Summary

This thesis presents, to our knowledge, the first attempt at utilizing *Pointer Assertion Logic* and *monadic second-order logic* in the context of pointer manipulation in Rust, as well as the application of *Graph types* for specifying *Data Structure Invariants* (DI) and demonstrating preservation of Rust data structures.

We formulated a transformation from *MIR* to *PALE* and demonstrated its application on a concrete example of a data structure manipulation program in Rust. We then attempted to prove the memory safety of this example and showed that the program does not violate the *DI*.

We have identified that *PALE* and *monadic second-order logic* present interesting avenues for further research in formal verification. We believe that these approaches have the potential to advance the field of formal verification in the context of *Rust*.

MASTER THESIS IN DISTRIBUTED SYSTEMS (DS)
COMPUTER SCIENCE

POINTER ASSERTION LOGIC & DATA STRUCTURE INVARIANT OF *MIR* PROGRAMS

ANDERS HAHN BOELT

Aalborg University

SUPERVISED BY
RENÉ RYDHOF HANSEN,
CO-SUPERVISOR DANNY B. POULSEN

JUNE 15, 2023

AALBORG UNIVERSITY
STUDENT REPORT

Computer Science
Aalborg University
aau.dk

Title:
Pointer assertion logic & data structure invariant of *MIR* programs

Theme:
Master Thesis in Distributed Systems (DS)

Project Period:
Spring Semester 2023

Project Group:
CS-23-DS-10-01

Participants:
Anders Hahn Boelt

Supervisors:
René Rydhof Hansen
Danny B. Poulsen (co-supervisor)

Copies: 1

Page Count: 42

Date of Completion:
15-06-2023

Abstract:

We investigate the use of the *Pointer Assertion Logic Engine* (*PALE*) in the context of *Rust*. We demonstrate how it is possible to translate *Rusts Mid-Level Intermediate Representation* (*MIR*) into the language *PALE*, where program annotations and *Graph Types* are used for semi-automatic formal reasoning over the program store, along with proof of *Data Structure Invariants* (*DI*) preservation.
We provide a concrete example of modeling an *intrusive circular doubly linked list* data structure from the *Rust-for-Linux* project in *PALE*. We define *DI* over the said data structure, we attempted to show the absence of memory faults.
In conclusion, we find that *PALE*, along with *monadic second-order logic*, is an interesting field for further exploration, although our transformation of *MIR* to *PALE* lacks a concrete proof of its validity.

Acknowledgements I would like to thank my supervisors *René Rydhof Hansen* and *Danny Bøgsted Poulsen* for their guidance and support during my Master Thesis.

I couldn’t resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

— C. A. R. HOARE

CONTENTS

Contents

1 **Introduction** 1
- 1.1 Pointer assertion logic 1
- 1.2 Mid-level intermediate representation and formal verification 3
- 1.3 Related Works 3
- 1.4 Contribution 4
- 1.5 `unsafe_list` data structure example 4

2 **Background** 11
- 2.1 Rust Mid-Level intermediate representation 11
- 2.2 Graph types 13
- 2.3 Monodic second order logic over graph types (M2LGT) 16
- 2.4 Pointer Assertion Logic 17

3 ***MIR* to *PALE* transformation** 25
- 3.1 MIR to PALE transformation 25
- 3.2 Rust Data types in PALE 34
- 3.3 Validation results 37

4 **Discussion & Conclusion** 43
- 4.1 Discussion 43
- 4.2 Conclusion 44

A **Linux Kernel Rust `unsafe_list` MIR Code** 45

B **Linux Kernel Rust `unsafe_list`** 49

C **Rust code used as examples in Section 3.1** 51

D **`Unsafe_list` PALE Model unsimplified** 53

Bibliography 61

1 INTRODUCTION

Rust, with its emphasis on memory safety and low-level system programming, has been gaining popularity as a safer alternative to languages like *C* and *C++*. Its strong type system and ownership based memmory model, adids in the privation of common programming errors such as *null pointer dereferences*, and *data races*. However, the complexity of the language and its novel features, including the separation of safe and unsafe Rust code, present new challenges for ensuring the correctness of Rust programs [1].

With the acceptance of *Rust* into the *Linux* kernel, there is a growing need for formal verification techniques specifically for *Rust*. The *Linux* kernel, being a critical piece of software, demands a high level of assurance in terms of correctness, security, and reliability. Formal verification can provides systematic approach to analyze Rust code and prove its correctness properties, ultimately enhancing the trustworthiness of the *Rust*, and in turn also its adoption in the *Linux* kernel [1].

The master thesis is structured as follows: Chapter 1: examines the argument for using *PALE* as the underlying logic for our verification efforts. It also explores the use of *MIR* as the underlying language for verification. Additionally, it presents a running example in *Rust* of a pointer manipulating program over a data structure, namely `unsafe_list`. We conclude the chapter by highlighting the contribution that we believe this thesis presents to the field of formal verification methods. Chapter 2: Covers the background theories related to graph types, *PALE*, and *MIR* to provide a foundation for the subsequent chapters. Chapter 3: Presents our verification efforts on a specific example, which involves a function that manipulates pointers in a data structure within the Linux kernel. The example specifically focuses on the `remove` operation in the `unsafe_list` data structure. Chapter 4 discusses the results of our verification efforts and presents the conclusions drawn from our thesis. It provides an analysis and discussion of the verification outcomes and their implications. Finally, it concludes the thesis by summarizing the key findings, contributions, and potential future directions for further research.

1.1 POINTER ASSERTION LOGIC

This section discusses the selection of *PALE* as the underlying logic, and explains why Separation Logic was not chosen.

Pointer Assertion Logic Engine (*PALE*) was introduced in [13] as a framework for verifying the partial correctness of programs that manipulate data structures. PALE specializes in verifying a wide range of data structures, that are expressible as *Graph Types* (see Section 2.2 for details). A more comprehensive explanation of PALE can be found in Section 2.4.

The following listing presents several key points supporting (+) and opposing (-) the choice of *PALE* as the underlying logic in this thesis:

(+) Graph types simplify the specification of data structures involving pointers, such as doubly linked lists and trees.

(+) We focus primarily on pointer manipulation occurring within data structures, making the benefits of *Graph Types* relevant.

(+) *PALE* automatically generates numerous checks of interest, including null pointer dereference and pointer separation in the heap (see Section 2.4 for more information).

(+) Not utilizing separation logic as the underlying logic may yield interesting observations.

(+) *PALE* leverages a *weak monodic second-order logic* in MONA, eliminating the need to calculate weakest preconditions [13].

(+) Has not been used to verify *Rust* programs before.

(-) *PALE* lacks a formal specification for its semantics.

(-) *PALE* can only describe data structures expressible as *Graph Types*.

(-) The tooling around *PALE* and MONA is outdated.

Alternatively, Separation Logic (*SL*) [14] was considered as another approach. *SL* is a formal logical framework that extends traditional *Hoare logic* [5] to reason about programs store. *SL* was initially proposed by Reynolds in Reynolds as a means of reasoning about pointer-based programs, in a more intuitive manner then *Hoare logic*.

Here are several points supporting (+) and opposing (-) for chose of Separation Logic:

(+) *SL* offers greater expressiveness compared to *PALE*, due to *PALE*s dependes on *Graph Types*.

(+) Supported by Automated Theorem Provers, such as Coq-Iris [3].

(-) Lack of automaton compared to *PALE*.

(-) The complexity of *SL*, and the time limitation of the thesis.

(-) Dependency on weakest precondition.

An interesting observation is that *PALE* and *SL* were developed around the same time in history, but *SL* has received significantly more development in recent years. As a result, we believe it would be interesting to explore any apparent benefits of choosing *PALE* over *SL*, in regards to the verfication of *Rust* progrmas.

By selecting *PALE*, we can leverage its specialized focus on verifying data structures expressible as *Graph Types*. Additionally, PALE's automatic generation of relevant checks, such as null pointer dereference and pointer separation in the heap, is a valuable feature.

Although there are arguments against *PALE*, such as the lack of a formal specification for its semantics and the limitations of *Graph Type* see Section 2.2, we believe that exploring PALE in the context of this thesis can yield valuable insights and potentially uncover unique observations. The historical parallel between *PALE* and *SL*, coupled with the potential benefits of *PALE*s specialized focus, make it a compelling choice to investigate in comparison to the more extensively developed *SL*.

1.2 MID-LEVEL INTERMEDIATE REPRESENTATION AND FORMAL VERIFICATION

This section discusses the selection of Rusts *mid-level intermediate representation* (*MIR*) as the underlying language used for our verification.

The following list covers some points supporting (+) and opposing (-) the choice of *MIR* as the underlying language for our verification:

(+) *MIR* provides a simplified representation of high-level *Rust* programs. It abstracts away some of the complexity of Rust, making it easier to reason about and analyze.

(+) Most of the language constructs in *MIR* are some what similar in *PALE*, the underlying logic used in our verification.

(+) Is was also an argument for the adoption of *MIR* into the *Rust* compiler [11], that *MIR*, should eventually be suitable for safety proofs.

(-) Given that *MIR* represents high level *Rust* as a (*CFG*) may complicate the modeling of loop constructs and other intricate language features, this was also identified as a limitations of *MIR* in [11].

(-) As *MIR* is integrated as part of the *Rust* compiler, there may be information that is not available at the *MIR* level but is needed for our verification. This limitation requires careful handling and potential workarounds.

While there may be challenges in handling specific language features or limitations due to the simplified nature of *MIR*, we believe that the benefits of using *MIR* as the underlying language for verification outweigh the potential drawbacks. With careful consideration and appropriate techniques, we see *MIR* as a provides a suitable basis for modeling and reasoning about *Rust* programs.

1.3 RELATED WORKS

In recent years, there have been several notable attempts at formal verification of Rust programs. This section provides an overview of some of the prominent projects and approaches in this domain.

RustBelt The RustBelt project focuses on developing formal verification techniques for Rust's ownership and borrowing system. By combining *separation logic*, type systems, RustBelt aims to provide rigorous guarantees about memory safety and data race freedom of the Rust standard library [1].

Creusot *Creusot* translates *MIR* functions into *WhyML*. They also introduced *PAREALITE*, a specification language used for annotations of Rust programs, which is later translated into annotations in *WhyML*. The main idea behind *Creusot* is the introduction of prophecies, which are able to speculate about the result of a mutable borrow [2].

RustHorn & RustHornBelt *RustHorn* translates Rust programs into *constrained Horn clauses* (*CHC*s). By utilizing the Rust borrow system, it can overcome the scalability problems of *CHC* [10]. In *RustHornBelt*, *RustBelt* and *RustHorn* are combined to enable support for *first-order logic* (*FOL*) specifications for safe APIs that are implemented in `unsafe` *Rust* code [9].

1.4 CONTRIBUTION

This master's thesis aims to bridge the gap between *MIR* and *PALE* by developing a transformation process that enables the formal verification of *MIR* programs. The following is a list of the contribution of this master thesis to the filed of *formal methods*:

- To our knowledge this thesis presents the first attempt at translating *Rust* to *PALE*.
- We present the first attempt at formalizing, and proving *data structure invariant* in *Rust*.

1.5 `unsafe_list` DATA STRUCTURE EXAMPLE

We have chosen an example from the *Rust for Linux* [16] project, which is a collective effort to bring Rust into the Linux kernel. The specific example we have selected is the `unsafe_list`, an implementation of an *intrusive circular doubly-linked list* (*ICDLL*) data structure. This data structure is commonly used, for example, as the underlying data structure of work queues[1].

We have chosen *ICDLL* as our focus because it aligns well with PALE's utilization of Graph types and the prevalence of pointer manipulations involved in the operations performed on this data structure.

We will begin by providing a brief informal definition of *ICDLL*, outlining its general structure and providing an informal description of its soundness. We will then proceed to explain the concrete implementation of the `unsafe_list`, where we will present the relevant operations that will be verified later in this thesis.

We will use *ICDLL* and its implementation in *Rust* as the basis for our verification efforts.

