.com/Zeta611/mir-escape-analysis`.

4.2 Sample analysis

The result of the escape analysis on the code shown in Figure 4.1 is given in Figure 4.2. The analysis successfully identified the escaping allocations in the functions `escape_1`, `escape_1_mod`, `escape_2`, and `escape_3`, and moreover, it did not generate false positives for `non_escape_1` and `non_escape_2`. The analysis reported false positives for `escape_1` and `escape_1_mod`, but the analyzer did not suffer from any false negative, as expected.

Another notable point is that the analysis reports about the `implement_test_escaping_Person.clone` function. This actually is the implementation of the `Clone` trait for the `Person` struct, which is automatically generated by the `rustc` compiler. This would not have been possible if the analysis was implemented for the Rust surface language rather than an intermediate representation.

4.3 The `aho-corasick` crate

The `aho-corasick` crate [9] was tested and successfully found 8 escaping monomorphic functions. The analysis took 173 seconds. While the analysis was sound and the results were indeed true positive, the crate did not contain any non-escaping allocation to be elided.

The analysis demonstrated a tolerable running time, yet there remains significant room for performance optimization. The algorithms presented in section 3.2 have not been optimized for performance, and they involve multiply nested loops.

The major shortcoming of the analysis is that it does not handle polymorphic functions. This is inherited from the limitation of MIRAI, which does not handle polymorphic functions—it is a non-trivial task to figure out which trait an ADT implements statically.

In addition, library crates like `aho-corasick` typically do not *consume* `Box`es; They rather *produce* `Box`es to be consumed by the users of the crate. This means that the analysis is not very useful for library crates, but the users of the crates can benefit more from our analysis. It would be an interesting survey to compare this result with the analyses of library consuming Rust programs.

```rust
use escape_analysis::{EscapeAnalysis, OptBox};
use escape_analysis_derive::EscapeAnalysis;
#[derive(Clone)]
struct Foo(i32);
#[derive(Clone, EscapeAnalysis)]
struct Person { age: OptBox<Foo> }
fn escape_1() -> Person {
    let p = Person { age: OptBox::new(Foo(42)) } /* Does not escape */;
    let crowd = vec![p];
    crowd[0].clone() /* Does escape, handled in `implement_test_escaping_Person.clone` */
}
fn escape_1_mod() -> Person {
    let p = Person { age: OptBox::new(Foo(42)) };
    let mut crowd = vec![];
    crowd.push(p.clone());
    if p.age.0.is_positive() {
        p
    } else {
        crowd[0].clone()
    }
}
fn escape_2() -> Vec<Person> {
    let mut crowd = vec![];
    crowd.push(Person { age: OptBox::new(Foo(42)) });
    crowd
}
fn escape_3(crowd: &mut Vec<Person>) {
    crowd.push(Person { age: OptBox::new(Foo(42)) });
}
fn non_escape_1() -> i32 {
    let p = Person { age: OptBox::new(Foo(42)) };
    p.age.0
}
fn non_escape_2() -> i32 {
    let mut crowd = vec![];
    crowd.push(Person { age: OptBox::new(Foo(42)) });
    crowd[0].age.0
}
fn main() {
    let _p = escape_1();
    let _q = escape_1_mod();
    let mut c = escape_2();
    escape_3(&mut c);
    let _a = non_escape_1() + non_escape_2();
}
```