1 Rust work queues implementation in the linux kernel `https://github.com/Rust-for-Linux/linux/blob/3dfc5ebff103ac99dca27644c09edbcc6dd8e9d1/rust/kernel/workqueue.rs`

Intrusive circular doubly-linked list An *ICDLL* is a data structure that organizes elements in a circular manner using doubly-linked pointers. Unlike traditional linked lists where the list nodes hold the data and pointers, in an ICDLL, the elements themselves contain the necessary pointers for linking them together.

A graphical representation of an example *ICDLL* is shown in Figure 1.1.

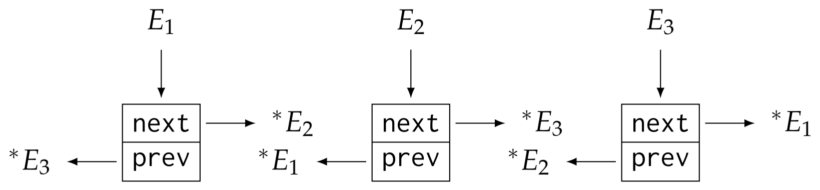

Figure 1.1: Example of *ICDLL*,

In terms of the safety of the Intrusive Circular Doubly-Linked List (ICDLL) data structure, we propose the following informal specifications:

As to the safety of the *ICDLL* data suture we pose the following informal specification:

- **Pointers within the List:** Every `next`, and `prev` pointer within the ICDLL must point to an element that exists within the list. This ensures that the links between elements are valid and do not reference elements outside the list.
- **List structure:** Every *next*, and *prev* pointer can only point to a neighboring elements with in the list.
- **Reachability:** The property of **reachability** states that given an element E that is present in the list, it should be possible to reach E from any other element E' in the list by following the `next` and `prev` pointers repeatedly.
- **Non-Null Pointers:** No pointer within the list can be null. This guarantees that all pointers are properly initialized and do not lead to undefined behavior.
- **Preservation of Properties:** Every operation performed on the ICDLL, including insertions, deletions, and modifications, must uphold the properties mentioned above. This ensures that the list's integrity and safety are maintained throughout all operations.

By specifying these properties informally, we establish a set of requirements that a given implementation must satisfy. Verifying that the implementation adheres to these specifications helps ensure the safety and correctness, preventing potential issues such as dangling pointers, null pointer dereferences, or invalid pointers within the list.

We will returnee to this specification later and give a more concrete specification using *Graph Types* in Section 3.2.

Rust Linux `unsafe_list` *data Structure*

In this section, we will present the general structure of the `unsafe_list` implementation [17], focusing on the remove operation. The main objective is to identify program points where unsafe operations (`unsafe {...}` blocks) occur and identify where any potential violations of the informal specifications for the (ICDLL).

We start by explain the structure of the `unsafe_list` data type, consider the code shown in Code listing 1.5.1.

```rust
pub struct List<A: Adapter + ?Sized> {
    first: Option<NonNull<A::EntryType>>,
}

pub unsafe trait Adapter {
    type EntryType: ?Sized;
    fn to_links(obj: &Self::EntryType) -> &Links<Self::EntryType>;
}

pub struct Links<T: ?Sized>(UnsafeCell<MaybeUninit<LinksInner<T>>>);

struct LinksInner<T: ?Sized> {
    next: NonNull<T>,
    prev: NonNull<T>,
    _pin: PhantomPinned,
}
```

Code listing 1.5.1: The overall structure of `unsafe_list` type [17].

`List<A: Adapter + ?Sized>` This constitutes the entry point of the aforementioned type. Within this context, there exists only one field, namely: `first: Option<NonNull<A::EntryType>`. This field represents a `NonNull` pointer pointing to the first element of the list. It is important to note that `first` is wrapped in an `Option` type, as the underlying `EntryType` may not necessarily contain any data, upon initialization of the `List`. The `?Sized` trait specifies that a given type has a constant size that is at compile time.

`LinksInner<T: ?Sized` This is the internal structure list, where `next: NonNull<T>` points to the next element of the list, and `prev: NonNull<T>` points to the previous element of the list. The type `_pin: PhantomPinned` ensures that these types cannot be moved in memory.

`Links<T: ?Sized>` The `Links` is a wrapper around the internal type `LinksInner<T>`, which itself is composed of two other types: `UnsafeCell` and `MaybeUninit`. The purpose of using `UnsafeCell` is to opt-out of the immutability guarantee for &T in case a shared reference to &T might point to data that is being mutated. MaybeUninit is used to represent values of type T that may not be fully initialized yet.

Trait / Type Adapter This defines an **unsafe trait** called Adapter. It serves as an abstraction for adapting user-defined types to be compatible with the intrusive list. It has an associated type EntryType that represents the type of the elements in the list. The trait also includes a method to_links that returns a reference to the Links struct associated with the given EntryType of the List. A small example of how one might use the train on a user-defined type is shown in Code listing 1.5.2

```
struct Example {
    v: usize,
    links: Links<Example>,
}

unsafe impl Adapter for Example {
    type EntryType = Self;
    fn to_links(obj: &Self) -> &Links<Self> {
        &obj.links
    }
}
```

Code listing 1.5.2: Example of Adapter trait implantation on a user-defined type [17].

We now turn our attentions to the implantation of the remove operation on a List consider the code shown in Code listing 1.5.3.

```
pub unsafe fn remove(&mut self, entry: &A::EntryType) {
  let inner = unsafe { self.inner_ref(NonNull::from(entry)) };
  let next = inner.next;
  let prev = inner.prev;

  let inner = unsafe { &mut *A::to_links(entry).0.get() };

  unsafe { inner.assume_init_drop() };

  if core::ptr::eq(next.as_ptr(), entry) {
      self.first = None;
  } else {
      unsafe { self.inner(prev).next = next };
      unsafe { self.inner(next).prev = prev };

      if core::ptr::eq(self.first.unwrap().as_ptr(), entry) {
          self.first = Some(next);
      }
  }
}
```

Code listing 1.5.3: `remove` Removal of an element in the list. All code in this listing is wrapped in `impl<A: Adapter +?Sized> List<A>` [17].

In this implementation, the `remove` function takes a mutable reference `&mut` to an entry of type `A::EntryType`. The function starts by obtaining a mutable reference to the internal `Links` structure associated with the `entry`. This is done using the `inner_ref` (see Appendix B Code listing B.0.1 for more details), function and converting the entry to a non-null pointer with `NonNull::from`. Next, the `next` and `prev` pointers are extracted from the obtained inner reference. These pointers represent the neighboring elements of the entry in the List that we are about to remove.

The `inner` reference is reinterpreted as a mutable reference to the `LinksInner` using raw pointer dereferencing. This operation allows direct access to the underlying memory of inner, this is also the reason that the function call is surround by `unsafe {}` block. Subsequently, the unsafe function `assume_init_drop()` is called, which deallocates the value of `inner` in the `Links` of the entry being removed.

The call to `assume_init_drop()` is the first place that we are violation the properties of *ICDLL* we specified early namely [List structure], by deallocating the internal links of `inner`, the list is no longer valid.

We the check if the element removed form the list was the only element in this case we assing it to `None`, if this was not the case we restore the structure of the list by the following two assignments
`unsafe { self.inner(prev).next = next };`

`unsafe { self.inner(next).prev = prev };`.

The function ends by checking if the element we removed was the first element of the list if this was the case we set `list.first = Some(next)`.

2 BACKGROUND

2.1 RUST MID-LEVEL INTERMEDIATE REPRESENTATION

As discussed in Section 1.2, we have chosen to use *MIR* as the underlying language for our verification efforts. This section will provide a brief introduction to the structure and syntax of *MIR*, and an informal explanation of the semantics of key operations in *MIR*.

For a formal explanation of the syntax and semantics of *MIR* we refer the reader to [4].

The structure of a *MIR* represent the control flow graph (*CFG*) of a given high-level *Rust* program [11].

***MIR* control-flow graph** A *control-flow graph*, represent a model of the flow of control between the basic blocks in a *MIR* function. We define a *CFG* as a (*directed cyclic labeled graph*) $G = (V, E)$, where $v \in V$ corresponds to a basic block in *MIR*, and $e = (n_i, n_j) \in E$ corresponds to a transfer of control from n_i to n_j, this is represented in *MIR* as *Terminators*.

Places Memory locations allocated on the stack, encompass function arguments, local variables. These specific locations are denoted by an index preceded by a leading underscore, such as `_1`. Moreover, a designated *local* (`_0`) is allocated specifically for storing the return value of a function.

***MIR* functions & Variable declarations** A program consists of one or more function declarations. Each function is defined the following syntax:

```
fn id({Place : Types}*) -> Types {Decl}{Block}
```

Here, `fn` denotes the function keyword, `id` represents the name of the function, and `{Place : Types}*` indicates a collection of input parameters to the function. `-> Types` specifies the return type of the function. In *MIR* representation, each function corresponds to a function in Rust. The set `{Decl}` consists of variable declarations in the form `let [mut] P:` τ, where `mut` is optional. It specifies whether the location referred to by `P` is mutable or immutable.

Basic Blocks The *MIR basic blocks* (`Block`) represent the vertices in the *CFG* . All changes in control flow of a *MIR* program occur between *basic blocks*. Each *Basic Block* has a unique ID with in a function, where $ID \in \mathbb{N}^+$:

```
BB_ID: { {Statements} Terminator}
```

and consists of a sequence of *Statements* followed by a single *Terminator*. Every *basic block* is terminated by a *Terminator*. [11].

Rvalues & Operands An `Rvalue` are expression that creates a value. `Rvalues`, and `Operands` (`Opr`) take the form shown in Table 2.1

Rvalue & Operands	*Syntactic* Reps.	*Semantic* meaning
`Refrence`	`& Place`	Represents a shared reference to a `Place`
`dereferencing`	`*Place`	Represents a dereferencing of a `Place`
`Mutable Refrence`	`&mut Place`	Represents a mutable reference to a `Place`
`Addition`	`Opr + Opr`	Evaluates the result of `Opr`, and adds the resulting values together
Operands		
`Constant`	`const` τ	Denotes a constant value of Type τ
`move`	`move Place`	Borrows the value stored at `place`
`copy`	`copy Place`	Copies the value stored at `place`

Table 2.1: Syntactic Reps., and semantic meaning of `Rvalues`, and `Operands`. For a more detailed description see [4]

Assignment & Terminators Assignment is of the syntactic form `Place = Rvalue`. *Terminator*s is the only construct in *MIR* that can describe edges between the vertices in the *CFG*. Table 2.2 shows the *Terminator*s, and gives and informal explanation of their semantic meaning.

Terminator	*Syntactic* Reps.	*Semantic* meaning
`goto`	`goto -> Bid`	Represents a direct transfer of execution to the specified target `Bid` basic block.
`switchInt`	`switchInt(Opr) -> [Target* otherwise: Bid]`	A conditional transfer of execution represents a control flow transfer that depends on a certain condition. It can be directed either to a set of `Target*` or to a specific `bid` basic block.
`function call`	`Place = i(Opr*) -> Bid;`	A function call stores the result of the call in `Place`, and after termination of `i()` transfers control to `Bid`.
`return`	`return;`	Returns the value stored at `_0`, to the caller of the function

Table 2.2: Syntactic Reps., and semantic meaning of `Terminalss`, where `Target* ::=` z `:` BB is a set of destinations, where z is a value over a enum, and `BB` is a given *basic block*. For a more detailed description see [4]

2.2 GRAPH TYPES

Graph types where introduced in [7], as means to expresses the structure of recursive data types in a more simple manner. This section will cover some of the theory behind *Graph types*, and also present the limitations that exists.

2.2.1 *Data Types*

Data types are represented as a specialized form of tree grammar, where the non-terminals are referred to as types. Within this framework, a main type is distinguished for a given recursive data type $\mathcal{D}$ we denote this as $Main\ \mathcal{D}$. To specify the production of a given data type, we use the following notation, as illustrated in this example for the data type $\mathcal{D}$ [7]:

$$T \rightarrow v(a_1 : T_1, \ldots, a_n : T_n)$$

In the given context, we have a type T and the types T_n that define the variant v of T. Additionally, $a_1, \ldots, a_n$ define the type-variant $(T : v)$. Let us denote the set of types as $T_\mathcal{D}$. The notation $T_\mathcal{D}(T : v)a = T_i$ indicates that T_i represents the type of a data field a belonging to the variant v of type T. Moreover, $V_\mathcal{D}$ represents the set of all variants in $\mathcal{D}$, and $V_\mathcal{D}T$ denote the set of variants of type T. $F_\mathcal{D}(T : v)$ denotes the set of data field of type T and a given variant v, as in the example above a is an element of the set $F^*_\mathcal{D}$ [7].

The formal definition a production of a data type can now be define as the function

$$\begin{aligned} x : F^*_\mathcal{D} \rightarrow T_\mathcal{D} \times V_\mathcal{D} \quad & \text{such that} \\ & dom\ x \text{ is finite and prefix closed} \\ & x(\epsilon) = (Main\ \mathcal{D} : v), \text{for some } v \\ & \forall a \in dom\ x, if\ x(a) = (T : v) \\ & \ - \quad v \in V_D T,\ and \\ & \ - \quad \alpha a \in dom\ x \Leftrightarrow \\ & \qquad a \in F_D(T : v) \wedge T_D(T : v)a = T' \\ & \qquad \text{where } x(\alpha a) = (T' : v') \\ & \qquad \text{for some } v' \end{aligned} \tag{2.1}$$

dom x are used as pointer values.

Data Type example As a simple example of a data type consider the simple integer list where the production can be define as follows [7]:

$$\begin{aligned} L &\rightarrow \texttt{nonempty(head : Int, tail: L)} \\ &\rightarrow () \end{aligned}$$

We can think of the type *Int* as a data type by representing with the production $Int \rightarrow 0() \mid 1() \mid 2() \mid \ldots$.

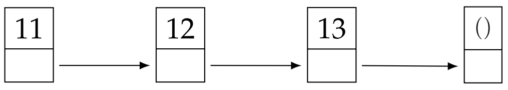

Figure 2.1: The tree structure of a simple Integer List Data type. Where () denotes an empty recorded

Data Types Expressibility The main problem with Data Types is that they can only represent trees structures consider the example of an Integer list containing 11, 12, 13 it can be depicted as the tree structure shown in Figure 2.1:

Usually when working with list we want constant time access toe the last element in the list, this can be accomplished by adding a new record to the tree structure this result in the the shape shown in Figure 2.2 The addition

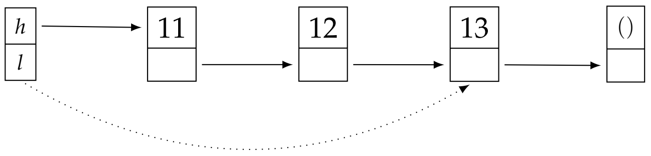

Figure 2.2: The "tree"structure of a simple Integer List Data type. Where () denotes an empty recorded, and h, and l denote the head and last element of the list

of the pointer to the last element of the list, creates a problem because it prevents the representation of the list as a tree structure. Since data types are specified using a tree grammar, this particular data type cannot be expressed due to the conflict introduced by the addition.

In a tree structure, nodes are connected in a hierarchical manner, where each node can have multiple child nodes. This allows for the representation of complex hierarchical relationships between data elements. However, the addition mentioned disrupts the hierarchical nature of the tree structure, the current definition of Data Types is not expressive enough to represent this structure.

To this extent, Graph Types are introduced as a simple extension of Data Types that enable us to express Data Types that exhibit graph-like structures [7].

2.2.2 *Graph type*

Graph types are simply put an extension of Data Types with what we call routing expressions, which are a superset of regular expressions, over what we refer to as the "backbone" of a graph type. The backbone is simply the values in the underlying Data type, which is are represented as canonical spanning tree. All edges that cannot be represented by the backbone, as seen in the limitation on Integer lists mentioned earlier, are functionally determined by the backbone (i.e., routing expression) [7].

We define the production of a Graph type as follows [7]:

$$T \to v(a_i : T_i, \ldots, a_j : T_j[R], \ldots) \tag{2.2}$$

Here a_i is a simple data field as in a normal *Data type*, and a_j is a routing field. Routing fields has an associated routing expression R.

The more formal definition of Graph types and Routing Expressions is as follows:

Definition 1. *Graph Type [7] We continue to use the notation $F_{\mathcal{G}}$ to denote the set of all fields, as defined in the Data Types (ref). Let $F^d_{\mathcal{G}}$ represent the set of data fields, and $F^r_{\mathcal{G}}$ represent the set of routing fields within $F_{\mathcal{G}}$. We introduce the function $R_{\mathcal{G}}(T : v)a$, which associates a given routing field a with a routing expression for the variant v of type T. Furthermore, let Data $\mathcal{G}$ denote the underlying data type of the graph type $\mathcal{G}$. It is obtained by removing all routing fields from $F_{\mathcal{G}}$, represented as Data $\mathcal{G} = F_{\mathcal{G}} \setminus F^r_{\mathcal{G}}$. Routing fields are all defined on Data $\mathcal{G}$.*

Given a data Type D, we define an alphabet Δ that comprises of the following directives also called letters, $\wedge, \$, \uparrow, \uparrow a$, and $\downarrow a$, where $a \in F_D; T$, and $(T : v)$, where $T \in T_D$, and $v \in V^T_D$. Let $\rightsquigarrow_x$ define the step relation (dom $x \times \Delta \times$ dom x), where $x \in Val\ D$. The transition of $\rightsquigarrow_x$ are defined as follows:

$$\epsilon \overset{\wedge}{\rightsquigarrow}_x \epsilon \qquad \alpha \cdot a \overset{\downarrow a}{\rightsquigarrow}_x \alpha$$

$$\alpha \cdot a \overset{\uparrow}{\rightsquigarrow}_x \alpha \qquad \alpha \cdot a \overset{T}{\rightsquigarrow}_x \alpha \quad \text{if } x(a) = (T : v) \text{ for some } v$$

$$\alpha \overset{\$}{\rightsquigarrow}_x \alpha \quad \text{if } \alpha \text{ is a leaf in } x \qquad \alpha \cdot a \overset{(T:v)}{\rightsquigarrow}_x \alpha$$

$$\alpha \cdot a \overset{\uparrow a}{\rightsquigarrow}_x \alpha$$

We use the notion $\alpha \overset{d}{\rightsquigarrow}_x \beta$ which means the β is reached from α by directive d. One thing to note is that we $\alpha \overset{d}{\rightsquigarrow}_x \beta$ is uniquely defined if it exists.

Definition 2. *Route and walk [7] We define a route as $p = d_1 \dots d_n$ is a word over the alphabet Δ We define a walk in x from $\alpha \in$ dom x to $\beta \in$ dom x along a route p is a unique sequence, $\alpha_0, \dots, \alpha_n = \beta$, such that $\alpha_{i-1} \overset{d_i}{\rightsquigarrow}_x a_i, \forall i, 1 \leq i \leq n$. We denote such a walk as $\alpha \overset{p}{\rightsquigarrow}_x \beta$.*

Definition 3. *Routing expression [7] Let R denote a routing expression on Data Type D, R is a regular expression over the Δ. Expression are constructed using the usual operators $+(union)$, $\cdot(concatenation)$, and $\star(iteration)$. The language recognized by R is denoted as $L(R)$. We define the set of all destinations $Dest_x(R, \alpha)$.*

To revisit the concept of pointers in programming languages, we can view routing fields as analogous to pointers. In this context, routing expressions serve as specifications that determine the target or destination to which the pointer is pointing [7].

Examples of Graph Types Firs we return to the example show that Data types are not sufficient to expres a list of integer's with a pointer to the last element of the list. First we can define the production for this List of integers as follows [7]:

$$\begin{aligned} H &\rightarrow (first : L, last : L[\downarrow frist \downarrow tail^{\star}\$ \uparrow]) \\ L &\rightarrow (head : Ltail : L) \\ &\rightarrow (head : Ltail : L) \end{aligned}$$

Using this production from the example list from earlier we can construct the graph Figure 2.3:

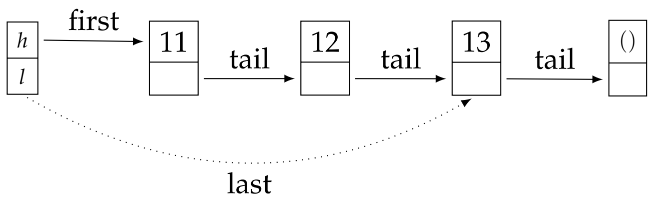

Figure 2.3: Representing a list of integers as a graph type with a first and last pointer [7].

The routing expression $\downarrow frist \downarrow tail^{\star}\$ \uparrow$, should be read as: " move down the $first$ pointer ($\downarrow first$) follow the *tail* until encountering a leaf ($\downarrow tail \star \$$) and the move up once ($\uparrow$)".

2.2.3 *Graph type limitations*

Graph types a limited in that they can only represent structures that are deterministic, which in the context of *recursive data type*, states that all pointers must be a functions of the underlying spanning tree. For example, we can not represent a pointer from the root to an arbitrary point in the tree structure [7].

Another limitation arises from the property of context-free grammar. It restricts our ability to represent pointers that are context-sensitive, meaning we cannot accurately represent structures where in a pointer is dependent on the structure of the underlying tree [7].

2.3 MONODIC SECOND ORDER LOGIC OVER GRAPH TYPES (M2LGT)

We use a monodic second-order logic over graph types also denoted (M2LGT) to express properties over graph types. We first introduce the notion of monodic second-order logic over Data types (M2LDT);

Definition 4. *M2LDT [7] We define the M2LDT on a Data type D as follows: Let x be a value variable of the Data type D, and let M be the set of addresses of D. The value variables and address set variables can be combined using the set operations $\cup$, $\cap$, and $\emptyset$ to form set expressions. The set $dom(x)$ represents the set of addresses, which can also be viewed as a set expression. Let α be a first-order variable in our logic, which we refer to as an address variable. We state that α represents an address of data type D. Value variables x of type D can be introduced using existential quantification ($\exists_D x$) or universal quantification ($\forall_D x$).*

We now define the basic formulas and connectives of the logic:

Definition 5. *M2LDT connectives and basic formulas Formulas of the logic define in Definition 4 are formed by the usual logic terms $\exists, \forall, \wedge, \vee, \neg$. We introduce the following basic formulas in the logic:*

$is^{\wedge}(\alpha)$	$\alpha = \epsilon$
$is_x\$(\alpha)$	$x(\alpha)$ *is a leaf variant*
$is_x(T:v)(\alpha)$	$x(\alpha) = (T:v)$
$is_x T(\alpha)$	$x(\alpha) = (T:v)$ *for some* v
$is_x walk(\alpha, \beta, R)$	$\exists p \in L(R) : \alpha \overset{p}{\leadsto}_x \beta$

$\alpha = beta$	
$\mathcal{E} = \mathcal{E}_2$	
$\mathcal{E} \subseteq \mathcal{E}_2$	
$\alpha \in \mathcal{E}$	
$\alpha \in \beta \star a$	
$\alpha \in \beta \overset{\star}{x} a$	$\alpha \in F_D x(\beta) \wedge \alpha = \beta \star a$

Where $\mathcal{E}_i$ be address set expressions. Formulas of the form $is_x^{\cdots}(\ldots)$ should be interpreted as follows: for example, $is_x walk(\alpha, \beta, R)$ is true if there exists a route $p \in \mathcal{L}(R)$ recognized by the language $L(R)$, and in such case, there must exist a walk from α to β denoted by $\alpha \overset{p}{\leadsto} o_x\beta$ [7].

Well-formedness Graph Types [7] To show how one might use this logic to prove properties of a given type lets consider the case of a simple graph type, where the notion of *will-formedness* is expressed in *M2LDT*.

Definition 6. *Well-formedness over Graph types: A graph type Data G, given data Fields $\alpha \in F_D$, where F_D is the backbone of Data G, and for a every transition $\alpha \overset{d}{\leadsto}_x \beta$, where $x \in Val; G$, there exits a unique β where the destination leads to a subtree of the specified type.*

The property shown in Definition 6 can be expressed in *M2LGT* as follows:

$$
\begin{aligned}
&\forall_D x : \\
&\quad \text{AND}_{T \in T_D, v \in V_D T} \\
&\quad \text{AND}_{a \in F^R(T:v)} \\
&\quad \forall \alpha \in dom\ x : \exists! \beta : is_x(T:v)(\alpha) \\
&\qquad\qquad\qquad \Rightarrow is_x walk(\alpha, R_D x(\alpha) a) a
\end{aligned}
$$

where $D = Data\ x$, where $\exists!$ is the abbreviations of " there exist a unique", and AND is the expansion over the corresponding indices.