Figure 4.1: A sample code to demonstrate the result of the escape analysis.

```
{ "test_escaping.escape_2": [
    [ "bb2[0]",
      { "expression": { "HeapBlock": { "abstract_address": 5001001, "is_zeroed": false } },
        "expression_size": 1 } ] ],
  "test_escaping.escape_3": [
    [ "bb1[1]",
      { "expression": "Top",
        "expression_size": 1 } ] ],
  "test_escaping.main": [
    [ "bb2[0]",
      { "expression": "Top",
        "expression_size": 1 } ] ],
  "test_escaping.escape_1": [
    [ "bb4[1]",
      { "expression": { "HeapBlock": { "abstract_address": 12000001, "is_zeroed": false } },
        "expression_size": 1 } ],
    [ "bb1[1]",
      { "expression": { "Transmute": { "operand": { "expression": { "CompileTimeConstant": { "U128":
       ↪  4 } }, "expression_size": 1 }, "target_type": "ThinPointer" } },
        "expression_size": 2 } ] ],
  "test_escaping.implement_test_escaping_Person.clone": [
    [ "bb1[0]",
      { "expression": { "HeapBlock": { "abstract_address": 7000001, "is_zeroed": false } },
        "expression_size": 1 } ],
    [ "bb1[0]",
      { "expression": { "HeapBlock": { "abstract_address": 7000001, "is_zeroed": false } },
        "expression_size": 1 } ] ],
  "test_escaping.escape_1_mod": [
    [ "bb2[2]",
      { "expression": { "Transmute": { "operand": { "expression": { "CompileTimeConstant": { "U128":
       ↪  4 } }, "expression_size": 1 }, "target_type": "ThinPointer" } },
        "expression_size": 2 } ],
    [ "bb1[1]",
      { "expression": { "HeapBlock": { "abstract_address": 4001001, "is_zeroed": false } },
        "expression_size": 1 } ],
    [ "bb9[1]",
      { "expression": { "HeapBlock": { "abstract_address": 15000001, "is_zeroed": false } },
        "expression_size": 1 } ] ] }
```

Figure 4.2: The result of running the escape analysis on the sample code given in Figure 4.1.

Chapter 5

Conclusion

In this thesis, we have presented a sound escape analysis for Rust MIR, aimed at optimizing program performance by eliding heap allocations. This analysis is particularly effective in addressing performance issues that arise from unnecessary heap allocations, whether these are the result of programmer oversight or the byproducts of automated code generation tools.

The analysis is sound, as it is based on the abstract interpretation framework. Our escape analysis algorithm was implemented on top of MIRAI, an abstract interpreter of Rust MIR, which is in turn based on Miri, an experimental MIR interpreter.

The analysis was evaluated on a sample code and the `aho-corasick` crate. The analysis successfully identified escaping allocations with low false positive rate; It actually did not raise any false positive for the `aho-corasick` crate.

5.1 Future work

Future work includes expanding the analysis to cover polymorphic functions. Currently, due to the limitation of MIRAI, the analysis does not handle polymorphic functions well. The exploration of polymorphic functions will be a promising opportunity for future work.

In addition, incorporating the code transformation routine to put heap-elidable data on the stack will greatly improve the usefulness of our analyzer.

Bibliography

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Prentice Hall, 1988.

[2] G. Thomas. “A proactive approach to more secure code.” (2019), [Online]. Available: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/ (visited on 12/14/2023).

[3] S. Klabnik and C. Nichols, *The Rust Programming Language*. No Starch Press, 2019.

[4] A. A. A. Donovan and B. W. Kernighan, *The Go Programming Language* (Addison-Wesley Professional Computing Series). Addison-Wesley, 2015.

[5] I. Bang, M. Kayondo, H. Moon, and Y. Paek, “TRust: A compilation framework for in-process isolation to protect safe rust against untrusted code,” in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 6947–6964, ISBN: 978-1-939133-37-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/bang.

[6] R. Jung, “Understanding and evolving the rust programming language,” PhD thesis, Saarland University, 2020.

[7] H. Venter, *Mirai*, https://github.com/facebookexperimental/MIRAI.

[8] X. Rival and K. Yi, *Introduction to Static Analysis: An Abstract Interpretation Perspective*. The MIT Press, 2020.

[9] A. Gallant, *Aho-corasick*, https://github.com/BurntSushi/aho-corasick.

초 록

이 논문은 러스트(Rust)의 중간 수준 중간어(MIR)를 위한 안전한 탈출 분석을 도입하고, 힙 메모리 할당을 안전하게 생략하여 러스트 프로그램을 최적화한다. 제시한 탈출 분석은 요약 해석에 기반을 두어, 기존의 MIRAI 요약 실행기를 확장하면서 러스트의 강력한 정적 타입 시스템을 활용하여 탈출할 수도 있는 힙 메모리 할당을 찾아낸다. 고안한 분석기는 샘플 코드와 공개적으로 이용 가능한 `aho-corasick` 크레이트에 대하여 평가하였으며, 오탐율을 보여준다. 이 연구는 러스트의 메모리 최적화에 기여한다.

주요어:탈출 분석, 메모리 최적화, 정적 분석, 요약 해석, 러스트