2.4 POINTER ASSERTION LOGIC

This section presents *Pointer Assertion Logic* (PAL) [13], which uses a *monadic second-order logic* on graph types (see Section 2.2), to express pointer directives. The explanation of PAL in this section will provide an understanding of its workings, which will be utilized later to formulate a transformation of *MIR* to *PALE*.

PAL enables quantification over heap records, supporting both set-based and individual element-based quantification due to its second-order nature. For convenient navigation within the heap, PAL employs routing expressions as described in Section 2.2.

We first define what a data structure invariant is see Definition 7

Definition 7. *Data structure invariant Let D be a data structure, and let ϕ denote a logical predicate that describes a property on D. We denote the evaluation of a data structure invariant as $IV_\phi(D)$. Let Q denote a program that performs some* *operation on the data structure D. For $IV(D)$ to hold, the following properties must hold at every program point:*

- *Preservation: $IV_\phi(D)$ holds true before and after the execution of Q. In other words, if $IV_\phi(D)$ is true before Q is executed, it must also be true after Q has completed.*
- *Initialization: $IV(D)$ must hold upon initialization of D.*

***PAL* Store Model** The memory model of *PAL*, consists of two main components: the *heap* and a set of program variables.

The *heap* is a data structure that stores records, where each record consists of multiple fields. Each field within a record can either hold a pointer or a boolean value. Pointers in *PAL* can either point to another record within the heap or have the special value `null`, indicating that they do not point to any valid record.

Program variables in *PAL* can be categorized into two types: *data variables*, and *pointer variables*. A *data variable* serves as the root of a data structure. *pointer variable* can hold a pointer value that can point to any record within the heap.

A *PAL* program is composed of a set of type, variables, and procedures declarations see Grammar 2.4.1.

$$
\begin{array}{lcl}
\texttt{pale} & ::= & (\texttt{declarations})^* \\
\texttt{declaration} & ::= & \texttt{typedecl} \;\mid\; \texttt{progvar} \\
 & \mid & \texttt{procedure} \\
\texttt{typedecl} & ::= & \texttt{type}\ T = \{\ (\texttt{filed}\ ;)^*\ \} \\
\texttt{field} & ::= & \texttt{data}\ p^{\oplus} : T \\
 & \mid & \texttt{pointer}\ p^{\oplus} : T[\texttt{form}] \\
 & \mid & \texttt{bool}\ p^{\oplus} \\
\texttt{progvar} & ::= & \texttt{data}\ p^{\oplus} : T \\
 & \mid & \texttt{pointer}\ p^{\oplus} : T \\
 & \mid & \texttt{bool}\ p^{\oplus} \\
\texttt{procedure} & ::= & \texttt{proc}\ n(\texttt{progvar}^{\otimes}) : (T \mid \texttt{void}) \\
 & & (\texttt{logicvar}\ ;)^* \\
 & & \texttt{property} \\
 & & (\{(\ \texttt{progvar}\ )^*\texttt{stm}\})^? \\
 & & \texttt{property}
\end{array}
$$

Grammar 2.4.1: PAL grammar, [13], [12]

The symbols $\oplus$ and $\otimes$ denote comma-separated lists that consist of one or more elements and zero or more elements, respectively. The variables T, p, b, and n are taken from the domains of type names, pointer variables or fields, boolean variables or fields, and procedures, respectively.

A `typedecl` is used to define a type where a number of `fields` can be defined. `Fields` can be either `data`, which corresponds to data fields in the backbone as defined in Section Section 2.2, `pointer`, which defines pointers in the underlying backbone where the destination is defined by a formula `form`, or `bool`, which can be used to model finite values.

A procedure consists of a name n, the formal parameters to the procedure `progvar`, the return type of the procedure (which can either be T or `void`), and lastly, the body of the procedure which consists of local variable declarations `progvar` and `stm` [13]. A statement is define as follows:

```
stm   ::=  stm stm
        |  asn⊕;
        |  procall;
        |  if (condexp) {stm} (else {stm})?
        |  while property (condexp) {stm}
        |  return progexp
        |  assert property
        |  split property (property)?
asn   ::=  lbexp = (condxp | procall)
        |  lptrexp = (ptrexp | procall)
```

Grammar 2.4.2: Statements and assignment [13]

The language allows for multiple assignment where the wright-hand side is evaluated first. Expressions are defined as follows:

```
condexp  ::=  bexp  |  ?  |  [form]
bexp     ::=  (bexp)  |  ! bexp
           |  bexp & bexp  |  bexp | bexp
           |  bexp => bexp  |  bexp <=> bexp
           |  bexp = bexp  |  ptrexp = ptrexp
           |  bexp != bexp  |  ptrexp != ptrexp
           |  true | false  |  lbexp
lbexp    ::=  b  |  ptrexp . b
ptrexp   ::=  null  |  lptrexp
lptrexp  ::=  p  |  ptrexp . p
procall  ::=  n ( (condexp | ptrexp)⊗ ) [formula]
```

Grammar 2.4.3: Expressions [13]

The operator `.` is of notable significance as it serves the purpose of dereferencing a pointer. It allows access to the data pointed to by a given pointer.

The logic allows for the definition of formulas with the following syntax:

$$\begin{array}{lll}
\texttt{form} & ::= & (\ \texttt{existpos} \mid \texttt{allpos}\ )\ p^{\oplus}\ \texttt{of}\ T : \texttt{form} \\
 & \mid & (\ \texttt{existset} \mid \texttt{allset}\ )\ p^{\oplus}\ \texttt{of}\ T : \texttt{form} \\
 & \mid & (\ \texttt{existptr} \mid \texttt{allptr}\ )\ p^{\oplus}\ \texttt{of}\ T : \texttt{form} \\
 & \mid & (\ \texttt{existbool} \mid \texttt{allbool}\ )\ s^{\oplus}\ \texttt{: form} \\
 & \mid & (\texttt{form}) \quad \mid \quad !\,\texttt{form} \\
 & \mid & \texttt{form \& form} \quad \mid \quad \texttt{form | form} \\
 & \mid & \texttt{form => form} \quad \mid \quad \texttt{form <=> form} \\
 & \mid & \texttt{ptrexp in setexp} \quad \mid \quad \texttt{setexp / setexp} \\
 & \mid & \texttt{setexp = setexp} \quad \mid \quad \texttt{setexp != setexp} \\
 & \mid & \texttt{empty (setexp)} \quad \mid \quad \texttt{bexp} \\
 & \mid & \texttt{return} \quad \mid \quad \texttt{n . b} \\
 & \mid & m\ ((\texttt{form | ptrexp | setexp})^{\otimes}) \\
 & \mid & \texttt{ptrxep < routingexp> ptrexp} \\
\texttt{predicate} & ::= & \texttt{pred}\ m(\texttt{logicvar}^{\otimes}) = \texttt{form}
\end{array}$$

Grammar 2.4.4: The syntax for PAL as presented in [13]

We use the identifiers m and s to denoted predicates and set variables. The `pos` and `ptr` are quantifiers over heap records, where s also includes the `null` value. Routing expression formulas of the form $\texttt{p}_1\texttt{<r>}\texttt{p}_2$ constitutes a walk as defined in Definition 2, which states that the formula is satisfied if there exist a walk from $\texttt{p}_1$ to $\texttt{p}_2$ that satisfy `r`.

Returning to the definition of procedures as seen in Grammar 2.4.1, where the `logicvar` is used to defining logic variables that can only be referenced in formulas `form`, and not a procedure body. We define `logicvar` as Grammar 2.4.5:

$$\begin{array}{lll}
\texttt{logicvar} & ::= & \texttt{Pointer}\ p^{\oplus} : T \\
 & \mid & \texttt{bool b}^{\oplus} \\
 & \mid & \texttt{set}\ s^{\oplus} : T
\end{array}$$

Grammar 2.4.5: Grammar of `logicvar` as presented in [13]

We can utilize `logicvar` to establish a relationship between the pre- and post-conditions of procedures. `Logicvar` serves as universally quantified variables in this context, enabling the specification of logical properties that hold universally.

Additionally, we extend `ptrexp` in formals to include the construct `return` | `n . p`. This extension allows for accessing the returned value in the post-condition of a procedure. Furthermore, this capability is also applicable within procedure call formulas, enabling the inclusion of the returned value in their formulas.

Set expressions contain the usual set operations, with the addition of an up operation `x ^ T.p` which denotes a set of records of a given type `T` that have a `p` successor to `x`. The grammar of `setexp` is defined in Grammar 2.4.6

```
setexp   ::=  s
            |  ptrexp ^ T . p
            |  { ptrexp⊕}
            |  setexp ∪ setexp
            |  setexp ∩ setexp
            |  setexp \ setexp
```

Grammar 2.4.6: Grammar of `setexp` as presented in [13]

The notion of routing expressions found in Section 2.2 is slightly more generalized in PALE, the grammar of routing expressions `routingexp` is defined in Grammar 2.4.7.

```
routingexp  ::=  p  |  ^T . p  |  [form]
              |  routingexp.routingexp
              |  routingexp + routingexp
              |  (routingexp)  |  routingexp ⋆
```

Grammar 2.4.7: Grammar of `routingexp` as presented in [13]

We can use `routingexp` to move up of down, `pointer`, `data` fields or `formula`, where the latter is a formula with extra free variable pos that filters away records that course the formula to evaluate to false when pos denotes on of them.

As a general rule, `pointer` fields are expected to adhere to the formula specified in their type declaration. However, in imperative languages, it is common to encounter situations where data structure invariants need to be violated. To accommodate such scenarios, we introduce pointer directives to override the specified formula. The pointer directives are denoted as `ptrdirs ::= (T . p [form])`$^{\oplus}$. These directives provide a means to deviate from the expected formula and allow for the necessary modifications in the behavior of the pointer fields.

Both the pointer directives defined in the type declaration and the ones overwritten need to be well-formed. This implies that at any given point in the program and its store, each pointer directive must uniquely identify one record. In other words, the directives should accurately specify the target location for each pointer field, ensuring consistency and coherence in the programs store.

Wrapping up this section, we now present the definition of `property`. A property is represented by the following structure: `property ::= [form ptrdirs]`. It denotes a set of stores where the following conditions must hold:

- The formula `form` is satisfied.
- A `data` variable must represent a disjoint acyclic backbone spanning the heap.
- Each `pointer` field must adhere to its respective pointer directive. If it fails to do so, it violates the property.

Properties are used in procedure as pre- and post-conditions and `while` loops as loop invariants, `split` statements which contain two properties, assumptions and assert. By satisfying these conditions, a property ensures the integrity and correctness of the program's state and memory.

single linked list example PALE

We now present an example of a PALE program and explain the different formulas and there meaning. The PALE program is shown in Code listing 2.4.1.

```
type List = {data next:List;}

pred reachable(pointer x:List, set R:List) = x<next*>R;

proc reverse(data x:List):List
  pointer P,Q:List;
  set R:List;
[reachable(x,R) & P=x & (x!=null => Q=x.next)]
{
  pointer t:List;
  if (x!=null) {
    if (x.next!=null) {
      t = reverse(x.next) [P=x & x!=null & x.next!=null & Q=x.next & !empty(reverse.R) & R=reverse.R union {x} & !(x in reverse.R) & reverse.P=Q];
      x.next.next = x;
      x.next = null;
      x = t;
    }
  }
  split [reachable(x,R) & (P=null | P.next=null) & (x!=null => x<next*>P)]
  return x;
}
[reachable(return,R) & (P=null | P.next=null) & (return!=null => return<next*>P)]

```

Code listing 2.4.1: Simple Linked list Implementation in PALE with program annotations [1]

1 Link to the sourer of the code shown in Code listing 2.4.1, https://www.brics.dk/pale/Examples/recreverse.pale

The code shown in Code listing 2.4.1, implements the simple data structure of a single linked list `List` in PALE, and the function `reverse` that transposes the order of the list. We only focus on the logic of the implementation, and not the implementation of the `reverse` procedure.

We first define the data type `List`, which contains a data filed `next`; We then define the predicate `reachable`,

```
pred reachable(pointer x:List, set R:List) = x<next*>R;
```

which states that given a `pointer` to `x` of type `List`, and `set R:List`, the routing expression `x<next*>R` which states that from the pointer `x` we can take any number of `next` pointer `*` and reached every element in `R` form `x`.

We then define the precondtion for the procedure `reverse`:

```
proc reverse(data x:List):List
  pointer P,Q:List;
  set R:List;
[reachable(x,R) & P=x & (x!=null => Q=x.next)]}
{...}
```

We first define three `logicvars` `pointer P,Q:List;`, and `set R:List`. Next we define the precondition of the procedure which states that `reachable(x,R)`, `P=x`, and `(x!=null => Q=x.next)`. We use the `logicvars` to relate variables in the procedure to correlating ones in the logic.

The next operation of interset is the recursive call to `reverse`, where we define a call invartion of the procedure:

```
t = reverse(x.next) [P=x & x!=null & x.next!=null & Q=x.next &
↪  !empty(reverse.R) & R=reverse.R union {x} & !(x in reverse.R) &
↪  reverse.P=Q];
```

A call invariant states what holds before the call to a given procedure. Taking the example shown above, starting from the first `form` `P=x & x!=null & x.next!=null & Q=x.next`, here we state the following:

- In the recursive call to `reverse`, the `logivar` `P` is equal to the current list `x`, and `reverse.P=Q`
- We also state that `x` and `x.next` are not equal to `null`.
- The remaining part of the formula states the following: in the recursive call to `reverse`, the set `R` connotations the current elements in `R` and the element `x`. We also state that `x` is not currently in the list `R`, and that `!empty(reverse.R)` states that `R` is not empty.

The last formula in the procedure is the postcondition, where we state that `reachable(return,R)` is satisfied, and `(p = null | p.next=null)` which means that we have reached the end of the list structure.

```
(return!=null => return<next*>P)
```

states that we can reach the list element `P` form the returned list structure.

2.4.1 *PALE guaranties*

PALE is translated into *MONA*, where the following properties of a given program is validated:

- Pointer directives of the form `pointer p : T [form]` are well-formed.
- No null pointer dereference can occur.
- At each cut-point of a given `data` variable, the following is satisfied:
  - `data` variables contain disjoint acyclic backbones spanning the heap.
  - All assertions and pointer directives are satisfied.
- All `assert` statements are valid.
- All cut-point properties are satisfiable.

We will not present the inner workings of *MONA*, we refer the reader to the manual [6]. Returning to the example shown in Code listing 2.4.1, we can verify that all the properties stated above are valid.

3 *MIR* TO *PALE* TRANSFORMATION

The main idea of this thesis is to model the behavior of *MIR* programs within the *PALE* while preserving their essential characteristics. This involves transforming *MIR* programs into an equivalent representation in *PALE*, ensuring that the resulting *PALE* programs capture the same underlying behavior as the original *MIR* programs.

By transforming *MIR* programs into the *PALE*, we gain the ability to provide annotations and demonstrate that a given *MIR* program does not exhibit certain program behaviors which are described in Section 2.4, and that data structure invariants are not violated as defined in Definition 7.

It should be noted that the *proof idea* provided for each semantic constructed of *MIR*, in the transformation serves as an informal argument for the correctness of the transformation, rather than a formal proof.

We hope to show that this transformation is simple and proves beneficial in proving the correctness of Rust programs.

3.1 MIR TO PALE TRANSFORMATION

This section focuses on the transformation of the *MIR* program into its equivalent *PALE* representations. In this transformation, we have the following simplifications:

- We have chosen to model only the primary types of the given *MIR* program. Consequently, types defined in the standard library of the *Rust* language are not considered in our transformation process.
- Our transformation specifically targets control-flow altering operations within the *MIR* program.

Additionally, we provide a series of *Proof Ideas* accompanying the transformation of control-flow altering operations in *PALE*. It is important to note that these proof ideas serve as informal arguments supporting the correctness of the transformation, without constituting formal proofs of equivalence.

MIR (memory locations) `Places` *in PALE*

To represent MIR `Places` in PAL, we can utilize global declarations for Places that live the intire duration of a function. Let's consider the MIR program depicted in Listing 3.1.1.

```
fn test(_1: i32) -> () {
    debug x => _1;
    let mut _0: ();
    let _2: *const i32;
    let _3: &i32;
    let _5: &i32;
    ....
}
```

Code listing 3.1.1: MIR example for memory location declarations

The function declaration will be discussed later. The code shown in Code listing 3.1.2 is the corresponding PALE program the models the MIR code shown in Code listing 3.1.2.

```
pointer 2_test : T
pointer 3_test : T
pointer 5_test : T
...
```

Code listing 3.1.2: PALE example for memory location declarations

The general idea of how we model `Places` in PALE is the following `Place_fun : T` where `Place` is a location in a MIR function we add the `fun` identifier if there are multiple functions in the same MIR program.

In the MIR example, both `*const` and `&i32` are pointers. They represent a pointer to a value of type `i32` this corresponds to `pointer p : T` in PALE.

Regarding the return location `_0` of functions, is not explicitly modeled in this context. The handling of return locations is managed by the procedure definition (`proc`) in PALE, as explained in Section 2.4.

Unsafe *Rust* in *PALE* In *MIR*, the notion of `Unsafe` is removed. We treat every *Place* as a memory location in *PALE*. This removal is not problematic, and we argue that we can capture this language construct in *PALE*, by overaproximation. By considering every *Place* as a memory location in *PALE*, we can effectively model and reason about the behavior of memory operations. Although the `Unsafe` keyword in Rust allows bypassing certain language restrictions and accessing low-level operations, we can handle these operations safely within *PALE* with annotations.

MIR Functions `fun` *and basic block* `Block` *in PALE*

To model function `funs`, and `Blocks` in PALE we simple transforms them into procedures in PALE, a simple example is the function shown in Code listing 3.1.1, specifically the function `fn test(_1 : i32) -> ()` this is simply translated into the equivalent *PALE* procedure shown in Code listing 3.1.3.

```
--- PALE ---
proc test(progvar 1_test : T) : void
property {...} property
```

Code listing 3.1.3: PALE procedure for the MIR function `fn test(_1 : i32)`.

We can transform *basic blocks* in the same manner, although basic blocks always have a return value of type `void`. Consider the shown basic blocs in Code listing 3.1.4,

```
--- MIR ---
bb0: {
  _10 = ...
}
--- PALE ---
proc bb0_test() : void {
  progvar 10_test_bb0 : T;
}
```

Code listing 3.1.4: Small example of basic blocks i MIR.

transforming this into PALE, constitutes a procedure `proc bb0_test() : void`, procedures that originates from a *basic block* do not contain any input parameters as our memory locations are stored globally, except for temporary Places within a given *basic block*. We model temporary Places in side a basic block, we use the following naming convention for `progvars` that live inside a basic block `progvar place_fun_bbi` where $i \in \mathbb{N}$, see Code listing 3.1.4.

Modeling MIR values in PALE Modeling MIR values in PALE poses some limitations. PALE does not support arithmetic values, meaning that the only values we can represent in a PALE program are boolean values. These boolean values can be either `true` or `false`.

While this might initially appear limiting, it is important to note that PALE is still capable of expressing the control flow operators of MIR, that are dependent on values. As an example consider the MIR Function `GT(move _4, const 0_i32)`, the function `GT` that test if `_4` is grater than `const 0_i32` which represent a 0 value of type `i32`.

We can model the behavior of `GT` as a PALE procedure `proc GT(bool vale1, bool value2)` see PALE code in Code listing 3.1.5.

```
--- PALE ---
proc gt(bool value1, bool value2) : bool
[true]
{
  bool res;
  if (value1 & !value2){ /* value1 > value2 */
    res = true;
  } else {
    res = false;
  }
  return res;
}
[true]
```

Code listing 3.1.5: MIR `GT` function modeled in PALE

For more detailed about the modeling of conditionals over arithmetic values in the *PAL* see [13].

MIR control-flow in PALE

In order to accurately model the program flow of a *MIR* program in *PALE*, it is necessary to define how each operation in *MIR*, which has the ability to split the program flow, is translated. These operations, known as terminators, include `goto`, `function calls`, `switchInt`, `asserts`, `panics`, and `return`. For more detailed information on these terminators, see Section 2.1.

By establishing the translation of each of these terminators, we can effectively capture the branching and control flow behavior of the *MIR* program within *PALE*.

Goto terminals `goto` $\rightarrow$ `Bid` To represent *MIRs* `goto` terminal in PALE, we can simply model them as procedure calls `proccall` see Grammar 2.4.1 for more detail. An example of this transformation can be seen in Code listing 3.1.6 for representing `goto` statement in PALE.

```
--- MIR ---      | ---- PALE ----
  goto -> bb1;   |    bb1() [form];
```

Code listing 3.1.6: MIR `goto` statements model in PALE

Proof Idea of *goto* Terminals : *Let G_{MIR} denote the (CFG) of a given MIR program P_{MIR}, and let G_{PALE} denote the CFG of the transformation $P_{MIR} \rightsquigarrow P_{PALE}$. Consider two nodes $BB1_{MIR}$ and $BB2_{MIR}$ in G_{MIR}, and let $BB1_{PALE}$ and $BB2_{PALE}$ denote the corresponding nodes in G_{PALE}. Assume that there exists an edge $BB1_{MIR} \overset{\texttt{goto -> bb2}}{\rightsquigarrow}_{MIR} BB2_{MIR}$ in G_{MIR}.*

We need to show that there exists an equivalent edge $BB1_{PALE} \overset{\texttt{BB2() [form]}}{\rightsquigarrow}_{PALE} BB2_{PALE}$ in G_{PALE}.

By applying the transformation, the edge $BB1_{MIR} \overset{\texttt{goto -> bb2}}{\rightsquigarrow}_{MIR} BB2_{MIR}$ *is transformed to* $BB1_{PALE} \overset{\texttt{BB2() [form]}}{\rightsquigarrow}_{PALE} BB2_{PALE}$ *in* G_{PALE}*. Thus, we have can argue that if there exists an edge* $BB1_{MIR} \overset{goto\ ->\ bb2}{\rightsquigarrow}_{MIR} BB2_{MIR}$*, there must exist an equivalent edge* $BB1_{PALE} \overset{BB2()\ [form]}{\rightsquigarrow}_{PALE} BB2_{PALE}$ *in* G_{PALE}*.*

Since the transformation from P_{MIR} *to* P_{PALE} *aims to preserve the control flow structure, we can argue that the transformed program* P_{PALE} *preserves the control-flow as the original program* P_{MIR}*, and thereby modeling the same behavior of* `goto` *statements in MIR.*

Function calls `place = i (...) -> Bid` We can model function calls from *MIR* in *PALE* as an assignment followed by a procedure calls.

```
--- MIR  ---
bb0: {
      _1 = process(true) -> bb1;
}

bb1: {
  _0 = _1;
  return;
}
fn process(_1: i32) -> i32 {
    bb0: {
        _2 = false;
        return;
    }
}
--- PALE ---
proc bb0() : void {
  1_bb0 = process(true) [ form ];
}

proc process(pointer 1_process : bool ) : bool
property
{
  2_process = false;
  return 2_process;
}
property
```

Code listing 3.1.7: MIR function, and panic terminals model in PALE, place declarations have be removed for simplicity in the MIR and PALE code

Proof Idea of `fucntion calls` : *We argue that the same reasoning applied to the transformation of* `goto` *terminals can be used for function calls and panics as well, to give an informal argument for the validtity of the transformation.*

Conditional terminals `switchInt` We can model MIRs conditional terminal `switchInt` as `if (condexp) stm (else stm)`? in PALE.

Consider the example MIR code shown in Code listing 3.1.8.

```
--- MIR ---
bb0: {
    _2 = Not(_1);
    switchInt(move _2) -> [0: bb2, otherwise: bb1];
}
bb1: {
    _0 = const false;
    goto -> bb3;
}
bb2: {
    _0 = const true;
    goto -> bb3;
}
bb3: {
    return;
}
```

Code listing 3.1.8: MIR `swtichInt` terminal *MIR* code based on *Rust* code in Appendix C Code listing C.0.1, function and place declarations have be removed for simplicity.

```
proc test(pointer 1_test bool ) : bool
property
{
  if 1_test == false {
    bb2_test() [ form ];
  } else {
    bb1_test() [ form ];
  }
}
property

proc bb1_test  : void
property
{
  0_test = false;
  bb5_test() [ form ];
}
property

proc bb2_test : void
property
{
  0_test = ture;
  bb5_test() [ form ];
}
property

proc bb3_test : void
property
{
  return 0_test;
}
property
```

Code listing 3.1.9: PALE program that models the *MIR* program shown in Code listing 3.1.8

Proof Idea of *swtichInt* Terminals : *Let $G_{MIR} = (V_{MIR}, E_{MIR})$ denote the CFG of a given MIR program P_{MIR}, and let $G_{PALE} = (V_{PALE}, E_{PALE})$ denote the CFG of the transformation $P_{MIR} \rightsquigarrow P_{PALE}$, into and equivalent PALE program.*

We aim to demonstrate that the transformation $f : P_{MIR} \rightsquigarrow P_{PALE}$, along with the graphs G_{MIR} and G_{PALE}, are isomorphic.

Assuming the existence of an edge $v_i \rightsquigarrow v_j \in E_{MIR}$, we argue that there exists an equivalent edge in $f(v_i) \rightsquigarrow f(v_j) \in E_{PALE}$. This argument is based on the observation that the behavior of the `switchInt` *construct can be equivalently modeled by the PALE construct* `if (condexp) stm (else stm)`$^?$.

Thus, for every edge $v_i \rightsquigarrow v_j \in E(G_{MIR})$, the transformation f preserves the edge and maps it to an equivalent edge $f(v_i) \rightsquigarrow f(v_j) \in E(G_{PALE})$.

MIR loop constructs model in PALE To model MIR loop constructs in in PALE, we need to analyses a given MIR program for cycles, this can be done either by hand or with the aid of algorithmic methods that can find *dominators* [8]. Let consider a simple example of a MIR program that contains a cycle see Code listing 3.1.10.

```
bb0: {
  goto -> bb1;
}

bb1: {
  _2 = const true;
  switchInt(move _2) -> [0: bb3, otherwise: bb2];
}

bb2: {
  _1 = const false;
  goto -> bb1;
}

bb3: {
  _0 = _1;
  return;
}
```

Code listing 3.1.10: MIR program that contains a while loop construct.

We can model the same program behavior in *PALE*, by the used of *if* statements, consider the code shown in Code listing 3.1.11

```
proc bb0() : void
property
{
  bb1() [form];
}
property

proc bb1() : void
property
{
  2_bb0 = true;
  if (2_bb0 == true) { bb2() [form]; }
  else {bb3() [form];}
}
property

proc bb2() : void
property
{
  1_bb2 = const false;
  bb1() [ form ];
}
property


proc bb3() : void
property
{
  0_bb0 = 1_bb2;
  return 0_bb0;
}
property
```

Code listing 3.1.11: PALE model for the MIR program shown in Code listing 3.1.10.

The code shown in Code listing 3.1.11, is the direct transformation of the *MIR* code shown in Code listing 3.1.10, by the use of *PALE* `if {} else {}` construct. Notice that in its current form, we can observe a cycle between the procedures `bb1` and `bb2`. This cycle constitutes the original while loop in the high-level *Rust* program. To simplify the program and capture the behavior of the while loop, we can utilize the *PALE* `while` statement. The modified program, depicted in Code listing 3.1.12, incorporates the *PALE* `while` statement to represent this looping behavior. We also move the statements inside every basic block procedure into the while loop and `proc bb1()`.

```
proc bb1() : void
property
{
  2_bb0 = true;
  while (2_bb0 == true) {
    1_bb2 = false;
  }
  0_bb0 = 1_bb2;
  return 0_bb0;
}
property
```

Code listing 3.1.12: Improved *PALE* model of the *PALE* program shown in Code listing 3.1.11.

Proof Idea of *Loop construct* : *Let G_{MIR} denote the (CFG) of a given MIR program P_{MIR} that contains a cycle, and let G_{PALE} denote the CFG of the transformation where f, is a bijective $f : P_{MIR} \rightsquigarrow P_{PALE}$. Let ϕ_{MIR} denote a circuit in G_{MIR}, where $\{bb1_1, bb2_2, \ldots bb3_n\} \in V_{MIR}$, where $n \in \mathbb{N}$ is the length of the circuit ϕ_{MIR}. For the transformation f to preserve the behavior of the P_{MIR}, the CFG G_{MIR} and G_{PALE} must be isomorphic. We argue that the transformation f preserves the same circuit in G_{PALE}, and therefor is f is isomorphic, under the assumption that* `while` *PALE exhibits the same behavior.*

Given the behavior of each *MIR* terminal, we believe that we have successfully captured the general control flow and the concept of memory in *MIR* using *PALE*.

Through the transformation process from *MIR* to *PALE*, we have preserved the essential control flow constructs and memory operations present in *MIR*. We argue that *PALE* provides suitable constructs and mechanisms to represent and simulate the behavior of *MIR* programs accurately.

3.2 RUST DATA TYPES IN PALE

This section will cover how we model Rust data types in PALE, we will only be explaining the different data types that we actually need. To model Rust user defined types in PALE we can use the `typedecl` language construct in PALE see Section 2.4; As an example consider the type `List` defined in Section 1.5 see code in Code listing 3.2.1.

```
pub struct List<A: Adapter + ?Sized> {
    first Option<NonNull<A:EntryType>>,
}

pub unsafe trait Adapter {
    type EntryType: ?Sized;
    fn to_links(obj: &Self::EntryType) -> &Links<Self::EntryType>;
}

pub struct Links<T: ?Sized>(UnsafeCell<MaybeUninit<LinksInner<T>>>);

struct LinksInner<T: ?Sized> {
    next: NonNull<T>,
    prev: NonNull<T>,
    _pin: PhantomPinned,
}
```

Code listing 3.2.1: Rust type `List` presented in Section 1.5

The code shown in Code listing 3.2.2, models the data structure of `List` in `unsafe_list` shown in Code listing 3.2.1.

```
type List = {
    data first: EntryType,
}
type EntryType = {
    bool value;
    data links : LinksInner;
}

type LinksInner = {
    pointer next : EntryType [form],
    pointer prev : EntryType [form],
}
```

Code listing 3.2.2: PALE type `List`

As noted early we do not model lower-level types in Rust, such as `Option` and `NonNull`. Similarly, we can abstract away the notion of traits, as they serve as a programming convenience. To represent the same structure, we can add a `links` data field to the `EntryType`. This `links` field would encapsulate the relevant functionality and relationships previously expressed through traits.

3.2.1 *Rust Kernel List Datastructure invariant*

We now present the datastructure invariant for the Rust Kernel List. We use the definition of data structure invariant found in Section 2.4 Definition 7.

We first define a graph type over the List as described in Section 2.2.

$$\begin{aligned}
List &\rightarrow (\texttt{first} : EntryType) \\
EntryType &\rightarrow (\texttt{data} : bool,\ \texttt{links} : LinksInner) \\
LinksInner &\rightarrow (\texttt{next} : EntryType,\ \texttt{prev} : EntryType[\phi_{prev_1}]) \\
&\rightarrow (\texttt{next} : EntryType[\phi_{\texttt{next}}],\ \texttt{prev} : EntryType[\phi_{\texttt{prev}_2}]) \\
&where \\
\phi_{next} &= \uparrow^{\star}\ {}^{\wedge} \\
\phi_{prev_1} &= \uparrow + {}^{\wedge} \downarrow \texttt{links.next}^{\star}\texttt{\$} \\
\phi_{prev_2} &= \uparrow +^{\wedge}
\end{aligned}$$

The first routing expression ϕ_{next} states that we follow the `next` pointer until we reach the root of `EntryType`. ϕ_{prev_1} uses a the nondeterministic union operator $(+)$, this is because the routing expression has to be context dependent, if we are a the root we must move down $(\downarrow)$ the `next` pointer until $(^{\star})$ a leaf $(\$)$ is reached, if we are not in a root we move up $\uparrow$ the *prev* pointer.

We can express the same invariant on the data structure in PALE see Code listing 3.2.3.

```
type List = {
    data first: EntryType,
}
type EntryType = {
    bool value;
    data links : LinksInner;
}

type LinksInner = {
  data next : EntryType;
  pointer prev : EntryType [this.next^EntryType.links={prev}];
}
```

Code listing 3.2.3: PALE type `List`, with data structure invariant

In annotated version of the `List` type, shown in Code listing 3.2.3, we have added the routing expression `this.next^EntryType.links={prev}` which states the set of `EntryType`s that can be reach `this` leaf, though a `next` pointer must only contain the `prev` leaf.

3.3 VALIDATION RESULTS

This section will present the transformation of the `unsafe_list` data structure described in Section 1.5. We will focus on modeling the following functions: `remove`, `inner_ref`. We use the *PALE* types define in Section 3.2, namely `List`, `EntryType`, and `LinksInner`.

Due to memory limitations in MONA, we encountered difficulties in directly modeling the `unsafe_list` in our translation process. A complete example of the transformation can be found in Appendix D using our transformation method described in Section 3.1.

Given these limitations, this section aims to demonstrate how one might annotate a program in *PALE*. We will now attempt to prove a simplified example of `unsafe_list` in PALE, the *PALE* program that models the `remove` function is shown in Code listing 3.3.1.

```
proc remove(data list : List, pointer entry : EntryType) : void
set S:EntryType;
pointer E:EntryType;
pointer L:List;
[
list.first<(links.next)*>entry
&entry<(links.next)*>list.first
&entry<(links.prev)*>list.first
&list.first<(links.prev)*>entry
&entry!=null
& E=entry & L=list
]
{
    pointer inner_links : LinksInner;
    pointer next : EntryType;
    pointer prev : EntryType;

    inner_links = inner_ref(entry) [true];
    next = inner_links.next;
    prev = inner_links.prev;

    assumed_droped(inner_links) [true];

   if (entry = list.first){
       list.first = null;
    } else {
        next.links.prev = prev;
        prev.links.next = next;
        if (entry = list.first ){
            list.first = next;
        }
    }

}
[!(L.first<(links.next)*>E & L.first<(links.prev)>E )]
```

Code listing 3.3.1: Simplified transformation of the `remove` code shown in Appendix D

We will now explain the annotations and their significance in the context of the function `remove`.

(`List`) structure annotations pre-condition `remove` The precondition to the `remove` function includes several clauses that describe the relationships and properties of the elements involved:

- `list.first<(links.next)>entry,`and `& entry<(links.next)>list.first`:
  - The first clause states that we can reach the element `entry` by following the `links.next` pointers until we reach the node `entry`.
  - The second clause states that we can reach the element `list.first` by following the `links.next` pointers until we reach the node `list.first`.
  - These annotations are necessary to indicate that the element we are about to remove is currently present in the list.
  - Additionally, these annotations capture the fact that the list is circular, meaning that we can traverse the list from `list.first` and eventually reach any element using the `links.next` pointers.
- `& entry<(links.prev)>list.first,`and `& list.first<(links.prev)>entry`:
  - The first clause states that we can reach the element `entry` by following the `links.prev` pointers until we reach the node `entry`.
  - The second clause states that we can reach the element `list.first` from `entry` by following the `links.prev` pointers until we reach the node `list.first`.
  - These annotations are necessary to establish that the list is doubly linked and circular. They ensure that we can traverse the list in both forward and backward directions using the `links.prev` pointers.

By including these annotations in the precondition, we capture the necessary relationships and properties of the elements involved in the `remove` function. These annotations allow us to reason about the behavior of the function and ensure that the list is structured correctly before performing the removal operation. The additional annotations in the precondition of the remove function are as follows:

- `entry!=null`: This annotation states that the element entry is not null. It ensures that we are operating on a valid element that can not be null.
- `& E=entry`: This annotation states that the memory address of variable E is equal to the memory address of entry. It provides a reference to the element entry in the post-condition of the fucntion.
- `& L=list:` This annotation states that the memory address of variable L is equal to the memory address of list. It provides a reference to the list itself in the post-condition of the fucntion.

We also include the post-condition `!(L.first<(links.next)*>E & L.first<(links.prev)>E)`, which states the following:

This clause ensures that it is no longer possible to reach the element E from L.first by traversing the links.next pointers (`(links.next)*`) or the links.prev pointers (`(links.prev)*`).

By including this formal post-condition, we establish a clear verification criterion for the remove function, ensuring that it correctly maintains the integrity of the list structure after the removal operation We argue that this annotations should be enough to validate that correctness of the `List` is preserved by the `remove` operations.

`inner_ref` correctness We also state the correctness of Code listing 3.3.2.

```
proc inner_ref(data entry : EntryType) : LinksInner
pointer E: EntryType;
[entry!=null & E=entry]
{
    return entry.links;
}
[return!=null & return=E.links]
```

Code listing 3.3.2: Simplified transformation of the `inner_ref` code shown in Appendix D

The pre-condition ensures that the function is executed in a memory state where the `entry` variable is not pointing to a `null`. The use of the `logical` variable `E` allows us to access the input variable `entry` which in the post-condition of the procedure. By executing `inner_ref` in a memory state that satisfies its pre-condition, we end up in a memory state where the post-condition is satified,which means that the **`return`** value is guaranteed to be not `null`, and is equal to the `entry.links` variable.

`assumed_droped` correctness We also have to show that the function `assumed_droped` is correct, the annotations are shown in Code listing 3.3.3.

```

data garbage:EntryType;

proc assumed_droped(pointer links: LinksInner) : void
pointer Links :LinksInner;
[links!=null &  Links=links & garbage!=null]
{
    links.next = garbage;
    links.prev = null;
    garbage = links.next;
}
[Links.next=garbage ]
```

Code listing 3.3.3: Simplified transformation of the `inner_ref` code shown in Appendix D, also based on code shown in Appendix B.

We first define the global variable garbage, which is used as the deallocated value.

Next, we define the pre-condition of the function, which states that `links` is not null (`links != null`), that `logivar Links` is equal to links (`Links = links`), and `garbage` is not null (`garbage != null`). We believe that this annotations are sufficient to validate the correctness of Code listing 3.3.3.

3.3.1 *PALE to MONA results*

We convert the *PALE* program shown in Code listings 3.3.1 to 3.3.3 to MONA. *PALE* generates the following set formulas shown in Table 3.2.

Sets of Formula	Program point
0-Formula	stmt following call `assumed_droped` line 29
1-Formula	stmt following procedure `inner_ref` line 25
2-Formula	`remove`
3-Formula	`assumed_droped`
4-Formula	`inner_ref`

Table 3.1: MONA Formuals genreated from *PALE* program Code listings 3.3.1 and 3.3.2

For every *n-Formula* the properties shown in Table 3.1, are verified by *MONA*.

Formula	Property
n-PRECONDSAT	That the Preconditon holds.
n-NOTMEMFAILED	Pointer field formulas are valid, and that null pointer dereferences can not occur
n-NOCYCLES	Backbone does not contain cycles.
n-DATADISJOINT	Data variables are disjoint, and span the same cells as before execution.
n-DATACOMPLETE	Cells reachable from data variables are the same as before execution.
n-NOTASSERTFAILED	All assertions are valid.

Table 3.2: Formulas check by *MONA* see Section 2.4 for more detail, and [12]

In table Table 3.3, we show the result of our verification effort,

	Annotated					*Unannotated*				
n-Formula	0	1	2	3	4	0	1	2	3	4
PRECONDSAT	+	+	+	+	+	*	*	-	*	*
NOTMEMFAILD	-	-	+	+	+	-	-	+	-	-
NOCYCLES	-	+	+	+	+	+	+	+	-	+
DATADISJOINT	+	+	+	+	+	+	+	+	-	+
DATACOMPLETE	-	+	+	-	+	-	+	+	-	+
NOTASSERTFAILED	-	-	+	+	+	-	*	*	*	*

Table 3.3: Result of *MONA* verification, with annotations shown in Code listings 3.3.1 to 3.3.3, where (+) denotes valid , (-) denotes counter-example, and (*) denotes assertions where `true`. We highlight difference in gray.

Analysis of result shown in Table 3.3 The Table 3.3 indicates that some of the formulas have been satisfied, particularly the *2-Formula* and *4-Formula*, demonstrating that none of the listed formulas in Table 3.2 are voilated.

Regarding the *3-Formula*, we have successfully shown the validity of *NOTMEMFAILD*, *NOCYCLES*, and *DATADISJOINT*, which represents an improvement compared to the unannotated version.

However, we encountered violations in the *0-Formula*, specifically with the *NOCYCLES* and *DATACOMPLETE* conditions. The reasons behind these violations remain unknown, requiring further investigation to determine their underlying causes. The formula *0-NOTASSERTFAILED* was also violated,which indicates that after the call to `assumed_droped`, the post-condition of `remove` is no longer valid,

In regards to formulas *0- 1-*, and *2-Formula*, which are mentioned in Table 3.1, these are the main formulas that needed to be proven to demonstrate the validity of the `remove` function and its compliance with the *data structure invariant*. However, we were unable to successfully validate these formulas, indicating that our verification efforts in this regard have been unsuccessful.

4 DISCUSSION & CONCLUSION

4.1 DISCUSSION

***MIR* to *PALE* transformation** We still believe that exploring the logic of *PALE* further is valuable, although we have not been able to provide concrete proofs using our current approach. While we have presented arguments and informal reasoning for the correctness of the transformations, a rigorous formal proof is still pending.

Despite the absence of a concrete proof at this stage, the exploration of *PALE* in the context of *MIR* program verification of data structure invariants remains an interesting avenue for future research and development.

The approach we have taken to model `unsafe` *Rust* in *PALE* is an overapproximation, given that we do not utilize the guarantees imposed by the *borrow checker*. In our *PALE* model, we consider all *Places* as mutable. This overapproximation was necessary because *PALE* does not have the capability to express the fine-grained distinctions of memory locations as is possible in *MIR*.

Is *PALE* a good alternative to *SL* We believe that the utilization of graph types in the PALE language to express pointers inside data structures is a convenient and expressive way to handle pointer aliasing. Graph types provide a powerful abstraction that allows us to model and reason about complex relationships between objects in a more intuitive and structured manner.

MIR for deductive verification We believe that further development of formal methods for *Rust* should focus their efforts on the *MIR* level. This is primarily due to the fact that many high-level constructs are represented in a simpler and more uniform manner in MIR.

Limitations of *MONA* As mentioned in the section on Section 3.3, we encountered difficulties in providing a proof in the direct transformation of the *MIR* program `remvoe`. These challenges stemmed from the significant number of global variables required for the transformation, and the need to validate all formulas in Table 3.1 all global variables for every procedure call within the *PALE* program. Consequently, constructing a concrete proof for the correctness of the function `remove` became impractical.

We where also limited in our utilization *MONA*, as a consequence of lacking understanding of the translation process from *PALE* to the underlying logic of *MONA*. However, due to time limitations, we were unable to acquire this understanding.

As indicated by the results shown in Table 3.3, we were unable to demonstrate the correctness of the remove function, as we faced challenges in interpreting the counter-examples produced by the *MONA* tool.

4.2 CONCLUSION

In conclusion, we have demonstrated a potential transformation from *MIR* to *PALE*. However, it is important to note that this transformation lacks a comprehensive proof. Nevertheless, we believe that this methodology presents an interesting avenue for further research.

While we encountered challenges in demonstrating proving properties using *MONA*, we believe that further development of formal methods for *Rust* should focus on leveraging the advantages of *MIR*. The simplicity of its representations, make it a promising platform for the development of formal methods.

PALE and, in turn, *MONA* possess an interesting method for program verification. The simplified nature of graph types allows for intuitive definition of *data structures*. We have demonstrated that it is possible to formally verify some program properties of concrete yet simplified examples using this approach.

We believe that *monadic second order logic*, deserves more attention in the filed of formal program verification, it posses an interesting benefits for automation of program proofs.

A LINUX KERNEL RUST UNSAFE_LIST MIR CODE

The MIR code shown in Code listings A.0.1 to A.0.3, was generated from [17] using the command `rustc -crate-type=lib -Zunpretty=mir unsafe_list.rs`, using version `rustc 1.66.0-nightly (c97d02cdb 2022-10-05)` of the *rustc* compiler. The code shown in Code listings A.0.1 to A.0.3, only show the function `remove`.

```
fn remove(_1: &mut List<A>, _2: &<A as Adapter>::EntryType) -> () {
    debug self => _1;
    debug entry => _2;
    let mut _0: ();
    let _3: &LinksInner<<A as Adapter>::EntryType>;
    let mut _4: &List<A>;
    let mut _5: NonNull<<A as Adapter>::EntryType>;
    let mut _6: &<A as Adapter>::EntryType;
    let mut _10: &mut MaybeUninit<LinksInner<<A as Adapter>::EntryType>>;
    let mut _11: *mut MaybeUninit<LinksInner<<A as Adapter>::EntryType>>;
    let mut _12: &UnsafeCell<MaybeUninit<LinksInner<<A as Adapter>::EntryType>>>;
    let _13: &Links<<A as Adapter>::EntryType>;
    let mut _14: &<A as Adapter>::EntryType;
    let _15: ();
    let mut _16: &mut MaybeUninit<LinksInner<<A as Adapter>::EntryType>>;
    let mut _17: bool;
    let mut _18: *const <A as Adapter>::EntryType;
    let mut _19: *mut <A as Adapter>::EntryType;
    let mut _20: NonNull<<A as Adapter>::EntryType>;
    let mut _21: *const <A as Adapter>::EntryType;
    let mut _22: Option<NonNull<<A as Adapter>::EntryType>>;
    let mut _23: NonNull<<A as Adapter>::EntryType>;
    let mut _24: &mut LinksInner<<A as Adapter>::EntryType>;
    let mut _25: &mut List<A>;
    let mut _26: NonNull<<A as Adapter>::EntryType>;
    let mut _27: NonNull<<A as Adapter>::EntryType>;
    let mut _28: &mut LinksInner<<A as Adapter>::EntryType>;
    let mut _29: &mut List<A>;
    let mut _30: NonNull<<A as Adapter>::EntryType>;
    let mut _31: bool;
    let mut _32: *const <A as Adapter>::EntryType;
    let mut _33: *mut <A as Adapter>::EntryType;
    let mut _34: NonNull<<A as Adapter>::EntryType>;
    let mut _35: Option<NonNull<<A as Adapter>::EntryType>>;
    let mut _36: *const <A as Adapter>::EntryType;
    let mut _37: Option<NonNull<<A as Adapter>::EntryType>>;
    let mut _38: NonNull<<A as Adapter>::EntryType>;
```

Code listing A.0.1: MIR code for `unsafe_list` part 1

```
    bb0: {
        _4 = &(*_1);
        _6 = _2;
        _5 = <NonNull<<A as Adapter>::EntryType> as From<&<A as Adapter>::EntryType>>::from(move _6) -> bb1;
    }

    bb1: {
        _3 = List::<A>::inner_ref(move _4, move _5) -> bb2;
    }

    bb2: {
        _7 = ((*_3).0: NonNull<<A as Adapter>::EntryType>);
        _8 = ((*_3).1: NonNull<<A as Adapter>::EntryType>);
        _14 = _2;
        _13 = <A as Adapter>::to_links(move _14) -> bb3;
    }
    bb3: {
        _12 = &((*_13).0: UnsafeCell<MaybeUninit<LinksInner<<A as Adapter>::EntryType>>>);
        _11 = UnsafeCell::<MaybeUninit<LinksInner<<A as Adapter>::EntryType>>>::get(move _12) -> bb4;
    }

    bb4: {
        _10 = &mut (*_11);
        _9 = &mut (*_10);
        _16 = &mut (*_9);
        _15 = MaybeUninit::<LinksInner<<A as Adapter>::EntryType>>::assume_init_drop(move _16) -> bb5;
    }

    bb5: {
        _20 = _7;
        _19 = NonNull::<<A as Adapter>::EntryType>::as_ptr(move _20) -> bb6;
    }

    bb6: {
        _18 = move _19 as *const <A as Adapter>::EntryType (Pointer(MutToConstPointer));
        _21 = &raw const (*_2);
        _17 = eq::<<A as Adapter>::EntryType>(move _18, move _21) -> bb7;
    }

    bb7: {
        switchInt(move _17) -> [false: bb9, otherwise: bb8];
    }

    bb8: {
        Deinit(_22);
        discriminant(_22) = 0;
        ((*_1).0: Option<NonNull<<A as Adapter>::EntryType>>) = move _22;
        goto -> bb16;
    }
```

Code listing A.0.2: MIR code for unsafe_list part 2

```

        bb9: {
            _23 = _7;
            _25 = &mut (*_1);
            _26 = _8;
            _24 = List::<A>::inner(move _25, move _26) -> bb10;
        }

        bb10: {
            ((*_24).0: NonNull<<A as Adapter>::EntryType>) = move _23;
            _27 = _8;
            _29 = &mut (*_1);
            _30 = _7;
            _28 = List::<A>::inner(move _29, move _30) -> bb11;
        }

        bb11: {
            ((*_28).1: NonNull<<A as Adapter>::EntryType>) = move _27;
            _35 = ((*_1).0: Option<NonNull<<A as Adapter>::EntryType>>);
            _34 = Option::<NonNull<<A as Adapter>::EntryType>>::unwrap(move _35) -> bb12;
        }

        bb12: {
            _33 = NonNull::<<A as Adapter>::EntryType>::as_ptr(move _34) -> bb13;
        }

        bb13: {
           _32 = move _33 as *const <A as Adapter>::EntryType (Pointer(MutToConstPointer));
            _36 = &raw const (*_2);
            _31 = eq::<<A as Adapter>::EntryType>(move _32, move _36) -> bb14;
        }

        bb14: {
            switchInt(move _31) -> [false: bb16, otherwise: bb15];
        }

        bb15: {
            _38 = _7;
            Deinit(_37);
            ((_37 as Some).0: NonNull<<A as Adapter>::EntryType>) = move _38;
            discriminant(_37) = 1;
            ((*_1).0: Option<NonNull<<A as Adapter>::EntryType>>) = move _37;
            goto -> bb16;
        }

        bb16: {
            return;
        }
    }
```

Code listing A.0.3: MIR code for `unsafe_list` part 3

B LINUX KERNEL RUST UNSAFE_LIST

```rust
unsafe fn inner(&mut self, ptr: NonNull<A::EntryType>)
    -> &mut LinksInner<A::EntryType> {
    unsafe { (*A::to_links(ptr.as_ref()).0.get()).assume_init_mut() }
}
unsafe fn inner_ref(&self, ptr: NonNull<A::EntryType>)
    -> &LinksInner<A::EntryType> {
    unsafe { (*A::to_links(ptr.as_ref()).0.get()).assume_init_ref() }
}
```

Code listing B.0.1: `inner`, returns a mutable reference to the `LinksInner` of a given list `self`, and `inner_ref` returns a reference. All code in this listing is wrapped in `impl<A: Adapter +?Sized> List<A>`.

C RUST CODE USED AS EXAMPLES IN ??

```
fn test(x : bool) -> bool {
    if x == false {
        return false
    } else
    {
        return true
    }

}
```

Code listing C.0.1: Rust code used in the example shown in Section 3.1 for `swtichInt`.

```
fn test(mut x :  bool) -> bool {

    while true  {
        x = false;
    }
    return x;
}
```

Code listing C.0.2: Rust code used in the example shown in Section 3.1 loop constructs.

D UNSAFE_LIST PALE MODEL UNSIMPLIFIED

```
type List = {
    data first: EntryType;
}
type EntryType = {
    bool value;
    data links : LinksInner;
}

type LinksInner = {
    data next : EntryType ;
    pointer prev : EntryType [];
}

data none : EntryType;
data dropped_Links : LinksInner; /*Droped value */
data dropped_EntryType : EntryType; /*Droped value */

data 4_test : List;
data 2_test : EntryType;
data 3_test : LinksInner;
data 6_test : EntryType;
data 5_test : EntryType;

pointer 1_self : List;
pointer 2_entry: EntryType;

pointer 7_test : EntryType;
pointer 8_test_bb2 : EntryType;

pointer 10_test : LinksInner;
pointer 11_test : LinksInner;
pointer 12_test : LinksInner;
pointer 13_test : LinksInner;

pointer 16_test : LinksInner;

```

```
bool 17_test;

pointer 18_test : EntryType;
pointer 19_test : EntryType;
pointer 20_test : EntryType;
pointer 21_test : EntryType;

pointer 22_test : EntryType;
pointer 23_test : EntryType;
pointer 24_test : LinksInner;
pointer 25_test : List;
pointer 26_test : EntryType;
pointer 27_test : EntryType;
pointer 28_test : LinksInner;
pointer 29_test : List;
pointer 30_test : EntryType;
bool 31_test;
pointer 32_test : EntryType;
pointer 33_test : EntryType;
pointer 34_test : EntryType;
pointer 35_test : EntryType;
pointer 36_test : EntryType;
pointer 38_test : EntryType;
pointer 37_test : EntryType;

proc remove(pointer 1_self : List, pointer 2_entry : EntryType
 ↪  ) : void
[1_self !=null & 2_entry !=null]
{
  4_test = 1_self;
  6_test = 2_entry;
  5_test = 6_test;
  bb1() [true];
}
[true]



proc bb1() : void
[true]
```

```
{
  3_test = inner_ref(4_test , 5_test) [true]; /* inner */
  bb2() [true];
}
[true]

proc bb2() : void
[true]
{
  pointer 14_test_bb2 : EntryType;

  7_test = 3_test.next; /* next */
  8_test_bb2 = 3_test.next; /* prev*/
  14_test_bb2 = 2_test;
  13_test = to_links(14_test_bb2) [true]; /* inner */
  bb4() [true]; /*bb3 abstacted away*/
}
[true]

proc bb4() : void
[true]
{
  pointer 9_test_bb4 : LinksInner;
  10_test = 11_test;
  9_test_bb4 = 10_test;
  16_test = 9_test_bb4;
  16_test = dropped_Links;
  bb5() [true];
}
[true]

proc bb5() : void
[true]
{
  20_test = 7_test;
  19_test = 20_test;
  bb6() [true];
}
[true]

```

```
proc bb6() : void
[true]
{
  18_test = 19_test;
  21_test = 2_test;
  17_test = 18_test = 21_test;
  bb7() [true];
}
[true]

proc bb7() : void
[true]
{
  if (17_test) {
    bb9() [true];
  } else {
    bb8() [true];
  }
}
[true]

proc bb8() : void
[true]
{
  22_test = dropped_EntryType;
  22_test = none;
  1_self.first = 22_test;
  bb16() [true];
}
[true]

proc bb9() : void
[true]
{
  23_test = 7_test;
  25_test = 1_self;
  26_test = 8_test_bb2;
  24_test = inner(25_test, 26_test) [true];
  bb10() [true];
}
```

```
[true]


proc bb10() : void
[true]
{
  24_test.next = 23_test;
  27_test = 8_test_bb2;
  29_test = 1_self;
  30_test = 7_test;
  28_test = inner(29_test, 30_test) [true];
  bb11() [true];
}
[true]

proc bb11() : void
[true]
{
  28_test.prev = 27_test;
  35_test = 1_self.first;
  34_test = 35_test;
  bb12() [true];
}
[true]


proc bb12() : void
[true]
{
  33_test = 35_test;
  bb13() [true];
}
[true]


proc bb13() : void
[true]
{
  32_test = 33_test;
  36_test = 2_entry;
  31_test = 32_test = 36_test;
  bb14() [true];
}
[true]

proc bb14() : void
[true]
{
  if (31_test) {
    bb15() [true];
  } else {
    bb16() [true];
  }
}
[true]



proc bb15() : void
[true]
{
  38_test = 7_test;
  37_test = dropped_EntryType;
  37_test = 38_test;
  1_self.first = 37_test;

}
[true]



proc bb16() : void
[true]
{
}
[true]

proc to_links(pointer entry : EntryType) : LinksInner
[true]
{
  return entry.links;
}
[true]


proc inner_ref(pointer 1_self : List, pointer 2_entry :
↪  EntryType) : LinksInner
[true]
{
  pointer entry_links : LinksInner;
  entry_links = 2_entry.links;

  return entry_links;
}
[true]


proc inner(pointer 1_self : List, pointer 2_entry : EntryType)
↪  : LinksInner
[true]
{
  pointer entry_links : LinksInner;
  entry_links = 2_entry.links;

  return entry_links;
}
[true]
```

BIBLIOGRAPHY

[1] Anders Hahn Boelt and Mathias Krog Holdgaard. *Is the Usage of Unsafe Rust a Threat to Safe Rust?* Tech. rep. 9rd semester, Pre-Specialisation. Dept. Computer science, Allborg University, 2023. URL: https://projekter.aau.dk/projekter/da/studentthesis/is-the-usage-of-unsafe-rust-a-threat-to-safe-rust(77fa2217-7015-4727-a0aa-dcb28f6f2d9b).html (visited on 06/02/2023).

[2] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. "Creusot: A foundry for the deductive verification of rust programs". In: *Formal Methods and Software Engineering* (2022), pp. 90–105. DOI: 10.1007/978-3-031-17244-1_6.

[3] Elizabeth Dietrich. "A beginner guide to Iris, Coq and separation logic". In: *CoRR* abs/2105.12077 (2021). arXiv: 2105.12077. URL: https://arxiv.org/abs/2105.12077.

[4] Simon Vinberg Andersen Felix Cho Petersen Mathias Knøsgaard Kristensen. "RUST'S BORROW SYSTEM IN STATIC ANALYSIS". 2022.

[5] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.

[6] Nils Klarlund and Anders Møller. *Mona version 1.4: User manual*. BRICS, Department of Computer Science, University of Aarhus Denmark, 2001.

[7] Nils Klarlund and Michael I. Schwartzbach. "Graph Types". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '93. Charleston, South Carolina, USA: Association for Computing Machinery, 1993, pp. 196–205. ISBN: 0897915607. DOI: 10.1145/158511.158628.

[8] Thomas Lengauer and Robert Endre Tarjan. "A Fast Algorithm for Finding Dominators in a Flowgraph". In: *ACM Trans. Program. Lang. Syst.* 1.1 (Jan. 1979), pp. 121–141. ISSN: 0164-0925. DOI: 10.1145/357062.357071. URL: https://doi.org/10.1145/357062.357071.

[9] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. "RustHornBelt: A Semantic Foundation for Functional Verification of Rust Programs with Unsafe Code". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 841–856. ISBN: 9781450392655. DOI: 10.1145/3519939.3523704.

[10] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. "RustHorn: CHC-based verification for Rust programs". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43.4 (2021), pp. 1–54.

[11] *Mir Proposal*. URL: https://rust-lang.github.io/rfcs/1211-mir.html (visited on 06/02/2023).

[12] Anders Møller. *PALE syntax*. URL: https://www.brics.dk/pale/grammar.html#basic (visited on 05/23/2023).

[13] Anders Møller and Michael I. Schwartzbach. "The Pointer Assertion Logic Engine". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Also in SIGPLAN Notices 36(5) (May 2001). June 2001.

[14] Peter O'Hearn. "Separation Logic". In: *Commun. ACM* 62.2 (Jan. 2019), pp. 86–95. ISSN: 0001-0782. DOI: 10.1145/3211968. URL: https://doi.org/10.1145/3211968.

[15] J.C. Reynolds. "Separation logic: a logic for shared mutable data structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[16] *Rust for Linux*. URL: https://rust-for-linux.com/#rust-for-linux (visited on 06/02/2023).

[17] Filho Wedson Almeida. *Intrusive circular doubly-linked lists*. URL: https://github.com/Rust-for-Linux/linux/blob/bc22545f38d74473cfef3e9fd65432733435b79f/rust/kernel/unsafe_list.rs (visited on 06/02/2023).